Smart Pathfinding:
Extending Pathfinding with Agent Capabilities

Department of Computer, Control and Management Engineering
Dottorato di Ricerca in Computer Science – XXIX Ciclo

Candidate
Davide Aversa
ID number 1148462

Thesis Advisor
Prof. Stavros Vassos

Co-Advisor
Prof. Massimo Mecella

A thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
2016/2017
To Gioia
Contents

1 Introduction 1

2 Literature Review 7
  2.1 Classical pathfinding ................................. 7
    2.1.1 World representation .............................. 7
    2.1.2 Graph search .................................... 10
  2.2 Incremental search .................................. 13
    2.2.1 Incremental update of the $g$-value ............... 14
    2.2.2 Incremental update of the $h$-value ............... 18
    2.2.3 Sub-tree recomputation ............................ 21
  2.3 Agent-Centered Search ................................ 22
  2.4 Symmetry-Breaking pathfinding ........................ 25
  2.5 Hierarchical and abstraction-based pathfinding .......... 27
  2.6 Multi-Agent Pathfinding ............................... 31

I Inventory-Aware Pathfinding 35

3 The Inventory-Aware Pathfinding Problem (IAP) 37
  3.1 Motivation and examples ............................... 38
  3.2 Related work ......................................... 42
  3.3 The scope of the IAP problem .......................... 42

4 The General IAP Problem and Variants 45
  4.1 The General IAP problem (G-IAP) ....................... 45
  4.2 The effects of drop actions variations ................ 47
  4.3 The unlock relation assumptions ....................... 48
  4.4 Inventory Size ....................................... 50
  4.5 Classification for IAP problems ....................... 50
    4.5.1 IAP expressive power for different Unlock assumptions .... 50
    4.5.2 IAP expressive power for different drop specifications .... 51
  4.6 The Layered-Items Graph Model (LIG) .................. 53
    4.6.1 From the standard pathfinding graph to the LIG model .... 54
    4.6.2 IAP problem in the LIG model ..................... 55
  4.7 The Unbounded Monotonic IAP problem ................. 57
    4.7.1 Properties of LIG under IAP-SU-UM assumption ......... 57
  4.8 Other IAP problem specifications ...................... 58
    4.8.1 Pick and Drop cost ................................ 58
    4.8.2 Inventory encumbrance ............................. 59
    4.8.3 Persistent item effect ............................. 59


5 **Inventory-Aware JPS (IA-JPS)** 61

5.1 The classical Jump Point Search .............................. 61
   5.1.1 Overview of JPS ........................................ 61
   5.1.2 Pruning mechanism ..................................... 63
   5.1.3 Jumping procedure ..................................... 63
5.2 Extending JPS for IAP ...................................... 64
   5.2.1 IA-JPS in the LIG model ................................. 65
5.3 Experiments for IA-JPS .................................... 66
   5.3.1 The experimental setting ................................. 66
   5.3.2 Experiment description and results ....................... 68
5.4 Conclusion ...................................................... 74

6 **Optimization: Filtering the Search Space** 77

6.1 Optimal filtering in Monotonic IAP ............................ 77
6.2 Sub-optimal filtering .......................................... 79
   6.2.1 Local \( \theta \)-filtering ................................ 79
   6.2.2 Fringe (global) \( \theta \)-filtering ......................... 81
6.3 Implementation of filtering in IA-JPS: F-IA-JPS ............... 82
   6.3.1 Descendant Pruning .................................... 83
   6.3.2 IA-JPS specific implementation problems ............... 83
6.4 Experiments with F-IA-JPS .................................. 84
   6.4.1 Experiment Parameters .................................. 85
   6.4.2 Experiment 1: No-Path Scenario ......................... 86
   6.4.3 Experiment 2: Not required items ....................... 87
   6.4.4 Experiment 3: Filtering Sub-Optimality ................. 88

7 **Optimization: Map Preprocessing** 91

7.1 The IAP preprocessing algorithm ............................. 92
   7.1.1 Step 1: The offline preprocessing ....................... 92
   7.1.2 Step 2: the online pre-search ......................... 93
   7.1.3 Step 3: IAP search with hints .......................... 95
7.2 Strategies for choosing \( \hat{K} \) .................................. 96
7.3 Recursion and convergence in preprocessing .................. 98
7.4 Experiments for map preprocessing ........................... 99

8 **Conclusions and Future Work on IAP** 103

8.1 Heuristic improvements ...................................... 103
   8.1.1 True-Distance Heuristic ................................ 104
   8.1.2 Preprocessing Based Heuristic ............................ 104

II **Belief-Aware Pathfinding** 105

9 **The Belief-Aware Pathfinding Problem (BAP)** 107

9.1 Motivation and examples ..................................... 107
9.2 Literature review ............................................. 109
9.3 Definition of the Belief-Aware Pathfinding problem ........... 110
9.4 The planning-execution cycle of BAP .......................... 112
Chapter 1

Introduction

Character navigation is a core component in many game genres. In early videogames, movement was hard-coded by developers with characters following predetermined and fixed paths on the screen\(^1\) or generated by a fixed set of deterministic rules\(^2\). However, videogames quickly started to become more complex and fixed-moving or rule-based characters began to appear aimless, unrealistic and, more often, incapable of achieving their purpose at all. Tasks such as reaching the point where the player clicked in Real Time Strategy (RTS) games or move to the nearest cover point in First Person Shooter (FPS) games requires that the character is able to calculate at real time an appropriate path to the destination.

The problem of computing a path that can bring the character from one location to another within a game world in a reasonable way, e.g., by avoiding obstacles and using the fastest way to go, is called **pathfinding** and search-based algorithms are a common solution to this problem (Figure 1.1). However, while in other fields searching for the pathfinding problem is limited to find the *shortest-path* between two locations, in the context of videogames this is usually not enough. As the pursuit of realism in videogames increases, the pathfinding problem is loaded with many additional requirements and constraints which have the purpose of producing more believable results for the navigation of characters. For instance, a lot of work is carried out in the videogames industry in making the resulting paths smoother, triggering the right animations or interacting with the other characters in a more human-like way \([70, 97, 1, 77]\).

On the other hand, there are also other aspects of navigation that have been studied in other AI research fields, e.g., robotics, knowledge representation, agent beliefs, agent communication, task-based reasoning, which have been explored less in the game industry in conjunction with pathfinding. A critical observation is that pathfinding in videogames is typically treated *separately from the other AI aspects* of game characters that may specify for instance the memory of the character and the task-based decision-making. In contrast, in the AI literature, it is common to treat pathfinding *in conjunction with other agent-related aspects*, such as for instance the beliefs of the agent, and in fact provide more “holistic” agent architectures.

As it is clear that “holistic” AI agent architectures and approaches are not suitable for game characters (at least for the time being), the challenge has always

---

\(^1\)This can be clearly seen in the movement of the spaceships in games such as *Space Invaders* (Taito, 1978) or *Galaga* (Namco, 1981).

\(^2\)We can find an excellent example of how simple reactive rules can generate addictive AI behaviors in *Pac-Man* (Namco, 1980) \([71]\).

\(^3\)Image courtesy of Red Blob Games [http://www.redblobgames.com/pathfinding/a-star/introduction.html](http://www.redblobgames.com/pathfinding/a-star/introduction.html).
been to identify practical subsets of the developed AI frameworks that can be integrated in the game developing pipeline in a clear and robust way. This applies also to pathfinding which has indeed benefited from advances in heuristic search, e.g., [11], in symmetry breaking representations that reduce memory and expanded nodes, e.g., [72, 31], etc.

At the same time pathfinding is a particular AI aspect of game characters as it is essentially the place in the AI of game characters where actually resource-heavy search takes place for most of the games. This provides a wide spread “entry point” for other AI-related search-based techniques to be combined with pathfinding and provide a more sophisticated navigation decision-making. In other words, much like in AI research navigation planning is often treated a part of a more general agent architecture and decision-making, in videogames pathfinding can be seen as a concrete basis for attaching agent-related aspects and arrive to mini-agent architectures with pathfinding at their core.

For a better understanding, we can look at Figure 1.2 for a comparison between the actual game character AI model and the proposed one. In Figure 1.2a we can see that the pathfinding component barely touches the decision-making layer.
Pathfinding is considered just an “actuator” component in the same way of animations or local collision avoidance. However, in this way, we are not fully taking advantage of the decision-making capabilities for a more sophisticated path planning solution. With smart pathfinding work towards this direction by expanding the pathfinding scope up to the decision-making layer (Figure 1.2b).

In this work, we explore two main directions in smart pathfinding solutions. In Part I we will explore the Inventory Aware Pathfinding (IAP) problem, in which we try to answer the question “What is the fastest way for a character to go from point A to point B when they can also pick up items that open blocked pathways?”. In Part II, instead, we will look at the Belief Aware Pathfinding (BAP) problem, in which we try to answer the question “What is the most promising way for a character to go from point A to point B taking into account their recent observations?”.

Each of these questions can be answered by taking advantage of well-known AI agent-based approaches, and the challenge is to identify appropriate forms for restricted AI problems that fit well in the videogames setting and can provide practical solutions. For instance, pathfinding with capabilities can be formulated as a special case of classical planning [25, 67] that merges navigational pathfinding with task-based deliberation. Furthermore, we identify some future opportunities that arise in this “pathfinding-centric” game character approaches and discuss promising next steps for research and future work.

Before moving on, it is important to identify some example game scenarios that will be useful in the next chapters.

Scenario S1: turn-based tactical game.

The first game scenario is a turn-based tactical shooter similar to the games of the XCOM series. In this game the player controls a small group of characters. The characters can move by a fixed set of steps per turn. The goal of the game is to eliminate all the enemies in the map.

In this game, much of the fun and challenge for the player lies in the ability to interact with the environment exploiting the map structure to overcome the numeric

advantage of the computer opponents. For this reason, the complex and detailed game map allows various types of interactions: doors can be closed and opened by finding a key or they can be blown up if the player (or the characters) can find the right tools; rooms can be set on fire to slowdown or trap characters, etc. In this scenario, smart pathfinding can be used for the enemy AI in order to interact in a believable way with the complex environment.

Scenario S2: AI companion character in FPS.

The second game scenario is a First-Person-Shooter (FPS) action game with an AI companion. In fast-paced games, smarter pathfinding (or even smarter decision-making) is not the number one requirement for the enemy AI because of their short lifespan that makes difficult for the player to notice complex behaviors. However, to be successful, AI companions are required to behave at a much higher believability level, closer to a human-like level of abilities. As the companion follows the player for big parts of gameplay, any behavioral flaws can be easily noticed during the player experience.

In this scenario, the AI companion can receive commands by the user. Some commands may require planning capabilities and it is expected that the AI companion can handle the command by itself without the user providing a step-by-step solution. There is also the possibility to use the AI companion as a source of information, for instance, when reaching a closed door, the companion can scout in order to suggest a safer or faster alternative route to follow.

The ability of the companion to make decisions and reasonable mistakes is important in this setting. As navigational aspects and game interactions are very much related to this kind of behavior, smart pathfinding can provide concrete methods in generating such behaviors.

Reading paths

The thesis covers both a theoretical view that lays out the general framework for the problems we consider, as well as a more practical view that focuses on approaches that can be embedded easily inside the programming environment of videogame developers.

As a help to interested readers to find faster the type of information they are most interested in, we propose two paths each of which focuses on one of the two aforementioned views:


- For the “practical approaches” to smart pathfinding with emphasis kn implementation and evaluation, interested readers can focus on Chapters 1 “Introduction”, Chapter 3 “The Inventory-Aware Pathfinding”, Chapter 5 “Inventory-Aware JPS”, Section 6.2 “Sub-optimal filtering in IAP”, Section 6.3 “Implementation of filtering in IA-JPS”, Chapters 7 “Optimization: Map Preprocessing”, Chapter 9 “The Belief-Aware Pathfinding Problem”, Chapter
Contributions

In this work, we contribute to Smart Pathfinding in the following three directions:

C1. Specification of a formal framework for Inventory-Aware Pathfinding and search-based methods for Inventory-Aware Pathfinding. We will cover this direction in Chapters 3, 4 and 5.

C2. Preprocessing and optimization methods for Inventory-Aware Pathfinding. We will cover this direction in Chapters 6 and 7.

C3. Specification of a formal framework for Belief-Aware Pathfinding and search-based methods for Belief-Aware Pathfinding. We will cover this direction in Chapters 9, 10 and 11.

Part of this work is reported in the following publications:


Finally, there are some additional original articles and software:


• “movingai2json”, a small utility package for working with the MovingAI map format. https://www.npmjs.com/package/movingai2json.

• “YoshiX”, a unit-test like framework for running experiments over a large set of parameters. https://github.com/THeK3nger/yoshix.
Chapter 2

Literature Review

As we said in the introduction, pathfinding is concerned with finding a path, e.g., a sequence of locations, that bring a character from a location A to location B in an arbitrary environment. Typically, the problem specification requires the fastest path to be found. In particular in videogames, apart for the extensive literature on solving the basic form of pathfinding in efficient ways with respect to runtime, e.g., [21, 35, 31, 34] and memory, e.g., [87], there are few directions that identify interesting pathfinding-related problems that look beyond “going from point A to point B” behaviors. For instance, Strutevant [88] identifies challenges in the single-agent and multi-agent setting, as well as challenges related to a “social aspect” of pathfinding, e.g., when a character needs to find a seat in a room or approaches another character in order to initiate a conversation in a polite way.

In this thesis, we are concerned with identifying similar scenarios. We focus less on directions for improving the aesthetics of the resulting paths, and are concerned more on the way information and available interactions in the game world can be used algorithmically to provide different (not “smoother”) paths which can support a more interesting or believable behavior for characters with respect to navigation.

A paper that gives a similar survey on pathfinding approaches in games is the work of Botea et al [14]. There, the authors distinguish the methods in these main categories: Hierarchical Abstraction, Symmetry Elimination, Triangulation-Based Pathfinding, Real-Time Search and Multi-Agent Pathfinding, and give an outline for the idea behind the methods. Our work can be seen as an updated, larger map of the area, that also includes more classical categories such as Incremental Search and Agent-Centered Search, along with a more extensive report on the representative methods in each one. Finally, here we illustrate the algorithmic methods with detailed pseudo code and provide insights that relate the methods with each other.

2.1 Classical pathfinding

2.1.1 World representation

Pathfinding is the problem of moving a character from one point to the other. However, there is no algorithm that can work directly on the geometry of the real or game world in which the character lives. Most of the algorithms that we will see later rely, in fact, on a simplified abstraction of the environment in the form of a pathfinding graph. The pathfinding graph is a undirected weighted graph in which each vertex (also called node) assumes the role of a representative location in the world and each edge \( (v_i, v_j) \) represents the cost for the character for going from \( v_i \)
to \( v_j \).\(^1\)

Still, it is clear that no real map or game map comes with a perfect graph representation attached. Therefore, the first step is to abstract this graph directly from the geometry of the map. There are several ways to perform this step.

### Waypoints (or Point Graph)

![Figure 2.1. An example of Point Graph representation for a map.](image)

The most straightforward way to abstract the complex geometry of a map is to manually identify relevant positions in the map and use them as vertexes for the pathfinding graph. This has the advantage of being very easy to understand and produce a limited number of vertexes. However, the necessity of manual intervention and the fact that pathfinding is limited to a very restricted set of locations makes the point graph a less popular approach \[59\].

### Tile Graph

Another simple but widely used abstraction is the tile graph \[59\]. This representation uses a bidimensional grid to tessellate the world map into regular squares\(^2\) called tiles.

A tile graph is not usually represented explicitly in graph form, instead, they are represented by the set of individual tiles and their connection with the neighbors’ tiles. For example, a generic map \( M \) can be represented in tile form as a set of tiles \( M = \{m_{rc}\} \) each of which represents a square \( l \times l \) in the real environment. A tile can be free or blocked depending on the fact that the corresponding real-world square is occupied by a physical obstacle or not. Finally, we define the neighbors of a tile \( m_{rc} \) as the set of free tiles that are directly accessible from \( m_{rc} \). Note that a

---

\(^1\)Usually this cost is simply the distance between the location represented by \( v_i \) and the location represented by \( v_j \). However, in general, this cost can take into account also other properties, e.g., traversing a swamp or water can be more difficult than passing on simple ground.

\(^2\)Hexagons are another popular polygon used for tile tessellation. However, in order to not make the discussion more complicated than necessary, we will refer only to square tiles.
2.1 Classical pathfinding

Figure 2.2. An example of Tile Map representation for a map. Note, in gray, the tiles that are partially obstructed by the geometry and are therefore considered as blocked.

The tile map defined in this way can be easily represented in a graph by considering a node for each tile\(^3\) and an edge connecting each node with its neighbors.\(^4\)

Navigation Mesh

Figure 2.3. An example of Navigation Mesh for the given world geometry. In this example, we used the polygon center of mass as graph node.

The main problem with tile graphs is that many areas of the map can be over-represented. Imagine a map with very large rooms or open areas with few obstacles.

\(^3\)For this reason, when we work with grid or tile map the terms node, tile and location are basically the same. In this work we will use node to all these elements whenever is possible.

\(^4\)The cost of these edges are 1 for the horizontal and vertical neighbors, and \(\sqrt{2}\) for all the diagonal neighbors.
Because the tiles are all the same size, in those regions, we can end up having hundreds of tiles even if we are not interested in a detailed path in that area. For instance, this can be only a transition area for many game characters and we just want them to walk in a straight line without exploring every tile. On the other hand, increasing the tile size is not a viable solution. This will effectively reduce the number of vertexes in these open regions, but, at the same time, pathfinding can become difficult—or even impossible—in all the narrow regions and small rooms that are now under-represented.

Navigation Mesh (often called navmesh) is a solution to this problem. They have been known since the 1980s with the name of meadow maps [3], but they were popularized in digital entertainment only in the 2000 with the explosion of massive 3D environments in games. Since then, navigation meshes have become extremely popular in game industry and they are actually one of the basic world representations included out-of-the-box in all the modern game engines and development tools.

The idea behind navmesh is extremely simple. Instead of tessellating the world map with fixed polygons, we can cover the navigable space of the map with a mesh of arbitrary shaped convex unobstructed polygons (usually triangles or quads). This has two major benefits: first, we are no more constrained to use the same size for all the regions of the map but we can use a fine-grained set of polygons for the narrow regions and large polygons for the open areas; second, because all the polygons are convex and unobstructed is guaranteed that we can move inside the single polygon in a straight line.

The resulting pathfinding graph for a navmesh can be done in two ways. In the first one we can create a vertex representing the geometric center of the polygon and an edge connecting each adjacent polygon in the mesh. The second method, often referred as portal model, assigns a vertex for each edge between two pairs of adjacent polygons and connects every vertex with the location on the other edges of the two polygons. The portal model generates more vertexes and edges in the pathfinding graph but allows an easier smoothing of the resulting path and, in general, produces a more natural character movement [59].

2.1.2 Graph search

Once we obtain a graph representation of the game world, we can use it to find the shortest path between the starting and destination vertexes. This is an old problem in graph theory that goes back to the early days of computer science and many solutions have been proposed in literature. In the next paragraphs, we will briefly talk about the two main graph search algorithms that are still widely used to solve pathfinding problems.

Dijkstra algorithm

The Dijkstra Algorithm [21] is one of the first proposed solution to the shortest path problem in graph theory. However, the mathematical problem Dijkstra wanted to solve was slightly different from the pathfinding problem. Where pathfinding tries to find the shortest path between a single starting point and a single destination, the Dijkstra algorithm finds the shortest path between a starting point and everywhere else.

Because of this, the Dijkstra is very wasteful in resources and computational time if used as a pathfinding algorithm. Nonetheless, it is widely used as a tool to identify properties of a game map and as a tool for tactical analysis in heavy
multiagent settings \cite{59}. Moreover, it introduces basic concepts that are used in more advanced algorithms such as A* \cite{35} or D* \cite{84,85} that we will see shortly.

Algorithm 1 Dijkstra Algorithm pseudo-code

1: function Dijkstra(Graph, start)
2:   Q ← \text{Set}()
3:   \text{// Initialization}
4:   for each \( v \in \text{Graph} \) do
5:     costSoFar\[v]\ ← \text{Inf}
6:     cameFrom\[v]\ ← \text{Null}
7:     ADD(Q,v)
8:   costSoFar[start] ← 0
9:   while Q is not empty do
10:      \( u \) ← vertex in Q with minimum distance\[u\]
11:      REMOVE(Q,u)
12:      for each \( v \in \text{NEIGHBORS(Graph, u)} \) do
13:         alt ← costSoFar\[u\] + cost(Graph,u,v)
14:         if alt < costSoFar\[v]\ then
15:            costSoFar\[v]\ ← alt
16:            cameFrom\[v]\ ← u
17:   return distance, previous

Dijkstra algorithm works as follows. In the initialization phase, the algorithm puts every node of the graph in the set of unvisited nodes and writes a distance label for each node in the graph with the value 0, for the starting node, and infinite for all the other ones. Then the algorithm extracts from the unvisited set the node \( u \) with the smallest distance value (in the first iteration, it is the starting node). Then, the algorithm explores all the unvisited neighbors of \( u \) and updates the distance labels by adding together the distance of \( u \) and the cost of going from \( u \) to the current neighbor node (Algorithm 1, line 15). If this new distance value is smaller than the previous one, the algorithm updates a label pointing to the node it comes from (\( u \)). The algorithm ends, when all the nodes in the graph have been explored.

The result of the algorithm is a list of previous pointers. In this way, choosing any node \( n \) on the graph, we can follow the chain of previous pointers up to the starting node. The resulting sequence of nodes represents the shortest path from \( n \) to the starting position.

The A* algorithm

The main problem with the Dijkstra algorithm for pathfinding is that it completely ignores the goal position. Even if we tweak the Dijkstra algorithm to consider the destination (for instance, be terminating the algorithm as soon as the goal is reached), it will continue to expand first all the neighbors of the starting node, and then all the neighbors’ neighbors, and so on and so forth. In other words, it searches in all directions with the same priority producing a circular fringe. However, it is clear that when we are looking for a route to a specific location, there are some directions that are more promising than others, namely the nodes in the direction of the goal. An algorithm that takes into account this kind of information is called informed search algorithm or heuristic search algorithm.

The A* algorithm \cite{35} is probably the most used heuristic search algorithm, to the extent that it is commonly used as a synonymous of pathfinding in videogames.

\footnote{We define with fringe the set of nodes in a graph that are candidate to be expanded in the next iteration of the algorithm.}
The main advantage over Dijkstra is that, when we need to choose the next note to be expanded, rather than just picking the node with the minimum distance value, A* takes the “most promising node” that can lead to the final shortest path. The way the algorithm can decide if a node is the “most promising” or not is encoded in the heuristic function \( h(n) \), a function that returns an estimate of the distance of any node from the goal.\(^6\)

\[ f(n) = g(n) + h(n) \tag{2.1} \]

where \( g(n) \) is the \( g \)-score, representing, similarly to the “costSoFar” value of the Dijkstra algorithm, the cost of the best path from the starting position to the node \( n \), and \( h(n) \) the heuristic function.

Choosing a good heuristic function can have dramatic effects on the performance of the algorithm. If a heuristic \( h_1 \) is better than \( h_2 \)—that is, \( h_1 \) returns always a value

---

\(^6\)We can note that we can reduce A* to the behavior of the Dijkstra algorithm by using a heuristic function that returns \( h(n) = 0 \) for all the nodes in the graph.
closer to the real cost of the path from \( n \) to the goal than \( h_2 \) then the algorithm using \( h_1 \) will expand less nodes in order to solve the pathfinding problem than the same algorithm using \( h_2 \).\cite{20} An important desired property of the heuristic function is the \textit{admissibility}. A heuristic function is \textit{admissible} if and only if it never overestimates the real cost to get to the nearest goal node. The A* algorithm guarantees that the resulting path is optimal only if we use an admissible heuristic.

Finding a good heuristic function is problem specific but, in pathfinding, two common choices are the \textit{Euclidean distance}, the traditional way to measure distance on a plane, and the \textit{Manhattan Distance}, the distance between two points in a grid based on a strictly horizontal and vertical path. However, note that, depending on the problem, they may not be always admissible. For instance, the Manhattan distance it is not admissible if we allow diagonal movement on a grid\footnote{It is possible to extend Manhattan distance to be admissible on grids that allow diagonal movements. This is called \textit{octile distance} \cite{34}.} while the Euclidean distance may not be admissible if we allow portals between different location of the same map.

Another desirable property of the heuristic function is \textit{consistency}. A heuristic function is \textit{consistent} if and only if \( h(\text{goal}, \text{goal}) = 0 \) and if it respects the \textit{triangular inequality}, that is

\[
h(n, n') \leq c(n, n') + h(n, \text{succ}(n, a)) \tag{2.2}
\]

for every node \( n \) and for every action \( a \) that can be done in \( n \). In other words, a heuristic function is consistent if its estimate is always less than or equal to the estimated distance from any neighboring vertex to the goal, plus the step cost of reaching that neighbor.

Note that every consistent heuristic is also admissible, however, it is possible to find an admissible heuristic that is not consistent. For instance, imagine a heuristic function for a grid map in which we can move only in the horizontal or vertical direction. If we chose a heuristic function that for each node randomly select a value between the Euclidean and the Manhattan distance, this heuristic is admissible (because both Euclidean and Manhattan distance are admissible in this scenario) however is not consistent because there is no clear relationship between heuristic estimations at each step.

### 2.2 Incremental search

\( A^* \) (and all the other simple pathfinding algorithms) start from two very strong assumptions. First, they assume that the map is perfectly known by the agent during the search time. Second, they assume that the map never changes from when the path is found and the execution is completed.

These two assumptions are difficult to satisfy in practice. In the real-world situations, it is probable that the agent doesn’t know precisely the state of all the environment and that such state often changes as well. In video games, because the game environment is stored in the computer memory, we can have a perfect representation of the world, but the environment is constantly changing by the player’s actions. Moreover, in order to create a good illusion of intelligence in the game characters, sometimes we do not want to build agents that have a perfect knowledge of the map and, therefore, we still have to assume partial knowledge of the map.
The main problem of A* with unknown or dynamically changing environments is wasted computational time. The A* algorithm, in fact, is fully contained in its execution; after the algorithm returns a path all the search done is completely discarded. If during navigation the character finds a discrepancy between the search pathfinding graph and the real world, the agent must start a new pathfinding search from scratch. In the case of unknown or dynamic environments these discrepancies can be frequent. It would be more efficient to save the information obtained in the previous search in order to solve the new path faster. This is what Incremental Search Algorithms try to exploit.

There are three ways of performing incremental search, each of which defines a special class of incremental search algorithms.

### 2.2.1 Incremental update of the g-value

In the first class, the algorithms modify the cost of the connections in the pathfinding graph associated to the map (namely the g-value). In other words, these algorithms transform the original pathfinding graph into a new, up to date, graph based on the current exploration of the map. We now describe three representative algorithms of this class.

#### The D* algorithm

The D* algorithm is one of the first successful proposals for an incremental-search algorithm [84, 85] that is able to navigate unknown or dynamic maps. First, when D* needs to search in unknown regions of the map, it uses the free space assumption, therefore, at the beginning, it considers that all the tiles on the maps are free to pass. Then applies the same techniques of A* but adds two new states for the search nodes: RAISE and LOWER.

To understand how D* works, first we need to define the following functions:

- **cameFrom(X) = Y**, which returns the back pointer from the node X to Y. Because D* searches from the destination to the starting point, at every time, the sequence of back pointers represents the current optimal path computed from the agent current position X to the destination G.

- **t(X)**, given a node n returns the tag associated with it (among RAISE, LOWER, OPEN or CLOSED).

- **f(X)**, returns the sum of the optimal (in terms of cost) path from x to the goal. This can be computed following the back pointers from X to the goal and summing all the individual edge costs.

- **k(X)**, returns the estimated cost of the optimal path at the current state of knowledge. Every time in a state X we have that **k(X) < f(X)**, the state is marked with the RAISE tag; otherwise with the LOWER tag.

The complete pseudocode for the algorithm is given in Algorithm 3 and Algorithm 4. We can summarize the following main steps:

1. It searches for a path from the destination to the starting point using the actual knowledge of the map and using the free space assumption for everything that is unknown.
2. When an obstacle is detected on the current optimal path, all the nodes affected are put back into the open list and marked as RAISE.

3. The RAISE information is propagated backward along the parents (towards the starting point) increasing the cost in the pathfinding graph. If the RAISE node neighbors can be lowered in cost, then their parent node is updated, marked as LOWER and put back in the open list as well.

4. The propagation stops when the LOWER front reaches the character. At this point the new path can be reconstructed by following the updated parent pointers chain from the current character position to the goal.

Algorithm 3 The original D* algorithm.

1: function DStar(start, goal)
2: \( h(G) \leftarrow 0 \)
3: do
4: \( k_{\min} = \text{PROCESS-STATE}(\) )
5: while \( k_{\min} \neq -1 \) and current state still in open list
6: if \( k_{\min} = -1 \) then Destination Unreachable; Exit end if
7: while true do
8: while goal is not reached and no discrepancy found do
9: Execute Current Optimal Path
10: if goal is reached then
11: Goal Reached; Exit
12: else
13: \( Y \leftarrow \) The state in which we found the discrepancy.
14: MODIFY-Cost\( (Y, X, \text{newCost}) \)
15: do
16: \( k_{\min} = \text{PROCESS-STATE}(\) )
17: while \( k(Y) < f(Y) \) and \( k_{\min} \neq -1 \)
18: if \( k_{\min} = -1 \) then Destination Unreachable; Exit end if

It is clear, however, that D* must retain the value of \( f(X) \) and \( k(Y) \) for every node of the map. This forces the pathfinding system to allocate much more memory than A* during the execution of the algorithm, and in addition, to keep this memory allocated in between pathfinding queries. Moreover, if a map changes frequently, we will end up with several update and execute steps.

The LPA* algorithm

The Lifelong Planning A* (LPA*) is another dynamic incremental heuristic search algorithm for pathfinding when cost and edges in the pathfinding graph are constantly changing [51].

LPA* is algorithmically based on A*. In fact, during the first search, assuming an empty knowledge of the environment, LPA* expands exactly the same nodes than A*. All the subsequent search instances are potentially faster because LPA* is able to identify the parts in the pathfinding graph that remained identical and recompute only those parts that are changed and that are necessary for finding the new optimal path. As a consequence, LPA* is more effective when the changes are few and near the goal.

In order to identify the changed region of the map, LPA* uses the \( g \)-value, that is the same of A*, and a new rhs-value.\(^8\)

\(^8\)The name rhs comes from the values of right-hand sides grammar rule in DynamicSWSF-FP [76], the work who inspired LPA*. 
The \( \text{rhs} \) value can be considered a one-step look-ahead for \( g \) and it is defined as follows:

\[
\text{rhs}(n) = \begin{cases} 
0 & \text{if } n = \text{start} \\
\min_{n' \in \text{pred}(n)} (g(n') + c(n', n)) & \text{otherwise}
\end{cases}
\] (2.3)

where \( \text{pred}(n) \) is the set of predecessors of the node \( n \). Then, we define a node as \textit{consistent} if and only if \( \text{rhs}(n) = g(n) \) and \textit{inconsistent} otherwise.

If every node in a map is consistent, then we can just backtrack from any goal to the starting node by following the predecessors with the minimal cost \( g(n') + c(n', n) \). If a node is inconsistent we need to update its \( g \)-value. However, LPA* does not try to make all nodes consistent, instead, it uses a priority queue to focus the update only on those nodes that are useful for reaching the goal. The key of the priority queue is given by \( \min(g(n), \text{rhs}(n)) + h(n, \text{goal}) \) with a tie-break on the \( g \)-value.

A detailed pseudocode of LPA* can be found in Algorithm 5. Note that LPA* does not require a closed list. In fact, keeping track of the inconsistent nodes is enough to guarantee that there is no unnecessary vertex re-expansion.
Algorithm 5 Pseudocode of the LPA* algorithm.

1: function CalculateKey(n)
2:     return \[\min(g(n), rhs(n)) + h(n, goal), \min(g(n), rhs(n))\]
3:
4: function Initialize()
5:     \(U \leftarrow \emptyset\)
6:     for each \(n \in N\) do
7:         \(rhs(n) \leftarrow g(n) \leftarrow \infty\)
8:     \(rhs(start) \leftarrow 0\)
9:     Insert\((U, start, CalculateKey(start))\)
10:
11: function UpdateVertex(u)
12:     if \(u \neq start\) then
13:         \(rhs(u) \leftarrow \min_{u' \in \text{pred}(u)}(g(u') + c(u', u))\) end if
14:     if \(u \in U\) then Remove\((U, u)\) end if
15:     if \(g(u) \neq rhs(u)\) then Insert\((U, u, CalculateKey(u))\) end if
16:
17: function ComputeShortestPath(start, goal)
18:     while Peek\((U)\) < CalculateKey\((goal)\) or rhs\((goal)\) \(\neq g(goal)\) do
19:         \(u \leftarrow \text{Pop}(U)\)
20:         if \(g(u) > rhs(u)\) then \(g(u) \leftarrow rhs(u)\)
21:             for each \(u' \in \text{Neighbors}(u)\) do
22:                 UpdateVertex\((u')\)
23:             else \(g(u) \leftarrow \infty\)
24:                 for each \(u' \in \text{Neighbors}(u) \cup \{u\}\) do
25:                     UpdateVertex\((u')\)
26:         end if
27:
28: function Main()
29:     Initialize()
30:     while true do
31:         ComputeShortestPath\((start, goal)\)
32:         Wait for change in the edge costs.
33:         for each edge \((u, v)\) with changed cost do
34:             \(c(u, v) \leftarrow \text{the new cost}\)
35:         UpdateVertex\((v)\)
36:

The D*-Lite algorithm

The D*-Lite algorithm is a version of the original D* algorithm based on LPA* [50]. The algorithm produces the same paths of D* with the same performance, however it is much shorter and easier to implement.

D*-Lite is algorithmically identical to LPA* but with few important changes. The first one is that D*-Lite makes the free space assumption like D*. This allows LPA* to work even in an unknown environment. The second modification is that it evaluates the cost (encoded into the \(g\) and \(rhs\) function) from the goal to the current position instead of the opposite direction. In D*-Lite the \(rhs\) function is the following

\[
rhs(n) = \begin{cases} 
0 & \text{if } n = goal \\
\min_{n' \in succ(n)}(g(n') + c(n', n)) & \text{otherwise}
\end{cases}
\] (2.4)

where \(g\) this time is the cost of the optimal candidate path from the current state to the goal.

Considering these two modifications, the pseudocode of D*-Lite is almost exactly the same of LPA*. 
2.2.2 Incremental update of the $h$-value

The second class of algorithms consists of approaches that address incremental search by updating the heuristic function. We know from $A^*$ theory that the more our heuristic function is capable to guess the real cost of the optimal path from $n$ to the goal, fewer nodes will be expanded during the search. The basic idea of those algorithms is that we can use the information acquired during the previous search to improve the value returned by the heuristic function.

The Adaptive $A^*$

The Adaptive $A^*$ algorithm is the most representative algorithm of the $h$-value changing class [49]. There are different ways to update the heuristic function and the algorithm behavior slightly changes for every one of them. However, the most used and generic way to update the heuristic function is given by:

$$h(n) = g(\text{goal}) - g(n),$$

where the value for the $g$ function comes from the previous search of Adaptive $A^*$ algorithm.

However, we need to ensure that this heuristic remains consistent during the execution of the program. Therefore, we can identify the following three cases:

1. The starting position changes. In this case we do not do anything. In fact, we can easily verify that the equation defined in equation (2.5) is consistent respect to the goal position.

2. The goal position changes. In this case we need to update the heuristic function for every node with the following new value:

$$h(n) = \max(\hat{h}(n, \text{goal}'), h(n) - h(\text{goal}')),$$

where $\hat{h}$ is the user defined heuristic function (for instance the Euclidean distance) and $\text{goal}'$ is the new goal node.

3. There is a change in the edge cost, that is, the map changed somewhere. In this case there are two possibilities. If all the changing costs increase, then the algorithm does not need to do anything because the heuristic function is still consistent. On the other hand, if one of the costs decreases, consistency is no more guaranteed.

A detailed pseudocode of the algorithm is provided in Algorithm 6 and Algorithm 7. It is clear that the limitation of having only increasing costs is too limiting in many cases, especially for navigation in dynamic terrains. Fortunately, the problem is addressed in the next algorithm.

The Generalized Adaptive $A^*$

The Generalized Adaptive $A^*$ (GAA*) is an improvement of Adaptive $A^*$ that is capable to handle also cases in which the edge cost is decreasing [92]. GAA* adds a new consistency procedure at the end of each loop in the main function of Adaptive $A^*$ (Algorithm 7, line 33). This new consistency procedure is described in Algorithm 8.

1: function InitializeNode(n)
2:   if search(n) ≠ counter and search(n) ≠ 0 then
3:     if g(n) + h(n) < pathcost(search(n)) then
4:       h(n) ← pathcost(search(n)) − g(n)
5:       h(n) ← h(n) − (∆h(counter) − ∆h(search(n)))
6:       h(n) ← max(h(n), ĥ(n, goal))
7:       g(n) ← ∞
8:     else
9:       g(n) ← ĥ(n, goal)
10:    search(n) ← counter
11: function ComputePath(start, goal)
12:   while g(goal) > min_{n' ∈ open list}(g(n') + h(n')) do
13:     delete the node n with the smallest f-value in open list
14:     for each action a ∈ A(n) do
15:       InitializeNode(succ(n, a))
16:       if g(succ(n, a)) > g(n) + c(n, a) then
17:         g(succ(n, a)) ← g(n) + c(n, a)
18:       tree(succ(n, a)) ← n
19:       if succ(n, a) ∈ open list then
20:         REMOVE(open list, succ(n, a))
21:         INSERT(open list, succ(n, a)) with f-value g(succ(n, a)) + h(succ(n, a))

Algorithm 7 Pseudocode of Main function for the Adaptive A* algorithm.

1: function Main(start, goal)
2:   counter ← 1
3:   ∆h(1) ← 0
4:   for each n ∈ N do
5:     search(n) = 0
6:   while start ≠ goal do
7:     InitializeNode(start)
8:     InitializeNode(goal)
9:     g(start) = 0
10:    open list ← ∅
11:   INSERT(open list, start) with f-value g(start) + h(start)
12:   ComputePath(start, goal)
13:   if open list = ∅ then
14:     pathcost(counter) ← ∞
15:   else
16:     pathcost(counter) ← g(goal)
17:   change start, goal costs
18:   start ← theNewStart
19:   newGoal ← theNewGoal
20:   if goal ≠ newGoal then
21:     InitializeNode(newGoal)
22:     if g(newGoal) + h(newGoal) < pathcost(counter) then
23:       h(newGoal) ← pathcost(counter) − g(newGoal)
24:       ∆h(counter + 1) ← ∆h(counter) + h(newGoal)
25:       goal ← newGoal
26:   else
27:     ∆h(counter + 1) = ∆h(counter)
28:     counter ← counter + 1
29:   update the increased action costs (if any)
Algorithm 8 The Consistency Procedure for GAA*

1: function ConsistencyProcedure( )
2:   open list ← ∅
3:   for each node-action pair (n, a) do
4:     if n ≠ goal and c(n, a) is decreased then
5:       INITIALIZE_NODE(n)
6:       INITIALIZE_NODE(succ(n, a))
7:       if h(n) > c(n, a) + h(succ(n, a)) then
8:         h(n) ← c(n, a) + h(succ(n, a))
9:       INSERT(open list, n, h(n))
10:  while open list is not empty do
11:     delete n′ with the smallest h(n′) from open list
12:     for each node-action pair (n, a) such that succ(n, a) = n′ do
13:       INITIALIZE_NODE(n)
14:       if h(n) > c(n, a) + h(succ(n, a)) then
15:         h(n) ← c(n, a) + h(succ(n, a))
16:       INSERT(open list, n, h(n))
2.2 Incremental search

2.2.3 Sub-tree recomputation

In the last class, we can find algorithms that are able to recompute only the sub-tree that has been affected by the change. They usually include systems to identify the sub-trees affected by one change and, therefore, they can run a new A* instance exactly from the point in which the two search trees start to diverge.

The Fringe Saving A* algorithm

The Fringe Saving A* (FSA*) is a main representative of this third class of incremental heuristic search algorithms [91]. FSA* works by recovering the exact state of the open list at the moment in which the current search instance may deviate from the previous one. Thanks to the instantaneous performance for those parts of the search tree that are in common in between searches, FSA* can outperform LPA* in many cases.

The FSA* algorithms work in six steps.

1. Recovering of the closed list. In this step, we want to identify a set of positions that can be safely be assumed as visited in between two search instances S1 and S2. The idea is to attach an index to the expanded nodes representing the iteration in which a node n has been expanded. We call this number ExpandedID(n). When we start the second search we need to consider the changed tile n’. If the position n’ becomes blocked, then we can safely assume that the new search S2 will expand, in the same order, at least every node such that ExpandedID(n) < ExpandedID(n’). Similarly, if the position n’ becomes unblocked, we can do a similar assumption for all the nodes n such that ExpandedID(n) < 1 + min_{n'' \in succ(n')} ExpandedID(n’’). All those cells can be safely put again in the closed list. We call these nodes reusable nodes.

2. Early termination. If the goal node falls in the set described by the previously discovered steps, then we do not need to run a new search at all. We can therefore return the path.

3. Recovering of the open list. All the unblocked nodes on the border of one or more reusable node must be added directly back to the open list.

4. Recovering of g-values and parent pointers. The g-value and parent pointers of nodes in the reusable set are guaranteed to be correct. However, we cannot say the same for nodes on the fringe. There are two important cases. In the first one we have nodes in the open list that were blocked in the previous search and, therefore, cannot have an updated g or parent. In the second case, we have nodes that point to a node not in the reusable set. Also in this case, we need to update the g and parent values. The solution to both cases is to find any adjacent node ˆn that is in the reusable set. We set the value of the fringe node to g(ˆn) + 1 and the parent to ˆn.

5. Sorting the open list. We transform the new open list in a binary heap restoring the ordering. Note that, if the open list is implemented with a binary heap, transforming a set into a binary heap in a single step is more efficient than inserting the elements one by one in an empty binary heap [19].

6. Restarting A*. At this point, we can simply restart A* with the new closed and open lists. Because the state of A* is completely determined by those two lists, the effect is a fast forward search in the search state of the current A* instance.
A complete pseudocode for this algorithm is provided in Algorithm 9, 10, 11, 12, 13 and 14.

### Algorithm 9 Initialize for FSA*

1: function InitializeFSA()  
2: BlockId[0] ← 0  
3: for each node n do  
4: GeneratedIteration[n] ← 0  
5: ExpandedIteration[n] ← 0  
6: ExpandedId[n] ← 0  
7: m ← 0  
8: open list ← ∅  
9: g(start) ← 0  
10: Push(open list, start)  
11: GeneratedIteration[start] ← 1  
12: Iteration ← 1

### Algorithm 10 ComputeShortestPath for FSA*

1: function ComputeShortestPath()  
2: BlockId[Iteration] ← ∞  
3: while open list ≠ ∅ do  
4: n ← POP(open list)  
5: ExpandedIteration[n] ← Iteration  
6: ExpandedId[n] ← m  
7: m ← m + 1  
8: if n = goal then  
9: return True  
10: else  
11: for each n’ ∈ Succ(n) do  
12: if GeneratedIteration[n] ≠ Iteration or g(n) + 1 < g(n’) then  
13: g(n’) ← g(n) + 1  
14: GeneratedIteration[n’] ← Iteration  
15: Parent[n’] ← n  
16: Push(open list, n’)  
17: return False

### Algorithm 11 CellReusable for FSA*

1: function CellReusable(n)  
2: if ExpandedId[n] < BlockId[ExpandedIteration[n]] then  
3: return True  
4: return False

### 2.3 Agent-Centered Search

The incremental heuristic search algorithm we have seen in the previous section can be used to navigate unknown and dynamic terrains. However, each one of those algorithms return a full path from the current character position to the destination. If the map is not changing frequently, there are good reasons for this. However, if the map is completely unknown or it is highly dynamic we are, in practice, wasting time looking for a complete path that probably will be proven wrong in few steps.

**Agent-Centered Search** tries to address this problem with the basic idea of interleaving a search step (planning) with an execution step. Instead of looking
for the complete path from start to destination, it looks for a path up to a limited radius from the character, then executes this partial path, then searches again and so on, until it reaches the destination.

The basic idea of agent-centered search is not exclusive of pathfinding of a specific algorithm. For instance, we can find one most common applications of agent-centered search in the AI algorithms for games such as Chess or Reversi [48]. In those case the agent-centered nature is forced by the fact that the search space is so huge that it is impossible to perform a complete search in one shot. However, there are algorithms that have been developed with the agent-centered search in mind. The most famous of them is the Learning Real-Time A* algorithm.

The Learning Real-Time A* algorithm

Learning Real-Time A* (LRTA*) is an agent-centered search algorithm that stores in memory a value for each node encountered during the agent navigation and uses dynamic programming techniques to update those values [54].
Algorithm 14 Main function for FSA*

1: function MainFSA( )
2: InitializeFSA( )
3: while True do
4:     if not ComputeShortestPath( ) then
5:         exit
6:     while BlockId[Iteration] ≤ ExpandedId[goal] do
7:         Identify the path using the parent pointers.
8:         Wait for traversability changes.
9:         UpdateMazeTraversability( )
10:    RetrieveFringe( )

Figure 2.4. Example of the value-calculation step in LRTA*. In the first image, we can see the initialized values. In the second image, we assume as correct all the values just outside the local search space and we update the value in it by following backward the shortest path from the edges to the current character location. The image is taken from [48].

The LRTA* algorithm execution can be divided into four steps.

1. Initialization. At the beginning of the first search query, LRTA* initializes all nodes in the map with the value of the heuristic function $h(n)$.

2. Search step. In this step LRTA* performs a complete search of the local space. The local space is represented by all the non-blocked states inside the search radius of the agent. In practice, this step consists of measuring the real cost of reaching a state just outside the local space.

3. Value-calculation step. Starting from the nodes just outside the local space, the algorithm assigns new values to the nodes in the local space. For example, assume we start from a boundary node with the value of 5. Then, following the shortest path to the agent location computed in the previous step, we start assigning values 6, 7 and so on (Image 2.4). If we cross a node that was already updated we chose the minimum of the two values.

4. Action-selection step. The algorithm chooses the action that minimizes the cost for reaching the goal. This is done by following the annotated nodes in decreasing order.

5. Action-execution step. The agent executes the selected action (or actions). After one step (or after reaching the boundaries of the local space) we come
2.4 Symmetry-Breaking pathfinding

back to step 2 with a new searching phase in a local space centered around
the new agent’s position.

A character using LRTA* will reach the destination in a very sub-optimal way,
trying several dead-ends before finding the right path to the goal. However, after
multiple search instances with the same goal, the algorithm learns the right values
for the query and the resulting path will converge to the optimal path.

Other LRTA*-like algorithms

LRTA* has several problems, especially for interactive characters in games. The
slow rate of learning is the biggest one. A character using LRTA* will follow a good
exploratory path at first, but then will start to repeat the same mistakes (e.g., by
entering multiple times the same dead-end hallways) in a predictable way that can
damage the believability of the character. Moreover, the learning process is based
on a specific goal: when the goal changes –and this is a very common possibility in
video games and real-life pathfinding applications– most of the learned information
must be discarded.

During the last 20 years, several works have been done on LRTA* to overcome
these drawbacks. For instance, D-LRTA* [17, 18] and following extensions [57] try to
address the problem of the slow and limiting learning by inserting sub-goals around
the map, which act as anchor locations on the map using preprocessing or caching of
off-line executed paths. About the problem with changing the goals, min-max LRTA*
[52] focuses on moving goals and LRTA*(k) [38, 39] extends the heuristic propagation
mechanism of the original LRTA* allowing the algorithm to update the heuristic
estimates of up to $k$ states, not necessarily distinct, at the same time. This provides
faster convergence to the optimal search time.

2.4 Symmetry-Breaking pathfinding

More recently, a different approach emerged among pathfinding research. Instead
of improving search performance for general pathfinding and search problems, some
researchers started to provide very fast algorithms for specific pathfinding instances.
We can classify these algorithms as Symmetry-Breaking Pathfinding algorithms
because they exploit the structure of specific pathfinding domains in order to obtain
a huge speedup in the search times. These algorithms, in fact, usually take strong
assumptions on the structure of the map or the pathfinding graph.

Swamp

Swamp is a symmetry-breaking algorithm that can identify some regions of the
map that are searched by the other algorithms (called swamp regions) [72] but in
fact can be skipped. The algorithm is completely independent from the heuristic
function and the actual search algorithm used for the final pathfinding step. Once
the algorithm has identified all the swamp regions, they can be treated as blocked
in the following steps of the search without losing the optimal path in the process
(unless, of course, the goal or the starting point is in one of those regions in which
case they are considered normally).

\footnote{This behavior is explained by the fact that, because in the value-calculation step only values in
a small radius around the character are incremented. If the dead end is big enough, the algorithm
needs multiple iterations before the values in the dead end get big enough to allow the algorithm to
skip it entirely.}
Algorithm 15 The Swamp Detection Algorithm

```
1: function GrowSwamps(sizeLimit)
2:   swamp⁰ ← ∅
3:   seed ← DetectSeed()
4:   t ← 1
5:   for each s ∈ seeds do
6:     region ← ExtendSeed(s, sizeLimit)
7:     if region is not empty then
8:       swampsᵗ ← swampsᵗ⁻¹ ∪ {region}
9:       t ← t + 1
10: end for
11: function ExtendSeed(s, sizeLimit)
12:   radius ← 0
13:   size ← 0
14:   while radius < MAX and size < sizeLimit do
15:     cluster ← GetReachable(seed, radius)
16:     currentswamp ← TrimToSwamp(cluster, radius)
17:     if size(currentswamp) > size then size = size(currentswamp) end if
18:     radius ← radius + 1
19:   return largest swamp found that had size less than sizeLimit
```

Algorithm 16 The Trimming to Swamp Algorithm

```
1: function TrimToSwamp(s, group)
2:   B ← boundaries of group
3:   for each (v₁, v₂) ∈ B do
4:     P₁ ← FindPath(v₁, v₂, swampᵗ⁻¹)
5:     P₂ ← FindPath(v₁, v₂, swampᵗ)
6:     if length(P₂) > length(P₁) then
7:       for each vᵢ₂ ∈ P₂ do
8:         if vᵢ₂ ∈ group then
9:           remove vᵢ₂ from group
10:          add vᵢ₂ from B(group)
```

Formally, a swamp is a set of positions such that, for any shortest path \( P \) that pass through it, or \( P \) starts from the swamp or end in it or there is another path \( P' \) that has the same cost of \( P \) but does not pass in it. In order to find a swamp region, we need to start from a seed node. A location \( s \) on the map is a seed if satisfies all the following properties.

1. \( s \) is unblocked.
2. At least a node above or below \( s \) is blocked.\(^{10}\)
3. At least a node to the left or to the right of \( s \) is blocked.

After we have identified the seeds of a map, we can start to grow them into swamps following Algorithms 15 and 16. In short, the algorithm starts with a swamp⁰ containing only the seed. Then, it starts expanding it by adding all the nodes reached by the seed in \( k \) steps (forming the candidate swamp set swamp⁰). Finally, the algorithm trims from the candidate swamp all the nodes that do not satisfy the above swamp properties. If the swamp region reaches the maximum size or remains the same as in the previous iteration (swamp⁰ = swamp⁰⁺¹), the algorithm stops and proceeds to the next seed.

\(^{10}\)We consider the map borders to be always blocked.
2.5 Hierarchical and abstraction-based pathfinding

The Rectangular Symmetry Reduction algorithm

The Rectangular Symmetry Reduction (RSR) algorithm exploits one of the most common structures in grid map: rectangular regions [34]. Rectangular empty regions are common in games, where they are used as rooms, but can they can be identified even in more complex and open maps by what is called rectangular decomposition. The strength of the RSR algorithm relies on the ability to skip entirely the search in these rectangular regions. In fact, the algorithm can assume that the borders of rectangular regions act like a portal that connects to the opposite side of the rectangle.

The algorithm works according to the following steps.

1. We decompose the grid map into a set of obstacle-free rectangles \( \mathcal{R} = \{ R_i \} \). For each rectangle \( R_i \) we remove all the internal nodes with the exclusion of the nodes on the boundaries.

2. We connect the nodes on the boundaries of \( R_i \) with macro-edges. A macro-edge connects a node on the edge of the rectangle with the node on the opposite side.\(^{11}\) The cost of these macro-edges can be easily assigned by the sizes of the rectangle.

3. When we ask for a path from \( x \) to \( y \) and these nodes are in a rectangular region, we temporarily insert the nodes back in the map.

4. We perform the pathfinding query with any algorithm we want (e.g., A*).

Depending on the map, RSR can speed up the computation by an order of magnitude with respect to regular A*. However, the algorithm is based on the strong assumption that the map is abstracted with a grid and, moreover, that the grid has uniform and not dynamic costs.

The Jump-Point Search algorithm

Jump-Point-Search (JPS) is a more recent technique introduced by Harabor and Grastien [31], that has proven extremely successful to navigate uniform cost grid-based maps. In some sense, it is a direct extension of the Rectangular Symmetry Reduction algorithm described above.

JPS operates by identifying every node that might end up being on the optimal path, i.e., the so-called jump-points, and discards the rest. Thus, instead of expanding all reachable nodes, JPS jumps from a potential turning (jump) point to another turning (jump) point, expanding only those nodes that might require a change of direction.

As JPS is fundamental for a big part of this work, we will talk more in detail about it in Section 5.1 when we will use JPS to allow practical performance in Inventory-Aware Pathfinding.

11This works for 4-connected grid maps. If we allow diagonal movement we need to include a more complex set of macro-edges. However, for simplicity, we only consider the basic set of macro-edges.
returning the optimal result. However, both those algorithms perform search at the lowest level possible: node by node.

In real life, we do not plan our movement in such level of details. For instance, if we want to reach the particular shop in our town, we do not plan the path step by step, instead, we use a more high-level plan such as “go to the front door, then reach the first crossroad, then go left” and so on.

This is the basic idea behind the hierarchical and abstraction-based pathfinding algorithms presented in this section. Computing a high-level path is faster because the node in the high-level pathfinding graph can abstract hundreds of nodes. Moreover, the resulting high-level paths are more robust to small changes in the map. For instance, an high-level path such as “go to room B, then to room C” remains valid even if there is a small obstacle in the middle of the room B while a detailed low-level path may be invalidated by the same obstacle if it is placed on the path.

Finally, hierarchical pathfinding can be easily “distributed” over multiple frames in the game engine. At the beginning of the movement, the pathfinding system can compute only the high-level path and then, compute the low-level path while the agent moves from a high-level region to the other.

However, in general, hierarchical pathfinding is not optimal, even though the sub-optimality is usually negligible except for specific pathological cases.

Hierarchical A*

The Hierarchical Pathfinding A* (HPA*) is probably the most used hierarchical approach for pathfinding in video games [13]. HPA* follows all the general aspects of hierarchical pathfinding but it differs in the map-independent way it performs the map abstraction.

Map abstraction is provided by covering the whole map with a set of disjoint regular rectangular areas called clusters. For instance, given a map of 40x40 tiles, we can cover it with 16 squares of size 10x10. Then, we scan the borders between clusters and we identify a possible set of entrances. An entrance is a set of locations on the edge between a node on the border of a cluster $c_i$ such that there is an adjacent location on a different cluster $c_j$. The set of entrances will be the set of vertexes in the high-level graph.

To find the edges of this high-level graph we proceed in two steps. First, we add an edge between two adjacent entrances connecting two different clusters $c_i$ and $c_j$. We call all these kind of edges inter-edges and they all have cost 1 because they connect two adjacent locations. Then, we connect each pair of entrances inside the same cluster $c_i$. We call these intra-edges. The cost of these edges is more complex and requires the computation of the optimal path between the entrances. However, this is a completely offline procedure and all the heavy computation can be done in advance, stored in a file (and, in the videogame setting, shipped with the game). Finally, once we have the first high-level graph, we can reiterate the algorithm building more high-level structures. The complete preprocessing algorithm pseudocode is shown in Algorithm 17.

When a character asks for a path from start to goal, the algorithm first inserts the start and goal nodes to the high-level graph as nodes and then connects them to the already existing nodes in the cluster of start (and goal). Finally, the algorithm searches for a path in the highest level and then expands this path into the lower levels until we reach the lowest, physical, level. The expansion step can be done all at once or as the character reaches the entrances of the various clusters.
Algorithm 17 The preprocessing algorithm for HPA*

1: function AbstractMaze( )
2: \( E \leftarrow \emptyset \)
3: \( C[1] \leftarrow \text{BuildCluster}(1) \)
4: for each \((c_1, c_2) \in C\) do
5: \( \text{if adjacent}(c_1, c_2) \text{ then } E \leftarrow E \cup \text{BuildEntrances}(c_1, c_2) \) end if
6: function BuildGraph( )
7: for each \( e \in E \) do
8: \( c_1 \leftarrow \text{getCluster1}(e, 1) \)
9: \( c_2 \leftarrow \text{getCluster2}(e, 1) \)
10: \( n_1 \leftarrow \text{newNode}(e, c_1) \)
11: \( n_2 \leftarrow \text{newNode}(e, c_2) \)
12: \( \text{addNode}(n_1, 1) \)
13: \( \text{addNode}(n_2, 1) \)
14: \( \text{addEdge}(n_1, n_2, 1, 1, \text{INTER}) \)
15: for each \( e \in C[1] \) do
16: for each \( n_1, n_2 \in N[e] \) where \( n_1 \neq n_2 \) do
17: \( d \leftarrow \text{searchForDistance}(n_1, n_2, c) \)
18: \( \text{if } d < \infty \text{ then } \text{addEdge}(n_1, n_2, 1, d, \text{INTRA}) \) end if
19: function AddLevelToGraph(int \( l \))
20: \( C[l] \leftarrow \text{BuildCluster}(l) \)
21: for each \( c_1, c_2 \in C[l] \) do
22: \( \text{if adjacent}(c_1, c_2) \text{ then continue } \) end if
23: for each \( e \in \text{getEntrances}(c_1, c_2) \) do
24: \( \text{setLevel}((\text{getNode1}(e), l)) \)
25: \( \text{setLevel}((\text{getNode2}(e), l)) \)
26: \( \text{setLevel}((\text{getEdge}(e), l)) \)
27: for each \( c \in C[l] \) do
28: for each \( n_1, n_2 \in N[e] \) where \( n_1 \neq n_2 \) do
29: \( d \leftarrow \text{searchForDistance}(n_1, n_2, c) \)
30: \( \text{if } d < \infty \text{ then } \text{addEdge}(n_1, n_2, 1, d, \text{INTRA}) \) end if
31: function Preprocessing(int maxLevels)
32: \( \text{AbstractMaze}() \)
33: \( \text{BuildGraph}() \)
34: for \( l \leftarrow 2; l \leq \text{maxLevel}; l++ \) do
35: \( \text{AddLevelToGraph}(l) \)

The Gateway Heuristic

HPA* uses abstraction by searching on the higher level and then expanding the high-level path on the physical level. However, this is not the only approach possible. The Gateway Heuristic approach, instead, uses the abstraction level to build a more informed heuristic that can be used directly on the low-level. This allows to have almost the same performance gain of HPA* but preserving optimality [11].

The Gateway Heuristic algorithm also starts with a preprocessing step. In this step, the algorithm decomposes the map into regions. Then, similarly to HPA*, the algorithm scans the boundaries of these regions and identifies a set of gates. A gate is a set of adjacent edges that connect a region to another region. Two edges \((x, y)\) and \((p, q)\) are adjacent if each pair of the nodes \(x, y, p\) and \(q\) is adjacent in

\[\text{In the original paper, the authors use a flood-filling algorithm to divide the maps into \"rectangular\" regions. This algorithm is very limiting and inefficient for many maps, e.g., when the map is composed by rooms not aligned to the map axis. However, the Gateway Heuristic approach works for any kind of decomposition algorithm, so we will assume that the map is decomposed with some optimal algorithm.}\]
the traditional sense.\textsuperscript{13}

Finally, the preprocessing algorithm computes and stores the distance between each pair of gates. This information will be used to compute the following heuristic function:

\[ h^G(n, g) = \min \sum_i \sum_j h^l(n, G_i) + H(G_i, G_j) + h^l(G_j, g) \]  \hspace{1cm} (2.7)

The heuristic \( h^l(n, G_i) \) computes the octile distance between \( n \) and the nearest point in the gate \( G_i \). The value \( H(G_i, G_j) \) instead is exactly the distance between gates we precomputed in the preprocessing step.

This heuristic transfers part of the complexity from the search to the heuristic function computation. However, the final results are comparable with the HPA\(^*\) algorithm.

\textbf{Partial Refinement A\(*\)}

\textbf{Partial Refinement A\(*\)} (PRA\(*\)) is a hierarchical pathfinding algorithm that is very similar to HPA\(*\). However, instead of overlapping a fixed structure over the map (the clusters), PRA\(*\) obtains the abstraction on the basis of the local features of the map such as cliques and orphans [90].

The abstraction algorithm works as follows. Starting from the lowest level, the algorithm identifies all the cliques, that are all the sets of 1, 2, 3 or 4 (at max) fully connected nodes. All the cliques found in this way will be a single node in the higher level of abstraction. Then, we merge the orphans (nodes with a single connection) to the unique connected clique. We can then repeat the procedure on the higher level until we end with a single node producing in this way a hierarchy of abstractions.

To find a path using this abstraction hierarchy we can introduce a simple algorithm called \textit{QuickPath} that works as follows:

1. Suppose we want to find a path between \( x_i \) and \( x_j \). The first step consists in exploring the abstraction hierarchy until we reach a common parent. For instance, the high-level node \( X^k_i \) in the \( k \)-th abstraction level. This node represents the path from \( x_i \) and \( x_j \) in the \( k \)-th level.

2. We start refining the path by expanding \( X^k_i \) into the \( (k - 1) \)-th level of abstraction.

3. Suppose that \( X^k_i \) is the common representation of \( X^{k-1}_i \) and \( X^{k-1}_j \) in the \( k \)-th level. Now, we need to find out which node of the \( (k - 2) \)-th level in \( X^{k-1}_i \) is connected with a node in \( X^{k-1}_j \). The existence of this node is guaranteed by the fact that, if two nodes in the \( k - 1 \) level have a common parent in the \( k \) level, then the two nodes are connected and there is a path in any lower level of abstraction.

4. At the end of this step, we find a node \( X^{k-2}_1 \) in \( X^{k-1}_i \) that is directly connected to a node \( X^{k-2}_2 \) in \( X^{k-1}_j \). To complete the refinement, we need a path in \( X^{k-1}_i \) from the \( k - 2 \) abstraction of \( x_i \) to \( X^{k-2}_1 \) and from \( X^{k-2}_2 \) to the \( k - 2 \) abstraction of \( x_j \). However, because the abstraction is built around cliques we know that there is a direct edge connecting these two nodes.

\textsuperscript{13}Note that gates are quite different from the HPA\(*\) entrances. An entrance is a set of nodes on the boundary of a region. A gate is the actual set of edges that connects two regions.
5. At this point, either the path is the final path \((k = 0)\), or we can iterate the algorithm again on every high-level step obtaining a path in the \((k − 3)\)-th level. And so on.

\text{PRA}\ast\text{ is directly based on QuickPath. however, it introduces three main enchantments. First, while QuickPath refines the path blindly, PRA}\ast\text{ uses }A\ast\text{ to search for the optimal path on an abstraction level. Second, to minimize the cost of the search operation, we allow }A\ast\text{ to expand only the nodes that are abstracted by the path we are refining. In other words, each abstracted path defines a “corridor” of valid nodes in the lower abstraction level and we are allowed to search only in there. Finally, instead of refining the path all at once, we refine only the }n\text{ steps on each abstraction level. We then refine the remaining nodes on demand. The pseudocode for the PRA}\ast\text{ search algorithm is described in Algorithm 18.}

\begin{algorithm}
\caption{The search algorithm for \text{PRA}\ast\text{}}
\begin{algorithmic}[1]
\Function{PRA}{abstractGraph, start, goal, n}
\State GetAbstractionHierarchy\((start, goal)\)
\State \text{s} \leftarrow \text{GetStartLevel\((start, goal)\)}
\State \text{path} \leftarrow \emptyset
\For{\text{each level} \in \text{[s..1]}}
\State \text{path} \leftarrow \text{RefinePath\((path, start[\text{level}], tail(path))\)}
\EndFor
\State truncate \text{path} to length \(n\)
\State \Return \text{path}
\EndFunction
\Function{RefinePath}{path, start, goal}
\State \Return AStar\((start, goal)\) subject to nodes in \text{path}.
\EndFunction
\end{algorithmic}
\end{algorithm}

\section{2.6 Multi-Agent Pathfinding}

With Multi-Agent Pathfinding we refer to the problem of finding the optimal path from \(start_i\) to \(goal_i\) for every agent \(a_i \in A\) 
but taking into account the position of every agent during the execution of their path. To understand why this problem is different from, and much more complex than, the single-agent pathfinding problem applied on multiple, let us consider the following scenario. Consider a map composed by two rooms connected by a long one-tile wide corridor and 10 characters that have to move from the first room to the second one. If we apply the single-agent pathfinding techniques to each character independently from the others we obtain the same path passing from the corridor. However, when we execute this path the characters will start to clash together creating a blockage or a deadlock at the start of the corridor. A multi-agent pathfinding algorithm, instead, can handle this problem by assigning to each character the order with which each of them have to pass through the corridor.

Unfortunately, finding an optimal solution to the Multi-Agent Pathfinding has been proven to be a PSPACE-hard problem \cite{43} with respect to the number of agents. This means that the problem is impractical except perahps for a very small number of characters. Videogames typically handle this problem by applying single-agent pathfinding to each character and relying to local collision avoidance algorithms when the characters get close together. This solution works well in many cases, especially because game designers can craft their levels in such a way that removes the possibility of local deadlocks. However, this approach occasionally fails and there are cases in which this is not enough for the problem. Real-Time Strategy
games, tactical and strategic behavior of character, complex level designs; all these situations require smarter AI that is able to solve even a simple form of multi-agent pathfinding.

There are mainly two different approaches for multi-agent pathfinding. The first one relies on a centralized system that can assign the optimal path to each agent [6, 56]. The second one, tries to decompose the problem into several sub-goals that can be solved more easily [23], which however may lose completeness and optimality in the process. Neither of these approaches are good enough for videogame applications. Instead, during the last decades, there have been developed several suboptimal techniques based on A* and/or map abstraction. In the next paragraphs, we will explore the main ones.

Local Repair A*

With Local Repair A* (LRA*) we refer to a class of algorithms in which multi-agent constraints are considered only during the execution step [82]. In LRA*, every agent computes their path independently from every other agent. Then, during the execution, when the collision is imminent, a new path is computed from scratch.

This approach is quite inefficient, and cycles and deadlocks are indeed frequent. To relieve this problem, many algorithms introduce an agitation parameter. This introduces random noise into the pathfinding algorithm and produces a noisier path (facilitating the disentanglement of cycles and deadlocks).

Cooperative A*

A more refined solution is given by the Cooperative A* algorithm (CA*) [82, 83]. The problem with LRA* (and A* in general) is that it searches in a snapshot of the map state, in other words, it searches assuming that the map is frozen in time. However, it is obvious that multi-agent pathfinding requires to reason about the future position of every other agent.

Cooperative A* solves this problem by extending the state representation into a third time dimension (this is also called space-time A*). According this extension, all the character actions move the agent in space and time. Moreover, CA* introduces also a new action wait that is used to keep the character still but moving it in time. For instance, if we represent the state of the character with \((x, y, t)\) the action Right will bring the agent from \((x, y, t)\) to \((x + 1, y, t + 1)\) while the action Wait will bring the character from \((x, y, t)\) to \((x, y, t + 1)\).

Using this search space representation, called reservation table, the algorithm can solve the problem with a series of greedy sequential operations:

1. The agent \(a_i\) computes a path from \(start_i\) to \(goal_i\) producing a space-time path \(p_i\).
2. The algorithm marks all the nodes in \(p_i\) as occupied and starts computing the path for the agent \(a_{i+1}\).
3. Continue like this until all the agents have a suitable path.

It is also interesting to look at the heuristic function used by CA*. Even if it is possible to use the standard function for traditional A*, a more efficient solution is to use the True Distance Heuristic. This heuristic represents the cost of the optimal path assuming that \(a_i\) is the only agent on the map.
2.6 Multi-Agent Pathfinding

Figure 2.5. A pathological case showing that Cooperative A* is not complete. In this figure, a character must navigate from $S_1$ to $G_1$ whilst the other must navigate from $S_2$ to $G_2$. Even if a solution to this problem exists (e.g., Character 1 waiting in the small hole until Character 2 completes), this solution cannot be found by regular Cooperative A*.

Note also that a greedy algorithm like Cooperative A* in not complete. There are problem instances that cannot be solved by CA* as shown in Figure 2.5. However, these problems are less common in games and practical applications.

**Windowed Hierarchical Cooperative A***

Cooperative Pathfinding can be improved by relying on abstraction and search windows. This is the main objectives of the **Windowed Hierarchical Cooperative A** (WHCA*) [82].

Hierarchical abstraction is used to provide a more sophisticated and fast heuristic to CA*. The abstraction in WHCA* is composed only by the spatial level, therefore, all the other agents are ignored during the high-level search. Searching for an optimal path on the high-level, then, is just a more efficient way to compute the True Distance Heuristic we presented in the previous section.

Windowing, instead, solves some of the major problems in CA*. There are, in fact, three major issues in Cooperative A*. First, it is sequential with respect to the agents. Imagine that the first agent’s goal is in the middle of a narrow corridor. This will block that path to all the subsequent agents making the problem unsolvable. The second issue is that CA* is dependent on the agents’ execution order. A pathfinding query may be solved with a particular order of agents, but unsolvable for all the other sequences. Third, all the paths are computed up to the destination and may require complex computations to avoid blockage that will practically never occurs.

These problems can be solved by interleaving character search and execution with a limited search depth (windows). Each character, in turn, will search for $k$
steps ahead and then pass to the next character. This has also the additional benefit of spreading the computation on all the character in a more even way. To ensure that the agents are headed to the right direction, the high-level search on the abstraction is not limited.

**Cooperative PRA***

**Cooperative PRA*** (CPRA*) is a direct improvement of WHCA* with the techniques used in PRA* [89]. The algorithms integration is straightforward. CPRA* uses PRA* for all the hierarchical abstraction levels, then, on the lowest level, it applies WHCA*. The windowing process is already embedded in PRA* thanks to the partial progressive refinement of the high-level paths. Moreover, we obtain a big improvement in the WHCA* application thanks to the fact that search is limited into the *corridors* projected on the lowest-level by the PRA* abstraction.
Part I

Inventory-Aware Pathfinding
Chapter 3

The Inventory-Aware Pathfinding Problem (IAP)

As we discussed before, the goal of pathfinding is to find a path that takes a character from a starting node to a destination node avoiding obstacles on the map. But, what if the character can also interact with objects along the way (and not just avoid them)? [46] This is a much more complex scenario that pushes the problem farther from regular pathfinding and is in fact closer to the AI automated planning domain as for instance classical STRIPS planning or Goal-Oriented Action Planning [25, 67].

In the general case, there is no standard solution for this problem in the videogame setting. A direct approach is to encode both task-planning and path-planning (pathfinding) into a combined planning problem and use a general-domain planning system to solve it. Unfortunately, these planning systems need expert knowledge to implement them inside videogames which can be a barrier for game developers, and in fact, have impractical performance even for pathfinding problems that can be easily solved by A*-based algorithms (Section 5.3.2).

The issue is that A* pathfinding approaches make use of heuristics that are very effective (e.g., variations of Manhattan distance) while general-domain planning systems use domain-independent heuristics that cannot be as effective in the specific domain of pathfinding. Note that this is not a decision of style or implementation. Manhattan distance is an admissible (and very helpful) heuristic for pathfinding problems, but it is not helpful for the more general setting we described; perhaps specialized heuristics can be investigated for planning systems that consider a navigational component in the planning problem, but these are not developed yet in the literature.

Instead, in this work, we try to identify sub-classes of the planning problem that can be solved using a pathfinding algorithm by introducing a minimal amount of extensions which, at the same time, is complex enough to be useful in many game situations. In other words, we want to solve restricted planning along with pathfinding instead of solving pathfinding as a special case of a bigger planning problem. As the restricted capabilities that we consider are similar to carrying objects in the character’s inventory, we will use the term inventory-aware pathfinding (IAP) to refer to this approach.
3. The Inventory-Aware Pathfinding Problem (IAP)

Figure 3.1. An example environment for a IAP problem. The red character must reach the king in the room. However, in order to kill the dragon (soft obstacle) it must first get the sword (item) in the other room.

3.1 Motivation and examples

An Inventory-Aware Pathfinding (IAP) problem instance is a hybrid composition of two different problems. The first one is the item constraints problem: in order to reach the goal we need to have a certain number of items to overcome certain obstacles, but to reach those items we may require other items as well, and so on, until we reach a state in which we can solve all these constraints or we have a proof that it is not possible to do so.\(^1\) The second one is pathfinding: we need to perform pathfinding to know if a path exists from the starting point to an item and, therefore, which items are required to reach the goal. Moreover, instead of looking for the optimal solution for each problem, we want to find overall optimal path/plan, typically in terms of shorter distance.

Therefore, the two problems are entangled and it is not possible to solve one without considering the other one. At the same time, the naive solution of interleaving between the two problems is highly inefficient.

To illustrate this, consider a simple example situation in which we have two items and two obstacles and \(S\) and \(G\) as starting and goal locations. A constraints solution to the IAP problem is a path \(P = S \ldots \{I_i, O_i\}^* \ldots G\) in which \(I_i\) is the location of the \(i\)-th item and \(O_i\) is the location of the object unlocked by the \(i\)-th item. The IAP problem requires two additional constraints that come from pathfinding. First, we want that in the solution every \(I_i\) comes before \(O_i\), which means that the solution is valid if we reach the item \(I_i\) before we traverse the obstacle unlocked by \(I_i\). The second constraint is that the sum of the distances along the path must be minimal. In the 2 items/2 obstacles example above there are 9 possible paths:

1. A path in which we use no item (\(SG\)).
2. A path in which we use only the first item (\(SI_1O_1G\)).

\(^1\)In isolation, this problem can be seen as a plain constraint satisfaction problem (CSP) such as the eight queens puzzle or Sudoku.
3.1 Motivation and examples

Figure 3.2. An abstract representation of all the possible path to the destination in a scenario with two items (the keys) and two obstacles (the doors).

3. A path in which we use only the second item ($SI_2O_2G$).

4. 6 paths that use both objects:
   (a) $SI_1O_1I_2O_2G$
   (b) $SI_2O_2I_1O_1G$
   (c) $SI_1I_2O_1O_2G$
   (d) $SI_2I_1O_1O_2G$
   (e) $SI_1I_2O_2O_1G$
   (f) $SI_2I_1O_2O_1G$

Then, for each of these paths, we have to measure the total path length by running a series of pathfinding queries. Assuming that we have some kind of caching system to avoid duplicate searches, we need to compute 11 paths ($D(S,I_1)$, $D(S,I_2)$, $D(I_1,I_2)$, and so on for each consecutive pair in the candidate solutions). The number of those required pathfinding instances increases according the following equation:

$$2n^2 + n + 1$$

where $n$ is the number of items on the map.\(^2\) At the same time the number of candidate paths increases exponentially. In particular, if we call $PN$ the number of possible paths in a IAP instance with $n$ items, we have the following equation:

$$PN > 2^n \times n!$$

In other words, we need to test a quadratic number of pathfinding instances over a factorial number of possible paths. Fortunately, we can greatly improve the performance of the algorithm by solving the pathfinding and the item constraint problem at the same time. In fact, this allows us to focus our search on the “most promising paths”. The main idea is to convert the IAP problem into a $2^n$ different pathfinding problems by searching into the $2^n$ maps corresponding to the different possible inventories: from the maps with no items in the inventory up to the maps with full inventory. In the following chapters, we will explore this solution and the way to optimize it by taking advance of the similarities between these map variants.

Finally, one may wonder why we need a pathfinding system that is able to solve puzzles with items and locked passages. After all, these kinds of problems are

\(^2\)This formula comes from enumerating the two sets of paths from start to every item and from every obstacle to the goal ($2n$), all path between items plus all paths between obstacles ($n(n-1)$), all the paths between items and obstacles ($n^2$), plus the only path from start to goal.
designed for the players and there is no reason for an agent to solve them in place of the player. However, even if we often use examples in which the AI character plays the role of the main player, this is not the ideal application for this technique. There are more interesting settings for IAP, indeed. Two of them are described in the next two paragraphs.

“Reach the player”.

In the turn-based tactical shooter scenario S1 described in Section 1, the player faces enemies in a complex environment. The player can perform several actions in order to slow down the strategic attack of the enemies, for instance, the player can decide to set a gateway on fire. In this situation, the enemy characters should be able to overcome this obstacle not only by avoiding the gateway but also by resolving the cause of the problem.

This can be done at the behavior or decision-making level of the character, e.g., using a set of reactive rules encoded in the character’s behavior tree (BT) [45]. A possible way to encode this is as follows.

- The character’s task is to reach the player.
- When this fails due to the gateway being blocked by the fire, this activates the set of behaviors to overcome the fire, for instance, “go to the nearest fire extinguisher” or “go to the nearest bucket of water”.
- After successfully completing these behaviors, the character can recompute the path to the player and pass through the gateway.

Note that this requires a considerable increase in the complexity of the character decision making specification for handling a number of similar or all possible cases in a manual almost hard-coded way. Apart from the developing effort, the resulting path is not necessarily optimal and may also appear strange to the player. Consider for example that there is an alternative, but longer, path for reaching the player without passing the burning gateway, while reaching the fire extinguisher and then reaching the player is, in fact, immediate using an item in the next room. Getting and using the fire extinguisher is then the optimal (and desired) solution but getting the behavior tree to support this involves even more manual work, e.g. by specifying when each of the reactive rules should be preferred compared to pathfinding solutions. As the complexity of the environment increases, this becomes hard to implement and maintain.

As a consequence, handling these cases using regular pathfinding and the higher-level decision-making specification of the character is problematic in two ways. First, it requires a lot of manual work concerning encoding possible cases that the character will use an item, and second, it does not allow a systematic way to evaluate the alternatives to pick the fastest (or most interesting) one. Instead, if we use Inventory-Aware Pathfinding we just need to specify which nodes of the map contain items that allow characters to traverse burning or other types of affected gateways, and all navigational behavior is handled automatically by a single, but more sophisticated, pathfinding procedure.

Behavior trees are a standard way of “reactive planning” that is used in game development to specify and execute high-level reactive decision making for characters.
“Find a medkit”.

In the game scenario S2 described in Section 1, the AI companion can be asked to perform complex behaviors such as “find a medkit” or “get some ammo”. These items are placed on the map also as a means to provide challenges to human players. It is possible that a closed door blocks the access to them and that this requires a bit of planning to open. Similar arguments as for the case in scenario S1 also apply in this case. Being able to solve this problem is necessary for the AI companion to obtain the requested objects or for notifying the player of the requirements before starting to execute the task, for instance, saying that the closest medkit is behind a door that can be opened only by a red key. Inventory-Aware Pathfinding allows to remove the complexity of navigational reasoning from the decision-making procedure and push it directly where it belongs, to the pathfinding service.

**Figure 3.3.** Elizabeth in *BioShock: Infinite* (Irrational Games, 2013) is an AI companion which, among the other tasks, will randomly provide ammo, medkits and money for the player. This kind of character acts in a small radius around the player but we can use IAP to provide them with more mobility or the capability to provide dynamic hints and suggestions.

“Procedural Content Generation (PCG) validation”

In addition to online path planning, the IAP problem can be also useful for offline procedural map generation. Suppose we want to generate a dungeon with keys, doors and other locked rooms opened by satisfying certain conditions, e.g., a dungeon like the ones in *The Legend of Zelda* (Nintendo, 1986). If we put keys and doors randomly, we will likely have a final dungeon that is unsolvable for the player. Instead we can use IAP in design phase for doing the following things:

1. Check whether a generated dungeon is solvable. That is, we know that it is possible to find a valid path from start to destination eventually using items to unlock some paths. A common problem with random generation for item-locked path is that some items are locked behind the same obstacle that are supposed to unlock.4 IAP allows to identify this anomaly very quickly.

---

4In general, we can have a dependency cycle among $n$ items (e.g., item 2 is required to reach item 1 and item 3 is required to reach item 2, but item 1 is required to reach item 3).
2. Check all the possible solutions to the dungeon (for instance, in order to avoid that there is a trivial solution that allows the player to skip the intended one).

3. Check the shortest solution in the generated dungeon.

4. Analyze the complexity of the generated dungeon (e.g., number of backtracking required in terms of detours to find items that unlock obstacles, number of alternative solutions, and so on).

Moreover, in addition to the verification tasks, we can extend the techniques presented in this part to reverse the IAP problem and use it directly in the generation of the above dungeon, giving the control to the developer to specify some of the above parameters (i.e., minimal path, number of detours needed, number of objects required and so on).

3.2 Related work

At first glance, the problem is an instance of a Constraint Satisfaction problem (CSP) or an Automated Planning problem. However, both approaches cannot solve efficiently the IAP problem as will be shown in the next chapter and it is reported in my Gamasutra article [4].

CSP can solve the item/doors constraints in the high-level problem—that is, which is the smallest amount of items needed to reach the destination—however, without performing multiple pathfinding instances, it is impossible to know if such set of items corresponds to an optimal path in the real map.\(^5\) At the same time, automated planning techniques such as STRIPS [25], are not able to solve IAP problems efficiently because of the lack of specific heuristics and optimizations for handling the pathfinding component.

Therefore, the proposed IAP solutions are based on fast pathfinding optimizations (in particular we exploit symmetry breaking techniques such as the one described in Section 2.4) and multi-agent pathfinding (Section 2.6). In fact, we can imagine the item-constraints as made by invisible dummy agents that move away from the door/blockage under certain conditions and allow the character to pass. In particular, the 3D space-time map extension of Cooperative Pathfinding is very similar to the multi-layered map that we will use for IAP in the following sections and many of the properties and optimizations for it can be adapted to IAP.

3.3 The scope of the IAP problem

This type of navigation problems we described can be seen as a particular subclass of automated planning. It can also be seen as a pathfinding query in which some areas of the map will be unblocked (traversable) only by visiting a specific node first in order to obtain a corresponding item or capability. The latter approach has been adopted in this work where we explore a hybrid pathfinding/task-planning solution for this type of problems and it is reported in conference paper [5].

In short, given a bidimensional map\(^6\) \(M = \{m_{i,j}\}\) in which each position \(m_{i,j}\) can be free or blocked, consider \(M_K\) a subset of \(M\) such that the positions in \(M_K\)

---

\(^5\)E.g., consider a map with a single door that offers a shortcut between start and destination. Even if the item opening this door is not required to reach the goal, it is part of the optimal solution.

\(^6\)We consider a grid map for simplicity but the concept can be easily extend to generic map representations.
are free if and only if the character has visited in the previous step some special nodes $K$ according to a relation $R \subset M_K \times K$.\footnote{Note that this is a general relation in the sense that a position in $K$ can unlock more than one positions in $M_K$ and one position in $M_K$ can be unlocked by more than one position in $K$.} For instance, $M_K$ can be composed by doors that can be opened by acquiring a key or pressing a button in a specific location but we can imagine also more creative scenarios, for instance, where $M_K$ contains tiles “on fire” and the character needs to collect a fire extinguisher to pass through.

Here, we are interested in path planning in the context of agents who carry objects—an “inventory”—that can influence the navigation process. In such settings, reaching the destination may depend on whether the agent has acquired a certain object, such as a key to open a door, or a swimming suit or boat to pass across water, as is typical in many videogame worlds. We will use “items” to refer to objects or capabilities that can be acquired by the agent, and can be used to traverse or open some blocked nodes in the map.

There are several variants we can consider for this problem, e.g., with bounded or unbounded inventory and other parameters. As in regular pathfinding, we are interested in finding the shortest path from start to destination but we also consider that some locations along the path may require the agent to have previously visited other special nodes to “unlock”.

Some of the most indicative variants will be introduced in the next sections. For most of our work we will consider a practical one, namely, IAP-UM-SU on a grid map, which assumes unbounded inventory and monotonic effects, that is, that an item can only unlock obstacles and never limit the character’s reachable locations.

We note that this is a one-shot complete-knowledge task, in the sense that the agent has all the relevant information at the outset—the map nodes and their connectivity, the blocked nodes, the items at nodes, and the items that open blocked nodes—and will deliberate offline in order to find the shortest path that brings them to destination.

While there are various straightforward solutions for dealing with pathfinding in videogames in the context of characters with different capabilities, e.g., characters that can fly or swim, and characters with different sizes, there is no documented approach for dealing with this type of inventory-driven pathfinding where objects and capabilities can be acquired in the course of executing the path. Similarly, in the academic literature for pathfinding search, there is no approach, to the best of our knowledge, that handles this particular variant of pathfinding. Of course, inventory-driven pathfinding can be formalized as a planning problem, e.g., by appealing to classical STRIPS planning, and using actions with appropriate precondition and effects to represent navigation, collecting objects, and using objects to open a blocked location. However, as we will show, being a domain-independent approach, this is an impractical approach for this task and is not suitable for applications like videogames, for which pathfinding requests ought to be resolved almost instantly.

Finally, we note that it is not possible, in general, to know in advance if an inventory item in the map will be needed or not in order to traverse the optimal path. This is indeed something to be discovered as part of the planning process and, as such, no item may in general be ignored at the outset.
Chapter 4

The General IAP Problem and Variants

In Section 3.3 we discussed the scope of the IAP problem and gave a brief informal definition of a particular variant. That definition is limited to the case in which the character can carry an unbounded amount of items in its inventory and the map is represented as a grid, two assumptions that are practical and we will use in the later chapters. Here, we present the IAP problem in its most general form and, at the same time, introduce a formal definition for many of the elements that we will use later.

4.1 The General IAP problem (G-IAP)

First, we need to introduce some elements.

Definition 4.1. We define an IAP pathfinding labeled graph as tuple \( G = (V, E, E_I, K, cost) \) representing the static information for the problem. This includes:

- A graph \( (V, E, cost) \) in which the character is able to move. Each vertex in \( V \) is a location, and edges in \( E = V \times V \) show the connectivity of locations.
- A cost function \( cost: E \rightarrow \mathbb{R} \) representing the cost of traversing an edge \( e \).
- A finite set of items \( K \).
- A special set of edges \( E_I \subseteq E \). These edges can be traversed by the character depending on the character inventory.

Definition 4.2. We define an IAP state as a tuple \( S = (v, I, loc) \) representing the dynamic properties of the map and character. This includes:

- A function \( loc: K \rightarrow V \) returning the vertex in which the item \( k \) is located.
- The character inventory \( I \subseteq K \) representing the set of items the character carries in a specific moment.
- The current position \( v \) of the character.

Definition 4.3. We define an IAP unlock relation, \( Unlock \subseteq \mathcal{P}(K) \times E \), representing all the inventory combinations that can unlock the edge \( e \). This relation describes the semantic of the dynamic properties of the map. We say that \( I \) unlocks \( e \) if and only if \( (I, e) \in Unlock \).
Next, we must define the set of actions $A$ that the character can perform in each vertex. In addition to the classical, $\text{traverse}(e)$ actions represented by the graph edges, we need to add two actions for the inventory manipulation: $\text{pick}(k)$, that adds to the inventory the item $k$, and $\text{drop}(k)$ that drops the item $k$ at the current location.

- $\text{traverse}(e)$. This action is the regular movement action. It takes an edge $e = \langle v_i, v_j \rangle$ and moves the character through it from $v_i$ to $v_j$. This action keeps the standard prerequisite that the character must be in $v_i$. However, in IAP there is the additional prerequisite that, if the edge is in $E_I$, then the character current inventory must satisfy the $\text{Unlock}$ relation. Formally, given the pathfinding state $S = \langle v, I, \text{loc} \rangle$, the action $\text{traverse}(\langle v_i, v_j \rangle)$ is valid if and only if $v = v_i$ and $\langle I, \langle v_i, v_j \rangle \rangle \in \text{Unlock}$

After this action, starting from current state $S = \langle v_i, I, \text{loc} \rangle$ we obtain a new state $S' = \langle v_j, I, \text{loc} \rangle$.

- $\text{pick}(k)$. With this action, the character can take the item $k$ in the current location. As a consequence, the character gets a new inventory $I' = I \cup \{k\}$. As a prerequisite, given the current character location $v$, we need that $\text{loc}(k) = v$. After this action, starting from the current state $S = \langle v, I, \text{loc} \rangle$ we obtain a new state $S' = \langle v, I', \text{loc} \rangle$.

- $\text{drop}(k)$. This is the inverse of the $\text{pick}(k)$ action. With this action, the character drops the item $k$ and gets a new inventory $I' = I - \{k\}$. After this action, starting from the current state $S = \langle v, I, \text{loc} \rangle$ we obtain a new state $S' = \langle v, I', \text{loc}' \rangle$. There are multiple ways to update the $\text{loc}$ function after a drop. Each way can drastically affect the complexity and the power of the algorithm. Here, we will not make any assumptions but we will discuss practical options in Section 4.2.

All these actions have a cost associated. The cost for the $\text{traverse}$ action is indicated by the cost function $\text{cost}$ for the correspondent edge. For $\text{pick}$ and $\text{drop}$, instead, we can define a constant or variable cost. In the later chapters, we assume that $\text{pick}$ and $\text{drop}$ have zero cost (however, we can add any cost function as needed by the scenario in question).

Now, we can define the concept of an IAP sequence in the IAP problem as any sequence of actions $\sigma = a_1a_2a_3\ldots a_n$, where $a_i$ is one of $\text{traverse}$, $\text{pick}$ or $\text{drop}$ action for appropriate arguments. A valid IAP sequence, instead, is a sequence $\sigma$ such that, given an initial inventory $I_0$, any action in the sequence is valid, that is, every action satisfies their prerequisites. For a valid IAP sequence we extract the IAP path $\rho$ which is the sequence of locations corresponding to the $\sigma$ sequence of actions.

**Definition 4.4.** We are now ready to specify an Inventory-Aware Pathfinding problem instance. The following are given:

1. A pathfinding graph $G$ as defined in Definition 4.1.
2. An initial state $S = \langle s, I_0, \text{loc} \rangle$ as defined in Definition 4.2, where $s$ is the starting location in $G$, $I_0$ is the initial inventory (usually empty) and $\text{loc}$ the location of the items in the graph.
3. An unlock relation $Unlock$ as defined in Definition 4.3.

4. A goal position $g$ representing the target location in $G$.

Then, a solution to the IAP problem is any IAP sequence $\sigma = a_1a_2a_3 \ldots a_n$ such that, given $\rho = v_1v_2 \ldots v_n$ the IAP path corresponding to $\sigma$, $v_1 = s$, $v_n = g$ and the sequence is valid for $I_0$. Moreover, given the cost of a sequence as

$$cost(\sigma) = \sum_{a \in \sigma} cost(a)$$

(4.1)

where $cost(a)$ is the action cost function, we define an optimal solution $\sigma^*$ as any sequence such that:

$$cost(\sigma^*) \leq cost(\sigma)$$

(4.2)

for every valid sequence $\sigma$ that is a solution to the IAP problem.

### 4.2 The effects of drop actions variations

The drop action needs more elaboration. In fact, the way we define the drop action effect can dramatically change the complexity of the problem. In the above definition of IAP we only specified the effect of drop on the character inventory but we did not define what happens to the dropped items. For this, there are three main practical alternatives in the context of videogames:

**Drop-and-Respawn (DR).** We assume that the item is removed from the character inventory, it disappears from the current location, and it re-spawns in its original place.

**Drop-and-Forget (DF).** In this drop alternative, every item that is dropped stays at the current location but it is considered lost as far as the offline search is concerned. In other words, we allow an item to be picked only once during a single search instance; a character cannot move to a location, drop the item, perform other moves, and come back to pick it up later in the same offline search.

**Drop-in-Place (DP).** This is the most realistic drop alternative and consists of leaving the item in the current character location and allowing the character to pick it up again during the same offline search. However, this requires changing the $loc$ function during the execution of the pathfinding instance. This increases the complexity because now the $loc$ function must be considered as a stateful object that can remember the items location and it is affected by $pick$ and $drop$ actions.

In robotics and other realistic settings, Drop-in-Place seems to be the only meaningful drop definition. In fact, in the real world we never watch a dropped item disappear or magically come back to its original location. However, in videogames all these approaches can be used. The use of spawn points for items, weapons and power-ups is very common in many games. In other games, dropped items disappear after a short while and therefore we can assume safely that dropped items can be forgotten.

There are many examples in video-game history for every one of the approaches above. In Role-Playing Games, and other games in which we want our game world
to be as much as realistic as possible, games prefer the Drop-in-Place approach. In *The Elder Scrolls V: Skyrim* (Bethesda, 2011), for instance, items can be dropped anywhere and stay there even after a long time. In *The Legend of Zelda: The Wind Waker* (Nintendo, 2002), instead, many items respawn at the origin after being used or dropped. This is done especially for items that are required in order to advance in the game or solve specific puzzles. The goal of DR, here, is to avoid that the player can get stuck by accidentally throwing away a fundamental item in an unreachable location. Finally, action packed games such as *First-Person Shooters*, have a very immediate use of items and they are usually destroyed after use or when the player decides to drop them.

The drop action choice also affects the power of IAP algorithms. In fact, different types of drop actions can solve different problems. We can call as IAP the set of problems that can be solved by the Generalized IAP, then, we call IAP-DR, IAP-DF and IAP-DP the set of problems that can be solved by a IAP character with the DR, DF and DP drop variation respectively. Now, given a problem $p$ we may be interested in which drop implementation can solve $p$ and what is the relation among these classes.

### 4.3 The unlock relation assumptions

The *Unlock* function defined in Section 4.1 is another important element of the IAP definition. In fact, we can define different assumptions on the unlock behavior and each of them can affect the behavior and the capabilities of a IAP algorithm. In general, an edge $e$ is unlocked by an inventory $I$ if and only if $\langle I, e \rangle$ is contained in the *Unlock* relation of the specific graph. Given this relation, it is useful to look at two functions: $\text{unlockedBy}(I)$ returning the set of possible edges that can be unlocked by the inventory $I$ and $\text{canUnlock}(e)$ returning the set of items that can unlock $e$.

Without any assumption, we are free to use any specific set of items to unlock a certain obstacle. For instance, we can have an edge $e$ such that $\text{canUnlock}(e) = \{\{a\}, \{b\}\}$ meaning that $e$ is unlocked if the character inventory is the set containing only item $a$, or the only item $b$ but it will remain locked if the inventory contains $a$ and $b$ or any other combination of items.
While this allows us to define any kind of exotic items and unlock conditions, this can be unnecessarily complex for common items. For instance, if \( a \) and \( b \) represent keys, it does not make much sense that the character is unable to unlock \( e \) if it carries both keys. We want to specify the \textit{Unlock} relation such that \( e \) is unlocked by every inventory that contains \( a \) or \( b \). However, if \( K \) is composed by many items, it will be tiresome to explicitly list every possible subset of items containing \( a \) or \( b \) in the \textit{unlock} relation.

To handle this problem, we will define some assumptions that can be added to the \textit{Unlock} relation in order to represent properties of particular items or obstacles.

**Independent-Items Unlock Relation**

A common property for the \textit{Unlock} relation is that each item contributes to the unlocked/locked state of an obstacle \textit{independently} from other items. In other words, if an obstacle is unlocked by the item \( a \) (or by its absence), then every inventory containing \( a \) (or not containing \( a \)) will unlock the obstacle.

This property allows us to adopt a more compact way to specify the \textit{Unlock} relation (instead of listing every possible inventory). We define an \textit{unlock condition} for an obstacle as a set of \textit{item conditions}. Each item condition can appear in two ways:

1. Positive items, \( k \), meaning that the item \( k \) must be contained in the unlocking inventory.
2. Negative items, \( \bar{k} \), meaning that the item \( k \) must \textit{not} be contained in the unlocking inventory.

An obstacle with an unlock condition \( UC \) is unlocked by an inventory \( I \) if and only if \( I \) contains all the positive items listed in \( UC \) \textit{and} not contains all the negative items listed in \( UC \).

For instance, if we have a door with the following unlock condition \( UC = \{a, \bar{b}\} \), then the door is unlocked by any inventory containing item \( a \) \textit{and not containing} the item \( b \). For instance, \( \{a\} \) and \( \{a, c\} \) both unlock the door while \( \{c\} \) and \( \{a, b\} \) keeps the door locked.

**Monotonic Unlock Relation**

If no obstacle has an unlocking condition containing negative items, then we say that the unlock relation is \textbf{monotonic}. A monotonic unlock relation captures the scenario in which no item can block a previously unlocked object. This scenario is quite common for many physical items, most noticeably the case in which we have locked doors and keys to open them.

A monotonic unlock relation is a very powerful assumption and has a huge benefit if combined with an unbounded inventory size: it makes the drop action useless. Under the monotonicity assumption, in fact, there is no reason to drop an item because carrying more items corresponds to the possibility to reach more locations in the map.

We will talk more of the effect of the monotonic unlock assumption in the later sections.
4.4 Inventory Size

There is another aspect in IAP that can affect the behavior and the overall complexity of the algorithm. This last aspect is the inventory size. There are two possible alternatives:

**Unbounded Inventory.** In this case, we assume that each character can carry around an unbounded number of items.

**Bounded Inventory.** In this case, we add the limit that \(|I| \leq N\) in every moment. Therefore, the character can carry at most \(N\) different elements.

4.5 Classification for IAP problems

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Variations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of drop action</td>
<td>DR</td>
<td>Dropped items respawn in the original location.</td>
</tr>
<tr>
<td></td>
<td>DF</td>
<td>Dropped items are ignored.</td>
</tr>
<tr>
<td></td>
<td>DP</td>
<td>Items are dropped at current location.</td>
</tr>
<tr>
<td>Assumptions on unlock</td>
<td>UG</td>
<td>Unconstrained.</td>
</tr>
<tr>
<td></td>
<td>UI</td>
<td>Each item acts independently.</td>
</tr>
<tr>
<td></td>
<td>UM</td>
<td>Monotonic. No negative items.</td>
</tr>
<tr>
<td>Size of the inventory</td>
<td>SB</td>
<td>Bounded Inventory</td>
</tr>
<tr>
<td></td>
<td>SU</td>
<td>Unbounded Inventory</td>
</tr>
</tbody>
</table>

In the previous section, we identified three different axes over which we can constraint G-IAP. We can see a summary of these directions in Table 4.1. From the table, we can derive 18 different IAP variations. However, not every assumption is independent from each other. For instance, we have seen that if we choose an unbounded inventory and a monotonic unlock relation, then we can totally ignore the drop action.

We will explore how IAP complexity and computational power change when varying these different algorithm parameters and assumptions.

4.5.1 IAP expressive power for different Unlock assumptions

Adding constraints to the Unlock relation will necessarily limit the type of problems IAP can express. As we said before, a completely generic Unlock relation can be used to represent any possible unlock condition for an obstacle.

The independent item effect (UI) assumption, instead, has limited description power. We can easily see that, considering only one edge \(e\), the general relation \(Unlock = \{\{a\},\{b\}\}\) cannot be represented using the notation we defined for the UI assumption.

In the same way, the monotonic assumption, is even more limited than UI. It is, in fact, impossible to describe unlock conditions in which an obstacle is blocked only if the character inventory contains a specific item.
4.5 Classification for IAP problems

4.5.2 IAP expressive power for different drop specifications

In this section, we want to explore how an algorithm’s capability to solve IAP problems changes depending on the specific drop action assumption. In this case, the problem is not on which problem we are able to define using a specific drop action, but rather which problems we are able to solve. In fact, not every problem that can be solved with a specific choice can be solved by an algorithm using another option (and vice versa).

First, we can easily verify the following proposition.

**Proposition 4.1.** Given three IAP variants, namely IAP-DF, IAP-DP and IAP-DR which differ only for the type of drop action, if we consider the set of possible problems that can be solved by each variation, the following relations are true

1. IAP-DF ⊂ IAP-DP
2. IAP-DF ⊂ IAP-DR
3. IAP-DP ∩ IAP-DR ≠ ∅
4. IAP-DP ∉ IAP-DR and IAP-DR ∉ IAP-DP
5. IAP-DF ⊂ (IAP-DP ∩ IAP-DR)

In order to prove the first two relations, we can see that in both cases we can simulate a IAP-DF implementation by simply dropping an item and never picking it up again. Therefore, we conclude that necessarily, IAP-DF ⊆ (IAP-DP ∩ IAP-DR), i.e., if a problem can be solved by a IAP character with a DF drop action, then can be solved by a character with any drop action.

A more interesting result is represented by relations 3 and 4, namely that IAP-DP ∩ IAP-DR is not empty and that IAP-DP and IAP-DR are not one a subset of the other. We can verify this by showing two problems, one that can be solved by a IAP character with a DF drop action, then can be solved by a character with any drop action.

In the first case, we can look at the scenario shown in Image 4.2 in which we have two rooms closed by some obstacles that can be unlocked only by a specific combination of items. Item $k_A$ is in an open area while item $k_B$ is inside the first room. We can see that we need item $k_A$ in order to enter the room but, once inside, we cannot leave the room carrying both item $k_A$ and $k_B$. However, we need both items to access the second room with the goal.

It is easy to see that this scenario cannot be solved by a Drop-in-Place (DP) character because it is impossible to leave the first room with both items. However, this scenario can be solved by a Drop-and-Respawn (DR) character. The IAP-DR character can, in fact, go into the first room, take $k_B$, drop $k_A$, leave the room, take again $k_A$ at the spawn point and finally reach the goal.

To prove that IAP-DP ∉ IAP-DR, instead, in Figure 4.3 we show another scenario that can be solved by IAP-DP but not by IAP-DR. In the image we can see that, in order to reach the goal $g$ we need item $k_C$. A IAP-DP character can easily solve this problem with the following steps:

1. Get item $k_A$.
2. Enter in the small lobby and get item $k_B$.
3. Leave item $k_A$ on the floor, enter in the next room and get $k_C$. 

In order to prove the first two relations, we can see that in both cases we can simulate a IAP-DF implementation by simply dropping an item and never picking it up again. Therefore, we conclude that necessarily, IAP-DF ⊆ (IAP-DP ∩ IAP-DR), i.e., if a problem can be solved by a IAP character with a DF drop action, then can be solved by a character with any drop action.

A more interesting result is represented by relations 3 and 4, namely that IAP-DP ∩ IAP-DR is not empty and that IAP-DP and IAP-DR are not one a subset of the other. We can verify this by showing two problems, one that can be solved by a IAP character with a DF drop action, then can be solved by a character with any drop action.

In the first case, we can look at the scenario shown in Image 4.2 in which we have two rooms closed by some obstacles that can be unlocked only by a specific combination of items. Item $k_A$ is in an open area while item $k_B$ is inside the first room. We can see that we need item $k_A$ in order to enter the room but, once inside, we cannot leave the room carrying both item $k_A$ and $k_B$. However, we need both items to access the second room with the goal.

It is easy to see that this scenario cannot be solved by a Drop-in-Place (DP) character because it is impossible to leave the first room with both items. However, this scenario can be solved by a Drop-and-Respawn (DR) character. The IAP-DR character can, in fact, go into the first room, take $k_B$, drop $k_A$, leave the room, take again $k_A$ at the spawn point and finally reach the goal.

To prove that IAP-DP ∉ IAP-DR, instead, in Figure 4.3 we show another scenario that can be solved by IAP-DP but not by IAP-DR. In the image we can see that, in order to reach the goal $g$ we need item $k_C$. A IAP-DP character can easily solve this problem with the following steps:

1. Get item $k_A$.
2. Enter in the small lobby and get item $k_B$.
3. Leave item $k_A$ on the floor, enter in the next room and get $k_C$. 

4. Come back to the first room and take again $k_A$.

5. Use $k_C$ to reach the goal room.

However, we can see that this solution is not suitable for a IAP-DR character because when the character drops item $k_A$ at step 3, item $k_A$ will be transported back to its original spawn point. As the spawn point for item $k_A$ is unreachable from dropping location, the character is now locked in the room where item $k_B$ is originally placed.

Now, the only remaining question is whether the relation 5 is true or not, that is if IAP-DF = (IAP-DP ∩ IAP-DR) or just a subset of it. Also in this case, the answer is negative and we can find a problem instance that can be solved by IAP-DP and IAP-DR, but it is impossible to solve for a IAP-DF character (Figure 4.4). In this scenario, at some point, the character needs to drop item $k_A$ to enter the room with item $k_B$. This is not a problem with IAP-DR (we can drop $k_A$ and take $k_A$ again from the spawn point) or for IAP-DP (we can temporarily leave $k_A$ outside the door), but it is a huge problem for IAP-DF that, on drop, will lose access to $k_A$ for the rest of the pathfinding search.
4.6 The Layered-Items Graph Model (LIG)

Working with a IAP graph in the way we specified it in the previous section is quite different from what we see in regular pathfinding. In particular, we have actions, edges that are conditionally enabled or disabled based on the current state of the character. Trying to implement solutions for this problem can be challenging. Fortunately, starting from the previous IAP problem specification, it is possible to build a different representation of the problem that can be solved by a traditional pathfinding algorithm and that is equivalent to a IAP problem with a Drop-and-Respawn or Drop-and-Forget kind of drop action.\footnote{While it is possible to extend the LIG model to the general case, in this work we focus on Monotonic Unbounded IAP (IAP-UM-SU) in which we do not need a drop action at all.}

We call this new representation Layered-Item Graph Model (LIG Model). The basic idea is to transform the graph $G$ into a new graph $\hat{G}$ that embeds all the
information about actions, prerequisites and the state of the character. In this way, we can map a path in \( \hat{G} \) into a IAP path in \( G \) and vice versa.

### 4.6.1 From the standard pathfinding graph to the LIG model

**Definition 4.5.** Given the original pathfinding graph \( G = \langle V, E, \text{cost} \rangle \), we can compute \( \hat{G} = \langle \hat{V}, \hat{E}, \text{cost}' \rangle \) as follows.

- A vertex \( \hat{v} \) in \( \hat{V} \) is a tuple \( \langle v, I \rangle \). The set of vertices \( \hat{V} \) is populated by the Cartesian product of \( V \) and the set of possible inventories \( \mathcal{P}(K) \).
- We define a function \( \hat{v}^k : \hat{V} \rightarrow \hat{V} \) as a function that returns the first item in \( \hat{v} \). In other words, this function projects a vertex in \( \hat{G} \) into the vertex in the original graph corresponding to the *same location*. In this work, whenever we have a vertex \( \hat{v} \) we implicitly assume that \( v \) is the result of \( \hat{v}^k \).
- We define then, a function \( \hat{v}^l : \hat{V} \rightarrow \mathcal{P}(K) \) returning the second element of \( \hat{V} \), that is the inventory with which the node is labeled.
- Then, given an inventory \( I \), we define the *layer edges set\( \hat{E}_I \) as follows.

\[
\hat{E}_I = \{ \langle \hat{v}, \hat{w} \rangle : \langle v, w \rangle \in E \text{ and } \text{Unlock}(I, \langle v, w \rangle) \text{ and } \hat{v}^l = \hat{w}^l \}\]  

for every pair of vertexes \( \hat{v} \) and \( \hat{w} \) in \( \hat{V} \).
- We define \( V_I \) as the set of vertexes with the same inventory \( I \).

\[
V_I = \{ \hat{v} : v^l = I \}\]

- Then, we define the two set of *portal edges* corresponding to the *pick* and *prop* action in the G-IAP problem. The first set corresponds to the action *pick* (\( \hat{E}_{\text{pick}} \)) and it is defined by the following equations

\[
\hat{E}_{\text{pick}} = \{ \langle \hat{v}, \hat{w} \rangle : v = w \text{ and } \exists k \text{ loc}(k) = v \text{ and } w^l = v^l \cup \{k\} \}\]  

(4.5)

Instead, the second one to the *drop* action (\( \hat{E}_{\text{drop}} \)) is defined as

\[
\hat{E}_{\text{drop}} = \{ \langle \hat{v}, \hat{w} \rangle : v = w \text{ and } \exists k w^l = v^l \setminus \{k\} \}\]  

(4.6)

- We can define the edges of \( \hat{G} \) as the union of all the edges defined above.

\[
\hat{E} = \hat{E}_{\text{pick}} \cup \hat{E}_{\text{drop}} \cup \left( \bigcup_{I \in \mathcal{P}(K)} \hat{E}_I \right)
\]  

(4.7)

- Finally, the \( \text{cost}' \) function can be defined as the action-cost function for IAP.
This definition of the LIG model may seem to be more complicated than before. However, everything appears more intuitive if we introduce the notion of layer. A layer \( L_I = (\hat{V}_I, \hat{E}_I) \) is the sub-graph obtained by using only the vertexes in \( \hat{V}_I \) and the edges defined in \( \hat{E}_I \). We can see that a layer \( L_I \) represents exactly the original graph when the character has the inventory \( I \).

Under this definition, portal edges appear more clearly. Portal edges for the \textit{pick} action are directed edges that connect the location in \( L_I \) where the item \( k \) is, to the same location in the layer \( L_{I \cup \{k\}} \). In other words, following the edges in \( \hat{E}_{\text{pick}} \) moves the character in a new “copy” of the map representing the new state with the new item. Similarly, portal edges for \textit{drop}, are directed edges connecting every vertex in a layer \( L_I \) with the corresponding node in the layer \( L_{I \setminus \{k\}} \). That is, following a node in \( \hat{E}_{\text{drop}} \) has the same effect of the \textit{drop} action, moving the character into a new “copy” of the map reflecting the changed inventory.

Now, it is important to demonstrate how to map the original IAP problem into the LIG model and that a solution in the LIG model corresponds to a unique solution in IAP.

### 4.6.2 IAP problem in the LIG model

The power of the LIG model is that it allows us to transform an IAP problem into a traditional multi-goal pathfinding problem. In this way, any traditional pathfinding algorithm in literature that supports multiple goals can be used to solve an IAP problem. In fact we can transform an IAP problem \( \langle G, S, g \rangle \) (see Definition 4.4) into a multi-goal pathfinding problem \( \langle \hat{G}, s, Goals \rangle \) over the LIG graph where the set of goals \( Goals \) contains all the vertex corresponding to the same location regardless of the character inventory. More formally, given the original goal \( g \),

\[
Goals = \{ \hat{v} \in \hat{V} : \hat{v}^t = g \}
\]  

We now show a proposition showing the equivalence between the LIG model and the original IAP formulation.

**Proposition 4.2.** Given a IAP problem \( \langle G, S, g \rangle \) (as defined in Definition 4.4) and the corresponding LIG model problem \( \langle \hat{G}, s, Goals \rangle \), suppose \( \sigma \) a valid sequence in the first problem and \( \hat{p} \) a path in the LIG graph. Then, the following statements are true:

1. There exists a bijective function \( \lambda(\sigma) \) such that \( \lambda(\sigma) = \hat{p} \) and \( \lambda^{-1}(\hat{p}) = \sigma \).

2. If \( \sigma \) is a solution for the original problem, then \( \lambda(\sigma) = \hat{p} \) is a solution for the LIG model problem.

3. If \( \hat{p} \) is a solution for the LIG model problem, then \( \lambda^{-1}(\hat{p}) = \sigma \) is a solution for the original IAP problem.

To prove Proposition 4.2, first we show a way to build this \( \lambda \) function and its inverse \( \lambda^{-1} \).

**Definition 4.6.** We define that \( \lambda : \Sigma \rightarrow P \) is a function that brings any valid sequence \( \sigma = a_1a_2 \ldots a_n \) in the original IAP problem into a path \( \hat{p} = \hat{v}_0\hat{v}_1 \ldots \hat{v}_n \) in the LIG model graph as follows

1. \( \hat{v}_0 = (s, I_0) \) corresponding to the initial state of the original IAP problem.
2. Then, the $\hat{v}_i$ vertex is obtained by applying action $a_i$ in $\sigma$ to $v_{i-1}$ in the following ways:
   
   (a) If $a_i = traverse(v_i, v_j)$, then $\hat{v}_i = \langle v_j, I_{i-1} \rangle$.
   
   (b) If $a_i = pick(k)$, then $\hat{v}_i = \langle v_{i-1}, I_{i-1} \cup \{k\} \rangle$.
   
   (c) If $a_i = pick(k)$, then $\hat{v}_i = \langle v_{i-1}, I_{i-1} \setminus \{k\} \rangle$.

**Definition 4.7.** We define the inverse function $\lambda^{-1} : P \rightarrow \Sigma$ as a function that brings any path in the LIG graph into a sequence in the original IAP problem. Given an input path $\hat{p} = \hat{v}_0 \hat{v}_1 \ldots \hat{v}_n$, then the function returns a sequence $\sigma = a_1 a_2 \ldots a_n$ such that:

1. If $\hat{v}_i \neq \hat{v}_{i+1}^\downarrow$, then $a_i = traverse(\langle \hat{v}_i^\downarrow, \hat{v}_{i+1}^\downarrow \rangle)$.

2. Else, we look into $I_i$ and $I_{i+1}$. By construction, this two inventories differ only by one item. Named $k$ this item, then:
   
   (a) If $I_i = I_{i+1} \cup \{k\}$, then $a_i = drop(k)$.
   
   (b) If $I_{i+1} = I_i \cup \{k\}$, then $a_i = pick(k)$.

Now, we need to prove that the function defined by Definition 4.6 returns a valid path in the LIG graph and that the function in Definition 4.7 returns a valid sequence in the Generalized IAP problem. Before this, we show that any action in the IAP problem always corresponds to an edge in LIG and vice versa.

**Proposition 4.3.** Every edge in the LIG graph $\hat{G}$ corresponds to an unique valid action $a$ on the original IAP problem.

**Proof.** Given a generic character state at the $S = \langle v, I \rangle$ we have three possible cases:

- The action $a$ is $traverse(\langle v_i, v_j \rangle)$. The action prerequisites are that current position is $v_i$ and that exists and edge $\langle v_i, v_j \rangle \in E$ and $unlock(I_i, \langle v_i, v_j \rangle)$. This action produces a new state $S' = \langle v_j, I \rangle$. This, by definition corresponds to the only edge in $\hat{E}_I$ in LIG graph $\hat{G}$ that connects $\hat{v} = \langle v_i, I \rangle$ to $\hat{w} = \langle v_j, I \rangle$.

- The action $a$ is $pick(k)$. This action prerequisites are that the current location of the character corresponds to the location of the item $\langle loc(k) = v_j \rangle$. This action produces a new state $S' = \langle v_i, I' \rangle$ with the same location but a new inventory $I' = I \cup \{k\}$. Also in this case, this corresponds by definition to the only edge in the $\hat{E}_{pick}$ that connects $\hat{v} = \langle v_i, I \rangle$ to $\hat{w} = \langle v_j, I' \rangle$.

- The action $a$ is $drop(k)$. This action prerequisites are that the current inventory contains the dropped item $k$. This action produces a new state $S' = \langle v_i, I' \rangle$ with the same location but a new inventory $I' = I \setminus \{k\}$. Also in this case, this corresponds by definition to the only edge in the $\hat{E}_{drop}$ that connects $\hat{v} = \langle v_i, I \rangle$ to $\hat{w} = \langle v_j, I' \rangle$.

The other direction can be easily constructed by using the same strategy used for $\lambda^{-1}$.

As we have seen before, each edge in LIG connecting $\hat{v}$ to $\hat{w}$ is a valid action for a IAP agent in state $S = \langle v^i, v^f \rangle$, that is the state corresponding to $\hat{v}$, and it will bring the agent in the state $\hat{S} = \langle v^f, w^f \rangle$, that is the state corresponding to $\hat{w}$.

Then is easy to see that every path in the LIG graph correspond to a valid sequence of actions in the original IAP problem. Therefore, we can state the following proposition.
Proposition 4.4. Given a $\lambda^{-1}$ function defined as in Definition 4.7, given a path $\hat{p}$ from $\hat{v}$ to $\hat{w}$ in $\text{LIG}$, the sequence $\sigma = \lambda^{-1}(\hat{p})$ is a valid sequence that moves the agent from $S_v = \langle v^I, v^l \rangle$ to $S_w = \langle w^I, v^l \rangle$.

This can be proved by induction starting from a path composed by a single edge and moving to a path of size $n$.

It is easy to see now, that if the path in the $\text{LIG}$ graph starts from the vertex corresponding to the initial state of the IAP problem and ends to one of the vertexes corresponding to the goal state, then $\lambda^{-1}$ maps a solution in the $\text{LIG}$ model to a solution in the original IAP problem.

A similar reasoning can be done for the other direction—a solution sequence in IAP will map to a solution to the $\text{LIG}$ problem).

However, in practice, this is not a practical solution even for many simple cases with few items and small maps. Passing from $G$ to $\hat{G}$ exponentially increases the search space (potentially we have $2^{|K|}$ more vertexes and edges than before). For this reason, in the next chapters we will present some more advanced techniques that can exploit the properties of the $\text{LIG}$ space.

4.7 The Unbounded Monotonic IAP problem

In the rest of this work, we will focus our attention to the simpler case of Unlimited Monotonic IAP problems (IAP-SU-UM). This combination shows many interesting properties that can make the problem tractable and practical, especially in real-time videogame applications.

The stronger advantage of IAP-SU-UM is the possibility to completely ignore the drop action. In fact, the combination of an unlimited inventory with the fact that picking a new item always increases the reachable states makes unnecessary to throw an object away.

4.7.1 Properties of $\text{LIG}$ under IAP-SU-UM assumption

The most evident effect of removing the Drop action, is the removing of the drop edges $\hat{E}_{\text{Drop}}$ from the $\text{LIG}$ graph $\hat{G}$. In this scenario, the $\text{LIG}$ graph is composed only by the various layers connected by the pick portal edges at the respective items locations. As in the Unlimited Monotonic case, pick portals are the only inter-layers edges, from now on we will refer to them simply as portals.

The topological structure of the $\text{LIG}$ graph is now more evident. In fact, we can represent the $\text{LIG}$ graph as a graded partial ordered set of layers as shown in Fig. 4.6.

- First, we define a partial ordering on the set of layers using the subset property of the inventories with which the layers are labeled. For instance, we say that $\mathcal{L}_A \leq \mathcal{L}_B$ if and only if for the labeling inventories is true that $A \subset B$ and, therefore, the two properties are equivalent.

- Then, we define as rank of the graded poset the cardinality of the labeled inventory. For instance, the rank of the layer $\mathcal{L}_{\{a,b\}}$ is 2 because $|\{a, b\}| = 2$.

Now, we can show some interesting properties of the $\text{LIG}$ graded poset of a UM-IAP.

\[\text{An Hasse Diagram is used to represent a finite partially ordered set by using set elements as vertexes and an an edge for each pair of elements that satisfies a particular ordering relation. In LIG we use the layer's inventory inclusion relation (i.e., the inventory of } x \text{ is a subset of the inventory of } y).\]
Proposition 4.5. We define \( \text{reach}(s, I) \) a function returning all the locations \( g \) such that there exists a path that brings the character from \( s \) to \( g \) in \( L_I \). Given another layer \( L_J \) such that \( L_I < L_J \), it follows that \( \text{reach}(s, I) \subset \text{reach}(s, J) \).

In other words, the number of locations that can be reached in each layer monotonically increases while we add items to the character inventory. This is a direct consequence of the monotonicity assumption and will be important later.

4.8 Other IAP problem specifications

There are other extensions to the IAP problem that we do not discuss here for simplicity. In this section, we will introduce them briefly.

4.8.1 Pick and Drop cost

In the previous sections, we assumed that the cost for the \textit{Pick} and \textit{Drop} action were free. However, this assumption is not always true. First, picking an action can require some work for the character. This can be required by a gameplay element—e.g., we can intend the pick action as a “mine iron from a mining spot”—or a graphical specification—e.g., the character must perform a “pick” animation before moving on. Second, we can consider as “item” a much more general concept such as “kill a monster” or “perform a certain task in that area”. These are much more complex actions, and sometimes entire sub-tasks, that cannot be considered free for the character.

In all these cases in which we cannot neglect the cost of this actions, we need to add a cost to the \textit{Pick} (and eventually \textit{Drop}) action.

This extension can be handled straightforwardly by the layered model presented previously. We just need to add a cost to the “portals” edges between layers. However, some optimizations presented in the later chapters will be affected by this change. For instance, in Monotonic Unbounded IAP we guarantee that it is always better to take as much as items as possible if they are on the path. This is not true anymore if we add a cost to the \textit{Pick} action.
4.8 Other IAP problem specifications

4.8.2 Inventory encumbrance

A more interesting extension is the inventory encumbrance. In the previous sections we never considered the items effects on the characters mobility. It is not uncommon in video-games, that certain items affect the speed of the characters. For instance, carrying around a heavy item can be useful to unlock a certain door, but will significantly slow down the character. An optimal solution, in this case, require a more complex interleaving of the Pick and Drop action in order to minimize the time the character has that item in the inventory.

In the cases in which a LIG model can be constructed, this effect can be modeled by increasing the cost of every edge depending on the inventory, for instance by uniformly scaling the cost of each layer $L_I$ by $K(I)$ where $K$ is a slow-down factor depending on the inventory.

However, note that with the inventory encumbrance the overall effect of carrying an item is no more monotonically positive. Because of this, many of the optimizations we will use later for the unbounded inventory case are no more applicable even if we are using an unbounded inventory with encumbrance. The most obvious difference is that we need to provide a Drop action even if we have infinite space to allow the character to move on a faster layer.

As a final remark, note also that we can have positive encumbrance (e.g., carrying a bike allow us to go faster). In this case, we could use a negative $K$ slowdown value. Note, however, that we must be careful not to break the admissibility of the heuristic function used in the pathfinding algorithm.

4.8.3 Persistent item effect

In this thesis, we will assume that an obstacle can be unlocked by an item if and only if the character is carrying the item while is passing the obstacle. However, in many real-world scenarios this is not always the case. For instance, a door unlocked by a key remains unlocked also if the character drops the key after unlocking it.

We call this item effect persistency. This kind of obstacles and items are not modeled by the above IAP problem definition. A general solution to this problem needs to transform the objects into statefull entities, so that they are able to remember when they have been unlocked or not.

However, under the monotonicity assumption, there is an easier solution. If the original problem is an unbounded, then there is no Drop action and, therefore, we can ignore the problem. If, instead, the original problem was bounded, instead of removing the item from the inventory, we can increase the inventory size by 1 every time the character drops an item but, at the same time, considering the item in the inventory for the path planning search.
Chapter 5

Inventory-Aware JPS (IA-JPS)

Since we are motivated by videogame worlds, we cannot talk about Inventory-Aware Pathfinding without providing a fast and practical implementation of it. Even if the problem’s worst case scenario complexity remains exponential in the number of items, we would be interested in an algorithm that can manage a small but practical number of items on the map, e.g., 5 or 6 items. However, a naive implementation of IAP can hardly reach practical performance even for such a limited number of items.

From Chapter 4, we have seen that a IAP is essentially a pathfinding query in a huge expanded space referred as Layered Item Graph (LIG). For this reason, a good starting point for an efficient IAP algorithm is a pathfinding algorithm that works as fast as possible.

In this and the later chapters we will work on Jump-Point Search (JPS), an award-winning algorithm which has attracted much attention within the game community. As all other pathfinding approaches, it is not able to cope with such inventory-driven scenarios natively: it will either yield the best path to the goal that does not resort to acquiring extra capabilities or output no solution. In this chapter then we show a straightforward and minimal extension of JPS that can be elaborated in a principled way to accommodate IAP. This basic implementation will be the foundation on which we will build further optimizations in the later chapters.

5.1 The classical Jump Point Search

Jump-Point-Search (JPS) is a technique introduced by Harabor and Grastien in [31] and further elaborated in [32, 33], that has proven extremely successful to navigate uniform cost grid-based maps. As we briefly anticipated in Chapter 2, JPS is ultimately a speedup approach based on the well-known A* [35] and it belongs to the class of symmetry-breaking based algorithms.

5.1.1 Overview of JPS

JPS operates by identifying every node that might end up being on the optimal path, i.e., the so-called jump-points, and discards the rest. Thus, instead of expanding all reachable nodes, JPS jumps from a potential turning (jump) point to another turning (jump) point, expanding only those nodes that might require a change of direction (Algorithm 19).

Technically, JPS operates like A* by working through the grid systematically, maintaining an open list, and selecting and opening nodes from that list using the
same best-first evaluation function as A*; but JPS is also a beam search in that it prunes as it goes.

When a node \( x \) in the open list is expanded, A* retrieves all its unblocked adjacent nodes and adds them to the open list. Instead, JPS retrieves only some adjacent nodes of \( x \), constructs a vector of travel from \( x \) through each of them, and identifies the first so-called “jump point” that occurs along that vector—a final successor of \( x \)—which is finally added to the open list. Informally, a jump point represents a location in which the traveling direction in the optimal path may change.

Since intermediate nodes from node \( x \) to each jump point are not explicitly handled or stored (in the closed list), once the goal is found and the path needs to be assembled, JPS must “fill in the gaps” between jump-points to generate a straight or diagonal path.

**Algorithm 19 Jump Point Search - Identify Successor**

1: function IDENTIFYSUCCESSORS(current, start, goal)
2:     successors(current) ← ∅
3:     neighbors(current) ← prune(current, neighbors(current))
4:     for each \( n \in \) neighbors(current) do
5:         \( n \) ← JUMP(current, direction(current, n), start, goal)
6:     insert(successors(current), n)
7: return successors(current)
5.1.2 Pruning mechanism

Given a node $x$ reached via a parent node $p$, JPS prunes from any node $n \in \text{neighbors}(x)$ of $x$ such that (assuming corner-cutting is not allowed) either:

1. there exists a path $\pi'$ from $p$ to $n$ that does not go through $x$ and is shorter than path $\pi = \langle p, x, n \rangle$; or

2. there exists a path $\pi'$ from $p$ to $n$ that does not go through $x$, has the same length as $\pi = \langle p, x, n \rangle$, but $\pi'$ has a diagonal move earlier than $\pi$.

For example, in Figure 5.1a and 5.1b, pruned neighbors of $x$ are marked in gray; the remaining neighbors of $p$ are marked as white and are referred to as the direct successors of node $x$. The natural successors of $x$ are the direct successors if one were to assume no obstacles whatsoever. In the absence of obstacles, non-natural successors are pruned. But the presence of obstacles may preclude the pruned rules to discard some non-natural successor, which therefore end up in the set of direct successors. Such non-natural successors that could not be pruned form the set of forced neighbors of $x$. In Figure 5.1c, the patterned cells $z_1$ and $z_2$ are forced neighbors of $x'$: they are not natural successors of $x'$, but they could not be pruned because the move $\langle p', z_1 \rangle$ cuts corner and is hence not legal. Importantly, only straight movements may produce (up to 4) forced neighbors.

5.1.3 Jumping procedure

As explained by [32], JPS applies to each actual successor—natural or forced neighbor—of the current node $x$ a simple “jumping” procedure to replace each neighbor $n$ with an alternative successor $n'$ that is further away, the next “jump point.” Technically, a jump point is a node that contains a forced neighbor. In Figure 5.1c, node $x'$ is a jump point. Intuitively, the fastest way to reach $z_1$ and $z_2$ from $p'$ is via $x'$, and hence node $x'$ becomes a “turning point.” The pseudo-code for the jump procedure is shown in Algorithm 20.

When moving in a straight manner, the only natural neighbor is explored recursively, in the corresponding straight direction, until either an obstacle, a jump point (i.e., a node having a forced neighbor; e.g., node $x'$ above), or the goal is encountered. In the first case, the path is deemed failed, all nodes in it are ignored,
and nothing is generated. If, however, a node $n'$ having one or more forced neighbors—jump points—or being the actual goal is reached, then $n'$ is generated as a next successor of $x$ and is added to the open list; effectively “jumping” from $x$ to $n'$ without adding any of the intermediate nodes into the open list. So, in Figure 5.1c, the final successor of node $x$ is jump point node $x'$, which has nodes $z_1$ and $z_2$ as a forced neighbors.

When moving diagonally, as in Figure 5.1b, node $x$ has three natural neighbors (white cells in $x$ surroundings): two straight (north and east) and one diagonal (east north). JPS then recourses over the diagonal neighbor only if both straight neighbors produce failed paths. When there is a non-failed straight jump, then the node in the diagonal path is also considered an (indirect) jump point—a potential turning point—and is added to the open list (node $y$).

### 5.2 Extending JPS for IAP

In this section, we show that JPS can be further elaborated in a principled way to accommodate inventory-driven path planning. The new algorithm, which we call inventory-aware JPS (IA-JPS), is obtained by modifying JPS in three simple ways.

The **first modification**, as one would do with any search approach, involves extending the state representation to account not just for the location of the agent, but also the current inventory. So, for a map $M$ a state is a pair $\langle x, I \rangle$ where $x$ is a node in $M$ and $I$ is a subset of elements from set $\mathcal{I}$ of all possible items. As in this work we do not consider the cost for obtaining or carrying an item, during search when an agent is at a location that has items, these are placed all in their inventory instantly. The items that an agent carries allow them to traverse also nodes that are marked as blocked but are labeled by $\text{req}$ with a set of items such that the intersection with the inventory $I$ is non-empty.

The **second modification** to JPS involves treating any node containing some capability or object as an “intermediate” goal. In other words, during the (recursive) jumping process, when an *intermediate* node $x$ is generated such that $\text{obj}(x) \neq \emptyset$, then the process is deemed complete and $x$ is considered a jump point (and thus added to the open list). We call such jump point nodes, *inventory jump points*. Note that when an item $i$ is already contained in the state then acquiring another instance of the same item $i$ (by visiting a node where the second instance lies) does not change the state representation in the search process and does not generate an inventory jump point. In the case though that two items $i, i'$, such that $i \neq i'$ happen to unlock the same nodes in the map, acquiring the essentially “duplicate” item $i'$ is handled in the generic way by the algorithm. In Algorithm 21, you can see the modified *jump* procedure. Note that it differs from the original one only for lines 9 and 10.

Finally, observe that, contrary to what one would expect, we do not require any change to the pruning mechanism and hence we took a “lazy” approach to inventory “finding.”

In classical JPS, when a new (jump point) node is retrieved from the open list for expansion, *all* directions towards natural and forced neighbors of the node in question are considered for further “jumps.” The natural successors represent the same direction the agent was traversing when a jump point was found, whereas the forced neighbors represent the turning directions that the agent may need to consider (see [31, Lemma 1]). This implies, for example, that it is not necessary to consider the parent of the jump node, as this would involve undoing the path traveled so far. However, in the context of inventory-based path planning, if the
node being expanded is an inventory jump point, then the agent ought to consider all possible directions, including that undoing the path traversed so far! The fact is that, with the new inventory acquired, nodes that looked “blocked” before may have now become traversable.

So, the third modification included in IA-JPS is to treat inventory jump point nodes as the starting node, thus applying the jumping process towards all possible directions. This is depicted in Figure 5.1d (assume the key opens all doors). When jumping east from x, the node with the item becomes a jump point. From there, the agent must consider jumping towards all directions, not just east. In fact, the agent should consider returning to node x as a new jump point, now though holding the key. Later, the agent will possibly generate node y as a jump point too, because the node north to it is now open to the agent and will allow her to visit area A. A similar argument can be given for the north door and area B (using the north-west direction).

Let us refer with IA-JPS to the algorithm obtained from implementing the above three modifications to JPS.

**Algorithm 21** Inventory-Aware JPS-Jump

```
function JUMP(current, direction, start, goal)
    n ← STEP(current, direction)
    if n is obstacle or is outside the map then
        return null
    if n = goal then
        return n
    if ∃n′ such that n′ is forced then
        return n
    if n is an item location then
        return n
    if direction is diagonal then
        d1 ← a direction obtained rotating direction 45° c.w.
        d2 ← a direction obtained rotating direction 45° c.c.w.
        if JUMP(n,d1,start,goal) is not null then
            return n
        if JUMP(n,d2,start,goal) is not null then
            return n
    return JUMP(n,d2,start,goal)
```

5.2.1 IA-JPS in the LIG model

On closer inspection, we can see the above three modifications as the extending of JPS to the **Layered Item Graph (LIG)** model we discussed in 4.6.

- The first modification corresponds to the node labeling of LIG. In fact, we can see that for each location x in M we are adding a different search node for every possible inventory. This is exactly the procedure we used to obtain the vertexes \( \hat{V} \) set in LIG. Ultimately, the search node used in the original JPS
  \[ s = (f,g,parent,x) \]  
  
  is extended with the new inventory information
  
  \[ s_{IA} = (f,g,parent,x,inventory) \]

- The second modification corresponds to the **portals** (or inter-layers edges) in LIG. With them, we allow JPS to have access to different item-layers and,
Finally, the third modification represents the fact that when we search on a new item-layer, we should consider that layer as new.

For this reason, we can imagine IA-JPS as the simplest application of JPS to the extended LIG model. In the later chapters, we will exploit the properties of LIG in order to provide several improvements to the standard IA-JPS algorithm.

5.3 Experiments for IA-JPS

5.3.1 The experimental setting

For the experiments, we implemented a version of IA-JPS for the P4 Simulator (Python Path Planning Project\footnote{https://bitbucket.org/ssardina/soft-p4-simulator/branches/}). We took the JPS implementation there and we applied the three modifications. The process required more work on the platform itself than to the algorithm. In fact, we needed to implement a way for the p4 simulator to handle items and conditional obstacles on the map. However, once
the platform was able to understand items, tweaking JPS into IA-JPS just required to extend the search node and to add new conditions to the \textit{jump} procedure as explained in the previous section.

![A screenshot of the p4 simulator user interface in action.](image)

\textbf{Figure 5.3.} A screenshot of the \textit{p4 simulator} user interface in action.

The \textit{p4 simulator} can read maps written in the \textit{Moving AI}\textsuperscript{2} [86] format. Thanks to this, we were able to use the full \textit{Moving AI} benchmark database for our experiments. However, also in this case we needed to extend this format for inventory-aware pathfinding problem instances. We added a new keyword for the \texttt{.map} format, namely \texttt{key} that can be used to specify the position of items and obstacles inside the map.

\begin{verbatim}
    type octile
    height 20
    width 20
    key 16 4 13 10
    key 17 8 1 15
    map

    ...  
\end{verbatim}

The \texttt{key} keyword works according this format.

\begin{verbatim}
    key <item_x> <item_y> <obs1_x> <obs1_y> ... <obsN_x> <obsN_y>
\end{verbatim}

We use a \texttt{key} entry for each item we want on our map. Then we specify a pair representing the item location on the map. Finally, we list all the pairs representing

\footnote{http://movingai.com/benchmarks/}
locations that are blocked unless the item is in the character inventory. This format is not the most general solution and, in fact, is limited to a subset of the IAP problem. In particular, we are assuming that no obstacle can be unlocked by more than one item and that one item can unlock multiple locations. However, how we have seen in the previous chapter, this is not a limitation of the IAP algorithm but rather an easy way to extend MovingAI format in order to accommodate a common IAP scenario.

The $p4$ simulator software is implemented in Python. The absolute value runtime of the algorithms reported are often higher than the one that can be achieved by C++ implementations that would typically be employed in videogames. Moreover, the JPS implementation provided by $p4$ in the version we used was sub-optimal performance-wise. In fact, it failed to achieve the 20x speed-up with respect to $A^*$ of JPS original paper. However, because we are interested in the effect IAP has on JPS, we can still use JPS as a baseline and focus on the differential effects of the various IA-JPS parameters. We are not interested into a comparison with state of the art pathfinding algorithms, but rather on how IAP modifications affect JPS.

All the experiments are run on an Intel i7 3.2GHz machine (on a single core) with 8GB of RAM. Nonetheless, the particular implementation platform does not alter the core findings and conclusions and, of course, one would always resort to the most efficient platform at deployment time.

5.3.2 Experiment description and results

We are now going to present five different experiments for IA-JPS in which we will try to show the validity of the LIG approach, the behavior of the parameters on IA-JPS and, finally, identify the conditions under which IA-JPS is practical for videogame scenarios, e.g., for a small amount of items. In the first three experiments, we used the benchmark consisting of maps from Baldur’s Gate II: Shadows of Amn (BioWare, 2000) that are scaled to $512 \times 512$ nodes. From each of the 75 maps we took 50 pathfinding instances that are known to be realizable and have a length between 150 and 250 steps. In the fourth experiment, instead, we run the experiments on synthetic map procedurally generated in order to provide specific IAP instances. Finally, in experiment 5, we measure the algorithm performance when there is no path available.

In all the experiments, we will assume items to be keys and the conditional obstacles to be locked doors. We think that following a real scenario helps the reader understanding better the problem in exam. However, every time you read about key and door you can replace them with any item and any obstacle fits better your real game scenario.

Experiment 1: Random placement of unnecessary keys over real game maps; analysis per number of keys.

First, we consider the case where regular maps are extended to include keys but not doors, so that the keys are not required (and every key is different one from the another), and we compare to running regular JPS. This experiment aims to evaluate the overhead of IA-JPS over JPS for cases that JPS is optimal for IAP. Effectively, the keys can be seen just as “noise” for the planner.

To extend the standard key-free maps, we added 0 – 100 distinct keys in a random way in each map and run IA-JPS and JPS on those instances. The runtime of the algorithms is reported in Figure 5.4. As expected, there is an overhead for considering keys, even when not required for the final path, which scales well (almost linearly) over large numbers of keys. In particular, the overhead of running
5.3 Experiments for IA-JPS

![Graph showing runtime of algorithms with keys randomly placed on the map.](image)

**Figure 5.4.** Runtime of algorithms with keys randomly placed on the map.

IA-JPS over for the case of having 100 keys on the map is equivalent to running approximately 10 regular pathfinding queries with JPS.

It is important to understand why even huge number of keys such as 100 does not affect the algorithm enough to make it impractical. First, we need to remember that all keys are not required for reaching the destination, this means that search will probably be completed in the lowest layer in LIG. There is no need to explore extensively the upper layers and, therefore, even if such layers are an exponential amount, the algorithm will avoid explore them extensively. Second, runtime is affected only by the keys which are actually taken into account during search. In fact, keys are the only way for the algorithm to have access to the upper layers. As we spread the keys uniformly on the map, not every node containing a key will be explored by the algorithm before reaching the destination. 100 keys are probably a large number for such maps, but it ensures that at least a few of those keys will appear in the random paths we try in this experiment. In the average case, only 1/10th of the keys fall in the search horizon of IA-JPS.

However, this is true only if a path exists and it is not too complex as a pattern in the map. If the destination is unreachable (because of items or because it is physically disconnected from the main map), then IA-JPS is forced into searching every layer, for every possible inventory. The effect is that, if we set an unreachable destination, IA-JPS does not terminate in a reasonable amount of time even for 10 keys. We will explore this case more in details in Experiment 5.

From this experiment, we see that we need to investigate where we will start to see an exponential overhead with respect to the small number of keys that are actually encountered during search in these random scenarios. The next experiment intends to explore this further by placing the keys in a position that are relative to the shortest path solution and, therefore, are necessarily caught by the search algorithm.

**Experiment 2: Placement of unnecessary keys on the path over real game maps; analysis per number of keys.**

Here, we consider again unnecessary keys but, following the discussion in the previous section, we look into the case that they are selectively inserted in three areas:
Figure 5.5. Runtime of algorithms with keys placed on the path.

1. At the beginning of the path (BEG). In this scenario, all the keys are clustered around the starting position of the character.

2. At the end of the path (END). In this scenario, all the keys are clustered around the character’s destination.

3. Distributed evenly on the shortest path (MID).

We assume that one location can be occupied by at most one key, therefore, keys are scattered appropriately in a small area around the starting location (BEG), the destination (END) or around the shortest path. We tested IA-JPS, plain JPS, as well as IA-A*, a straightforward inventory-aware variant of standard A* over LIG.

Figure 5.6. Speedup for the cases of Figure 5.5 compared to IA-A* (log scale).

As argued, the overhead is expected to increase when more keys are reached.
during search. The runtime of IA-JPS compared with that of JPS (for reference) is reported on Figure 5.5, while the speedup of IA-JPS compared to IA-A*, in terms of how much faster is the runtime of IA-JPS, is reported on Figure 5.6. Note that every configuration (BEG, END and MID) is compared with IA-A* in the same configuration, e.g., IA-JPS (BEG) is compared to IA-A* (BEG).

The results validate the expected effect of the placement of unnecessary keys on the runtime performance, with the (BEG) distribution being the worst configuration and the (END) distribution the best. This result can be easily explained by referring to the LIG model. The BEG configuration corresponds to a map that can early spread search focus over multiple layers (because portals are located near the start) while the END configuration represent a model with many portals near the destination. This means that most of the search is confined into the base layer and only near the end we can access above layers (and, because we are near the end, we will find the destination before the search needs to go too deep in the item-layers).

The exponential speedup we observe in the BEG scenario is due to the known speed-up of JPS over A*, but applied over all those conditioned cases of the search space for every combination of keys that is encountered. As the number of these combinations is exponential to the number of keys, the speedup of IA-JPS then is also exponential. In fact, when we explore a smaller number of conditioned cases such as in the END scenario the exponential speedup is less evident (note that Figure 5.6 is in logarithmic scale).

These results validate that the benefits of JPS are carried over in the inventory-driven setting. In particular, in the case of just 4 keys close to the start location, IA-JPS is more than 300 times faster than IA-A* (IA-A* is not able to cope with more than 4 keys).

**Experiment 3: Placement of unnecessary keys on the path over real game maps; analysis per path length.**

![Figure 5.7. Runtime of IA-JPS with keys at the beginning of the path.](image)

In this experiment, we test IA-JPS over the most challenging scenario from the previous analysis, i.e., the BEG scenario, with respect to the path length. We
consider 100 realizable paths with no length restriction per map. We then compute
the optimal path and construct inventory-driven instances where keys are placed in
the beginning of the path. Figure 5.7 shows the runtime of IA-JPS with respect to
plan length for the cases in which 0, 2, 4, and 8 keys are added in the beginning of
the path.

When no keys are added, IA-JPS reduces to regular JPS. This can be used as
reference to see the impact of the number of keys for different path lengths, compared
to non-inventory paths of the same length that are solved with JPS. Observe that
there is an approximately linear increase of the runtime of IA-JPS as the path length
increases. While, as expected, more keys induce more effort to IA-JPS, it is still able
to solve problems with long paths, which would be impractical due to memory and
time constraint for any A* version doing complete node expansions.

Experiment 4: Incremental scenario of necessary keys over synthetic
maps; analysis per number of keys.

![Graph showing runtime performance when keys are necessary.]

**Figure 5.8.** Runtime performance when keys are necessary.

Here, we focus on the case where keys are actually needed in order to reach
destination. As there are no benchmark maps for this setting, we built synthetic
512 × 512 maps of two types starting from completely empty maps. Then, for each
number of necessary keys ranging from 0 to 10, we report the average runtime of
IA-JPS over 200 pathfinding instances.

In the first type of maps, we build artificial rooms that separate the empty map
from left to right by putting vertical walls and a door between adjacent rooms. We
impose a “sequential” traversal of the rooms in the sense that door $d_i$ cannot be
reached without crossing door $d_{i-1}$, and key for door $d_i$ can always be found in
the map region before $d_i$. The destination is behind the last door (Figure 5.10a). IA-JPS’s performance is shown in Figure 5.8.

For the second type, we impose a “detour” behavior. For $n$ keys, we start from
the destination, build a “room” around it, and then create a random entry point
with a door $d_n$. We then chose a random point in the map, place key $k_n$ for door $d_n$,
5.3 Experiments for IA-JPS

Figure 5.9. Runtime performance when there is no possible path.

Figure 5.10. An example of sequential map (a) and detour map (b).

and repeat the process now with key $k_n$ (until all keys and doors are configured). Unlike the first case, here doors are all reachable by the agent from the starting state, but solutions require the agent to go back and forth along all the map as the key for door $d_i$ is closed in the room blocked by door $d_{i-1}$ and so on (Figure 5.10b).

IA-JPS’s performance is shown in Figure 5.8.

The two map typologies provide quite different behaviors. In sequential maps, each new item increases extensively the available search. A character with no items will be only able to move inside the first region (representing around 20%-30% of the whole map). Once the character gets the first item, it can move in the previous region plus a new slice of the map, and so on. On the other end, in a detour map, the search space remains almost the same for the full search. In fact, a character with no item can still search all the space outside the small rooms.

Also, the expected average path length differs noticeably between the two map typologies. In the “detour” scenario the average path length is larger than in the
“sequential” scenario as well as the other experiments by a factor 3 (450-500 steps on average). In fact, in this scenario the character has to go from room to room, traversing multiple times the full width of the map. The placement of rooms is done randomly and we only place a small number of rooms, the whole area of the map is not necessarily covered though, and this is why the average path length is not larger than the map dimensions.

Observe also that the runtime of IA-JPS for 10 keys in the “sequential” scenario is comparable to the runtime of IA-JPS in Experiment 2 with 10 unnecessary keys in the challenging BEG scenario. Finally, note that other approaches, including JPS, cannot be applied to find a path here as they will return no solution.

Experiment 5: Key performance overhead for unreachable destinations.

Here, we test IA-JPS over instances that are not solvable, which in fact identifies the overall worst case scenario (as all the search space is explored), but also provides an upper-bound for the worst case when keys are needed, as discussed in the previous section. We produced 200 paths for keys from 1 to 10 over the synthetic maps of Exp. 4 but we surrounded the destination by walls. The results, reported in Figure 5.9, confirm a higher runtime than all the other experiments. For 10 keys, the runtime is approximately 7 times more than the BEG scenario, while it starts to show the inherent exponential nature of inventory-driven pathfinding. Consider though that due to the symmetry-breaking benefits of IA-JPS, reasoning about this scenario is nonetheless possible, while it is not feasible for IA-A* to even handle beyond 4 keys.

Whenever a path is unreachable, the algorithm’s performance gets worse by an order of magnitude. This is the main limitation of IA-JPS. Even if IA-JPS is able to prune away a great part of the search space, as soon as a path is unreachable, it is forced to explore the full exponentially LIG enlarged search space. In the next chapter, we will propose additional pruning techniques that are able to mitigate this problem.

General-purpose planning

Finally, it is possible to solve inventory-driven pathfinding by using general action-based planning [27]. Such techniques in fact go beyond what IA-JPS can handle, as they can mix path planning with more general dynamic reasoning, such as delivering packages or solving puzzles. However, the performance of such general approaches is not competitive for the particular case of inventory-driven pathfinding. We performed experiments with state-of-the-art planners FF [42] and LAMA [79] on inventory-free maps that confirm this. FF could only parse maps of size up to 120 × 120, and took a couple of seconds to synthesize paths of length below 100. LAMA on the other hand was able to parse maps of size 512 × 512 (as those used in our experiments with IA-JPS), but could not solve many of those maps in several minutes. It is important to remark that this is hardly surprising, and not even a fair comparison, since such planners rely on domain-independent heuristics and are meant to be used when known domain heuristics are not available.

5.4 Conclusion

We can summarize the main results for IA-JPS in the following points:

- IA-JPS performance is equivalent to JPS when there are no items on the map.
• **IA-JPS** performance depends on the position of the items. **IA-JPS** search time is much worse if many items are clustered around the starting location than if the same items are clustered around the destination.

• In the worse case, **IA-JPS** still has practical performance for 4-5 items on the map, whenever a solution exists.

• **IA-JPS** main issue in practice is when no solution can be found. In this case, the algorithm must search the map multiple times for each item combination.

In conclusion, the work on **IA-JPS** shows that this kind of **IAP** solution is promising. In the following chapter, we will show how to optimize the algorithm to increase the item limit and, more important, a way to greatly reduce the exponential explosion of the case in which no solution can be found.
Chapter 6

Optimization: Filtering the Search Space

In the previous chapter we presented a minimal but practical implementation of an Inventory-Aware Pathfinding (IAP) algorithm. As we said before, IA-JPS can be seen as the application of Jump-Point Search JPS to the LIG model of the original map. It is clear, however, that in IA-JPS we did not exploit any of the properties of the LIG space and, in fact, when the search moves from a layer to another the algorithm simply starts a complete new search from scratch. This suggests us that there is a lot of space for optimization.

In this chapter, we will explore this direction. In particular, we will look at some techniques for pruning the search space on the upper level of the LIG graph.

6.1 Optimal filtering in Monotonic IAP

As we discussed in Chapter 4, the IAP problem we are investigating has monotonic item effects and, therefore, there is no item that, once picked, reduces the number of locations that the character can visit. To understand better how we can use this to filter out nodes from the search space, consider in the open list two nodes \( n \) and \( m \) corresponding to the same physical location on the map \( n \downarrow = m \downarrow \) but belonging to two different layers \( L_{I_n} \) and \( L_{I_m} \).

Because the two nodes represent the same location, they represent two potential paths whose only difference is the amount of collected items in the previous steps. At this point, we can ask ourselves how the two nodes are related and, in particular, if we can arbitrarily drop one of them or we are forced to continue to search on both layers.

The first thing to consider is whether \( L_{I_n} < L_{I_m} \) (that is, the inventory on the \( n \) layer, \( I_n \), is a subset of the inventory in the \( m \) layer, \( I_m \)), \( L_{I_n} > L_{I_m} \) or if there is no relation between the two layers.\(^1\)

If an ordering relation exists between the two layers (for instance \( L_{I_n} < L_{I_m} \)), then we know from Proposition 4.5 that any location that can be reached on the layer \( L_{I_n} \), can also be reached in the layer \( L_{I_m} \) but the last one can potentially reach more locations (and thus be more promising for looking for the goal). It is easy to see that, as the heuristic function does not change from layer to layer, two different search instances starting from \( n \) and \( m \) will expand the exact same nodes for each

\(^1\)See Section 4.7.1 for more details on the partial ordering relation between layers in the LIG model.
free state in the layer of \( n \). However, if the goal is in one of the states that can be reached only in the layer of \( m \) then only that search instance will find a solution.

From this we may be tempted to assume that the layer ordering, namely the subset relation between the inventories collected in the different nodes, is a sufficient condition to filter out \( n \) in favor of \( m \). After all, \( m \) can find everything \( n \) can find and more, therefore we can discard the duplicated computation represented by continuing to expand \( n \). However, this is still not enough. In fact, imagine that \( m \) represents the uppermost layer, the one in which the agent collected every possible item in the map. It is true that from \( m \) we can reach for sure a solution, however, it is clear that the cost of collecting all the items can be huge and unnecessary if only one of those item is actually needed to reach the goal. As a matter of fact, we need to take into account also the cost for reaching \( n \) and \( m \), the \( g \)-value of \( n \) and \( m \).

Now, we can formalize the filtering in the following proposition.

**Proposition 6.1.** Given two nodes \( n \) and \( m \) belonging to two different layers \( L_i \) and \( L_{i_m} \). A node \( n \) can be filtered (e.g., removed from the open list) without losing optimality, if the following conditions are all true:

1. \( n^\downarrow = m^\downarrow \), that is, the location represented by \( n \) and the location represented by \( m \) are the same (Locality Condition).
2. \( L_{i_n} < L_{i_m} \), that is, the inventory in the node \( n \) is a subset of the inventory in the node \( m \). (Item Subset Condition).
3. \( g(m) \leq g(n) \), that is, the cost for reaching \( m \) is less or equal than the cost of reaching \( n \) (\( g \)-Cost Condition).

Figure 6.1. The example scenario with the obstacle and a door \( D \) that allow the character to shortcut. Because the red path that goes through the door has more items and it can reach faster the dashed tile, the algorithm can stop searching on the other layer (the blue path that goes around the door).

Common situations in which the filtering conditions activate is when a certain item (or set of items) will open a potential shortcut in the map as depicted in
6.2 Sub-optimal filtering

Fig. 6.1. In this example scenario, a character starts from $S$ and needs to reach a goal on the other side of the map. A door $D$ blocks the access to a shortcut. From this, we imagine two possible paths: one for when the character has a key (that we call red layer) and one for when the character does not (blue layer). In general, the algorithm keeps searching on both paths, however we can perform an optimal filtering and discard future search in the blue layer.

To see that, we will focus on the dashed tile where the two paths meet again. The $g$-value for the blue path is $g_{\text{blue}} = 18^2$ while the $g$-value for the red path is $g_{\text{red}} = 7$. Because on the red path the character has one key while on the blue one it has none, this tile satisfies also the Item Subset Condition. Therefore, we can prune the blue node in the open list and, as a consequence, no more future search will be done starting from that node.

What we have done considering only nodes on the paths, in reality, involves every node in the search space inside the current search fringe.

6.2 Sub-optimal filtering

6.2.1 Local $\theta$-filtering

The filtering condition in Proposition 6.1 is strong and effective but it is involved only on particular situations. It is possible, however, to relax some of the constraint leading to a more effective filtering condition at the expense of losing path optimality. Consider the same situation of the previous section with two nodes $n$ and $m$ such that the filtering condition is not because $g(m) > g(n)$. The question who arise is “how much $g(m)$ is greater than $g(n)$?” In fact, depending on the difference between $g(m)$ and $g(n)$ we may decide that it is worth to prune a complete sub-tree in the search space for that amount of additional cost. We can therefore provide a first sub-optimal filtering condition.

**Proposition 6.2.** Given two nodes $n$ and $m$ belonging to two different layers $\mathcal{L}_{l_m}$ and $\mathcal{L}_{l_m}$. A node $n$ can be locally $\theta$-filtered (e.g., removed from the open list), if the following conditions are all true:

1. $n^d = m^d$, that is, the location represented by $n$ and the location represented by $m$ are the same (Locality Condition).
2. $\mathcal{L}_{l_n} < \mathcal{L}_{l_m}$, that is, the inventory in the node $n$ is a subset of the inventory in the node $m$. (Item Subset Condition).
3. $g(m) - g(n) \leq \theta$, that is, the difference between the cost for reaching $m$ and the cost for reaching $n$ is less than a threshold $\theta$ (Relaxed $g$-Cost Condition).

In practice, we decide a certain value $\theta$, and we define that the filtering condition is applicable to $n$ even if reaching $m$ costs more than reaching $n$, provided that this additional cost is less than $\theta$. But, what is the physical meaning of the $\theta$ threshold? The answer is in the following lemma.

**Lemma 6.1.** If a path in the IAP problem exists and $p^*$ is the optimal one, the cost $C$ of a path $\hat{p}$ obtained by an algorithm using the relaxed filtering condition in Proposition 6.2 on the same problem, will be $C(\hat{p}) \leq C(p^*) + k\theta$ where $k$ is the number of items on the map.

\footnote{For simplicity we consider diagonal moves as 1 instead of $\sqrt{2} = 1.41\ldots$.}
We can imagine that each item encountered by the character during search carries with it the additional cost of going away from the optimal path to pick it up. Because we do this detour only if its total cost is less than $\theta$, we have at most $k$ possible detours of cost $\theta$.

However, this maximum sub-optimality is reached only if every Pick action entirely filters the search on the lower layers. Because this does not happen very frequently in practice, the sub-optimality obtained by this filter is much smaller.

![Figure 6.2](image.png)

**Figure 6.2.** An example scenario for local filtering.

To better illustrate local pruning behavior, we will look at a simplified scenario in which the pathfinding graph is small.

Consider a pathfinding map as the one depicted in Figure 6.2. The map is composed by a starting and goal location $s$ and $g$, the locations $a$, $b$, $c$ and $d$, and an item $k$ located in position $e$. We can now go over the search algorithm and filtering mechanism.

1. We expand $s \rightarrow \{a, e_k\}$. The current open list contains is $\{a, e_k\}$ where we mark as with subscript $k$ the nodes in which the character has the item $k$.

2. Now imagine that the next expanded node is $a \rightarrow \{c, b\}$. At this step the open list is $\{e_k, c, b\}$.

3. Now we expand $e_k \rightarrow \{a_k, c_k, d_k\}$. The $a_k$ node will be inserted in the open list only if $D(s, e) + D(k, a) < D(s, a) + \theta$ (g-Cost Condition). In the same way $c_k$ will prune out $c$ if $D(s, e) + D(e, c) < D(s, a) + D(s, c) + \theta$. Assuming these conditions are satisfied, after this step the open list contains $\{b, a_k, c_k, d_k\}$.

4. Note that $b$, a path on the layer with no item, is still in the open list. If $b$ is the next node to be expanded, the search will reach the destination and the final path will be the optimal one even if we used a sub-optimal filtering.

We can observe some effects of local filtering as follows.

- Local Filtering does not always prune all nodes at the lower levels even if a key is in $\theta$ range around the start.
- A layer is completely pruned only if all its nodes in the open list are subsumed by some other node. However, because of locality, this is unlikely to happen.
- The sub-optimality effect of $\theta$ for each layer may not apply if a “path” from a lower level reaches the goal first.

Therefore, we expect that the total sub-optimality for Local Filtering to be quite small on average but, at the same time, filtering may not be as effective as expected in many cases.
6.2.2 Fringe (global) $\theta$-filtering

The effect of local $\theta$-filtering is clearer only when the search reaches the destination at the layer where a key is found. In any case, even if we filter a node $n$ in a layer $\mathcal{L}_I$ in favor to a node $m$, this does not mean that we stop searching in the layer $\mathcal{L}_I$. In fact, alternative search directions in $\mathcal{L}_I$ will still be open in case the overall cost of searching from $m$ will go over the threshold.

A stronger filtering approach can be obtained by removing the “Locality Condition” from the filtering condition.

**Proposition 6.3.** Given two nodes $n$ and $m$ belonging to two different layers $\mathcal{L}_{I_n}$ and $\mathcal{L}_{I_m}$. A node $n$ can be globally $\theta$-filtered (e.g., removed from the open list), if the following conditions are all true:

1. $\mathcal{L}_{I_n} < \mathcal{L}_{I_m}$, that is, the inventory in node $n$ is a subset of inventory in the node $m$. (Item Subset Condition).

2. $g(m) - g(n) \leq \theta$, that is, the difference between the cost for reaching $m$ and the cost for reaching $n$ is less than a threshold $\theta$ (Relaxed Reach Cost Condition).

In practice, this filtering acts as an attractor toward each item in a certain radius. In other words, if a candidate path passes near an item, then the algorithm prefers to reach the item and start searching again from that position.

![Figure 6.3](image.png)

**Figure 6.3.** A visualization of for global $\theta$-filtering. The red inner circle has radius $g(k) - \theta$ and represents the safe zone for the lowest level. The $\theta$ variable is a way to reduce the safe zone and, therefore, to control how much pruning the algorithm will do. Every node on the open list that is outside the red inner circle and belongs to a lower layer than $k$ is automatically pruned away when the agent takes the item.

Understanding this behavior is not straightforward, therefore we will proceed step by step. We consider the same map as in local filtering but this time we apply fringe filtering. We focus our attention to what happen when we add $e_k$ to the open list. As fringe filtering has no locality condition, we can compute the $g$-Cost Condition for it.

$$g(k) - g(n) \leq \theta$$

that can be rewritten as
This implies that global filtering prunes any node in the fringe that is on a lower level and outside a circle centered in s with radius $g(k) - \theta$.\(^3\) This, in practice stops most of the search effort in the lower level, especially if an item is found near the start. The item location, becomes the only way to continue the search and, therefore, we obtain the final effect of a character taking a key just because it is “near enough”.

From this, we expect fringe pruning sub-optimality to be much more evident than the local pruning case. At the same time, we should see a much more aggressive pruning.

### 6.3 Implementation of filtering in IA-JPS: F-IA-JPS

Implementing the filtering algorithm on IA-A* is a straightforward operation. The first step is to implement a function that is able to test nodes filtering eligibility. We can see two functions implementing filtering test in Algorithm 22. The first one, checks if nodeA is better than nodeB and, therefore, allowing nodeB to pruned. The second one, instead, applies the first function to every node in the openlist but also on the closedlist. In fact, when we want to prune a node we need to check if there is already a better node in the open list, but also, that we did not already expand a better node in the past.

This check can be costly if we do not use some additional care in the way we store the open and closed list. In our implementation, we carry out the following optimizations:

1. We combined the open and closed list in a single data structure. This data structure provides hash table that can be used as an index to query a node using its location independently from its location in the open or closed list.

2. Given three nodes a, b, and c, if a is better than b and b is better then c, then a is better than c. We use this transitive property on the IsBetterThan function for reducing the number of comparison in the closed list. We do this by comparing a new node only with the strongest node in each location. This is implemented by sorting the bucket list in the hash table at point 1.

After that, we need to perform the actual filtering. Algorithm 23 shows the section of the full search algorithm in which a new node is pushed in the open list. First, we check if there exists a better node in the open or closed list using the ThereIsABetterNodeThan function described above. If this node exists, then we are already expanding a more promising route and we can skip (prune) the current node.

If there is no better node, then, it is possible that the new node is better than some other node in the open list (or descendants of nodes in the closed list). Consequently, we need to find these nodes and prune them. The FindSubsumedBy function scans the open list looking for any node that is worse (is subsumed) by the new incoming node. Note that, thanks to the optimization above to the open list, we can focus only on the nodes in the current location, if any, and complete this check in constant time most of the times.

Finally, we can add the new node to the open list and continue the algorithm to the next iteration.

\(^3\)Because we talk about the $g$ value, if there are obstacles in the circle the real shape is stretched accordingly. However, we can ignore this detail for now.
6.3 Implementation of filtering in IA-JPS: F-IA-JPS

Algorithm 22 n Is Better Than m

function IsBetterThan(nodeA, nodeB, θ)
moreKeys ← nodeB.keys ⊂ nodeA.keys
costLess ← nodeA.g ≤ nodeB.g + θ
return moreKeys and costLess

function ThereIsABetterNodeThan(nodeA, θ)
for each n ∈ openlist and closedlist do
    if n↓ = nodeA↓ and IsBetterThan(n, nodeA, θ) then
        return True
    return False

Algorithm 23 Filtering on Push

if ThereIsABetterNodeThan(n) then
    closedlist.append(n)
else
    subsumed ← FindSubsumedBy(n, openlist)
    SetAllVisited(subsumed)
    if DescendantPruningIsActive then
        subsumed ← FindSubsumedBy(n, closedlist)
        for cNode ∈ subsumed do
            DescendantPruning(cNode)
    openlist.push(n)

6.3.1 Descendant Pruning

If a subsumed node is in the open list, pruning this node is easy: we just remove the subsumed node from the open list. However, if the subsumed node is in the closed list, things are more complicated. In fact, we cannot remove the node from the closed list; a node in the closed list has been already expanded and its effect on the search algorithm has already been achieved. Instead, in order to have a real pruning, we need to find the descendant of the subsumed node that are still in the open list and remove them.

This task is called descendant pruning and it is implemented by tracking the ancestors for each node in the open list. Starting from the subsumed node in the closed list, we fast-forward to all its descendant and we mark them for pruning.

Implementing the descendant pruning efficiently requires several modifications to the search algorithm, and in some cases the benefits obtained from it are comparable to the additional cost of the operation. Moreover, it has no effect in fringe filtering because the filtering algorithm can directly act on the fringe of the search space. For this reason, its implementation should be weighed on the specific application scenario.

6.3.2 IA-JPS specific implementation problems

Reasoning about the filtering algorithm for IA-A* is helpful for understanding the main aspects of pruning in an inventory-aware problem. However, in IA-JPS, some additional problems arise from the JPS structure of jump-points. The JPS algorithm, in fact, adds in the open-list only a discrete pattern of jump-points. Such jump-points often do not overlap in between layers and, therefore, local filtering is usually highly ineffective.

Jump-point locations depend on three parameters: starting location, goal location and the map topology. When expanding jump-points on different layers, even if the goal location and the map are the same, the starting position is different: it may be
the pathfinding start location or the portal location that moved the search from a layer to another. Image 6.4 shows an example of this. As you can see, the jump-point expansion for the layer without the key (the red path) has a slight offset from the path with the key (the blue one). As a result, we end with two not-overlapping forced neighbors.

When we apply local filtering, the probability of satisfying the Locality Condition is very low. To solve this problem, we need to match non-overlapping jump points. A simple solution is to decide a radius $r$ and look for a matching jump-point in the $(r + 1) \times (r + 1)$ square centered on the node. With this, we can assume that all the jump-points in the square are somehow related to the testing node.

Obviously, the value $r$ adds to $\theta$ and, therefore, it is not possible to perform optimal filtering. Moreover, this adds a small overhead to the algorithm performance because the search algorithm needs to perform a small search around the node instead of looking only in the node position. However, this overhead is negligible for small values of $r$ that, as we will see, are usually enough for filtering in IA-JPS.

6.4 Experiments with F-IA-JPS

Now we proceed to show some experimental results for F-IA-JPS with threshold $\theta$ for local and fringe filtering. To that end, we developed a benchmark on top of python-based P4\textsuperscript{4} path planning simulator [58]. As the previous chapter, we started from a preexistent implementation of JPS and we proceeded to modify it in order to accommodate F-IA-JPS modifications. The algorithm and the benchmark suite are written in Python and the actual implementation of JPS is not optimal. For this reason, we chose to emphasize expanded nodes instead of CPU time as a better way to represent algorithm performance in a language/machine independent way.

However, this brings some problem when evaluating filtered-enabled algorithms. In fact, in this case there is an additional computation that makes each node potentially more costly. Therefore, we need to consider the performance cost of looking in the closed and open list for nodes to prune. In general, we discovered that, thanks to our indexed open and closed list data structure, this additional cost can be considered proportional to the number of expanded nodes. Table 6.1 shows the average filtering performance. We can look at these values as “additional nodes”. For instance, if a local filtered IA-JPS expands 1000 nodes, we can have a grasp of the real performance by adding a 10% more nodes (1100).

\textsuperscript{4}P4 path planning simulator: \url{https://bitbucket.org/ssardina/soft-p4-sim-core}.
6.4 Experiments with F-IA-JPS

Table 6.1. Summary of the effect of different filtering strategies to the total performance algorithm.

<table>
<thead>
<tr>
<th>Filtering Type</th>
<th>Average Additional Total Filtering Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local (no descendant pruning)</td>
<td>∼10%</td>
</tr>
<tr>
<td>Local (descendant pruning)</td>
<td>∼25%</td>
</tr>
<tr>
<td>Fringe</td>
<td>&lt;10%</td>
</tr>
</tbody>
</table>

The values in the table are expected. Local filtering has the additional cost of an indexed search inside the closed and open list. This needs on average 10% additional time and, in practice, it is consistently around this value. However, if we add descendant pruning, the cost rises quite enough. In fact, we need to add the cost of looking for the descendant of the subsumed node. This is on average 25%, but in reality, there are occasional higher spikes. Moreover, the variance of this value increases greatly with the size of the path because to longer path corresponds longer (and more frequent) searches from closed nodes to their descendant in the fringe.

Finally, fringe pruning is the most efficient one. Mostly, because it does not need descendant pruning as it affects any node in the fringe anyway, and it can perform pruning just on specific nodes, such as the item locations. On the other hand, it has to pass over every node in the open list and this can be more costly as it grows.

Another thing to take into account is the radius $r$ for the overlapping problem of IA-JPS. With radius $r$ we have $(2r + 1)^2$ location calls to the open/closed list data structure because we need to look for every node in the $(2r + 1) \times (2r + 1)$ square of interest. This overhead can be usually ignored for small radius values but it rapidly increases and can quickly become a problem. In practice, we rarely need a radius greater than 2, however, if it is needed, we suggest using a different approach. For instance, you can organize nodes into macro-nodes similar to how HPA* does for building the higher-levels. Then we can trigger locality just for macro-nodes and optimize the closed/open data structure for obtaining all the nodes into a macro-node in constant time. Note that fringe filtering does not need a radius, therefore this problem does not apply to it.

Having said this, we still think that focusing on expanded nodes for evaluating these filtering algorithms is the preferable way to go. The behavior with respect to the expanded nodes highlights the intrinsic value of the filtering algorithm that is independent from implementation details such as implementation language, data structures, and their future improvement.

6.4.1 Experiment Parameters

Before we go to the experimental results, we look into the parameters of the experiments.

In local F-IA-JPS we chose $r = 2$, as preliminary experiments showed this is able to handle well the case of non-overlapping jump-points that we identified in Section 6.3.2. Also, based on our analysis, it is illustrative to see threshold values for $\theta$ equal to 0, 50, 100 and 200. Finally, for the first experiment we used 4 indicative maps taken from the Moving AI Benchmark [86], namely “AR0011SR”, “AR0012SR”, “AR0602SR”, and “AR0013SR” (Figure 6.5) and additional maps from the same benchmark for the other experiments.
6. Optimization: Filtering the Search Space

6.4.2 Experiment 1: No-Path Scenario

In this first experiment we put ourselves in the same IAP setting with keys and locked doors of the IA-JPS experiments of Section 5.3. We start from where IA-JPS fails, the case in which no path is possible.

As we said in the previous chapter, IA-JPS performance is practical up to a certain amount of keys, however, if no path can be found IA-JPS is forced into exploring every possible exponentially exploded node. If a map originally contains \( m \) free states and we have an IAP problem with \( n \) items located in the map, IA-JPS will be exploring all the \( m^n \) states in the \( \text{LIG} \) space. This is a serious problem even for 3 items. Addressing this case is the number one priority.

Therefore, we tested various combinations of filtering strategies and \( \theta \) values running several pathfinding queries from a random starting position to an unreachable destination. In Fig. 6.6 we see that (regular) IA-JPS is clearly exponential in the number of keys, because, when there is no path the algorithm is forced to search the full search space. The optimal version of F-IA-JPS is still exponential with a small improvement. This can be explained by the overlapping problem of IA-JPS nodes. In fact, optimal filtering has a small effect in IA-JPS (while it is much more effective in IA-A*). The fact that nodes are missing other nodes on lower levels makes that very few nodes can be optimally pruned from the search space.

Allowing for suboptimal solutions, though, provides a wide range of speedup factors. The two groups of suboptimal algorithms, namely local-filtering F-IA-JPS and fringe-filtering F-IA-JPS, perform much better with a much slower exponential behavior, that is almost linear in the range of 0-10 keys for fringe-pruning F-IA-JPS.
6.4 Experiments with F-IA-JPS

However, the capabilities of fringe-pruning F-IA-JPS in the no-path scenario are not enough to choose it as the best strategy. In fact, even an algorithm that randomly prunes away an exponential number of nodes can perform linearly in this scenario even if it will fail in the case in which the path exists. For this reason, we need to look at the next experiment.

![Figure 6.6. F-IA-JPS over scenarios that no path exists.](image)

### 6.4.3 Experiment 2: Not required items

For evaluating a filtering strategy, we evidently need to show the number of expanded nodes in the case a path exists. As in this case the whole search space would not be explored, for higher variety on the results we used the 4 maps shown in Figure 6.5 plus 16 additional maps from the Baldur’s Gate II: Shadows of Amn (BioWare, 2000) database of the MovingAI benchmark [86]. We also used some of the search queries in the .scen file that comes along with the map files. In this way, we can compare our results with the preexistent data of the query, such as the optimal path length.

The number of expanded nodes for each variation are shown in Fig. 6.7. The first thing one observes is that the fringe-pruning strategy expands consistently more nodes than optimal filtering, but also more than the traditional IA-JPS. This can be easily explained by the greater suboptimality of fringe-pruning strategy with respect to the local-filtering strategy (as we will see in the following experiment). The basic idea is that the more we increase the sub-optimality, the more items the algorithm will find around the map. This, in turn, increases the number of layers the algorithm needs to explore. Moreover, the more items the algorithm finds, more the sub-optimality increases because the search is “driven into” the new items as the fringe-pruning strategy works.

However, note that the increase of expanded nodes for the fringe-pruning suboptimal variants compared to the optimal F-IA-JPS is relatively small in actual numbers over the cases when paths exist. This is because the challenging cases with high complexity are those that no path exists. In particular, note that in Fig. 6.7 the expanded nodes are in the order of hundreds while in Fig. 6.6 are in the order of tens of thousands.
We observe also that in the two graphs we see, for a small number of keys, F-IA-JPS is comparable to IA-JPS, hence having a practical runtime for use in a real videogame setting. Nonetheless, in general there may be many keys on the map that are not useful for a particular pathfinding query but bring the complexity up anyway. With the preprocessing step, we will introduce in the next chapter, we can address this by identifying a small subset of keys that are useful for each pathfinding query, and offer F-IA-JPS approaches that are practical over all scenarios.

As in many cases, we face a trade-off between choosing an algorithm that can save a lot of computational time when we face a no-path scenario but it can show large sub-optimality in the general case; or an algorithm that is consistently faster than standard IA-JPS but has worst performance in the no-path scenario (but, however, still much better than standard InvJPS).

![Figure 6.7. F-IA-JPS over scenarios that keys are not necessary for the path.](image)

### 6.4.4 Experiment 3: Filtering Sub-Optimality

Finally, it is important to measure the degree of suboptimality of the algorithms. In order to do this, we used the same maps of Experiment 2, regular maps with the same additional not required keys. Regular IA-JPS would be able to find the shortest paths that do not use keys. On the other hand, F-IA-JPS with threshold $\theta$ greater than zero favors detours for picking up keys, and the paths found are expected to be suboptimal (i.e., longer) due to these detours.

The results show an evident increase in the path length for fringe-pruning F-IA-JPS with a maximum of a 30% longer path when there are 50 unnecessary keys on the map, compared to optimal F-IA-JPS. On the other hand, the local variant shows a small 3-5% increase (Table 6.2).

Both results were expected. As we discussed above, local filtering helps a lot in pruning unnecessary search nodes, however, the sub-optimality condition triggers only in few cases. In fact, because not every path in a lower layer is pruned, there is always the possibility that a path (closer to the optimal one) is found on that level.

On the other hand, we have seen how fringe pruning items can attract the search effort. This generates a behavior in which the character tends to pick a key just because it is in a “reasonable radius”. This behavior can be less believable or more believable depending on the particular map: we can imagine that in a intricate map...
is reasonable for a character to collect many items at cost of a longer path (priority on collecting), but in a very open map probably it is better to try to reach the destination first (priority on speed).

Table 6.2. The sub-optimality ratio (defined as computed path length over optimal path length) for the three different sub-optimal filtering strategies.

<table>
<thead>
<tr>
<th>Filtering Type</th>
<th>Sub-Optimality Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local (no descendant pruning)</td>
<td>~5%</td>
</tr>
<tr>
<td>Local (descendant pruning)</td>
<td>~3%</td>
</tr>
<tr>
<td>Fringe</td>
<td>~33%</td>
</tr>
</tbody>
</table>
Chapter 7

Optimization: Map Preprocessing

In the previous chapter we tried to solve the problem of searching in the large Inventory-Aware space by filtering out parts of the search space on the basis of the properties of the LIG space. However, F-IA-JPS, even with some aggressive filtering mechanism is still problematic on the “no-path” case. Moreover, there is a property of the IA-JPS that leaves us unsatisfied and that it is not addressed at all by F-IA-JPS. In fact, the algorithm is still very sensitive to “not-required” items.

Both problems are strictly related, after all in the no-path scenario every item is not required. In the example we used until now, discarding not required items is easy, we just need to remove all items that are not connected to an obstacle. However, in the general case this problem is more complex. First, some keys may be not-required for reaching some locations but required for other locations. This definition of not-required is different from the one we used until now, but if we limit our attention to a single IAP pathfinding query, then we can see a lot of analogies.

Second, we may try to test for unreachable path first (for instance, by assuming all obstacles removed and using standard pathfinding), but unfortunately a path can be unreachable because of some unsatisfiable item-obstacle constraint (e.g., a dependency cycle like $k_1$ is behind $d_2$ but $k_2$ is behind $k_1$) and we cannot catch this by avoiding items and obstacles.

Fortunately, we can significantly improve on these problems by using a combination of hierarchical and preprocessing techniques. In this section, we introduce a preprocessing method that aims to help the IAP search procedure focus on a promising subset of items in the map and disregard the rest. In practice, given the set of all the items in the map $K$, we can find a subset $\hat{K} \subset K$ that is smaller than $K$ but is still enough for reaching the goal. The effect of this can be huge. As the search complexity is intrinsically exponential to the number of items on the map, disregarding even a small amount of items can significantly improve overall performance.

The proposed solution is capable of identifying unreachable locations in constant time, and identifying other special conditions (such as cyclic dependencies) even before the real pathfinding search starts. The preprocessing method for this task is divided in two steps:

1. An offline preprocessing step that generates a logical description of the connections between regions and doors on the map.

2. An online pre-search step that, given the starting and destination position
and the position of every item, can generate a subset of necessary (but, in
general, not sufficient) items for the specific path in question.

Unfortunately, this has an associated cost. In general, the algorithm may lose
its completeness, that is, the ability to find a path if one exists. However, in many
practical cases, it is possible to pay more computational time in the online pre-search
in order to obtain a subset $\hat{K}$ that is able to keep the pathfinding completeness.

As we will see, there are different strategies for generating such items subset
offering a trade-off between completeness and efficiency. Also, we want to highlight
that the preprocessing method is independent from the actual search method and
can be used with any inventory-aware pathfinding approach. However, here we will
show results for the F-IAP-JPS variants.

7.1 The IAP preprocessing algorithm

7.1.1 Step 1: The offline preprocessing

The goal of the offline preprocessing step is to analyze the game map and extract
information on the different regions that are connected by the obstacles. This is
crucial for the algorithm because it allows us to reason on the high-level structure of
the item-obstacle relation. In particular, using this information we will be able to:

1. Understand how different regions of the map are connected through the different
obstacles.

2. Understand how obstacles are related with each other. For instance, we can
see that, starting from a region certain obstacles can be encountered only if
we remove a specific obstacle first.

3. Understand how different map regions are connected. If we ask for a path
between two disconnected regions, then we can return a failure without even
starting with the search.

The algorithm takes as input the map representation annotated with the location
of every obstacle and generates two different outputs. The first one is the connected
areas map ($L_M$), a copy of the map in which every location is labeled with an
integer number and with the property that $\text{label}(x)$ is equal to $\text{label}(x')$ if and only
if it is possible to find a path between locations $x$ and $x'$ without using any door
(Figure 7.1a). In this way, we can easily identify the region for each location. If a
query is between two locations in the same area, we can simply limit our IAP search
to that single region and set $\hat{K}$ to the set of items in the region.$^1$

The second output is an obstacle connectivity graph $C$ in which each vertex is
a label in $L_M$ and each edge is a door connecting $l_i$ with $l_j$ annotated with the
connecting door $d$. $^2$ This connectivity graph (Figure 7.1b) is the key for performing
reasoning on the structure of the pathfinding problem seen from the item-obstacle
and topological structure. Using this graph, we can answer questions such as “which
obstacle I may encounter if I move from $s$ to $t$?” A path in this graph corresponds

$^1$You may ask why we still need IAP if we know that a start and destination are in the same
region (and therefore, no obstacle is in the way). The reason is that it is still possible that an item
can open a shortcut (see Figure 6.1 for an example) and therefore we need to search for it.

$^2$To be precise, $C$ is in fact a multi-graph because it is possible that two areas are connected by
multiple obstacles unlocked by different items.
7.1 The IAP preprocessing algorithm

7.1.2 Step 2: the online pre-search

This step is intended to be performed at the beginning of every pathfinding query as a prerequisite for the actual search. It takes as input the starting and goal location, the map, and the preprocessing pair $(\mathcal{L}_M, \mathcal{C})$ for the given map. The output is a set $\hat{K} \subseteq K$ of items that represent a restricted version of the total item $K$ to which inventory-aware search is restricted. In other word, the actual search will consider only keys in $\hat{K}$ and ignore the rest, saving a lot of potential search states.

Figure 7.1. (a) A map and the corresponding connected areas map $\mathcal{L}_M$. All the fixed and triggered obstacles are marked with zero while, any other region with an integer. (b) The connectivity graph $\mathcal{C}$ associated to the map on the left. We can easily spot some interesting information such as that region $A_2$ is unreachable, that there are two doors connecting $A_1$ and $A_3$, and that it is not possible to reach $A_6$ from $A_3$ without passing through $A_5$.

to a possible (but not guaranteed) solution to the item-obstacle problem. We will look much deeper on this problem in the next step.

The labeling map is constructed assuming that all obstacles are locked and running a binary connected area labeling algorithm, a fast parallelizable algorithm originally designed for image segmentation [30]. The connectivity graph, instead, is generated by searching for obstacles along the perimeter of each area and adding a connection edge for each connection that can be found.

There are two interesting properties that we need to note. First, the preprocessing algorithm depends only on the topology of the map and on the position of the obstacles. The position of the items is not required during this step. Therefore, we can recycle this step for different items configurations (for instance, if we want to spread randomly the items on the map). Second, this step can be done completely offline. For each map, and for each obstacle configuration, we can produce a tuple $(\mathcal{L}_M, \mathcal{C})$ at design phase and serialize the output on disk as appropriate.
Depending on the strategy adopted (Section 7.2), $\hat{K}$ may be a “best bet” small set that has high probability to be sufficient, or a larger “complete” set of keys in the sense that if a path exists then it is guaranteed that the destination can be reached with this subset of keys.

In the next section, we will see some concrete strategies for choosing $\hat{K}$ but, for now, we focus at the generic steps that all the strategies share.

Computing the NOS set

First, we use the connectivity graph in order to compute the so-called necessary obstacle sets (NOS): a set of sets of obstacles each of which corresponds to an acyclic path in $C$ from the area of the starting position to the area of destination.

Algorithm 24  Compute the Necessary-Doors-Set

Require: $s$ starting location, $g$ goal, $labels$ the preprocessed label map, $cGraph$ the connectivity graph.

\begin{verbatim}
startLabel ← labels[s]
endLabel ← labels[g]
paths ← []
if startLabel = endLabel then
    add(paths,[])
currentLabel ← endLabel
openPaths = [(null, currentLabel)]
while openPaths is not empty do
    currentPath ← pop(openPaths)
currentLabel ← currentPath.label
    if currentLabel == startLabel then
        append(paths,currentPath)
        continue
    adjacentLabels ← adjacent(cGraph, currentLabel)
    for label, door ∈ adjacentLabels do
        newPath ← insert(currentPath,(label,door))
        append(openPaths,newPath)
return PathToNDS(paths)
\end{verbatim}

Algorithm 24 shows how NOS can be computed. NOS contains high-level plans in terms of which doors the character need to pass in order to reach the destination. For instance, if the empty obstacle-set is in the NOS then the destination can be reached without passing through any obstacle. In the same way, a non-empty obstacle-set denotes that the destination can be reached by going through the obstacles in the obstacle-set. The set of all the possible alternatives, is the NOS set.

This set of obstacle-sets is necessary in the sense that if a path exists then all the obstacle in one of the obstacle-set must be necessarily unlocked reach the destination. However, as the NOS does not take into account the positions of the items, it is easy to see that such a obstacle-set may be not sufficient. For example, reaching obstacle $d$ in the obstacle-set \{d\} option may in reality require that the character passes through another obstacle to pick up the item for $d$. This can be computed at runtime using the keys location by expanding the high-level plan that a doors-set represents, as we will see soon.

Searching for all the acyclic paths in $C$ from starting area to destination is still an exponential task with respect to the number of regions in the high-level abstraction. However, in practice, connectivity graphs in practical maps are much different from the theoretical worst-case scenario. First, the connectivity graph is usually quite small
7.1 The IAP preprocessing algorithm

Figure 7.2. The example shows the NOS set for a pathfinding query from a starting position in area $A_1$ and a destination in area $A_3$. Looking at the connectivity graph we can see that all the possible acyclic paths from $A_1$ to $A_3$ need to pass through obstacle $d_1$ and $d_2$ or $d_2$ and $d_3$. Looking for the items unlocking one of these possible obstacles is necessary but, clearly not sufficient.

compared to the pathfinding graph. The number of areas are directly connected to the number of obstacles and, in turn, the number of obstacles usually depend on the number of real items. Second, the branching factor for a real-world map is usually small. A map can contain few areas that connect to 10 different areas, but for the greatest part, an area has, at most, an entrance and an exit. Finally, if needed, part of this computation can be done offline. For instance, we may save the NOS set for many common or difficult area to area paths.

Computing the NKS set

From NOS, we can compute the corresponding necessary item sets (NKS) by substituting each door in every set in NOS with a key that is required to open it and removing possible duplicates.

At this point we may decide to stop or to iterate the process. That is, we compute a NOS set for each start to item $k$ and we find the corresponding NKS (Algorithm 25). We will look better into this problem in the next section. For now, we just assume that there is a way for computing $\hat{K}$ from NKS. The most trivial one is to merge together all item-sets in NKS, or to choose just one set in NKS as $\hat{K}$. We call $\hat{K}$ the candidate item set.

7.1.3 Step 3: IAP search with hints

Once we have $\hat{K}$, we can use it as the item set $K$ for the IAP algorithm that will search for a path. This requires a small modification to the algorithm: we need to ignore any item on the map that is not contained in $\hat{K}$.

Note that we put no requirement to the algorithm, therefore it is possible to use IA-A*, IA-JPS or F-IA-JPS at will. However, depending on how we chose $\hat{K}$, using an optimal algorithm as final step for this preprocessing algorithm may not return
Algorithm 25 The Recursive Iteration

Require: \(s\) starting location, \(g\) goal, \(map\) the map, \(labels\) the preprocessed label map, \(cGraph\) the connectivity graph.

\[
K \leftarrow \text{NecessaryKeys}(s,g,labels,cGraph,\text{map})
\]

for \(i \leftarrow \{0, n - 1\}\) do

\[
\text{new}\, K \leftarrow K
\]

for \(k \in K\) do

\[
\text{tmp} \leftarrow \text{NecessaryKeys}(s,k,labels,cGraph,\text{map})
\]

\[
\text{new}\, K \leftarrow \text{union}(\text{new}\, K, \text{tmp})
\]

if \(K = \text{new}\, K\) then

break

else

\[
K \leftarrow \text{new}\, K
\]

return \(K\)

---

Figure 7.3. The Hasse Diagram of LIG layers for a IAP problem with \(K = \{a, b, c\}\). We imagine that preprocessing returns \(\hat{K} = \{b, c\}\). The parts highlighted in red are the only ones in which the search algorithm needs to go.

... an optimal path.\(^3\) In fact, it is possible that \(\hat{K}\) does not contain all the items needed for the optimal path, but contains enough items for reaching the destination anyway. Again, a good example involves the shortcut scenario. We may have decided to use the empty set as candidate set, and in fact, we can find a path using no items at all. However, if we add the shortcut item we can find a faster route.

7.2 Strategies for choosing \(\hat{K}\)

We now specify some strategies for generating the candidate item set \(\hat{K}\). Similar to the previous discussion for obstacles, under conditions we can also show that such set is “complete” in the sense that, using \(\hat{K}\), F-IA-JPS (or any other IA algorithm) can always find a solution if a path exists.

---

\(^3\)An optimal algorithm will return the optimal path assuming that only items in \(\hat{K}\) are present. However, this is not the optimal path in the original problem.
7.2 Strategies for choosing $\hat{K}$

**FirstNKS**

This is probably the simplest strategy for choosing $\hat{K}$. It consists in choosing the smallest set in $\text{NKS}$ ignoring all the others. This solution is greedy in the sense that it picks the option that looks most promising in terms of efficiency as it uses the minimal amount of keys. It is expected to have fast runtime as well as to produce suboptimal paths because in general the location of the keys matters for finding optimal paths (e.g., a path with more keys may be optimal because they are near the character).

**AllNKS**

A more cautious approach is to define $\hat{K}$ as the union of all the sets in $\text{NKS}$.

$$\hat{K} = \bigcup_{K_i \in \text{NKS}} K_i$$ (7.1)

It is expected to have slower runtime as it will use more keys but a better performance with respect to optimality as all options (and the corresponding keys) are considered in the search.

**Recursive Application**

Both approaches are incomplete in general: the generated $\hat{K}$ may not be sufficient for reaching the destination because a key may be behind a closed door, as we discussed earlier. Nonetheless, at runtime the locations of keys are available, therefore we can identify which keys are behind doors and which are the required extra keys, and add them to the set $\hat{K}$.

This can be done with a recursive procedure that starts with the set $\hat{K}_0$ generated directly by the NOS using FirstNKS or AllNKS before and at each step generates a set $\hat{K}_{i+1}$ that enlarges $\hat{K}_i$ with keys that allow to reach keys in $K_i$. Algorithm 25 describes this recursive step. We call these algorithms FirstNKS-$n$ and AllNKS-$n$ depending on which base strategy we want to use and using the $n$ parameter to denote the number of recursion steps.

If we define $S$ as one of the above strategies and $\text{nks}(s, g)$ as the function returning the $\text{NKS}$ set for a query from $s$ to $g$, we can define the recursion as:

$$\hat{K}_{n+1} = \hat{K}_n \cup \left( \bigcup_{k \in \hat{K}_n} S(\text{nks}(s, k)) \right)$$ (7.2)

For instance, we look at the same example map in Figure 7.1a and the same path from $A_1$ to $A_4$ as in Figure 7.2. Assume also that $k_i$ is the item that unlocks $d_i$. Now, we can imagine that we are choosing the AllNKS strategy and, therefore, our $\hat{K}_0$ set is composed by $k_1, k_2$ and $k_3$.

Consider that $k_1$ is in $A_1$, $k_2$ is also in $A_1$ and $k_3$ is in $A_5$. We can now apply the first recursion step to find $\hat{K}_1$. First, we find the value of $S(\text{nks}(s, k))$ for all the key in $K_0$. From this we obtain $\{\}$ for $k_1$ ($k_1$ and $s$ are in the same area), $\{\}$ for $k_2$ (as before) and $\{k_1, k_2, k_4\}$ for $k_3$ (because $k_3$ is in $A_5$ and we need to unlock $d_4$ to enter the area).

The item $k_4$ is a new item that was not present in $\hat{K}_0$ before, so we add it to the set and we get $\hat{K}_1 = \{k_1, k_2, k_3, k_4\}$. We can continue again to find the new set
\( \hat{K}_2 \) that, if we imagine the \( k_4 \) is in \( A_3 \), is identical to \( \hat{K}_1 \). Therefore, we can stop iterating.

### 7.3 Recursion and convergence in preprocessing

When we apply recursion to item-set strategy, we can identify two border cases for each approach: one when \( n = 1 \) which we call “optimistic” and one when the recursion step is repeated until a fixed point is reach which we call “recursive”. This leads to four variants, namely FirstNKS-Optimistic, FirstNKS-Recursive, AllNKS-Optimstic, AllNKS-Recursive.

The good news is, that, for every item-object configuration, it is always possible to reach a fixed point in which \( \hat{K}_n = \hat{K}_{n-1} \). We can summarize this property in the following proposition.

**Proposition 7.1.** Given an area labeling map \( L_m \), a connectivity graph \( C \) and the position of each item in the map. Given \( \hat{K}_n \) the set obtaining reiterating any strategy \( n \) times. There is always a number \( m \) after that \( \hat{K}_n = \hat{K}_* \).

This proposition is easy to prove. In fact, we know that the set of every item in the map \( K \) is a finite set and that the recursion algorithm iterates using the union operator for subsets of \( K \); an operation that is clearly monotonic and bounded by \( K \). We start from a subset of \( K \) and we keep merging other subsets of \( K \). In the end, in the worst case, the algorithm will converge to \( K \) itself.

A more interesting lemma is that there are many cases in which the fixed-point set is a strict subset of \( K \). A trivial case is when we ask for a path starting and ending in the same area. In this scenario, the \( NKS \) set is composed by the empty set alone. \( K_0 \) is the empty set and in this case, any other \( K_n \) cannot be different from the empty set (there are no items to iterate over). However, this is not the only case. As a matter of fact, this is the most common case in many game maps and it is where the optimization effect of the preprocessing method is useful.

Another important property is that, if we choose the AllNKS strategy, then the fixed-point set \( \hat{K}_* \) is complete in the sense of the next proposition.

**Proposition 7.2.** If there exists a path for an inventory-aware pathfinding instance, then it can always be reached by an IAP algorithm that uses only the item in \( \hat{K}_* \).

A formal proof for this proposition can be tricky. However, we can observe that when we iterate the recursive algorithm indefinitely, by definition, the resulting set contains every item needed to reach the destination plus every item needed to reach every other item in the set. Note, however, that the resulting set is complete, but it is not guaranteed that the path obtained by using this set will be optimal. In fact, the algorithm will miss eventual shortcut items (items that connects the same area) because they are not strictly required.

This is an important structural property of IAP that allows us to define parametrized classes for the IAP problem. We define a **Class n IAP problem** as a IAP instance that requires at most \( n \) steps with AllNKS strategy to converge to a complete set of items.

This definition is directly linked to the complexity of the IAP problem. Higher IAP problems require more convoluted paths, more items, longer paths and, in the end, more complex searches.

While the formal link between complexity and IAP Complexity Classes is left for future work, we can intuitively see that the notion of IAP classes corresponds to
what a game developer and players perceive as the “complexity” in a IAP problem. Class 0 problems are usually trivial problems in which items are just annoyances that the player have to collect along the way. Class 1 problems require the user to reach a specific location just for collecting an item required to proceed on the initial path. Class 2 problems are more complex, with a location that must be visited in order to reach another location containing required items for the original path. In class 3 and beyond problems is usually hard for human to keep in mind all the different levels of deviations and constraints required to reach the destination.

7.4 Experiments for map preprocessing

We now proceed to evaluate the preprocessing method, in particular the four variants FirstNKS-Optimistic, FirstNKS-Recursive, AllNKS-Optimstic, AllNKS-Recursive, in combination with two filtering methods from the previous chapter, namely optimal
F-IA-JPS, and local F-IA-JPS with $\theta = 100$ that are representative of the space of behaviors.

Once again, we used a map from *Baldur’s Gate II: Shadows of Amn* (BioWare, 2000) from the Moving AI benchmark [86]. One of the main problems with preprocessing experiments is that we want meaningful queries. This means that we want a meaningful map decomposition (where areas represent specific “rooms” of the map) and, most importantly, keys and doors placed such that they are required to reach the destination but, at the same time, they offer a challenge that can be solved by the character. However, if we take a map and we start randomizing both key/doors locations and start and destination points, then there is high probability that the query is not satisfiable. There is simply too much randomness involved. For this reason, we opted for using a single complex map (namely, map AR0602SR) which we manually filled with keys and doors locations. The final map can be seen in Fig. 7.5 where each area is colored with a different color.

We used 14 doors which divide the map in 13 different areas. We manually distributed 14 keys on the map (one for each door) in such a way that every key is reachable starting from the central area. This is ensured by avoiding cyclic requirements. That is, we avoided cycles such as to get key $k_1$ requires to get $k_2$ and reaching $k_2$ requires to get $k_1$. Maintaining this constraint is quite easy. We start from a desired region (in our case the yellow central region) and we list all the regions that can be reached from there in one step. Then we put all the keys for that regions inside this starting region. We then continue in this way by listing all the regions that can be reached in two steps and putting all the keys for that regions any of the 1-step regions or the starting regions. And so on.

Finally, we specified 300 random paths that start from the central area and end in a random location. As we said before, starting from the same area guarantees that there exists a solution for every path because of the way we placed the keys. In Fig. 7.6 we see the average expanded nodes over the set of paths for each combination of approaches.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{ar0602sr.png}
\caption{The map “AR0602SR” used in the experiments for the preprocessing algorithm.}
\end{figure}
7.4 Experiments for map preprocessing

As expected, optimistic approaches expand the least amount of nodes due to being incomplete: FirstNKS-Optimistic and AllNKS-Optimistic show an average of 61% completeness (with small differences in the combined search approach), that is, they return a solution for 61% of the solvable pathfinding instances. However, if they can solve the IAP problem, they return a solution extremely quickly with a number of expanded nodes comparable to not-IAP JPS and a small overhead for the online pre-search step.

There are a few additional interesting outcomes. First, AllNKS-Optimistic performs similarly to FirstNKS-Recursive in terms of expanded nodes, and the latter is complete in this scenario even though not complete in general. Second, the average performance of the preprocessing approaches is almost unaffected by the suboptimal nature of the combined search approach. As the suboptimal behavior in F-IA-JPS is mostly caused by detours for picking up unnecessary keys, with the preprocessing step such detours are avoided since most of the unnecessary keys are removed in advance. This is an important result because it shows that, in some sense, preprocessing can subsume filtering on certain cases.

In Fig. 7.6 we average over all different scenarios, e.g., cases where paths exist or
not, and over different behaviors, e.g., with respect to completeness. In Fig. 7.7 we look closer on the subset of paths where the approaches are complete, in order to compare the same search behavior, and report on the average expanded nodes over the path length. In this way, it is also more clear to see how the performance scales in terms of detours that typically happen in longer paths.

In the graph, we avoided showing the recursive version of the preprocessing strategies because in the subset of paths that can be solved by both recursive and optimistic strategies, they behave exactly the same (e.g., the recursion converges to the same optimistic set). This is another interesting discovery. Another way to look at this is that the recursive strategy can adapt to the complexity of the query. If we implement the recursive strategy and then we use it only on simple scenarios that can be solved also by the respective optimistic versions, we are not losing anything in terms of final performance.

Finally, we verify that the following common border cases are handled by our preprocessing method with a similar or better performance as regular pathfinding:

1. When there are only unnecessary keys the preprocessing method allows to disregard all keys. In this case F-IÅ-JPS would work exactly as JPS.

2. In the case in which there is no path from start to destination by a fixed obstacle (e.g., a wall or door that cannot be opened), the preprocessing method will identify this case and it returns with an answer in the pre-search step. This completely solves the problem we had with standard IA-JPS and F-IÅ-JPS.
Chapter 8

Conclusions and Future Work on IAP

In Chapter 5–Chapter 7 we have seen Inventory-Aware Pathfinding (IAP), a way to enhance pathfinding with a limited subset of planning and pro-active reasoning that it is extremely useful for spatial navigation: objects interaction and conditional obstacles.

We have formalized the problem in a mathematical way and we explored a simple but representative sub-class of General IAP, namely monotonic unbounded IAP (IAP-UI-SU). For this cause, we introduced a basic algorithm based on JPS called IA-JPS, and we proved that it is practical for a small set of items.

Then, we introduced improvements to IA-JPS through pruning of the search space, producing the filtered version F-IA-JPS, both in an optimal and sub-optimal way. We tested the new algorithm and we saw that the algorithm can greatly cut the number of search expanded nodes with little overhead.

Finally, we introduced a more refined hierarchical optimization based on pre-processing and high-level search in order to eliminate from the search space the greatest amount of items as possible (sometimes, even sacrificing the algorithm completeness).

However, we think that there are more possibilities for IAP variants and optimizations. In the following sections, we propose one direction that can be explored in the future.

8.1 Heuristic improvements

As we have seen in the previous chapters, all the experiments have been done using a basic Euclidean Distance Heuristic. Unfortunately, the Euclidean Distance is a poor heuristic in the IAP setting.

The main problem is that it completely ignores items. Search is always directed towards the goals, even if we know that is not possible to solve the problem with the current inventory (for instance, we can obtain this information after the online pre-search step). On the other hand, we cannot directly include items in the heuristic functions because, in order to achieve this, we should know which is the inventory corresponding to the optimal path. But if we knew that, we can avoid IAP entirely as well.

However, there are several improvements over the standard distance heuristic that we can use to speed-up the process.
8. Conclusions and Future Work on IAP

8.1.1 True-Distance Heuristic

A straightforward solution to the heuristic problem is to apply the True-Distance Heuristic similarly to the case of Cooperative A* (Section 2.6). To do that we can proceed as follows:

1. Compute the optimal path from the destination to the starting point assuming that all the not-fixed obstacles in the map are removed; in the monotonic case, this is equivalent to searching for a path with the character having all the possible items. If a path exists, go to Step 2, otherwise we can end with a failure because there is no solution.

2. We use the saved closed-list, in particular the $g$-values of the visited nodes, as the Heuristic Value for the real IAP search.

The idea, in short, is to perform a preliminary search to develop a much more accurate heuristic function. This will not save the real IAP search from exploring in the wrong direction if an item is needed, however, it will make that search much more efficient.

However, this has some drawbacks. If the heuristic is extremely accurate for guiding search to the goal, we can have more problems when such goal is unreachable in the current layer and the search should move in the opposite direction for reaching the right item.

The threshold between these two cases needs to be explored more in detail.

8.1.2 Preprocessing Based Heuristic

The preprocessing step we discussed in Chapter 7 gives much information on the structure of the IAP problem that we can use to remove not promising keys from the map. It appears obvious to use such information for the heuristic function too.

The most important information we can extract from the connectivity graph and from the Necessary Obstacle Sets (NOS) associated to a query is which layer is not promising for a solution. In fact, it is true that the preprocessing information cannot tell us which layer contains the optimal solution, however, it can say which layer contains no solution for sure.

Consider a NOS containing a single set of obstacles $\{d_1, d_3, d_4\}$. From this set, that is in fact called necessary, we can see that the character must necessarily pass $d_1$, $d_3$ and $d_4$. There is no alternative. As a consequence, any item-layer associated with an inventory that does not contain one of the items unlocking those obstacles is for sure not the right layer.

We can use this information to develop a per-layer heuristic function. We will call $\mathcal{K}$ the set of all the inventories in which we know it is impossible to find a solution. Then we can define the following heuristic function:

$$ h(n, \text{goal}, L) = \begin{cases} h(n, \text{goal}) & \text{if } I \notin \mathcal{K} \\ \min_i h(n, i) & \text{otherwise} \end{cases} \quad (8.1) $$

where $h(n, m)$ is the standard heuristic function and $i$ is the closest item not in the inventory. The effect of this is heuristic is to push the search towards an item if the preprocessing function tells us that a give inventory cannot achieve a solution.
Part II

Belief-Aware Pathfinding
The algorithm for Inventory-Aware Pathfinding is not the only way to make pathfinding smarter. There are, in fact, other proactive behaviors that can be integrated into the search phase of pathfinding.

Following a different direction, pathfinding can be extended to allow characters to navigate in an environment where nodes may be in an unknown state. This problem is apparently related to incremental search algorithms such as D* but, in reality, we are concerned with a more high-level problem. We do not want only characters that are able to navigate in an unknown environment, we want characters that are able to reason over their past experience and how they imagine the world state to be. We call this information the character’s beliefs about the world.

The concept of “belief” here is different than the data structure and math formulas of other algorithms designed to efficiently compute a path in an unknown environment. Here, we do not seek efficiency, but the believability so that the player can enjoy the illusion of intelligence the Non-Player Characters provide with their movements. For this reason, beliefs in this case, can guide the characters to even wrong paths, and can be influenced by the non-player characters “emotions” such as fear, hastiness, bravery or the simple desire of choosing the safest route instead of the fastest one.

In most commercial videogames the problem of beliefs is typically avoided for non-player characters, as they access the actual current state of the game map, and have complete information at every frame for most of the artifacts, e.g., doors. However, in some games it is a gameplay requirement for characters to navigate characters under incomplete information about the state of the game map. For instance, the game designer can be interested in characters that can provide to the player the illusion that they are really struggling to navigate the map or to keep track of the changes the player may be making to the environment. As the navigation of each character then is conditioned to the beliefs as defined above, we will use the term “belief-aware pathfinding (BAP)” to refer to such approaches.

9.1 Motivation and examples

Before we move on, we would like to showcase some cases that can be efficiently solved applying a BAP algorithm. We will refer to the game scenarios presented in Section 1 and point out cases for BAP.
Figure 9.1. A screenshot from *XCOM: Enemy Unknown* (Firaxis Games, 2012). Here we can see a classic example of fog of war, when parts of the map are out of sight for the player, they are “darkened” and enemy presence and obstacles state are not shown to the player.

“Navigate under a fog of war”.

In the tactical turn-based game scenario S1, the player may be moving his characters in an incomplete-information BAP scenario: enemies may be hidden by the fog of war, doors may be locked, and there may be other kind of unknown obstacles. In this situation, the player assumes that NPC enemies play according the same rules too, a fact which does not occur often in reality. In games, pathfinding is an omniscient entity that works on the basis of the real geometry of the world and ignores the perception/knowledge of the single character in the map.

Consider for instance the scenario where the player may decide to blow up a wall to form a shortcut to his advantage or lock a door in a common route in order slow down his enemies. In this situation, enemy characters should assume that the wall is still intact and the door is opened until they can perceive this information. Otherwise, the desired effect of slowing down the enemy would be lost and with it also the believability of the character behavior. Ideally, this should work in a per-character basis so each character has a personalized view of the game map. For example, even if one character has perceived the information that the door is locked, the other characters still need to try to use the door (if this is the fastest path) before realizing that it is locked.

“Player personalized paths”.

In the game scenario S2, personalized pathfinding based on the AI companion’s knowledge is a desired feature. In one case, the companion beliefs are used to keep track of the player’s beliefs and offer assistance conditioned to the information that the player possesses.

For example, the AI companion maybe able to suggest a path for helping the player to navigate a big map, similar to some games that display overlays on the game map when the player presses a specific key to guide him to his target (Figure 9.2). As the AI companion shares, in the player experience, the same knowledge of the player, it is expected to suggest only what is reasonable with respect to what the player knows. For instance, if the player never discovered a door, or a shortcut, the character companion should not be aware of them neither and the suggested path
Figure 9.2. The guiding light of Dead Space (EA Redwood Shores, 2008) is probably the most iconic path suggestion mechanism. When the player presses a specific button, an holographic light is projected on the ground to guide the players towards their destination.

should be the one that follows the player’s knowledge. This can avoid the “spoiler effect” that visual hints often have on the player experience.

In another case, the player may ask the AI companion to suggest a path that serves as maximizing the information that the player would get about the map and the hidden or uncertain node states. For instance, it is possible to encode the possibility of player’s ambushes, cover points, escape points, items locations and any other meaningful spots on the map.

9.2 Literature review

Navigation is probably the most important aspects of a character behavior. As stated by early psychology studies, by itself, the way figures move on the screen can communicate their emotions and humanity to the observer [36]. Belief-Aware Pathfinding does not address the movement problem of a character in its entirety, however, it tries to solve the navigational aspect of the movement by providing a way to include the internal beliefs and emotions of a character into the pathfinding search.

Exploring Non-Player character believability is not a new research field. However, defining what we mean for believable behavior for NPCs is not an easy task. A recent answer to this question is given by Hinkkanen, Kurhila and Pasanen in their work for evaluating the believability of Non-Player Characters in games [41].

The proposed framework tries to evaluate NPCs according three different directions: movement, animation and behavior. With movement, the authors address the problem of natural pathfinding, that is, how the NPCs move from one point to another. A believable NPC must be able to move smoothly and must be able to take into account other NPCs during the path. Animation, instead, is referred to the local movements of the character both in terms of mesh animation than in small local movements such obstacle avoidance, interaction with other objects through techniques like procedural animation and inverse kinematic chains. Last, behavior represents the proactive reasoning of the character: what the NPC decide to do, when and how.
This framework proposed an evaluation based on two tables, one for movement and animation, and one for behavior. Each table gives a score between 0 and 10 and the total score for the NPCs in the game is the multiplication of the scores of every individual table. Unfortunately, the parameters for each table are probably too much tailored for First-Person Shooter games to be a definitive answer to the believability problem in general. However, they can give us insights on the basic believability elements of the NPCs.

In this context, our BAP approach offers a possibility to improve the score of NPCs believability in both tables at the same time, by deciding a more suitable path from start to destination on the basis of the information the NPCs can reasonably have, and by providing a more unpredictable and logical human-like behavior.

### 9.3 Definition of the Belief-Aware Pathfinding problem

As we previously stated, Belief-Aware Pathfinding can be informally defined as a pathfinding query on a special map representing the current beliefs state of the querying character. We can imagine a first simple approach to BAP which consists in creating a copy of the map for each character in the game and using it for pathfinding queries. Because the map belongs to a single character, it can be updated only on character perception and, therefore, will represent the accurate state of beliefs of the character.

The above solution however has several problems. First, it requires big amount of duplication. We need to keep, update and maintain a full copy of the map for every character. This is not only a memory problem, in fact, such a large and sparsely updated memory is hard to optimize and will not benefit from caching. Second, we still have no way to represent character uncertainty. The above solution can represent the states in which the character is convinced of the state of a door, but tell us nothing about the cases in which the character is unsure (e.g., about a door that the character has not seen in a long time).

A more refined approach, consists of representing only the important beliefs on a per character basis, leaving the full map as it is. Following this approach, we can see BAP as a special case of reasoning over random graphs. In this case the information about the game map is represented as a connectivity graph with probabilities that denote the degree of belief of the character that a connection is traversable (open). If an edge in the random graph has no degree of belief, we call it fixed edge, otherwise the edge can change during the life of the character and therefore we call it moving edge.

Assuming that the full graph is given, the relevant information that we need to track for every character is the list of moving edges and the degree of belief that a character has for each specific moving edge. This is a huge improvement with respect to the previous solution. In fact, if we provide an index for the moving edges, instead of copying the map for each character, we just need to provide a belief vector $B = (b_1, b_2, \ldots, b_n)$ in which $b_i$ represents the degree of beliefs of the character for the $i$-th moving edge of the map.

With the help of the belief vector, we can define the Belief-Aware Pathfinding problem in the following way.

**Definition 9.1.** Given:

- A pathfinding map representation $G$.
- A starting location $s$ and a goal location $g$ on $G$. 
A belief vector $B$ describing the character’s knowledge over the state of moving edges in $G$.

A Belief-Aware Pathfinding solution is a strategy (or policy) for reaching $g$ from $s$ such that minimizes the expected path length.

As an example, consider a simple pathfinding graph $G$ composed by only 5 locations. The graph, shown in Figure 9.3 is composed by plain edges except for the edge from $b$ to $g$ that is a moving edge. This graph structure in which edges may or not exist with a certain probability $p$ are called random graphs. Each individual possible graph is called a realization of the graph. In our example $G_0$ and $G_1$ are the only two realizations of $G$.

It is fundamental to note that in a random graph talking about the optimal path does not make sense. As we do not know in advance in which realization of $G$ we are, we cannot say that a certain path, as a sequence of edges, can solve the problem as we cannot know if all the edges in the path are available or not before the character starts executing it.

![Figure 9.3. An example random graph. Each edge has unitary cost and the only moving edge $\langle b, g \rangle$ is believed to be open with degree $B_{bg} = 0.7$.](image)

In the scenario in Figure 9.3, the character, starting at $s$, needs to reach $g$. However, it is not sure if the edge between $b$ and $g$ exists or not as it is out of the character’s perception range.

In order to solve this problem, we need take into account every possible realization of the graph and imagine the path of a character on it. Because we only have one moving edge, we have only two possible realizations: $G_0$, in which the edge is closed, and $G_1$ in which the edge is open. In the first realization, there is only a single viable path ($p_0 = \langle s, a, c, g \rangle$) with length 3. In the second realization, instead, the shortest path is the one traversing the moving edge $\langle b, g \rangle$ ($p_1 = \langle s, b, g \rangle$) with length 2. As we know the degree of belief of the character for each realization, we can compute the expected shortest path length (ESPL) for the path between $s$ and $g$ by weighting each shortest path length with the degree of belief of the graph realization in which it is computed. We denote this as $\epsilon(s, g)$ and it is computed as follows in the previous example.

$$\epsilon(s, g) = 2B_{bg} + 3(1 - B_{bg}) = 2.3 \quad (9.1)$$

This value can be seen as the average path length for an omniscient character (i.e., who knows the real state of each moving edge) over multiple realizations of $G$ drawn randomly according the degrees of belief of every edge.

Unfortunately, this gives us little insight on how we can choose a first action from $s$ for our BAP character. There are, in fact, two conflicting aspects. At first glance, $p_1$ is a good candidate as it allows the character to reach the goal faster and it is closer to the theoretical ESPL value. On the other hand, $p_0$ is available
in every realization while there is a 30% chance that $p_1$ will not be available in the real map. Moreover, because the character is able to perceive that an edge is closed only when it reaches a node adjacent to the moving edges, we may want to take into account also the backtracking cost of such event. That is, the cost of going back and reaching the goal through a different path.

For these reasons, in the case of random graph, it is useful to think in terms of the next best action instead of thinking at a sequence of steps from start to destination that the character must blindly follow. We call this solution a strategy which describes the best action the character can take in any location in order to minimize the expected shortest path length from start to destination.¹

In the example above, a character starting from $s$ has 2 possible actions: move to $a$ or move to $b$.

1. We suppose the character decides to move along the path with $a$ as the first step ($p_0$). In this case the cost is the same for every possible realization and, therefore, the expected path length is the same as the real length. In math, $\varepsilon(s, g|a) = 3B_{bg} + 3(1 - B_{bg}) = 3$.

2. Now we imagine the character decides to move along the alternative path ($p_1$). In this case the cost is 2 if the edge is open, however if the edge is closed the character will first reach $b$ (cost 1), and then he needs to find an alternative path $⟨b, s, a, c, g⟩$ with cost 4 (backtracking cost). In math, $\varepsilon(s, g|b) = 2B_{bg} + 5(1 - B_{bg}) = 5 - 3B_{cg}$. Assuming the character beliefs on the $⟨bg⟩$ edge to be equal to 0.7, $\varepsilon(s, g|b) = 2.9$.

From the expected path length we can clearly see that, given the belief of the character, the benefits from trying to pass the edge $⟨b, g⟩$ are slightly greater than the risk of the additional backtracking cost.

It is easy to imagine that in more complex graphs, finding a solution in this way is quite complicated. First, the number of realization grows exponentially with the number of moving edges²; second, computing the backtracking cost quickly becomes an issue.

This problem is well known in literature with the name of Canadian Traveler Problem (CTP) [69]. However, while CTP is an important part of the theoretical framework of BAP, it remains just one aspect of BAP in its entirety. In the next section, we will talk about the other parts of a BAP character. We will talk more on the formal problems of CTP in Chapter 11.

### 9.4 The planning-execution cycle of BAP

As we have seen before, the BAP scope goes beyond the off-line search for a navigation strategy. We can find a good strategy, however, a complete BAP pathfinding must also verify this strategy, update the beliefs, provide a perception schema to get new information and handle the unexpected situations (e.g., when the belief for an edge to be open is 1 but, in the real map, it happens to be closed). Moreover, all this must be encoded in a series of techniques that are practical and suitable for video games applications.

A BAP approach needs, therefore, to interleave (at least) three phases as shown in Figure 9.4.

¹Note, however, that a path can be seen as a partial strategy in which there is specified only a small subset of locations and for every one of them there is only a single best solution.

²For a graph with $n$ moving edges, there are $2^n$ possible realizations of $G$. 
I. Planning phase

In the planning phase, the character computes a policy based on his degree of belief of the map state. This step is performed offline and consists of two sub phases as follows:

Map Instantiation and Synthesis. This step requires the character to embed its own beliefs into the global map. The global map represents the “template”, that is shared among all the characters, on which every character bases the map used in BAP’s later stages. The global map can be any pathfinding graph abstraction (e.g., NavMesh, grids, point graphs and other). A character takes the global map and merges with it its own belief data in order to produce a random graph. The belief data usually comes in the form of a meaningful map abstraction (a high-level representation of the map abstracted around the moving edges of the real map) that is computed once at the beginning of the game for each map, and the actual belief vector updated at the end of each BAP cycle.

Pathfinding. This step is where the actual strategy is computed. In the simplest cases, we can imagine a standard pathfinding query over the “most believable” realization of the input random graph. In more complex cases, instead, we can look at this step as a complete solution to the underlying CTP problem. In any case, an approximate solution to the navigation problem is often good enough for BAP. Looking for an optimal strategy is typically resource demanding and, at the end, often useless because the strategy is quickly invalidated by the dynamicity of the game environment.

II. Execution phase

In the execution phase, the character starts executing the computed policy. This can include following one or more steps of the computed path/policy. By itself, this step can be trivial. However, this is where the character needs to take important decisions on unexpected events and trigger a new planning step.

- The policy is incomplete. Sometime, we may compute only a partial policy, or a partial path because we do not want to waste resources into computing a full path that will be found invalid before the character reaches the destination.
- The policy is wrong. The preferred actions are all closed. While the character executes the policy, it starts discovering in which realization it is. Even if
our policy or candidate path is based on the most believable realizations, the reality can be completely different. Many policies can provide alternatives if a certain edge supposed to be open is closed, but unless we computed a full policy that covers every case, we may end in a situation in which our plan has no more options.

- **The policy is wrong. A much better option is open.** The dual problem is when a character discovers that an edge that was supposed to be closed is, in reality, open. At this point, the character can continue to follow the previous policy, but if the edge allows the character to reach the destination much faster, it is probably better to plan again.

- **The character beliefs are wrong.** The character may discover that his beliefs are wrong very often. In this case, the execution phase can decide to stop following a policy based on a such out of date idea of the environment.

We are not forced to take into account all these situations, but the decision about when to stop following a policy is the execution step duty.

### III. Perception phase

The execution phase is usually interleaved with the perception phase. In this step, the character’s beliefs are updated with new information coming from the execution and the surroundings of the character. There are two sub-phases in the perception phase: perception and beliefs update.

**Sensing and update.** For the perception sub-step, the character relies on a perception schema that describe how the character perceives the environment. BAP does not require a particular perception schema and, therefore, they can be designed at will. Note however that perception schema can affect the overall performance of the BAP character. Common design patterns are:

1. **Area knowledge.** Every time the character enters in a new area (as defined by the meaningful belief abstraction) the character instantaneously perceives the real state of every outgoing edge, even if they are out of the character’s line of sight. If the areas are small enough this is an easy and effective way to provide a perception schema for the character.

2. **Line of sight.** The character ray-traces any location associated with a moving edge to obtain the real state of the edge that can be seen by the character.

3. **Perception volume.** The character perceives the state of all the edges in its perception volume (usually a sphere). This is similar to the line of sight approach, but perception has a limited distance.

After the character obtains this new knowledge we need to merge it to the current beliefs. This depends on how the character stores its beliefs data. Usually, this is done by setting the beliefs for that particular edge to 1 (if open) or 0 (if closed). However, there may be exceptions, or other beliefs that are dependent on the value of the others, for instance, a character may know that if a certain blue door is open, then all the blue doors are open.
Knowledge decay. Another important aspect is how we may progressively invalidate the character’s beliefs for a particular edge if the state of that edge is not refreshed in a long time. This affects all the beliefs at once and it is executed periodically. This effect is called knowledge decay and we will talk more about it in Section 11.4.
Chapter 10

A basic boolean BAP solver

Developing a BAP character does not require necessarily a complete solution of the Canadian Traveler Problem. Instead of recording beliefs as values in the \([0,1]\) interval, we can imagine storing them as boolean values. Then, when we perform the planning step, we can simply generate a map realization in which all the edges believed to be “false” will be removed, and all the edges believed to be “true” will be included.

In some sense, we are removing much of the burden of reasoning about uncertainty from the planning phase—that can now be performed by a traditional pathfinding algorithm—to the sensing and execution step. All the complex reasoning about beliefs is now encoded into a single binary state. Obviously, this solution is much less powerful than the solutions based on the degree of belief; it is in fact no more possible to reason about how much probable is that a certain path is blocked and, at the same time, we completely discard the backtracking cost.

However, in many videogames, this level of details is unnecessary and a fast and “easy to understand” solution is preferable to more complex algorithms. Even if this algorithm lacks the flexibility provided by a more formal and general algorithm, it is important as a first step toward more complete BAP solvers and introduces several concepts that we will exploit in the following chapters.

In this chapter, we will try a first basic solution to the BAP problem that goes exactly in the direction described above. In the following algorithm, we ignore for a moment the problem of a generic BAP solution based on optimal strategy and we will provide, instead, an abstraction-based algorithm which return directly the most promising path on the basis of the character belief state. This proposed algorithm is described in its entirety, covering all the BAP phases and can be implemented independently from all the other BAP concepts.

10.1 GCA*: Gate connectivity A*

Before going into details with the BAP approach, we describe a simple hierarchical pathfinding approach over gates in the game map.

We assume that the game-world is represented as an octile map \(M\) such that each tile \(m \in M\) may be blocked or free, and appropriate functions exist for retrieving neighboring tiles and the state of the tile. We choose the grid representation for simplicity but note that the methods we present can be adapted to work for other representations.

Gate connectivity A* (GCA*) relies on a simple map decomposition of \(M\) into a set \(\mathcal{R}\) of \(n\) non-overlapping regions \(R_i \subseteq M\), such that \(\bigcup_{i=1}^{n} R_i = M\), and
$R_i \cap R_j = \emptyset$ for every $i \neq j$. Finally, we define $M_P$ as the set of tiles that are in the edge of one region in the sense that they have a neighbor that belongs to a different region.

We now define two important entities in the GCA$^\ast$.

**Definition 10.1.** Given $\mathcal{R}$, we define a **portal** $p$ as a pair $(m_i, m_j) \in M \times M$ such that:

1. $m_i \in M_P$ and $m_j \in M_P$
2. $m_i \in R_i$ and $m_j \in R_j$
3. $i \neq j$

We say that $p$ belongs to $R_i$ if and only if $m_i \in R_i$ or $m_j \in R_i$. As a consequence, any portal always belongs to exactly two regions.

**Definition 10.2.** Finally, we define a **gate** $G$ as a set of portals, such that:

1. Every portal $p \in G$ belongs to the same pair of regions $R_i$ and $R_j$.
2. The portals in $G$ form a consecutive horizontal or vertical line of one or more portals.

We also say then that gate $G$ connects regions $R_i$ and $R_j$ and use $\text{conn}(G)$ to denote the set $\{R_i, R_j\}$.

Given a map $M$ and an appropriate set of regions $\mathcal{R}$ (e.g., one that decomposes the map into coherent areas) and an appropriate set of gates $\mathcal{G}$ (e.g., one that segments the borders of regions into gates of bounded size), then we can define the **gate connectivity graph** $\Gamma_\mathcal{G} = \langle V, E \rangle$ as follows.

**Definition 10.3.** A gate connectivity graph is a tuple $\Gamma_\mathcal{G} = \langle V, E \rangle$ such that:

1. The set of vertices $V$ is the set of gates $\mathcal{G}$.
2. An edge $\langle G_i, G_j \rangle$ is in $E$ if and only if $\text{conn}(G_i) \cap \text{conn}(G_j) \neq \emptyset$. That is, the two gates connect at least one region in common.
3. Each edge $\langle G_i, G_j \rangle$ is labeled with the true distance between $G_i$ and $G_j$. The true distance is computed by finding the shortest path from the center of gate $G_i$ to the center of $G_j$ considering only the region in common between the two gates. If the regions in common are two, we use the shortest of the two paths.

As the map decomposition $\mathcal{R}$ is fixed, the gate connectivity graph $\Gamma_\mathcal{G}$ can be precomputed offline. If the memory requirements allow it, we can store the full paths between each gate and avoid entirely the online computation for the inter-gate paths.

Note that a gate connecting two regions may actually be just a couple of tiles wide, while two automatically decomposed regions may be connected through a large number of tiles as illustrated in the map of Figure 10.1. At the one extreme, one can consider every portal as a gate (that the NPC believes to be free or blocked), and at the other extreme one can consider just one gate per two neighboring regions. In Figure 10.1 the gate connectivity graph is composed by six gates that abstract the connections of the regions according to the latter extreme. In a more practical
Figure 10.1. (a) A small benchmark map decomposed in five regions, and (b) a corresponding gate connectivity graph. Gate $G_1$ is a set of three portals, each of which is denoted by a double arrow pointing to the two tiles of the portal.

scenario, however, we can put a maximum gate size of the number that is more appropriate in the specific game scenario.

Similar to more sophisticated hierarchical planning approaches, such as [13, 90, 87], GCA* first searches for a path in the higher-level portal connectivity graph, and then searches for low-level paths that realize each of the high-level moves between regions.

For a query between two generic locations $m_a$ and $m_b$, GCA* handles a request as follows:

1. If $m_a$ and $m_b$ belong to the same region, the algorithm returns the path $m_a \rightarrow m_b$. In this case we can completely ignore the high-level gate connectivity graph.

2. Otherwise, graph $\Gamma_G$ is expanded with two vertices $v_a$ and $v_b$ into graph $\Gamma^*_G$. New edges are added between $v_a$ and each gate that connects the region where $m_a$ is in, and similarly for the new vertex $v_b$ and $m_b$.

3. An $A^*$ search is launched on $\Gamma^*_G$ in order to find a high-level path of the form $v_a \rightarrow G_i \rightarrow \cdots \rightarrow G_j \rightarrow v_b$. If no path is found GCA* returns with failure.

4. Otherwise, the high-level path found is expanded by a regular low-level pathfinding algorithm such that for each step $G_i \rightarrow G_j$ a low-level path is found that connects $G_i$ and $G_j$ through tiles of the map $M$ while searching only inside the area that $G_i$ and $G_j$ connect. GCA* does not specify any pathfinding algorithm for this step. We can use any pathfinding algorithm such as $A^*$, JPS, and so on.

10.2 The BCGA* algorithm

Starting from GCA*, we now present a BAP approach on top of it. In this BAP algorithm, each non-player character (NPC) maintains its own gate connectivity map capturing their beliefs about the map state. Because in GCA* we provide an

Our definition of portals is inspired by [28] but we use a pair of tiles connecting exactly two regions, instead of a single tile that may connect multiple regions. A gate then is a (part of a) gateway in the sense of [11].
abstraction based on meaningful locations which are points of interest from the navigation point of view—such as doors or places where an obstacle may be put to obstruct the passage—the resulting gates are a super-set of the map moving edges. We can then translate the beliefs that a certain edge is open or not, into the belief that a certain gate is traversable or not.

This is a practical way to capture beliefs about connectivity as it essentially filters out the local obstacle avoidance inside each region, which may be also fine-tuned at execution. Also, each NPC need only maintain beliefs about a small subset of the map represented by the gates. This reduces the duplication required for providing a personalized belief vector to every character in game.

Depending on the resolution of the map, many portals can be grouped together to form a meaningful gate that corresponds to a part of the map that can be blocked during play in a way that actually obstructs navigation from one region to another. In this sense, the extreme case of putting together all portals in the border between two regions in a single gate works well, as the personalized portal connectivity map would hold information about the high-level connectivity of regions and not about the actual detailed position that the NPC should enter or leave the region. Nonetheless, since in very wide borders this detail may provide a nice refinement to the path followed by a belief-aware NPC, we expect that a better approach is to group portals together into gates according to an average size of gateways in the game. For example, in the map of Figure 10.1 a reasonable choice for the size of a gate would be three portals.

We assume that appropriate gates are formed based on a maximum size in tiles. This then gives some bounds on the memory that each NPC needs to allocate for the personalized beliefs as well as the total memory overhead for this BAP approach. In the worst case, the memory required for storing the beliefs for a single NPC is proportional to the size of the portal set $|MP|$, which with an average decomposition algorithm is expected to be much smaller than the total number of tiles in the map. The total memory needed for $n$ NPCs is then just the amount needed for a single one multiplied by $n$.

Belief representation in BGCA*

The belief vector $B$ for a character in BGCA* is represented by a list of boolean variables $b_i \in \{0, 1\}$ such that $b_i = 1$ represents the fact that the $i$-th gate $G_i$ is open (and, therefore, the character can traverse it), while $b_i = 0$ indicates that gate $G_i$ is closed.

In BGCA* we do not store intermediate values for the character beliefs. Every gate can be open or closed and no uncertainty is allowed. While this solution is not expressive as a more general BAP application, in terms of implementation it is very memory efficient because we can store each belief in a single bit and use bitwise operation to query and update the character beliefs for the map. For instance, given $B_R = \langle b_1, \ldots, b_n \rangle$ the real state of the map updated by a central system, and given $M(R_i)$ a binary mask associated with the $i$-th region, it is possible to update the character beliefs with a “bitwise and” between $M(R_i)$ and $B_R$. If we need to update more regions at once, we may first do and “bitwise or” of every region mask and then do the “bitwise and” with $B_R$.

Unfortunately, even if belief information is just a boolean value, as time goes by, the NPC may need to “forget” or discard some of his beliefs because they may have become obsolete. According the general BAP problem, this is the problem of knowledge decay. However, when considering boolean beliefs we cannot use a smooth gradual decay and, instead, we need to forget everything and assume the gate is
open again after some time. We will give more details about this problem in the perception phase section. For now, we note that each character needs another vector $T = (t_1, \ldots, t_n)$ storing the last time the state for $i$-th gate has been updated.

The cycle of the algorithm follows the three phases of Section 9.4, alternating between offline planning and online execution and perception.

The behavior of BGCA* can be tweaked by three parameters that we will describe in the details of the three phases.

1. The maximum number of repaths (i.e., planning and execution iterations) allowed, before BGCA* returns failure.
2. The perception radius of the NPC.
3. The time-window after which beliefs are invalidated.

I. Planning phase in BGCA*

In BGCA*, the planning step is executed using GCA*. However, we introduce an important preliminary step. Before the search starts, we create a personalized gate connectivity graph for the querying character and then we use it for the high-level search step.

The way in which this personalized gate connectivity graph is created starting from $\Gamma_G$ and then removing all the gates $G_i$ such that the character belief $b_i$ is 0. In practice, we want to change the A* neighbors’ expansion function so that it returns only the gates believed to be open by the character.

If a path in the personalized connection graph exists, then this phase returns this high-level path as a plan for the next phase. Otherwise, if the GCA* query on the personalized graph fails, we can provide different fallback behaviors such as the following:

1. Relax the problem. We can run a knowledge decay iteration by all beliefs that are older than a specified, that is, switching old closed beliefs to 1. At this point, we can run GCA* again.
2. Randomly explore the surrounding. If the character believes that the destination is unreachable, then we can imagine the character wandering a bit like he is searching for new information. This allows the character to get more information and the knowledge decay to “open” more gates in the character mind.

II. Execution phase in BGCA*

In the execution phase, BGCA* uses the high-level path computed in the planning phase. This plan is a sequence of gates such as $\Pi = (s, G_1, \ldots, G_k, g)$, where $s$ and $g$ are the starting and goal locations, and $G_1$ to $G_k$ is a sequence of $k$ gates.

The first step is to compute a low-level path from $s$ to $G_1$. This can be done with any pathfinding algorithm but we have to be sure that the search is limited to the region in which $s$ is.\footnote{In fact, if a region is not regular, it is possible that the shortest path between two gates (or start to gate) passes through another region, eventually through a gate believed to be closed.}

Then, the character can pass the gate through a specific portal (usually the one in the middle). Upon entering the new region, a perception step is invoked and a new path from $G_1$ to $G_2$ is computed (or read from the precomputed paths database).
However, if the destination gate is closed, the path is not realizable. In this case the execution step fails and the character returns back to the planning phase. Otherwise, the character will reach the destination $g$ and BGCA* finishes with success.

III. Perception phase in BGCA*

As is common in BAP, an important aspect for the final behavior of the character is the perception phase. This step handles how every character perceives the environment, how BGCA* updates the belief vector for every character and how obsolete information is removed from the computation.

For the first task, the definition of a perception schema, BGCA* uses $k$-regions wide instantaneous perception procedure. Every time a character enters in a new region, the character perceives the real state for every gate that is $k$ high-level steps away from the entering gate. We call this number $k$ \textit{perception radius} even though we are not referring to some geometrical radius like in the \textit{perception volume} schema (Section 9.4).

As we saw before, updating the character beliefs in the BGCA* case can be done by simply registering the updated states in the respective beliefs. However, we need also to update the timestamp vector $T$ with the current time.\footnote{Timestamps can be expressed in milliseconds, seconds, frames or iterations depending on the game architecture.} Timestamp is needed to perform a discrete analogous of the \textit{knowledge decay} presented in the previous section.

To better understand the effect and the necessity of knowledge decay in BGCA*, we look at two extreme situations. At one extreme, the NPC may focus solely on his perception, keeping only the beliefs that come from his field of view and discard anything else, and at the other extreme he may give strong trust to his memory, keeping all beliefs until they are revised by his perception.

In the first case, we are completely ignoring beliefs. The algorithm will consider all the gates open all the times with the exception of the ones in its perception radius. This guarantees that we always find a path (if it exists), but we may be forced to several new search instances every time the path brings the character to a closed gate. Moreover, as the character is not learning from its experience, in the next iteration it may go back to a closed gate and get stuck in an infinite loop between two regions.

In the second case, instead, we have the opposite problem. If a character believes that the destination is inside a room with a closed gate, it will always fail to find a high-level solution to the problem. The character will avoid to even try to reach the destination until he updates the state of the destination gate during his random exploration. In practice, because every closed gate will remain closed forever in the character’s beliefs, the character will be blocked by its own beliefs.

A middle ground is desired where beliefs are kept in memory only up to a certain time-limit so that they guide his navigation but without hurting too much the precision of his results. Note that it is actually desired that the NPC follows paths that may be obsolete at the time of execution or even fails to find a path when in fact there is one, but this should happen a limited amount of times that makes his navigation reasonable, and believable.

BGCA* adopts a practical solution that simplifies the handling of unknown gates by an \textit{open-door assumption}, i.e., the discarded beliefs are updated to beliefs that the corresponding gates are free. This is a common assumption followed by several approaches for pathfinding in unknown maps. BGCA* also adopts a practical
one-step strategy for discarding beliefs: essentially all beliefs are kept until the NPC gets to a point that a pathfinding request fails to find a solution. Then beliefs that are older than a specified time window are discarded.

10.3 Experiments with the BGCA* algorithm

10.3.1 Experiments setting for BGCA*

In order to investigate the correlation between the parameters of BGCA* and the performance of the algorithm, we have developed a BAP tool in the Unity game engine\(^4\) that automatically iterates over a set of configuration files and maps.\(^5\)

For each configuration and each map, the benchmarking tool acquires a map area decomposition using the flood-filling method of [11], generates the gate connectivity graph, and goes over a number of BGCA* pathfinding requests as specified in the configuration file. Unlike other benchmarks that generate paths with random start and destination tiles, here only random destinations are generated and the request uses the current position of the NPC as the start. This is needed in order to measure the use of BGCA* in a scenario where beliefs about the areas that the NPC visited recently can be used to drive the new paths requested. Moving the character in a new start destination can put the character in disadvantage because it may not have previous beliefs and information on the new location.

The BAP tool takes as input the following parameters:

- **max-gate-size**: the maximum size for gates.
- **iterations**: the number of destination requests,
- **initial-closed**: the ratio of closed gates,
- **shuffle-amount**: the ratio of gates to be updated,
- **shuffle-rate**: the frequency of updating gates,
- **max-repaths**: the maximum number of repaths,
- **update-radius**: the perception radius of the character,
- **belief-window**: the time-limit for discarding beliefs.

During the generation of destinations, positions that lie in the same area as the NPC are filtered out. The initial ratio of closed gates is specified in the parameters and a shuffling of them takes place every few iterations. This changes the state of the percentage of gates specified but keeps the total ratio of closed gates constant. The last three parameters correspond to the parameters of BGCA* as described in the previous section.

As far as the time-window is concerned, since the navigation of the NPC is simulated, we chose to specify the window in terms of the number of shuffling events that have occurred. For example, if a shuffling rate of 10% is specified, then belief window of 4 would correspond to the time required on average before 40% of the map has changed in a real-time running game.

\(^4\)unity3d.com
\(^5\)The benchmark code can be found at [https://github.com/THeK3nger/BDP-Benchmark](https://github.com/THeK3nger/BDP-Benchmark)
Regarding the map decomposition, as we said before, it can be manually provided or automatically computed. In this work, we will assume that an automated decomposition is obtained by using an algorithmic map partitioning algorithm, e.g., the counter-based graph-partitioning algorithm [28] based on the betweenness centrality of the portal edges, or the flood-filling algorithm used in [11] for their pathfinding gateway heuristic. While the first algorithm provides a more robust map decomposition, the second one always provides horizontal or vertical portals that are easier to use and store. On the other hand, this partitioning method does not perform well on every map, especially, in maps that grow diagonally with respect to the $x, y$ axes, in which case it tends to generate many non-intuitive regions.

We used maps taken from Dragon Age: Origins (BioWare, 2009) provided by the MovingAI benchmark database [86]. We formed two groups of representative maps of increasing size, each of which consisting of five maps. Group A contains “open” maps that allow multiple solutions to pathfinding requests, while Group B contains “maze-like” maps where one blocked passage may cut-off connectivity between big parts of the map. Group A consists of maps: orz500d, den504d, rmtst, arena, den101d. Group B consists of maps: lak302d, den005d, lak202d, den020d, orz601d. A picture of the maps (with the corresponding abstraction) is shown in Figure 10.2.

The test application reports a detailed log for each configuration file and map. As we intend to compare the performance of BGCA* with an omniscient pathfinder, the testbed also performs the same requests with an omniscient GCA*. This is so that the comparison is fair, as GCA* is also hierarchical and works in the same way but in one-shot using the accurate version of the current map.

### 10.3.2 Experimental results

We measure the outcome of the different configurations for BGCA* with respect to effort and precision. As far as the effort is concerned, we look into the number of total expanded nodes for the whole set of planning execution steps until the destination is reached or BGCA* quits. We also look into the number of expanded nodes for each execution step. We refer to this as the expanded nodes per repath and it corresponds to the effort needed for each iteration of the planning and execution.
Algorithm 26 Automatic Map Decomposition algorithm as presented in [11] and used in BGCA* experiments.

1: \(\text{currZone} \leftarrow 1\)
2: \(\text{while} \) there are no more free tiles \(\text{do}\)
3: \((x_{Left}, y) \leftarrow\) top leftmost free tile of the map
4: \(\text{shrunkR} \leftarrow\) \(\)false
5: \(\text{shrunkL} \leftarrow\) \(\)false
6: \(\text{while true do}\)
7: \(\text{\hspace{1cm}}\) \(\)\(\triangleright\) \(\)Mark line until hit wall or area opens upwards.
8: \(\text{\hspace{1cm}}x \leftarrow x_{Left}\)
9: \(\text{\hspace{1cm}}\text{zone}(x, y) \leftarrow \text{currZone}\)
10: \(\text{\hspace{1cm}}\text{while} \ (x + 1, y) = \text{free} \ \text{and} \ (x + 1, y - 1) \neq \text{free} \ \text{do}\)
11: \(\text{\hspace{2cm}}x \leftarrow x + 1\)
12: \(\text{\hspace{2cm}}\text{zone}(x, y) \leftarrow \text{currZone}\)
13: \(\text{\hspace{1cm}}\) \(\)\(\triangleright\) \(\)Stop filling area if right border regrowing.
14: \(\text{\hspace{1cm}}\text{if} \ (x + 1, y - 1) = \text{currZone} \ \text{then}\)
15: \(\text{\hspace{2cm}}\text{shrunkR} \leftarrow\) \(\)true
16: \(\text{\hspace{1cm}}\text{else if} \ (x + 1, y - 1) = \text{currZone} \ \text{and} \ \text{shrunkR} \ \text{then}\)
17: \(\text{\hspace{2cm}}\) \(\)\(\triangleright\) \(\)Undo line markings
18: \(\text{\hspace{1cm}}\text{while} \ (x, y) = \text{currZone} \ \text{do}\)
19: \(\text{\hspace{2cm}}\text{zone}(x, y) = \text{free}\)
20: \(\text{\hspace{2cm}}x \leftarrow x - 1\)
21: \(\text{\hspace{1cm}}\) \(\)\(\triangleright\) \(\)Goto same initial x-position in next line
22: \(\text{\hspace{1cm}}(x, y) \leftarrow (x_{Left}, y + 1)\)
23: \(\text{\hspace{1cm}}\) \(\)\(\triangleright\) \(\)If on obstacle, go right in zone until empty.
24: \(\text{\hspace{1cm}}\text{while} \ (x, y) \neq \text{free} \ \text{and} \ \text{zone}(x, y - 1) = \text{currZone} \ \text{do}\)
25: \(\text{\hspace{2cm}}x \leftarrow x + 1\)
26: \(\text{\hspace{1cm}}\) \(\)\(\triangleright\) \(\)Move further left until wall or opens upward.
27: \(\text{\hspace{1cm}}\text{while} \ (x - 1, y) = \text{free} \ \text{and} \ (x - 1, y - 1) \neq \text{free} \ \text{do}\)
28: \(\text{\hspace{2cm}}x \leftarrow x - 1\)
29: \(\text{\hspace{1cm}}\) \(\)\(\triangleright\) \(\)Stop filling area if left border regrowing.
30: \(\text{\hspace{1cm}}\text{if} \ (x - 1, y - 1) = \text{currZone} \ \text{then}\)
31: \(\text{\hspace{2cm}}\text{shrunkL} \leftarrow\) \(\)true
32: \(\text{\hspace{1cm}}\text{else if} \ (x, y - 1) \neq \text{currZone} \ \text{and} \ \text{shrunkL} \ \text{then}\)
33: \(\text{\hspace{2cm}}\) \(\)\(\triangleright\) \(\)break
34: \(\text{\hspace{1cm}}\) \(\)\(\triangleright\) \(\)break
35: \(\text{\hspace{1cm}}\text{currZone} \leftarrow \text{currZone} + 1\)

Cycle; that is, in practice, the time before the NPC starts moving in the game map. As far as precision is concerned we measure the ratio of destination instances that BGCA* and the omniscient GCA* agree about success or failure. This value is important because BGCA* will “give up” after a certain number of iterations of the planning-execution cycle. In real world games, however, this is exactly the desired behavior and, after a certain number of iterations, the target location probably moved in a different location anyway. Finally, we take into account the number of repaths per destination.

Our intention is to identify the parameters that achieve a balance between the NPC finding paths with a precision above 75% while inducing the least effort as possible and keeping the number of repaths to a small constant, e.g., 2 or 3.\(^6\)

For example, if we would allow the NPC to discard all beliefs and retry as many times as needed then he would eventually find a path if one exists. At the other extreme if we never allow the NPC to discard beliefs then at some point he may not be able to find paths outside of his area of vision when the last information he had

\(^6\)This is also because unlike other contexts where the shortest path is desired regardless of repath steps, here we are interested in behaviors that generate only a few “re-thinking” events for the NPC that are easier to be acknowledged by the player.
was that the surrounding areas next to his location were blocked.

The following results show how a balance can be achieved for the two groups of maps that we identified. We performed several experiments and we report here the ones that are more suitable for drawing conclusions. For each one we set the number of iterations to 500, and for all reported sets we set the ratio of closed gates to 20% and a shuffling ratio of 10% after every destination iteration. Also, we used a fixed number for the maximum size of gates to be at most 5 portals. As far as the parameters of \( \text{BGCA}^* \) are concerned we report on combinations of belief-window values of 2, 4 and 6, and update-radius values of 1 and 2. Figure 10.3 and Figure 10.4 show the effort for these experiment sets, where “W” stands for the belief-window and “R” for the update-radius.

From these graphs, we can conclude the following:

- The average effort per repaths of \( \text{BGCA}^* \) in the “open” maps of Group A is comparable with the one of omniscient \( \text{GCA}^* \). This means that as far as the NPC making the first move is concerned, \( \text{BGCA}^* \) is not inducing any significant overhead. As a matter of fact, \( \text{BGCA}^* \) just searches for a different path the cost of which is comparable to the computation of the classical shortest-path.

- The average effort per repaths of \( \text{BGCA}^* \) for the “maze-like” maps of Group B is greater than that of omniscient \( \text{GCA}^* \). However, it seems to be bounded by a factor of 2-3 times the effort of \( \text{GCA}^* \). This shows that some constant overhead is to be expected to maps of this type.
Figure 10.4. Total effort and effort per repath for maps in and Group B.
10. A basic boolean BAP solver

Figure 10.5. Precision and average number of repaths for the maps of Group A and B.

- The total effort of BGCA* is larger than that of omniscient GCA* which is expected as the NPC makes a number of repaths before reaching the destination.

- There is a trade-off between total effort and precision. A lower value for the belief-window can help the character to reach a precision above 90%, however leads to a larger total effort. On the other hand, an increase of the perception radius can help the character achieve a higher precision with the same effort. For instance, in Figure 10.5, every experiment with radius 2 leads to a precision score above 90% but the effort is comparable with the effort of the corresponding experiments with a perception radius of 1. The use of a radius larger than 1 is a way to achieve better performance by giving the NPC more perception power.

The last point motivated us to explore further a slightly more adaptive version of BGCA* such that in the planning procedure instead of performing a one-shot revision (point 2), relies on an incremental revision up to a smaller belief-window. Additional experiments showed that a variant of BGCA* that uses a variable belief-window from 6 to 4 (depicted as $W = 6-4$) has a more consistent performance with respect to precision in the maps of Group B, and in fact achieves superior performance with respect to precision in the maps of Group A.

Finally, as far as memory usage is concerned, the total number of gates shows the memory usage per NPC. Assuming that one bit is stored per tile and gate, a map with $n$ tiles requires $n$ bits for a single copy, $nm$ bits for $m$ characters keeping...
a full copy each, and $km$ bits for our approach where $k$ is the number of gates. For example, for map `den005d` in group B, $n$ is 17559 and $k$ is 571. Note that portals and gates are identified by means of an automated procedure for decomposing the map into regions and considering all tiles in the border between regions. In practice, the number of gates could be much lower if game designers identified directly those points of interest that are appropriate to be considered as gates, e.g., doors, passages, breakable walls, etc.
Chapter 11

A degree-of-beliefs based algorithm for BAP

The simple discrete solution presented in the above chapter (BGCA*) is effective in practice in many cases. However, it has limitations. First, the physical meaning of some parameters—such as the validity window—may be difficult to tweak and understand. Second, it completely ignores the uncertainty by assuming that everything the character is not sure about, it is considered open (open world assumption).

The combined effect of these two problems is that the overall algorithm is not very flexible. Suppose we want to design two different character behaviors. The first character may be cautious and move along the path that is more probable to be valid. The second one, instead, may be a more reckless character and decide to take more risks by trying paths even if they are probably not valid. How we can encode this behavior in BGCA*? We can tweak the validity window to emulate these two behaviors, but without a clear correspondence between this value and the desired behavior.

The keyword in this case is the word “probable”. The difference between the two behaviors is in how the characters decide on the basis of their measured probability that a path is valid or not. They can accept different level of probabilities, however, in both cases we need a way to estimate such value. We need, therefore, to do a step back and look at the planning phase of BAP from a more general point of view.

In this chapter, we will show that the BAP planning phases can be approximately solved by considering the underlying Canadian Traveler Problem as an instance of a Markov Decision Process problem. Finally, we present a improved BAP algorithm analogous to BGCA* that can be more easily tweaked and expanded.

11.1 The Canadian Traveler Problem as a Markov Decision Process

11.1.1 The Canadian Traveler Problem

As we said in the introduction to the Belief-Aware Pathfinding problem, the BAP planning phase overlaps the Canadian-Traveler Problem. Before we move on, we go over the steps of the BAP problem. First, we need to abstract the game map into a high-level representation similar to how we did in BGCA*. The idea is to divide the map into convex regions and identify on the edge of these regions a set of connection points called entrances. An entrance is composed by two nodes, one in
one region and the other in the other region, connected by an unitary edge (this is the same abstraction scheme of HPAn). It is important to represent as an entrance all meaningful dynamic obstructions that are present in the map (i.e., doors, gates, breakable walls and so on). We will call this kind of entrances moving entrances because they can be open (so that they are connected by an edge) or closed (with no edges between the two nodes of the entrance). For this kind of entrances each character stores the value \( b \) representing the character belief that the entrance is open into a belief vector \( B \). At the beginning, without previous knowledge, we can assume \( p(e) = 0.5 \) for all \( e \in E \).

The result of this abstraction step is random graph \( G \) representing the high-level navigation of the character in the environment. Because of the stochastic nature of the graph is not possible to find a fixed solution. Instead the goal is to find the best strategy or policy to navigate this graph.

This problem, the core of the BAP planning phase, is referred in literature as the **Canadian Traveler Problem** [69] or **Stochastic Shortest-Path Problem** [73]. The CTP is a generalization of the shortest path problem to graphs that are partially observable. For a character moving in it, the graph is revealed only while it is being explored, and the cost of exploration is taken into account even if they do not contribute to the final path.

In its general form, the problem is known to be PSPACE-complete [96] and therefore is not scalable to big problem instances. However, we will show that the for offline planning phase of a BAP algorithm we can produce an approximated solution that is more than enough in any practical application in video game setting. In fact, the offline planning phase of BAP has two main requirements. First, for practical implementations in games it requires that the high-level policy can be found very quickly. Second, an optimal solution is not required. Because of the dynamic nature of a game map, for a believable behavior an approximate solution is more than enough. Finally, we will see that in BAP the graph can change during the execution of the policy, the belief vector \( b \) is continuously updated according the character perception and the natural decay of knowledge (see Section 11.4). On one side, this allowa us to avoid searching for an optimal policy because the graph can change at any time, invalidating the optimality of the policy. On the other side, however, this makes the full problem (including both offline and online phase) more complex.

### 11.1.2 The Markov Decision Process

In its static form, the CTP problem can be directly mapped into a **Markov’s Decision Process (MDP)** [74]. A Markov decision process is a tuple \( \langle S, A, P(.,.), R(.,.), \gamma \rangle \) where

- \( S \) is a finite set of states.
- \( A \) is a finite set of transition actions between states.
- \( P_a(s,s') \) is the probability that action \( a \) in state \( s \) will lead to state \( s' \).
- \( R_a(s,s') \) is the immediate reward received after transition to state \( s' \) from \( s \).
- \( \gamma \in [0,1] \) is the **discount factor** which represents the difference in importance between future rewards and present rewards.

The solution to a MDP is a policy \( \pi \) that describes the best actions from each state that has to be performed in order to maximize the total expected reward.
11.1.3 From CTP to MDP

The Canadian Traveler Problem can be easily reduced into a Markov Decision Process instance using the following equivalences:

- \( S := V \), the states are the vertices in \( G \).
- \( A := E \cup E_r \), the actions are the edges between vertices in \( G \).
- \( P_a(s, s') := \hat{p}(e) \) where \( \hat{p} \) extends \( p \) such that if \( e \in E \) then \( \hat{p}(e) = 1 \).
- \( R_a(s, s') := -c(e) \), because maximizing the reward or minimizing the cost are equivalent.
- \( \gamma = 1 \).

The goal state is modeled as an absorbing state in which the character can loop indefinitely with zero cost.

This is an important reduction because we can start to solve this problem with all the state-of-the-art algorithms for MDP problems.

11.2 Exact solution to the CTP

There are two main algorithms for a MDP instance that are able to return the optimal policy for the MDP instance. For optimal policy we mean a policy that, if executed, lead us to the maximum expected reward, that is, on average, allow us to maximize the total reward on the Markov’s process. In CTP setting we will call it estimated shortest-path. The application of these algorithms to the shortest-path problem in stochastic graphs has been studied in its general setting in [7]. Bonet and Geffner in [12] provide optimality guarantees for real-time dynamic programming (RTDP). These algorithms have been also applied to navigation problems in robotics in [16] and [15].

Value Iteration

The value iteration (VI) algorithms compress the constraints of a MDP problem into a system of \( N - 1 \) equations where \( N \) is the total number of states:

\[
v = F(v)
\]

Thus, the desired solution \( v \) of this system of equations is a fixed point of function \( F \). The value iteration algorithm finds this fixed point by treating Equation 11.1 as an iterative process:

\[
v^{(n+1)} = F(v^{(n)}),
\]

given the initial state \( v^{(0)} = 0 \). More information about the value iteration algorithm applied to a CTP-like problem can be found in [16].
Policy Iteration

The idea of the second algorithm is to hypothesize a policy for all the nodes, and to solve the resulting linear system of Equation 11.1. The process is then repeated, using the previous solution to determine the new edge priority. The iteration terminates when the solution to the current system of equations yields the same policy as the previous solution.

There is extensive information in the literature about these methods. Here we presented only the main idea that is needed to understand the rest of the approach.

11.3 Monte-Carlo solutions to the CTP

The problem with the previous algorithms is that they converge to the optimal policy for every initial state to the target. For a pathfinding problem, this level of details is often unnecessary. In a pathfinding instance, the target may change with every query and we are interested only in finding the correct policy from the current state to the destination. Moreover, in a BAP instance the degree of belief distribution over the entrances changes during the execution of the path due to the character perception. Therefore, finding an optimal solution for every node in the graph is a waste of computational time.

For this reason, it is a good idea to use some non-optimal or near-optimal algorithm in order to achieve an acceptable solution in a practical amount of time. One way of solving this is through the Monte-Carlo Planning family algorithms. A good analysis of the Monte-Carlo approach for the Canadian Traveler Problem is presented in [24].

The Optimistic Policy (OMT)

The simplest policy we can imagine is the optimistic policy (OMT). According to this policy we find the most optimistic instantiation of the random graph $G$ and then we find the shortest path on this graph. The most optimistic instantiation of $G$ is obviously the one in which each random edge with a degree of belief $b > \phi$ is substituted by an open (fixed) entrance. The value $\phi$ can be arbitrarily chosen in order to specify how much the character has to be optimistic during the navigation.

This is clearly a non-optimal policy but can provide a sufficient behavior for the character in many cases, especially in video game with many characters and a dynamic environment. As we will see later, this algorithm corresponds to a floating-point version of the BGCA* presented in the previous chapter.

Hindsight Optimization (HOP)

A simple improvement over OMT is given by the hindsight optimization (HOP). After each query for a path, the hindsight optimization approach performs a sequence of iterations called rollouts.

In short, we randomly pick a random instantiation of $G$ of unknown roads using the actual knowledge degree of the character. If the resulting graph is not solvable (that is, there is no path from the starting point to the destination), the rollout counts as failed. Otherwise, the rollout counts as successful and we compute the distance from the character’s location to the goal in the random instance.

At the end we produce the average, over every successful rollout, of the estimated shortest-path (ESP) for each edge starting from the initial state. The resulting policy will be to pick the action corresponding to the minimum ESP.
This algorithm is obviously biased by the fact that the ESP is computed by an omniscient character, i.e., knows ahead of time which roads are traversable and hence follows the shortest goal path. However, this algorithm is very successful in practice.

From a game-development point of view, this algorithm has several advantages. The most important one is that this algorithm is any-time, that is, it can be interrupted any time after the first rollout and still produce a reasonable policy. The more time we spend on this algorithm, the more precise the policy will be. The tradeoff between precision and performance is in total control of the developer.

**Optimistic Rollout (ORO)**

A small but effective refinement over HOP is given by the optimistic rollout (ORO). The algorithm is exactly the same of the HOP version but, instead of using an omniscient character to “simulate” the path over the rollout instance, it simulates the optimistic policy over it. Hence, in each rollout the character follows a shortest path in the optimistic graph until it reaches the goal or a road which is blocked. In the latter case, it recomputes the optimistic distances based on the new information and follows a new path, iterating in this fashion until it reaches the goal. The total distance traveled then serves as the rollout cost. At the end, the average for each edge is computed in a similar manner.

This algorithm is more costly than HOP because it requires a character simulation for every rollout but provides a better estimation of the CTP solution. However, in some situations, the algorithm converges to a non-optimal solution as shown in [80].

The level of precision of this algorithm is rarely useful in videogame setting because it starts to be significant after many high-level steps. In videogame, both target and environment do not stay fixed for so long and a new computation is usually required anyway.

**Upper Confidence Bounds on Tree (UCT)**

For completeness, we introduce also one of the most effective Monte-Carlo algorithm for MDP. The UCT algorithm stands for upper confidence bounds applied to trees and it is presented in [47] as a general algorithm for MDP problems.

The difference between UCT and ORO is in how the character’s movements during each rollout are determined. While each rollout is independent in ORO, this is not the case in UCT. Without going into details, UCT choses its movement in order to maximize the benefit of exploration (trying new path that may be successful or terribly worse) and exploitation (chose the path that for sure provide good results). This allows to have a better sampling of the search space and then leading to a better solution.

However, because this algorithm implementation is much more complex than ORO or HOP, and it provides a level of precision that is not needed in games, it is hardly a good candidate for the BAP planning phase.

### 11.4 The Hierarchical Optimistic BAP algorithm (HO-BAP)

In this chapter, we will present a general algorithm for solving BAP problems based on a hierarchical approach and the Optimistic Policy presented in the previous section: the Hierarchical Optimistic BAP (HO-BAP). This algorithm is modular
such that each of its parts can be replaced with a different approach if needed. For instance, we can replace OMT with HOP or any other Monte-Carlo solution to CTP, or we can replace the abstraction algorithm with any other algorithm we think is more suitable for our setting. For this reason, even if we will provide specific algorithms for each component, we can consider HO-BAP as a “template” for any future BAP improvement and variation.

As described before, this algorithm starts from the idea that a BAP problem is divided into three main phases: planning, execution and perception. The planning phases can be described as a Markov Decision Process and we can find for it an approximated solution using a Monte-Carlo approach. Perception and Execution, on the other hand, are BAP specific problems and we will describe some methods for handling them.

11.4.1 The planning phase

As we said before, the planning phase for a BAP problem can be described as a MDP instance and approximately solved using Monte-Carlo algorithms. In HO-BAP we will use OMT to get the policy.

OMT has several advantages for BAP in games. First, its returning policy is just a path. This reduces the complexity of the system, it is easy to understand for people in the industry and experienced developers in classic pathfinding, and integrates well with the pre-existent infrastructure of game engines. Second, each query is approximately fast as a traditional hierarchical pathfinding query. As we said before, we pay this by obtaining a path that is clearly a sub-optimal policy. However, this is good enough to provide a good illusion of intelligence for the non-player character and convey the core points of BAP.

We now assume that we have a function $OMT(G, \theta, s, d)$ that, given a random graph and a starting location $s$ and goal location $d$ and the character optimistic threshold $\theta$, returns a policy $\pi(n)$ that given the current location tell us the next best action (represented as the best edge to travel). Because OMT returns a path, the corresponding policy for a location $n$ in the path is just the location on the next step and undefined for every other node. However, in order to maintain this step as much as generic as possible, we will keep the policy notation.

In HO-BAP we can set a threshold $\theta$ for the specific character, consider open each edge such that $b_e > \theta$ and search for a path in the resulting graph. In this case the policy $\pi(n)$ is a path from $s$ to $d$.

At this point the character can face two different scenarios:

1. **The policy $\pi(current) = e$ exists.** In this case, the character can start the execution phase by trying to traverse the edge $e$.

2. **The policy $\pi(current)$ does not exist.** In this case, the character must rely on some backup strategy. A failed policy may mean that there is no path from start to destination, but also that there is no path for which the character is confident enough. There are three different solutions at this point:

   - *Go in a random direction.* If no edge is good enough, this also means that we can take any direction without losing much. Going in a random direction can give the character new perceptions that can help the character find a policy in the next iteration. In any case, this will help to provide to the player the idea that the NPC character is actively trying to find a path in a complex situation.
• Go in a greedy direction. If we have no data for choosing among different direction we can fall back to a greedy approach in which the character will move to a region that is nearer to the destination. This can give similar benefits with choosing a random path and in some cases can put the character closest to the destination. However, note that using this strategy the character can move into clear dead ends.

• Algorithm-specific fallback. Depending on the algorithm we develop an ad-hoc solution. For OMT, for instance, we can try another query with a smaller θ. In this way, we have a larger probability to have a path with some open edges.

11.4.2 The execution phase

Once the Execution phase receives the policy and the first action, the character starts the execution. The most common problem that a character can encounter during the execution phase is that the chosen action is not realizable. For instance, a door that was believed to be open during the planning phase is found to be closed. In this case, the character is forced into update its knowledge and replans accordingly or, if the policy is good enough, use the next-best edge.\footnote{When we choose OMT in the planning phase, we cannot have a “next-best edge” because a path can have only one next location for every node in the path.}

However, there are many other cases in which the character needs to decide to replan or switch to a different approach. We call this the execution strategy of the character.

• How frequently to replan? This depends mostly on how good the policy function is and how fast the game world changes. If the game changes very fast, we can decide to force character replan every time it enters a new region. Otherwise, we can continue using the current partial policy until we find a discrepancy.

• How to handle new information? What should the character do when he enters a new region and finds a non-destructive discrepancy? For instance, the character may find that a door that he thought was closed, however, is open. The character can continue with the previous policy because this new information cannot block the character. However, this new information can be important because, for instance, the door may lead to a much shorter path. Measuring the impact of a wrong assumption like this on the path is feasible but outside the scope of this work and outside the practical constraints of the videogame application of BAP. We already explored some more detailed scenarios in Section 9.4.

11.4.3 The perception phase

Another important point in BAP is how the character’s sensing affects the character’s belief vector \(B\) over the random graph, that is, how the probability distribution of \(G\) changes over time in relation to the character actions and perceptions.

There are two main points to analyze as follows.
Sensing and update

The first important modification of the belief-vector $B$ is when the characters can perceive the actual state of the gate. In this case the character can simply set the corresponding edge degree of belief $b_i$ to 1 or 0 according to the actual perceived state of the connection.

Perception in BAP can be of very different types. We can assume that the character is able to perceive instantaneously the state of every entrances of a region as soon as it enters it. Or we can use more realistic but complex solution, such as using a perception sphere around the character so that it can update the entrances state only if they enter in the character radius.

Knowledge decay

The second important point is the knowledge decay. This variation is introduced to model the natural increase of uncertainty about the state of a random edge $e$ over time. In other words, if sufficient amount of time is passed without the character observing the edge state, then the belief degree $b_e$ for that edge approaches 0.5 (that is, the character has no information at all over the state of the door).

In practice, knowledge decay can be modeled with a simple function such as

$$b_e(t + 1) = b_e(t) + \alpha \left[ \frac{1}{2} - b_e(t) \right],$$

where $\alpha \in [0, 1] \subset \mathbb{R}$ is the decay speed parameter who specifies how fast the probability goes to 0.5 ($\alpha = 0$ means no decay at all whereas $\alpha = 1$ means instantaneous decay). However, any decay function can be used if it satisfies the following conditions:

1. $0 \leq b_e(t) \leq 1$ for every $t$.

2. $\lim_{t \to \infty} b_e(t) = 0.5$

The parameter $\alpha$ in (11.3) plays and important role. There are many different and interesting ways with which we can set up it. For instance, we can use the same $\alpha$ for every random edge (and therefore becomes an algorithm parameter) or can be assigned individually to each edge in order to model the fact that the character can have different bias on how fast the state of an obstacle changes during the game. For instance, we can change $\alpha$ according the tagged type of obstacle along the edge. If a door collapse (so that cannot be open again in the future) the character can set this probability to zero and $\alpha = 0$ in order to stop the decay for that door.

Another idea is to make this parameter adaptive by using this formula:

$$\alpha = E^{\omega},$$

where $\omega \in \mathbb{R}^+$ is a generic scale factor and $E \in [0, 1] \subset \mathbb{R}$ is some entropy measure over observations of the state of the edge in the past (or in a temporal window). The idea is that if a door is perceived often in a similar state (so that the entropy $E$ is near to zero), the $\alpha$ value tends to zero and the decay is very slow. On the other hand, if a door changes its state often, the entropy is very high (near 1), so the $\alpha$ is near 1 as well and the decay is very fast.
11.5 Experiment and results for HO-BAP

HO-BAP behavior corresponds to BGCA* with the difference that we are explicitly handling the uncertainty of the different degrees of belief for every obstacle. On the relation between BGCA* and HO-BAP we can make the following observations:

- The combined effect of knowledge decay and θ threshold corresponds to the validity window in BGCA*. When θ < 0.5, knowledge decay progressively frees the edges blocked in the character beliefs. However, unlike the validity window, this time we have a clear meaning for both knowledge decay and θ and, moreover, we can avoid adding a timestamp to the belief vector. In fact, the “validity time frame” is encoded into the α parameter of the decay function.

- If θ > 0.5 then knowledge decay acts no more like the validity windows. Instead, the knowledge decay progressively blocks uncertain edges. This leads to a character that is “scared” by trying edges which it is not certain enough that are free. This raises some additional problems that must be solved with ad-hoc solutions. While this can produce an interesting behavior that is worth exploring, it is safer to choose a θ less than 0.5.

- Uncertainty is not destroyed during search. In BGCA*, the implicit random graph representing the planning phase is continuously instantiated into a realization. Every time the validity window or the search algorithm switches the belief value from “true” to “false” or vice versa, we are creating a new realization of G that replaces the previous one. In HO-BAP this does not happen. We obtain a random graph realization only during the planning phase and it is discarded immediately after. In this way, we maintain the character beliefs in their original continuous form.

As BGCA* and HO-BAP are strictly correlated, we tested HO-BAP on the same test scenario for BGCA*. Then, we tried some combinations for the validity window size and θ and we found that θ = 0.75 and α = 0.15 produce approximately a behavior similar to a validity window of size 6. In fact, such α value is the value that approximately brings a belief equal to 1 under the threshold in 6 iterations.

As a rule of a thumb, we can compute the closed form for the decay function (11.3) of an edge such that \( b_e(0) = 1 \).

\[
b_e(t) = \frac{1}{2} + \frac{1}{2}(1-\alpha)^t
\]

(11.5)

Then, we can find which α put the belief under the threshold θ in W steps by solving the following equation for α.

\[
\frac{1}{2} + \frac{1}{2}(1-\alpha)^W = \theta
\]

(11.6)

The experimental results for that value of α and θ show that the behavior of HO-BAP overlaps to the corresponding BGCA* version.

11.6 Conclusion

We conclude that HO-BAP features most of the benefits of BGCA* but, at the same time, grants to the developers a superior expressive power thanks to the direct
access to the belief vector and knowledge decay. BGCA* is still considered, though, a simple approach that can be implemented easily in existing games.

Also, HO-BAP offers a more solid base point for future BAP enhancement. As we anticipated in the BAP introduction chapter, designers can decide to assign different thresholds or decay $\alpha$ to individual characters and obstacles, they can make knowledge decay adaptive, they can use a more sophisticated and precise solution for the planning phase, and more.

Performance wise, HO-BAP is comparable with BGCA*. As usual, a tuning of the two parameters $\theta$ and $\alpha$ can make a difference between good performance and good behavior of the character. HO-BAP introduces a small overhead for producing a realization of $G$, but this can be efficiently done at runtime or even on-demand while the search algorithm expands a node.
Chapter 12

Conclusions and Future Work

Before proceeding to draw conclusions, we will discuss the possible future outcomes and works that can follow from Smart Pathfinding applications. We already have seen some extension in the previous chapters briefly; now we will try to explore some of them in more detail.

Three main directions can be studied starting from this work: improve the algorithms performance, improve the algorithms expressiveness and improve the tooling. Finally, we can also imagine new types of Smart Pathfinding beyond BAP and IAP.

12.1 Algorithm performance improvements

Incremental IAP

The setting of Inventory-Aware Pathfinding offers a lot of opportunities for exploiting sophisticated pruning and symmetry breaking techniques. In fact, a “vanilla” IAP search method that considers all the search space actually performs a lot of duplicate work when it searches over regions both under the assumption that the character holds an item and under the assumption that he does not. As we seen before, two different item layers $L_I$ and $L_J$ share a great part of the “shape” of the map. Moreover, they differ in a more predictable way depending if $L_I < L_J$, $L_I > L_J$ or if there is no relations among them. In other words, a great deal of the game map is unaffected by the inventory of the character, and search techniques can be developed to take advantage of this observation. Note that here we are not talking about filtering as we did in F-IA-JPS, but about technique that are capable of reusing the search time spent on lower layers.

As a pathological example, consider a map composed by a single room and a blocked door that can be opened but it opens on a 1x1 room (Figure 12.1). Even if the locked door is, in practice, useless, a standard application of IAP search would explore the large room twice: one assuming that the character has no item in his inventory ($L_\emptyset$), and one per every combination of items he may pick up ($L_{\{k\}}$). However, it is clear that the relative distances inside the room remain the same in all cases, so it is strange and inefficient that we search in each case as if it is a completely different room every time. The key idea here is to transfer the knowledge obtained on a lower layer to the upper ones.

There is a lot of related work in the literature about pathfinding heuristic optimization in similar scenarios (we have explored many of these works in Section 2.2). For instance, in incremental heuristic search [53] the heuristic information is propagated in between consecutive searches on a slightly changing map. We can
straightforwardly imagine applying incremental heuristic search in order to propagate the heuristic information in between variants of the game map that differ only in the state of special nodes that open and close by means of items in the inventory of the character.

There are also other interesting optimizations that can be applied to IAP such as the work on the differential heuristic [29, 28], with which we can exploit the relative fixed heuristic between locations as discussed in [75]. The differential heuristic, in particular, appears to fit in very well in IAP setting. The differential heuristic stores the precomputed distance from every vertex to a certain set of locations \( P \) called pivot. Then, when we ask for the heuristic function from a certain \( x \) to \( g \) given a pivot point \( p \), we use the following formula:

\[
h_p(x, g) = |d(x, p) - d(x, g)|
\]  

(12.1)

and, therefore, given all the pivot locations,

\[
h(x, g) = \max \left\{ h_{\text{base}}(x, g), \max_{p \in P} h_p(x, g) \right\}.
\]  

(12.2)

In short, we are paying a better heuristic with the additional cost of storing the pivot distances.\(^1\) In IAP, doors and other moving obstacles are a natural choice for pivots. They, in fact, represent important fixed “landscape points” that may be crucial in the final computation. However, IAP pivots are potentially not distributed in an optimal way due to the fact that they are attached to some in-game meaningful location, e.g., they may be clustered together. It may be interesting to evaluate this approach over IAP problems or to start from the regular differential heuristic approach in order to develop a more specialized differential heuristic that can exploit the precise structure of the IAP LIG topology.

12.2 Improved Smart Pathfinding expressive power

Multi-agent IAP and BAP

Going back to the turn-based tactical game of scenario S1, even when every character can solve IAP problems in a perfect way, we cannot guarantee that, in

\(^1\)Given \( V \) the set of vertices in the pathfinding graph and \( m = |P| \) number of pivots, we need to store \( m|V| \) values in a look-at table. A memory optimized solution is presented in [29].
12.2 Improved Smart Pathfinding expressive power 143

a multi-character setting, the collective behavior achieved this way is optimal or believable for the player.

As an example, imagine that two enemy characters are blocked by a fire in a gateway. Imagine now that one character has a fire extinguisher while the other does not. We can then ask an inventory-aware path for each character. The second character will probably find a path that includes reaching a fire extinguisher in the map, even though the faster (and more reasonable) thing to do is to wait for the other character or ask the other character to put out the fire instead of looking for another extinguisher.

Finding a better solution involves the algorithm to be aware, not only of the character inventory but other characters’ inventories as well. We can see this as a primitive form of communication and collaboration between game character. In any case, the important fact is that we cannot compute the optimal path by performing individual searches. This problem is extensively studied in robotics and there is a lot of related work in the literature of multi-agent collaboration, both on the pathfinding aspect [94, 81] and the collaboration aspect [55, 68]. It is a challenging direction to investigate how we can integrate such reasoning into IAP to solve the communication issue at the same time while searching for the pathfinding solution.

Extending Cooperative A* (Section 2.6) is a good starting point in this direction. The main problem in this case is that the search space can easily become too big for practical computation. In a Cooperative Inventory-Aware A* we will both have duplicate maps expansions in the time direction (as described by Cooperative A*) and in the item direction for each character (as described in IAP) resulting in a massive multidimensional map. A simple idea to address this problem involves considering a single cooperative inventory that is shared among all the characters and a system to handle local constraints (e.g., the other characters need to wait for the character with the fire extinguisher to pass first). In this way, we will have just two additional dimensions, namely time and the global inventory.

Also in the BAP case, it is important to extend reasoning about beliefs into a multi-agent setting. A straightforward example is one that involves character communication. In a simple setting, it is reasonable to assume that characters of the same team can exchange information on the state of the world. For instance, consider the game of scenario S1 with two enemy characters meeting in a room. The first character may have updated information on some region of the map while the second character may have updated information on some other region. When the characters meet, they can revise their beliefs in order to obtain a more up-to-date of the current state of the game map.

There is a long list of approaches for belief revision and agent communication in the AI literature, e.g., dating back to seminal work in [2]. Also, the problem of merging data from different sources is a common problem in databases [40] and semantic web [78]. In the general case, a trust policy needs to be established among the different sources of information.

As with other AI-related aspects, the challenge is to identify appropriate sub-cases of the existing formalisms and extend them or fit them in the game setting in a practical way. The general assumptions are similar to what would be interesting in a videogame, e.g., there are scenarios where the communicating characters need to decide who has the most “trustworthy” beliefs (even among human players). There are also simple approaches to adopt in the navigational setting, for instance, the characters can be optimistic and keep the higher degree of belief on the traversability of a connection, or pessimistic by using the lower value, or use the most recent observation by time-stamping the time of the update.
Combining IAP and BAP.

IAP and BAP try to enrich classical pathfinding in orthogonal ways, and it is easy to think of problem instances that require both approaches at the same time, e.g., a character needs a key in order to open a door but he is not sure if the key is reachable or not.

Under such a scenario, wider types of uncertainty may be interesting to represent and reason with. Until now we have discussed only the case where a node on the map is traversable or not with some degree of belief. But what if we want to represent more complex uncertainty such as that position of an object is uncertain? For example, the character is unsure about the location of the key that is needed in order to reach his destination. Even though some very general AI framework could handle this, the point again is that practical restricted approaches are needed for the videogame setting.

A complete solution to this kind of problem is usually unnecessary, but there are two main reasons why it may be important to start thinking on some framework that is able to represent and capture the problems. First, even if a detailed solution is unfeasible and unnecessary, the knowledge representation structure that are built and automatically kept updated during the execution of the game can be used as heuristics and guides for fast approximated solutions or even reactive approaches (for instance, by switching the behavior of existing behavior trees). Second, it may offer a solution for some gameplay elements we still have not thought about.

About the first point, it is easy to see how IAP and BAP active world representation can be used to fuel an Environment Query System (EQS) that can be used, in turn, by other pathfinding algorithms or decision-making systems. BAP abstraction and beliefs representation can be queried whenever a character needs to choose his next destination, in this case the EQS can greedily select a room that brings the agent to the player and it is believed to be opened. In the same way, IAP item-obstacle navigation graph and the LIG structure can be used to provide a quick way to prune the map search space on the basis of the current character/player inventory.

However, when we start following the route of creating complex multi-purpose smart pathfinding systems, it is mandatory to keep attention to not push smart pathfinding too much into planning. In case such a wide formalism is needed, we can look for more efficient solutions in the generic planning domain.

12.3 Improved Smart Pathfinding tooling

Language for BAP representation.

The BAP paradigm is based on a random graph representation; however this is not a “videogame-friendly” notion. Even though it works well as a formalism, there is also the need to define an appropriate game-independent language for specifying the relevant belief-related aspects of the game in an intuitive way. The idea is that specifications expressed in this language can then be automatically compiled into a random graph, as per the requirements of the BAP phases. With such language, game designers could visually annotate the map with the desired uncertainty elements or use a script-like description, e.g., using intuitions from planning languages such as PDDL [27].

What is still missing is the ability to handle uncertainty about any location at the pathfinding level by integrating a general mechanism in modern game engines. There is the need for a general framework that can take as input any user-defined
uncertainty elements. For instance, game designers could visually specify the BAP-relevant elements on the map and the effects of them on the character navigation or by using a custom language, that may be inspired by planning languages such as PDDL [27]. The system could then automatically use this information to build a belief state for the character and to provide the adequate perception update loop.

For a simple example, we can come back to the scenario in which the only element of uncertainty are the doors connecting different rooms and areas in a big building. In the editing phase, a level designer can simply design the environment as usual. Then, they can annotate doors with a tag indicating that they are source of uncertainty for the characters. Then, a baking algorithm can perform the following actions:

1. As an input, it receives the low-level pathfinding graph (tile map, NavMesh or others).
2. It stores all the tagged BAP obstacles in the map in a list.
3. Assuming all the BAP obstacles as actual obstacles, it searches for all the connected components in the above pathfinding graph. We want, in other words, a list of all the regions that are independent from the state of the obstacles.
4. From the output of the previous point, it creates the belief graph for the characters.
5. It sets up the characters perception volume.

Note that all the requirements for BAP (belief graph, perception volumes and BAP controllers) can be automatically generated from the tagged obstacles and the raw pathfinding graph. After this process the designers only need to attach the BAP perception system to the character that needs to move according BAP rules. In addition, we can store in the BAP objects their nature (how fast the object change state? Is the change permanent?) and parametrize character perception and BAP application in order to obtain a certain behavior (e.g., do we want a more “brave” character or a more “cautious” one?).

For standard BAP, this is just an implementation problem. Instead, the challenge comes from the possibility to apply a BAP system to different uncertainty sources. In this case, we need a more powerful language to express what kind of uncertainty it represents and, more important, how the state of the object affects the navigation graph. For instance, we can try to represent objects whose position is unknown or that may affect the graph in a different position (e.g., the state of a switch opening/closing a door far away).

The uncertainty can be also not linked to physical objects in the map but on a more global state known a runtime. For instance, we can imagine a vampire game in which enemy vampires cannot traverse the sunlit areas of the map. The position of the sun is a dynamic continuous property that, in turns, can affect the shadow on the map. Because the sun position and shadows are predictable, building a belief graph for this scenario is possible, but it is definitely hard to integrate an easy tool that is able to encode this problem as easy as the standard BAP example presented above.

---

2Many popular game engines, such as Unity or Unreal Engine, provide a tagging system with which the designers can annotate game objects on the fly.
12. Conclusions and Future Work

12.4 Wider integration of smart pathfinding

In this section, we discuss some future work directions about integrating smart pathfinding into more high-level decision-making tools and techniques.

IAP and BAP for player modeling

As we stated in this work, the main goal of smart pathfinding is to merge into the pathfinding level some aspects that are typically handled in a higher level of decision making. However, we can imagine also situations in which smart pathfinding can be used to bring pathfinding information in decision making.

For instance, we can use smart pathfinding to build a basic player model. Consider a scenario in which the enemy characters must decide where to ambush the player. For this task, the strategic AI layer needs to answer the question “where will the player pass?”. If the game maintains a belief representation for the player, then we can use BAP in order to obtain the most probable route of the player in the environment by storing information on every passage or door that the players know to be open, and then asking to the BAP algorithm the path from the actual player’s position to the expected destination. The same idea can be applied to IAP, where in a map with can have character setting an ambush in the place around objects that they know–thanks to IAP–are needed for the player in order to advance into the level.

In other words, as we said before, smart pathfinding can be integrated into a Environment Query System used by high-level systems to decide targets both at strategic and individual level.

12.4.1 Selective BAP and IAP application in Behavior-Trees

In this work, we described IAP and BAP as alternatives to traditional pathfinding. In practice, every problem that can be solved by a traditional pathfinding algorithm can be solved by IAP and BAP, but not vice versa. However, despite all the optimizations introduced here or that can be possible to add in the future, smart pathfinding ability to solve additional problems requires additional computational time that cannot be avoided. This additional cost can be small or large, the fact is that it is not possible to guarantee that a game can provide room for this in every situation.

For this reason, we can think at BAP and IAP as additional tool that a character can call only when needed. To illustrate this, we can imagine a Behavior Tree for a game character in which designers provide all the usual reactive solutions to some inventory problem (e.g., if the door is closed, try to find a key in the next room and so on). But if everything fails, we can use IAP as last resort. In practice, we use all the standard pathfinding and decision-making tool but we use IAP as a way to solve problems that are more complex than the one designers thought of at design phase.

12.5 More Smart Pathfinding possibilities

In this work, we talked about two particular application of smart pathfinding: inventory and beliefs aware pathfinding. However, smart pathfinding is not limited to these two purposes. When we say that smart pathfinding is about pushing pathfinding into the proactive decision-making domain, we are referring to a wider spectrum of applications. In this section, we introduce other smart pathfinding domains that can be further explored.
Social-Aware Pathfinding

Consider a life simulation game in which many computer characters need to move in a limited environment. The challenge, in this example, is to create a meaningful and realistic character behavior considering the social and crowd aspects. Characters must move in a multi-agent setting, but, in addition to the goal of reaching a particular location, they must take into account how humans group and, most important, move in a social environment.

Games usually neglect this aspect for performance reasons. There are some insights on how to distribute groups of characters in a map and how to arrange individuals among these groups. However, movement of these groups of individuals from a group to another is usually left to a combination of pathfinding plus local flocking behaviors and obstacle avoidance.

However, *proxemics*, the study of human use of space and the effects of population density on behaviors and social interactions, tell us that the way people move and position themselves relative to one another is an important nonverbal communication process. Therefore, trying to put more attention on this aspect could be very useful in the game-player interaction. The ability for a character to position and move itself in a meaningful way appears to be much more important in the near future with a wider adoption of *Virtual Reality* (VR) games. In such games, the way characters interact with the *intimate, personal* and *social* space of the player can be a very effective way to communicate different emotions.

Outside videogames, *human crowd behaviors* is an increasingly important and active field of study, in particular for robotics and human-robot interaction (HRI). There are different aspects to consider. At the higher level, we are interested in how individuals group together and how these groups interact with each other in a limited environment [62]. Then, we want to see how crowds move in the environment, for instance, when people approach a bottleneck, and the way an individual behavior is correlated to emotions and social interaction [37, 61] and, finally, how two people walk side by side [60].

With this in mind, we can imagine a *Social-Aware Pathfinding*, on the same path of IAP and BAP. Instead of building a pathfinding system that can take into account the character beliefs and the interaction with items on the map, social-aware pathfinding takes into account the agent social relations with other characters. This can address two goals. First, we can have a pathfinding that tunes the output path on the basis of the character’s temper and emotions. Aggressive characters can decide to invade the personal space of other characters more often than sympathetic characters. Second, we can have a pathfinding algorithm that can output a path that avoids characters it dislikes or goes towards characters it likes.

12.6 Conclusions

In this work, we introduced two novel approaches toward *Smart Pathfinding*, namely *Inventory-Aware Pathfinding* (IAP) and *Belief-Aware Pathfinding* (BAP). With Smart Pathfinding, we want to include all methodologies that aim to enhance pathfinding with the capability to interact with objects, other characters and gameplay elements, and not be limited by the “reach the destination avoiding obstacles” core of classic pathfinding. In this context, IAP and BAP are two first formal instances of this idea that we hope can be explored further in future work.

With Inventory-Aware Pathfinding we explored the idea of a pathfinding algorithm that takes into account that the character can interact with objects while searching for a path. While this problem falls in the more general planning domain,
restricting the problem to a specific kind of interaction, we were able to incorporate a small planning subset inside the pathfinding algorithm itself. In the IAP case, we focused on the problem of a map in which certain locations can be accessed only if the character takes a specific item located in a different location first. This problem is inherently exponential in the number of items on the map, however, using a fast pathfinding algorithm as the basis and employing pruning techniques we were able to arrive to a practical solution that can handle using up to 6-8 items to find a path in the worst case.

For IAP we introduced a first algorithm based on Jump-Point Search (JPS), namely Inventory-Aware Jump-Point Search (IA-JPS), that showed promising results for maps with a small number of items on the map (e.g., up to 5). However, the algorithm performance was not practical whenever the final destination was unreachable (Chapter 5). Trying to address this problem, we first introduced more refined pruning mechanisms on the search space in the Filtered Inventory-Aware JPS (F-IA-JPS). These pruning methods sacrifice optimality for performance and are able to push the item limit. Unfortunately, the problems when the destination is unreachable were not solved this way (Chapter 6).

As a final optimization, we introduced a preprocessing step and a pre-search step (Chapter 7) that are able to prune from the computation items that are probably not required to solve the problem. In doing this, we may lose completeness, even though we can recover it by iterating the pre-search step until the algorithm converges to a fixed set of items. The preprocessing algorithm extracts the high-level relations between reachable locations and obstacles and using this information identifies a set of items that must be picked up in order to reach the destination. In this way, we can search a path with F-IA-JPS or any other IAP algorithm but limiting the search to only the items in this set and ignoring the others. The overall effect is that we can limit our F-IA-JPS search only on the smaller set of items that is useful to reach the destination instead of the full set of items. For what is concerning the case in which the destination is not reachable, the preprocessing method can catch this case in the pre-search step, before the real search is even started.

For Belief-Aware Pathfinding, instead, we focus on the problem of characters moving in a dynamic environment on the basis of their own beliefs of the map. That is, if a character knows that a certain path is unlocked, it will try that path even if it is closed in the real world. To tackle this problem, we introduced first a hierarchical algorithm that uses a binary representation of the character’s belief. The algorithm, namely Belief Gate Connectivity A* (BGCA*), is described in Chapter 10. Then, in Chapter 11 we proposed a more refined algorithm, namely Hierarchical Optimistic BAP (HO-BAP) that uses continuous values between 0 and 1 for the belief state of each obstacle. In this way, we can use in the computation the actual degree of belief of the character instead of a discrete boolean value as in BGCA*. This allows more flexibility and expressive power for the developers maintaining the same performance of BGCA*. However, it remains slightly more complex to implement in the setting of current game-engines on top of existing pathfinding systems.

Finally, we want to conclude highlighting that Smart Pathfinding is broad term that can include much more pathfinding ideas other than BAP and IAP. The idea is to imagine pathfinding as a more central part of “search-based” game AI, instead of considering it only as a tool for moving characters from a point A to a point B. There are many other types of smart behaviors that can be achieved beyond the use of items and beliefs. For example, we can imagine a pathfinding algorithm that is influenced by the personalities, goals, desires or moods of the involved characters. We mentioned one of these possibilities in the social-aware pathfinding section above, but all the others are still promising directions that it is worth exploring. Consider a
character who fears light. In a typical case, it decides to follow a longer, but in shade, path in order to reach his destination. However, in some situations, he can decide to take the risk of crossing small bright areas in order to take a much shorter path. How the character decides between alternatives may depend on some personality parameters. Proposing a “light-aware pathfinding” algorithm as a general framework similar to IAP is perhaps very domain-specific and less useful as a general approach. However, we hope that the basic idea behind Smart Pathfinding is clear: restricted navigational planning problems can be embedded in pathfinding itself to offer more informed gameplay-oriented solutions. We can exploit the search time used by the pathfinding system to facilitate and solve specific decision-making problems.

We think that this direction is worth—and fun—to explore and we hope that this work will motivate researchers and game developers to think about pathfinding from an alternative and more AI-oriented point of view.
Bibliography


Videogames