



UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Influence Map-Based Pathfinding Algorithms in Video Games

Michaël Carlos Gonçalves Adaixo

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Abel João Padrão Gomes

Covilhã, Junho de 2014

Influence Map-Based Pathfinding Algorithms in Video Games

Influence Map-Based Pathfinding Algorithms in Video Games

Dedicado à minha Mãe, ao meu Pai e à minha Irmã.

Influence Map-Based Pathfinding Algorithms in Video Games

Dedicated to my Mother, to my Father and to my Sister.

Influence Map-Based Pathfinding Algorithms in Video Games

Acknowledgments

I would like to thank Professor Doutor Abel Gomes for the opportunity to work under his guidance through the research and writing of this dissertation, as well as to Gonçalo Amador for sharing his knowledge with me on the studied subjects.

I want to thank my family, who always provided the love and support I needed to pursue my dreams.

I would like to show my gratitude to my dearest friends, Ana Rita Augusto, Tiago Reis, Ana Figueira, Joana Costa, and Luis de Matos, for putting up with me and my silly shenanigans over our academic years together and outside the faculty for remaining the best of friends.

Influence Map-Based Pathfinding Algorithms in Video Games

Resumo

Algoritmos de pathfinding são usados por agentes inteligentes para resolver o problema do caminho mais curto, desde a área jogos de computador até à robótica. Pathfinding é um tipo particular de algoritmos de pesquisa, em que o objectivo é encontrar o caminho mais curto entre dois nós. Um nó é um ponto no espaço onde um agente inteligente consegue navegar. Agentes móveis em mundos físicos e virtuais são uma componente chave para a simulação de comportamento inteligente. Se um agente não for capaz de navegar no ambiente que o rodeia sem colidir com obstáculos, não aparenta ser inteligente. Consequentemente, pathfinding faz parte das tarefas fundamentais de inteligencia artificial em vídeo jogos.

Algoritmos de pathfinding funcionam bem com agentes únicos a navegar por um ambiente. Em jogos de estratégia em tempo real (RTS), potential fields (PF) são utilizados para a navegação multi-agente em ambientes amplos e dinâmicos. Pelo contrário, os influence maps não são usados no pathfinding. Influence maps são uma técnica de raciocínio espacial que ajudam agentes inteligentes e jogadores a tomar decisões sobre o decorrer do jogo. Influence maps representam informação de jogo, por exemplo, eventos e distribuição de poder, que são usados para fornecer conhecimento aos agentes na tomada de decisões estratégicas ou táticas. As decisões estratégicas são baseadas em atingir uma meta global, por exemplo, a captura de uma zona do inimigo e ganhar o jogo. Decisões táticas são baseadas em acções pequenas e precisas, por exemplo, em que local instalar uma torre de defesa, ou onde se esconder do inimigo.

Esta dissertação foca-se numa nova técnica que consiste em combinar algoritmos de pathfinding com influence maps, afim de alcançar melhores performances a nível de tempo de pesquisa e consumo de memória, assim como obter caminhos visualmente mais suaves.

Palavras-chave

Inteligencia Artifical, Pathfinding, Spatial Reasoning, Influence Maps

Influence Map-Based Pathfinding Algorithms in Video Games

Abstract

Path search algorithms, i.e., pathfinding algorithms, are used to solve shortest path problems by intelligent agents, ranging from computer games and applications to robotics. Pathfinding is a particular kind of search, in which the objective is to find a path between two nodes. A node is a point in space where an intelligent agent can travel. Moving agents in physical or virtual worlds is a key part of the simulation of intelligent behavior. If a game agent is not able to navigate through its surrounding environment without avoiding obstacles, it does not seem intelligent. Hence the reason why pathfinding is among the core tasks of AI in computer games.

Pathfinding algorithms work well with single agents navigating through an environment. In real-time strategy (RTS) games, potential fields (PF) are used for multi-agent navigation in large and dynamic game environments. On the contrary, influence maps are not used in pathfinding. Influence maps are a spatial reasoning technique that helps bots and players to take decisions about the course of the game. Influence map represent game information, e.g., events and faction power distribution, and is ultimately used to provide game agents knowledge to take strategic or tactical decisions. Strategic decisions are based on achieving an overall goal, e.g., capture an enemy location and win the game. Tactical decisions are based on small and precise actions, e.g., where to install a turret, where to hide from the enemy.

This dissertation work focuses on a novel path search method, that combines the state-of-the-art pathfinding algorithms with influence maps in order to achieve better time performance and less memory space performance as well as more smooth paths in pathfinding.

Keywords

Artificial Intelligence, Pathfinding, Spatial Reasoning, Influence Maps

Influence Map-Based Pathfinding Algorithms in Video Games

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem Statement	1
1.3	Scheduling of the Research Work	2
1.4	Contributions	2
1.5	Organization of the Thesis	3
1.6	Target Audience	3
2	State-of-the-Art in Pathfinding	5
2.1	Introduction	5
2.2	Pathfinding	5
2.3	Search Space Representation	6
2.3.1	Grid-based	6
2.3.2	Waypoint-based	7
2.3.3	Mesh-based	9
2.4	Search Algorithms	10
2.4.1	Dijkstra's Algorithm	10
2.4.2	A* Algorithm	14
2.4.3	Best-First Search	19
2.4.4	Fringe Search Algorithm	20
2.5	Final Remarks	20
3	Influence Map-Based Pathfinding	23
3.1	Potential Fields	23
3.1.1	Overview	23
3.1.2	Potential Field Types	24
3.1.3	Local Minima	25
3.2	Influence Maps	26
3.2.1	Overview	27
3.2.2	Representations	29
3.2.3	Parameters	30
3.2.4	Detailed Algorithm	33

Influence Map-Based Pathfinding Algorithms in Video Games

3.3 Pathfinding with Influence Maps	35
3.3.1 Leading Idea	35
3.3.2 Dijkstra's Algorithm with Influence Maps	37
3.3.3 A* Algorithm with Influence Maps	38
3.4 Tests and Discussion	40
3.5 Results	42
3.6 Final Remarks	44
4 Conclusions	45
4.1 Comparative Remarks	45
4.2 Future Work	45
Bibliography	47
A Map Dataset and Results	53

List of Figures

2.1	Different grid-based map with 4 and 8 neighbors.	7
2.2	Waypoint-based map.	8
2.3	Visibility graph search space representation.	8
2.4	Visibility graph search space representation with overly complex graph.	9
2.5	Search space representation of a navigational mesh.	9
2.6	Navigational mesh with different node locations.	10
2.7	Dijkstra's algorithm search path.	12
2.8	Point-to-point Dijkstra's algorithm search path.	13
2.9	A* Algorithm Search Path.	15
2.10	A* search path results using different heuristics.	17
2.11	Search space explication comparison.	19
3.1	Attractive and repulsive force fields.	24
3.2	Opposing charges in same potential fields.	24
3.3	Asymmetric Potential Field. The path avoids facing the turret [Sul11].	25
3.4	Different types of potential fields.	25
3.5	Local minima problem with U-shaped obstacle.	26
3.6	Influence map representations over a game world.	27
3.7	Visual representation of influence values.	27
3.8	Repulsive and attractive propagators.	28
3.9	Possible final influence map representation.	29
3.10	Influence maps grid representations [Cha11].	30
3.11	Influence maps area and waypoint graph representations [Cha11].	31
3.12	Momentum comparison.	31
3.13	Comparison of exponential decay with different decay values.	32
3.14	Comparison of different exponential decay values.	33
3.15	Representation of grid based game environment and grid based influence map.	36
3.16	Dijkstra algorithm search comparison.	37
3.17	Dijkstra algorithm search comparison with influence maps.	38
3.18	A* algorithm search comparison.	39
3.19	A* algorithm search comparison with influence maps.	40

Influence Map-Based Pathfinding Algorithms in Video Games

3.20 Maps from Dragon Age: Origins	41
3.21 Maps from Baldur's Gate II.	41
3.22 HOG2 game maps with influence maps.	42
3.23 Overall results.	43
3.24 Overall gain results.	43
A.1 Map "AR0603SR".	54
A.2 Map "AR0603SR" test results.	54
A.3 Map "orz200d".	55
A.4 Map "orz200d" test results.	55
A.5 Map "den502d".	56
A.6 Map "den502d" test results.	56
A.7 Map "bcr505d".	57
A.8 Map "bcr505d" test results.	57
A.9 Map "lgt604d".	58
A.10 Map "lgt604d" test results.	58
A.11 Map "lak302d".	59
A.12 Map "lak302d" test results.	59
A.13 Map "AR0202SR".	60
A.14 Map "AR0202SR" test results.	60
A.15 Map "AR0411SR".	61
A.16 Map "AR0411SR" test results.	61
A.17 Map "AR0011SR".	62
A.18 Map "AR0011SR" test results.	62
A.19 Map "AR0400SR".	63
A.20 Map "AR0400SR" test results.	63
A.21 Map "den520d".	64
A.22 Map "den520d" test results.	64
A.23 Map "lak510d".	65
A.24 Map "lak510d" test results.	65
A.25 Map "lgt602d".	66
A.26 Map "lgt602d" test results.	66
A.27 Map "brc997d".	67
A.28 Map "brc997d" test results.	67

Influence Map-Based Pathfinding Algorithms in Video Games

A.29 Map "den600d".	68
A.30 Map "den600d" test results.	68
A.31 Map "arena2".	69
A.32 Map "arena2" test results.	69
A.33 Map "AR0701SR".	70
A.34 Map "AR0701SR" test results.	70
A.35 Map "AR0414SR".	71
A.36 Map "AR0414SR" test results.	71
A.37 Map "brc501d".	72
A.38 Map "brc501d" test results.	72
A.39 Map "AR0406SR".	73
A.40 Map "AR0406SR" test results.	73

List of Tables

3.1 Average of all test results.	42
3.2 Overall Speedup.	43
A.1 Map "AR0603SR" test results.	54
A.2 Map "orz200d" test results.	55
A.3 Map "den502d" test results.	56
A.4 Map "bcr505d" test results.	57
A.5 Map "lgt604d" test results.	58
A.6 Map "lak302d" test results.	59
A.7 Map "AR0202SR" test results.	60
A.8 Map "AR0411SR" test results.	61
A.9 Map "AR0011SR" test results.	62
A.10 Map "AR0400SR" test results.	63
A.11 Map "den520d" test results.	64
A.12 Map "lak510d" test results.	65
A.13 Map "lgt602d" test results.	66
A.14 Map "brc997d" test results.	67
A.15 Map "den600d" test results.	68
A.16 Map "arena2" test results.	69
A.17 Map "AR0701SR" test results.	70
A.18 Map "AR0414SR" test results.	71
A.19 Map "brc501d" test results.	72
A.20 Map "AR0406SR" test results.	73

Influence Map-Based Pathfinding Algorithms in Video Games

Acronyms

UBI	Universidade da Beira Interior
AI	Artificial Intelligence
PF	Potential Fields
APF	Artificial Potential Fields
RTS	Real-Time Strategy
BFS	Best-First Search
DFS	Depth-First Search
RPG	Role Playing Game
FPS	First Person Shooter
CPU	Central Processing Unit
RAM	Random Access Memory
OS	Operating System

Chapter 1

Introduction

Artificial intelligence (AI) has many definitions, but D. Poole [PMG98] describes it as "the study and design of intelligent agents". An intelligent agent being an autonomous entity that analyses its environment, makes decisions, and takes action accordingly in order to achieve its goal or objective [RNC⁺95]. AI goals include reasoning, learning, natural language processing, knowledge, perception, the ability to move and manipulate objects, and planning, which is used for pathfinding [Nil98].

1.1 Motivation

Path search algorithms, i.e., pathfinding algorithms, are used to solve shortest path problems by intelligent agents, ranging from computer games and applications to robotics. Pathfinding is a particular kind of search, in which the objective is to find a path between two nodes. A node is a point in space where an intelligent agent can travel. Moving agents in physical or virtual worlds is a key part of the simulation of intelligent behavior. If a game agent is not able to navigate through its surrounding environment without avoiding obstacles, it does not seem intelligent. Hence the reason why pathfinding is among the core tasks of AI in computer games.

Pathfinding algorithms work well with single agents navigating through an environment. In real-time strategy (RTS) games, potential fields (PF) are used for multi-agent navigation in large and dynamic game environments. On the contrary, influence maps are not used in pathfinding. Influence maps are a spatial reasoning technique that helps bots and players to take decisions about the course of the game. Influence map represent game information, e.g., events and faction power distribution, and is ultimately used to provide game agents knowledge to take strategic or tactical decisions. Strategic decisions are based on achieving an overall goal, e.g., capture an enemy location and win the game. Tactical decisions are based on small and precise actions, e.g., where to install a turret, where to hide from the enemy.

This dissertation work focuses on a novel path search method, that combines the state-of-the-art pathfinding algorithms with influence maps in order to achieve better time performance and less memory space performance as well as more smooth paths in pathfinding.

1.2 Problem Statement

A game engine is divided into a series of sub-systems, namely, graphics, sound, physics, networking, input, storytelling, artificial intelligence, etc. With the tendency to give more importance to graphics, game engines tend to consume and constrain the computational power of other game sub-systems, including artificial intelligence [BF03] [BF04].

Computer games are constantly pushing the limits of computers, consoles and mobile hardware, running at 30, 60 or even more frames per second (FPS). The amount of processor cycles and memory available varies from gaming platform, i.e., consoles, computers, mobile. Due to resource allocation sharing between game development sub-systems, game AI developers find themselves making trade-offs between memory space and speed in the design of the game AI, often choosing sub-optimal solutions [Buc05], as long as the game AI continues to be perceived as intelligent. Even so, computer games continue to increase in complexity [LMP⁺13], i.e., richer and bigger environments, more game entities per scenes, increased level of detail, thus slowing down the performance of pathfinding.

Therefore, faster and better pathfinding methods are needed to cope with processing and memory limitations and deliver results in real-time.

1.3 Scheduling of the Research Work

The research work that has led to the present dissertation involved a number of steps, namely:

- A continuous and comprehensive study of the literature regarding the subjects of artificial intelligence, namely, pathfinding, agent behavior, motion planning and strategic reasoning.
- A implementation of classical search algorithms, more specifically, Dijkstra's algorithm and A* algorithm.
- A implementation of spatial reasoning using influence maps.
- A novel implementation of classical search algorithms (i.e., Dijkstra's and A*), using influence maps.
- The writing of this dissertation.

1.4 Contributions

As far we are aware of, there is no algorithm combining influence maps with pathfinding algorithms, namely Dijkstra's algorithm and A* algorithm. Note that these two techniques have been used so far to address distinct issues.

The main contributions of this dissertation is the use of both technologies for smooth pathfinding solutions. On one hand, influence maps provide information layers of the game environment, whether it is used for tactics, strategic, or event-type information, e.g., sound or noise propagation, grenades exploding, war zones, etc.. On the other hand, Dijkstra's algorithm and A* algorithm are used for pathfinding.

The combination of these pathfinding algorithms with influence maps results in dynamic game agents, able to respond according to changes in the environment or in-game events, for example, when something in the scene is destroyed. Moreover, the resulting search uses less memory, converges faster to a solution, and naturally smooths the resulting path.

Influence Map-Based Pathfinding Algorithms in Video Games

Finally, let us mention that we have recently submitted the following paper for publication:

M. Adaixo, G. Amador, and A. Gomes, "Influence Map-Based Pathfinding Algorithms in Video Games" (submitted to IEEE Transaction on Computational Intelligence and AI in Games).

1.5 Organization of the Thesis

In general terms, this dissertation consists of four chapters, and is organized as follows:

- Chapter 1 is the current chapter. It introduces the field of game artificial intelligence, specifically focusing on motion planning, spatial reasoning and pathfinding.
- Chapter 2 deals with previous work done in game artificial intelligence, specifically pathfinding search techniques, search space representations and general optimization used in pathfinding search algorithms.
- Chapter 3 describes the technique that allows as to combine pathfinding with influence maps.
- Chapter 4 draws relevant conclusions about this dissertation and presents some issues for future work.

1.6 Target Audience

The scope of this dissertation lies in artificial intelligence in video games, specifically in the fields of pathfinding. This dissertation is likely relevant to all programmers and researchers in these knowledge fields and to all who are interested in applying pathfinding algorithms video games and/or robotics.

Influence Map-Based Pathfinding Algorithms in Video Games

Chapter 2

State-of-the-Art in Pathfinding

Commercial games are a middle ground between real life problems and highly abstracted academic benchmarks, which makes them an excellent testbed for artificial intelligence (AI) research. Among the many AI techniques and problems relevant to games, namely planning, natural language processing and learning, pathfinding stands out as one of the most important applications of AI research [LMP⁺13].

2.1 Introduction

In video games, artificial intelligence is used to simulate human-like intelligence and behavior of non-player characters (NPCs). The term game AI is often referred to a broad set of algorithms that include techniques from control theory, robotics, computer graphics, and computer science in general. Game AI for NPCs is centered on appearance of intelligence and good gameplay within environment restrictions (e.g., other units, obstacles), its approach is very different from that of traditional AI. Workarounds and cheats are acceptable and in many cases, the computer abilities must be toned down to give human players a sense of fairness [LMP⁺13].

In video games, avatars often require to move from one point to another in the environment. An avatar is a virtual representation of a game unit, whether it is controlled by a player (i.e., units) which enables the interaction with the world, or controlled with artificial intelligence, referred to as NPCs (i.e., agents, bots).

Search algorithms achieves success across a very wide range of domains. The success and generality of search for producing apparently strategic and human competitive behavior points to the possibility that search might be a powerful tool in finding strategies in video game AI. This remains true in multiplayer online strategy games where AI players need to consistently make effective decisions in order to foster the player fun. Search is already well embedded in the AI of most video games, with A* pathfinding present in most, and ideas such as procedural generation [JTB11] gaining traction [LMP⁺13].

2.2 Pathfinding

Pathfinding is defined as the search for the shortest path between two nodes in a graph. A path connects two nodes, A and B, through a set of arcs, and nodes. A node is a component that is used to represent a state and position and an arc (also known as edge) is a connection or reference from one node to another. The node that connects with an arc is a successor node.

In video games, pathfinding is necessary because we often require our avatars to move from one point to another in the game world. An avatar is a virtual representation of a game unit,

whether it is controlled by a player which enables the interaction with the world, or controlled with artificial intelligence, referred to as NPC (Non Playable Character), or BOT. Whenever a NPC needs to take an action such as changing its position, a path to their destination needs to be calculated while taking in consideration numerous factors, namely obstacles and travel costs, e.g, money, distance, time, or any other variable.

Graph search algorithms are algorithms that attempt to find paths between two nodes. If the path between the two nodes is the shortest path possible, it is said that the path is the optimal path [HNR68]. An algorithm is said complete, if there is a path between two nodes, and it successfully finds it. A near-optimal algorithm has a high probability of finding the optimal path, but does not guarantee it. While calculating a path for one NPC may be fast, in many games, for instance in real-time strategy games, where there may be hundreds of units, calculating a path for each unit will slow down gameplay or even turn the game unplayable, so any computational or memory optimization must be considered to minimize processing cost.

To search a path, we need to have a search space representation of our game world or game map. These representations are navigation graphs, i.e. waypoint-based, mesh-based, grid-based. The next section introduces the different types of search space representations.

2.3 Search Space Representation

Search space representation refers to the data structure that represents the graph containing the nodes of game world accessible areas. The data structure contains information about the world environment, including obstacles, enemies, points of interest and others. Search space representation can make a tremendous difference in path quality and performance, depending on which data structure is used; for instance, the number of nodes has an impact on path calculation time and memory costs. In short, algorithms tend to reach to a solution faster when fewer nodes are explored.

2.3.1 Grid-based

Most games run pathfinding algorithms on rectangular tile grids (e.g., The Sims, Age of Empire and Baldur's Gate). Once the optimal path is found, smoothing is done on the path to make it look more realistic [S00]. Grid-maps use a regular subdivision of the world into small shapes called tiles. Each tile has a positive weight that is associated with the cost to travel into that tile [Yap02].

Usually, a game agent is allowed to move in the four compass direction on a tile, see Figure 2.1 (a). However it is possible to also include the four diagonal directions as shown in Figure 2.1 (b). Non-square grid maps exist also in form of hexagonal tiles, see Figure 2.1 (c), and triangular tiles, as shown in Figure 2.1 (d).

Although grids are useful and easy to setup, they hold a huge drawback. Real-Time strategy (RTS) and Role Playing Games (RPG) can use hundreds of thousands of tiles to represent their game worlds. The resulting graphs can be extremely large, costly to search and take up large amounts of memory. Yngvi Bjornsson et al. [BEH⁺03] show that A* pathfinding algorithm performs better on octile grids, 8-connected tiles, and that IDA* pathfinding performs better on 4-connected tile

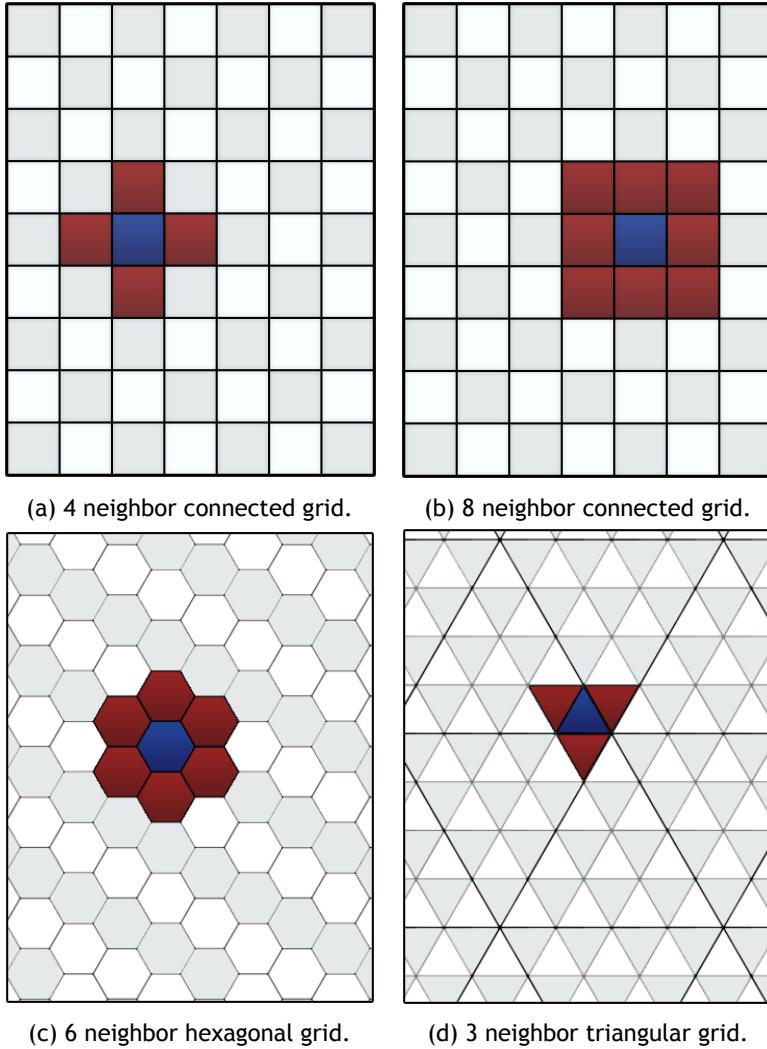


Figure 2.1: Different grid-based map with 4 and 8 neighbors.

grids.

2.3.2 Waypoint-based

Waypoint-based maps, also known as navigation graph or navgraph, represent all the locations in a game environment the game agents may visit and all the connections between them, as shown in Figure 2.2. Each node represents the position of a key area or object within the environment and each edge represents the path that a game agent can use to travel between those nodes. Furthermore, each edge has an associated cost. But, waypoint-based maps have a limitation, in which many areas of the game world have restricted access since there is path to them. Navigational meshes solve this problem by offering a more complete solution on navigable areas, as described in Section 2.3.3.

Waypoint-based maps can be constructed automatically via the visibility graph algorithm [AAG⁺85] [AAG⁺86]. A visibility graph (see Figure 2.3) is a set of pairs of vertices, points or nodes that can be seen from each other. If the world map is static (i.e., the map does not change over time during gameplay), this algorithm may be good enough, but not for dynamic maps that change over time. Using this latter technique requires that before every graph search, arcs have to be

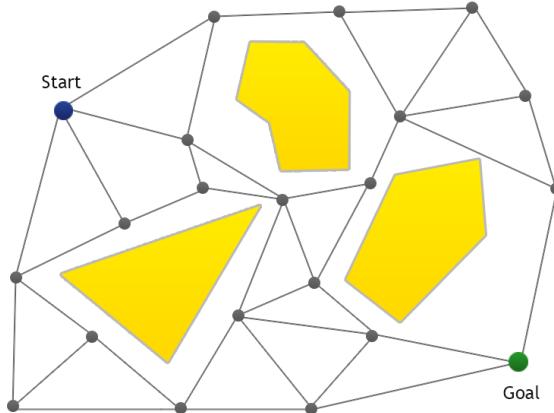


Figure 2.2: Waypoint-based map.

created to the starting node and goal node since their positions are dynamic. These nodes check all lines of sight to each existing node in the graph and add the connecting edges between them (see Figure 2.3). Upon adding the edges between all nodes, travel costs must be calculated using a heuristic (see Section 2.4.2.4). Once the graph search is complete, the starting node and goal node's edges are removed from the graph. For each new graph search, the line of sight algorithm has to be re-applied, leading to a increase in computational cost.

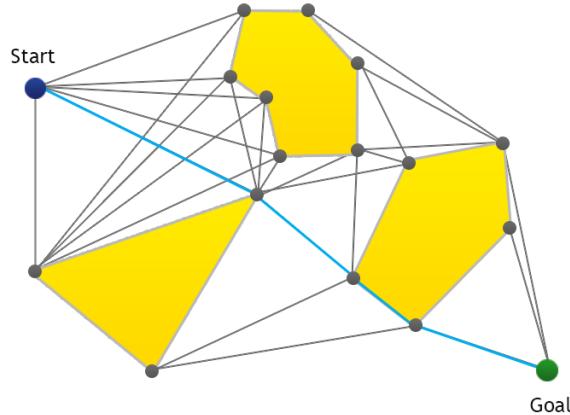


Figure 2.3: Visibility graph search space representation.

Compared to grid-based maps, waypoint maps use much less nodes to represent the walkable areas of game agents, greatly reducing memory costs and computational costs. Unfortunately, depending on the map (e.g., maps with many obstacles), a visibility graph may become complex very quickly. A great disadvantage of connecting every pair of nodes is that if there are N vertices, the graph ends up with N^2 edges.

The increasing complexity of these maps led to the use of graph simplification algorithms by removing redundant edges (see Figure 2.4). A possible optimization would be to look for nodes that are nearby, use a reduced visibility graph, or using a navigational mesh as described below.

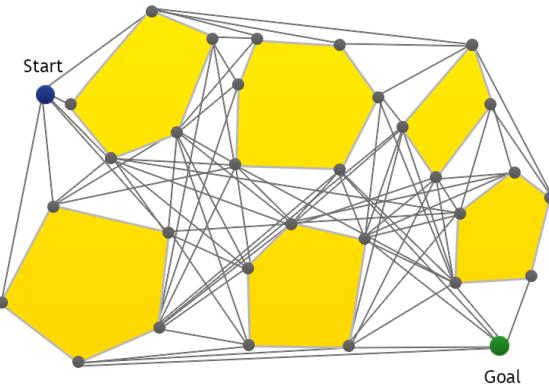


Figure 2.4: Visibility graph search space representation with overly complex graph.

2.3.3 Mesh-based

A navigation mesh (or navmesh, for short), is a set of convex polygons that describes the walkable area of a 3D environment [Boa02]. In other words, navigation meshes are commonly used to represent the walkable geometry within a virtual environment [OP11].

Algorithms have been developed to abstract the information required to generate navigation meshes for any given map. Navigation meshes generated by such algorithms are composed of convex polygons which when assembled together represent the shape of the map analogous to a floor plan [GMS03], as shown in Figure 2.5. The polygons in a mesh have to be convex since this guarantees that avatars can move in a single straight line from any point in one polygon to the center point of any adjacent polygon [Whi02].

Compared with a waypoint graph, the navmesh approach is guaranteed to find a near optimal path by searching much less data, and the pathfinding behavior in a navmesh is superior to that in a waypoint graph [Ton10].

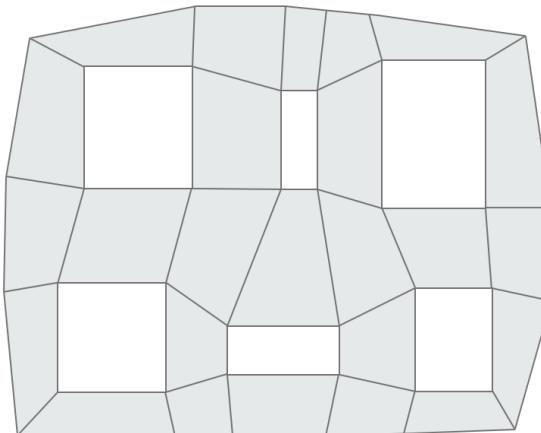


Figure 2.5: Search space representation of a navigational mesh.

There are many possibilities when trying to navigate through navmeshes, namely polygon center-based movement, edge center-based movement, vertex-based movement and hybrid combinations.

Polygon center-based movement, as illustrated in Figure 2.6 (a), uses the center of each polygon as a graph node position. Figure 2.6 (b) shows edge center-based movement which consists in moving through the edges between adjacent polygon, where the center of the edges serve as a graph node position. Vertex-based movement, as shown in Figure 2.6 (c), moves through the vertex of the non-defined areas, just like waypoint-based visibility graphs, although this alternative doesn't connect nodes in straight lines. Lastly, a hybrid combination of the previous methods can be employed, as illustrated in Figure 2.6 (d).

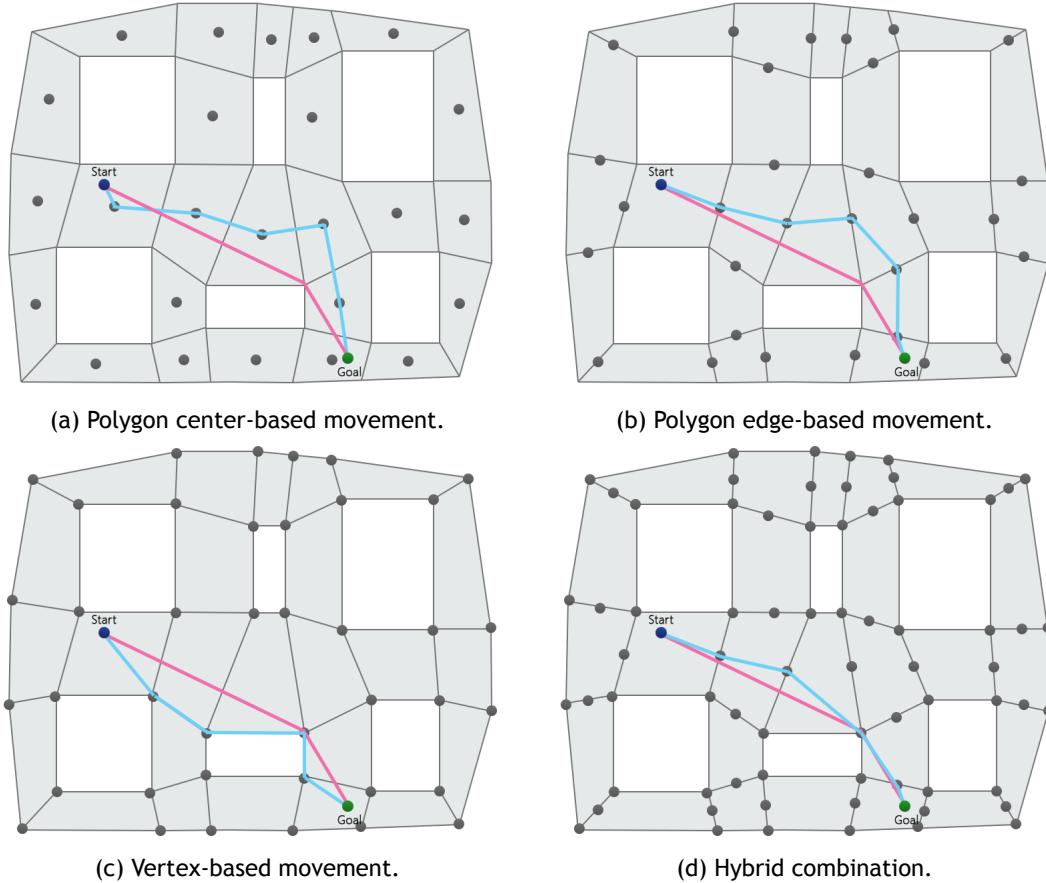


Figure 2.6: Navigational mesh with different node locations.

Figure 2.6 shows the methods described above, where the nodes are represented as grey dots, the blue path is the optimal path, and the pink path is a visually optimal path.

2.4 Search Algorithms

This section describes some of the most known algorithms for pathfinding, in chronological order, Dijkstra's algorithm, A*, Best-first search and Fringe search.

2.4.1 Dijkstra's Algorithm

In 1959, Edsger Dijkstra [Dij59] introduced a method to find the shortest path between two nodes of a weighted graph, where each edge would contain a positive traversal cost. Dijkstra's algorithm is both complete and optimal.

2.4.1.1 Leading Idea

Dijkstra's algorithm explores the whole graph while keeping track of the cost so far (CFS) value C of each node from the starting node. C values allows to know how far is the currently explored node from the starting node. At each node, the cost to travel to a successor node is added to its C cost. Since multiple paths can exist to a single node, successor nodes may already have a G cost, meaning that there is an existing path to that node, at which point an evaluation of a better path needs to be done. A new C cost is calculated from the current node to the successor node, and in case it has a lower value, the path is updated. This guarantees that the returned path is the shortest path possible, the optimal path.

During the search, Dijkstra's algorithm encounters nodes that fall into different states: unseen, unvisited and visited. When an unseen node is encountered, its C value is calculated. This unseen node now needs to be explored, so it is placed into a list, called the open list, which contains all nodes that need to be explored. A node is removed from the open list in order to be explored, and added to another list, called the closed list, which represents the nodes that have been visited, but visited once. The node popped from the open list to be explored is the node with the lowest C value. This means that it is the node that is closer to the starting node, that is, the shortest path so far. When there are no more nodes in the open list, the algorithm returns the shortest path from the starting node to the goal node.

2.4.1.2 Detailed Description

Dijkstra's algorithm is detailed in Algorithm 1. Dijkstra's algorithm starts by setting the start node's parent reference to null and its C cost to 0 (line 1). The starting node is added to the open list (line 2). For each iteration, while the open list is not empty, the algorithm removes the node with the lowest cost so far value from the open list and sets it to the current node N (lines 3-4). For each successor node S of N, a new cost so far value is calculated, $C(s) = C(n) + \text{traversal cost}$ (lines 5-6). Then, if S is unexplored, S's cost so far value is set to the new $C(s)$ value. S's parent is set to N, and S is added to the open list (lines 7-10). If S is in the open list, therefore, explored, and its cost so far value is greater than the new cost so far. S is updated with the new cost so far value, and S's parent reference is set to N (lines 11-13). If S is in the closed list, and S's cost so far value is greater than the new cost so far value. S's cost so far is updated with the new cost so far value. S's parent reference is set to N, and S is removed from the closed list and inserted into the open list (lines 14-18). N is then added to the closed list (line 19). Finally, the algorithm reconstructs the path (line 20).

The memory costs of Dijkstra's algorithm accounts for of the data contained in each node in the graph and the memory costs of the open and closed lists.

Dijkstra's algorithm can still be optimized. At each iteration, since the closest node to the start node is explored, a closer path to a closed node is highly unlikely to be found [Ang12]. Therefore, successor nodes that are already in the closed list can be ignored in the search process, what leads to a more concise algorithm. Removing lines 14 to line 18 from Algorithm 1 would be enough to reflect these changes.

Figure 2.7 shows Dijkstra's search and resulting search space exploration. Bright green node is the starting node, whereas the red node is the goal node. Light blue nodes are explored nodes inserted in the closed list. Light green nodes are called frontier nodes and are seen nodes that

Algorithm 1 Dijkstra's Search Algorithm

```

1: Clear start node's parent and C cost.
2: Add start node to open list
3: while open list is not empty do
4:   n = best node from open list
5:   for all Successor nodes S of n do
6:     Calculate new C cost
7:     if S is unexplored then
8:       Set S's C cost to new C cost
9:       Set S's Parent to n
10:      Add S to open list
11:    else if S is on the open list and S's C cost is greater than new G cost then
12:      Set S's C cost to new C cost
13:      Set S's Parent to n
14:    else if S is on the closed list and S's C cost is greater than new G then
15:      Set S's C cost to new C cost
16:      Set S's Parent to n
17:      Remove S from closed list
18:      Add S to open list
19:    Add N to closed list
20:  Reconstruct path

```

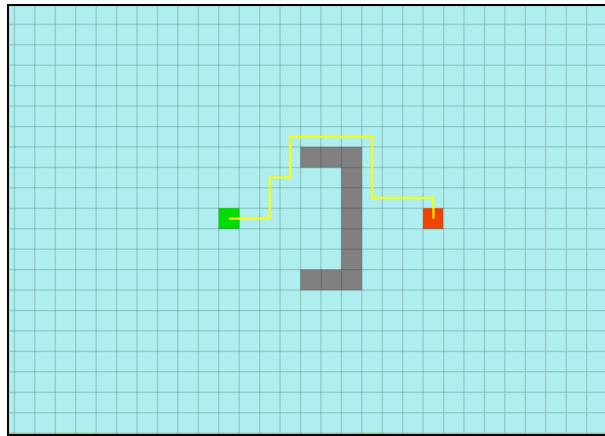


Figure 2.7: Dijkstra's algorithm search path.

are in the open list, waiting to be explored. Grey nodes represent non-walkable position of the grid map (e.g., walls). The yellow line represents the path from the starting node to the goal node.

2.4.1.3 Point-to-Point Variant

Point-to-point version of Dijkstra's is a variant of Dijkstra's algorithm for video games, since video games usually rely on point-to-point searches (see Figure 2.8). The original version of Dijkstra's algorithm is an all-points search, finding the shortest path from the start node to every single node in the graph. Dijkstra's algorithm explores the entire search space even after it has found the shortest path to the goal. In most cases, this leads to unnecessary computations, and the point-to-point version of Dijkstra's algorithm solves this problem and optimizes it for video games offering significant improvements over the original method [Ang12].

This version is based on the fact that once a node is visited, no shorter paths to that node

Influence Map-Based Pathfinding Algorithms in Video Games

may exist. This means that if the algorithm terminates as soon as the goal node has been explored, then the current path found to the goal is guaranteed to be the shortest path. This prevents further exploration of the search space. The algorithm remains optimal and complete, while reducing the number of explored nodes, reducing also processing and memory costs and improving search times.

The goal node may be found early on the search, but may not be immediately explored. This is due to other nodes being closer to the starting node than the goal node. In this case, Dijkstra's algorithm will continue the search and explore nodes closer than the starting node postponing the exploration of the goal node. This extra search can be prevented by simply terminating the search as soon as the goal node is found, but this early termination will not guarantee the optimality of the algorithm.

Algorithm 2 shows the improvements made over Dijkstra's original algorithm. Lines 14 to 18 were removed from Algorithm 1 and lines 5 and 6 were added to Algorithm 2. Additions to the algorithm have to do with checking the goal node, when the best node is selected from the open list. In case the node is the goal node, the algorithm reconstructs the path and returns it.

Algorithm 2 Point to Point Dijksta's Search Algorithm

```
1: Clear start node's parent and G cost.  
2: Add start node to open list  
3: while open list is not empty do  
4:   n = best node from open list  
5:   if n is goal then  
6:     Reconstruct path  
7:   for all Successor nodes S of n do  
8:     Calculate new G cost  
9:     if S is unexplored then  
10:      Set S's G cost to new G cost  
11:      Set S's Parent to n  
12:      Add S to open list  
13:    else if S is on the open list and S's G cost is greater than new G cost then  
14:      Set S's G cost to new G cost  
15:      Set S's Parent to n  
16:    Add N to closed list  
17: Reconstruct Path
```

Figure 2.8 shows Dijkstra's point to point search and resulting search space exploration.



Figure 2.8: Point-to-point Dijksta's algorithm search path.

2.4.2 A* Algorithm

The A* algorithm was introduced in 1968 by Peter Hart, Nils Nilson and Bertram Raphael [HNR68] as a variant of Dijkstra's algorithm. However, unlike Dijkstra's algorithm, A* does not ensure that the shortest path is found. Nevertheless, it tends to find the shortest path, using for that the heuristic of the later best-first search [Pee84] (see Section 2.4.3), which directs the search to the goal. A* is one of the most popular search algorithms, not only in games (e.g., pathfinding), but also in other single-agent search problems, such as AI planning, and used to be the de facto standard in pathfinding search. Both the memory and the running time can be important bottlenecks for A*, especially in large search problems [LMP⁺13]. A* returns an optimal solution (i.e., admissible) when all cost estimates are not higher than the real cost [DP85].

2.4.2.1 Leading Idea

A* algorithm explores a search graph in a best-first manner, growing an exploration area from the start node until the goal node is found, while remembering the visited nodes, to be able to avoid re-exploration of nodes and to reconstruct a solution path to the goal.

The A* algorithm works by combining the C value used in Dijkstra's algorithm and the heuristic estimate, H, to the goal, into a total estimated cost, referred as $F(n) = C(n) + H(n)$. A* differs from Dijkstra's algorithm on its selection of the node from the open list. Dijkstra's algorithm selects the node that has a lower C value, i.e., the node that is closer to the starting node, while A* selects the node that has a lower F value, which represents the node that has a shorter path so far and that is estimated to be closer to the goal.

2.4.2.2 Detailed Algorithm

A* algorithm, described in algorithm 3, starts by setting the start node values, F, C and H to zero, setting also its parent reference to null (line 1). The start node is added to the open list (line 2). While the open list is not empty, the node with the lowest F value is popped from the open list and is set as the current node (lines 3-4). If the current node is the goal node, the path is returned (lines 5-6). Otherwise, this node is then placed in the closed list as a visited node (line 7). A* checks all neighbors of the current node. For each neighbor N, if S is already in the open list, skip this node (lines 8-10). Otherwise, a tentative C value is calculated as in Dijkstra's algorithm (line 11). Then, if N is not in the open list or its C value is higher than the tentative C value, C value of N is updated with the tentative C value, a new H is calculated, F is updated, and the parent reference is set to the current node (lines 12-15). If S is not in the open list, S is added in the open list for exploration (lines 16-17).

2.4.2.3 Tie Breaking

Upon selection of the next node to retrieve from the open list, it may happen that two or more open nodes may have the same F value but different C and H values. In situations like this, a tie breaking method must be employed. A good tie-break is to use the node with the lowest C value, that would be the node closer to the starting node.

Algorithm 3 A* Search Algorithm

```

1: Clear start node's Parent and cost values ( F, C, H )
2: Add start node to open list
3: while open list is not empty do
4:   n = pop best node from open list
5:   if n is goal then
6:     Reconstruct path
7:     Add n to closed list
8:   for all Successor nodes S of n do
9:     if S is in open list then
10:      continue;
11:     Calculate new G cost ( Current C + traversal cost )
12:     if S is not in open list or new C cost is lower than S's G cost then
13:       Set S's Parent to n
14:       Set S's C cost to new C cost
15:       Set S's F cost to S's C cost + Heuristic estimate from S to Goal
16:     if S is not in open list then
17:       Add S into open list
18:   Reconstruct No Path

```

In A* algorithm, node exploration is based on the heuristic value, which is an estimation of the closeness to the goal node. This estimation may often be too optimistic and lead to the exploration of nodes that are farther from the start node before the nodes that are closer to the start node, resulting in the exploration of closed nodes whose C values are overly high and which can still be reduced. This means that closed nodes cannot be ignored as in Dijkstra's algorithm and need to be checked and updated if necessary [Ang12]. Unfortunately, this introduces the concept of re-exploring closed nodes, which does not occur in Dijkstra's algorithm.

Depending on the search problem and search space, the degree of node re-exploration can be quite large. This increases computational cost of A* algorithm, but has no effect on memory cost. Even with this increase in computational cost, total node exploration (i.e., the number of visited nodes) is still significantly less than in Dijkstra's algorithm.

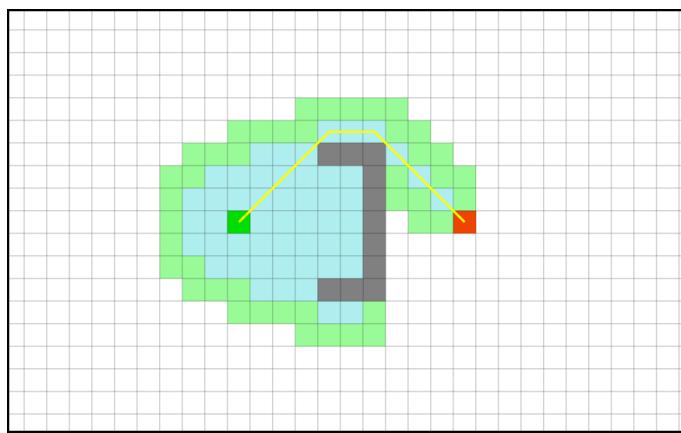


Figure 2.9: A* Algorithm Search Path.

Figure 2.9 shows the search space and resulting path (in yellow) found using A* search algorithm. The bright green node is the starting node, the bright red node is the goal node, the light blue nodes are the visited nodes and the light green nodes are the nodes to be visited, the frontier nodes. The dark grey nodes represent obstacles (i.e., walls).

2.4.2.4 Heuristics

An heuristics is an estimate of cost incurred in traveling between two nodes. This estimate is given by a function, $H(n)$. The heuristic can alter the search algorithm performance, making it work faster or slower, use more or less memory, be more or less accurate, and optimal or not.

In A* algorithm, the next node to explore is chosen based on its associated value, $F(n) = C(n) + H(n)$. Depending on the importance of $H(n)$ over $C(n)$, $F(n)$ will be different, leading to a different path. Some of the possibilities for the importance of different $H(n)$ values are as follows [Pat11] :

- If $H(n)$ is 0, only $C(n)$ will affect on the search resulting the same search result as Dijkstra's algorithm.
- If $H(n)$ is always lower than or equal to the cost of moving from N to the goal, A* is guaranteed to find the shortest path [HNR68]. The lower the estimation, the more nodes A* explores.
- If $H(n)$ is exactly equal to the cost of moving from N to the goal, A* only follows the best path and never explores any other node, making it very fast. Given a perfect heuristic, A* behaves perfectly.
- If $H(n)$ is sometimes higher than the cost of moving from N to the goal, A* will run even faster than using a perfect heuristic, but there is no guarantee to find an optimal path.
- If $H(n)$ is very high relatively to $C(n)$. In this case, only $H(n)$ affects the search. This returns the same search result as the best-first search.

A* can be fine tuned towards speed or accuracy depending on the needs of the search algorithm. In many video games a "good enough" path may be sufficient and by raising the estimation, the search will run faster, returning a nearly optimal path. On the other hand, when the estimation is lower or exact, a more accurate and slower search will return an optimal path from N to the goal.

The balance between accuracy and speed can be regulated dynamically in-game. By giving a weight to the heuristic cost, the algorithm is able to control how accurate the search needs to be. On a large search area, A* may be tuned to run faster on the hope that the path may be re-calculated with more accuracy when closer to the goal. This way, the slower search will run closer to the goal with a reduced search space leading to an equally fast search without loosing performance or path quality.

Note that $C(n)$ and $H(n)$ must have the same scale in order to ensure that the algorithm behaves correctly. For example, if $C(n)$ returns values in meters and $H(n)$ in kilometers, $H(n)$ will clearly have much more influence on the resulting value $F(n)$.

A number of well known heuristics have been used in A* algorithm. In a plane, with standard coordinates A_i and B_i , respectively, heuristics are:

- Euclidean distance [Bla04] is the straight line distance between two points A and B. In a

Influence Map-Based Pathfinding Algorithms in Video Games

plane, the Euclidean distance is defined as:

$$D_{\text{euclidian}}(\mathbf{A}, \mathbf{B}) = \sqrt{\sum_{i=1}^n (A_i - B_i)^2} \quad (2.1)$$

- The Manhattan distance [Bla06] is the distance between two points measured along axes at right angles. In a plane, the Manhattan distance is defined as:

$$D_{\text{manhattan}}(\mathbf{A}, \mathbf{B}) = \sum_{i=1}^k |A_i - B_i| \quad (2.2)$$

- The Chebyshev distance [Bla08], also known as maximum metric, is the greatest difference between two points. In a plane, the Chebyshev distance is defined as:

$$D_{\text{chebyshev}}(\mathbf{A}, \mathbf{B}) = \max(|A_x - B_x|, |A_y - B_y|) \quad (2.3)$$

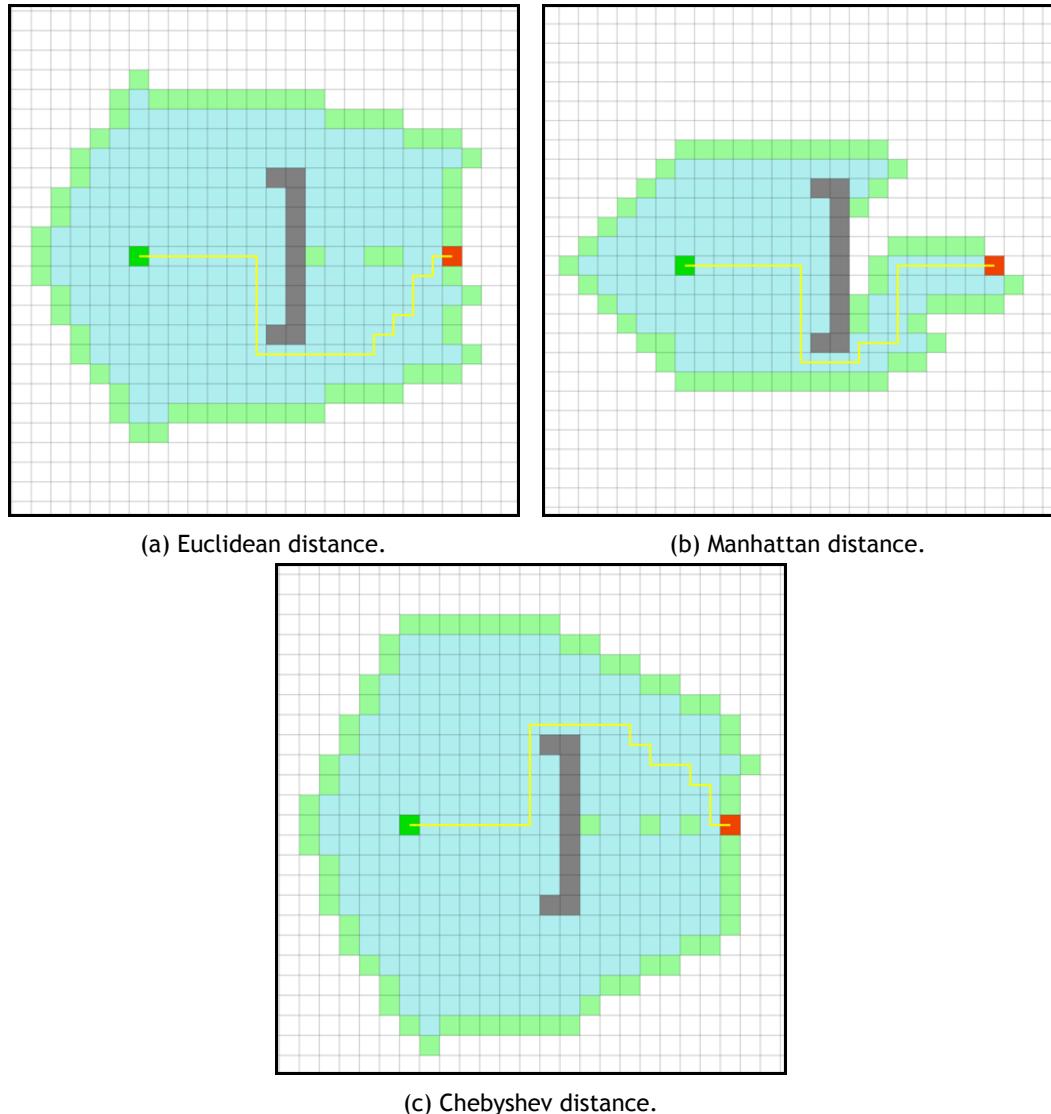


Figure 2.10: A* search path results using different heuristics.

Figure 2.10 (a) shows the resulting path of A* search algorithm using Euclidean distance. Eu-

clidean distance uses the square root operation which will reduce performance and should be avoided when possible. Using the euclidean distance, $H(n)$ will have lower values than Manhattan's or Chebyshev's distances, making the cost function $C(n)$ not match the heuristics. Heuristics that do not over-estimate the true distance to a goal node are said admissible. A* with an admissible heuristic returns optimal solutions [HNR68]. Figure 2.10 (b) shows A* algorithm's search path in a 4-connected grid using the Manhattan distance as the heuristic. Figure 2.10 (c) shows the resulting path of A* search algorithm using Chebyshev distance. Light green nodes are the frontier nodes, which are the nodes on the open list. Light blue squares are the visited nodes. Green is the starting node and red the goal node. Grey nodes represent non-passable nodes (e.g., walls).

Each of these heuristic costs are multiplied by a distance cost, D , which is based on the lowest cost between two adjacent nodes. In the absence of obstacles and terrains that have a minimum cost D , moving to a step closer to the goal should increase $C(n)$ by D and decrease $H(n)$ by D . When adding the two components $C(n)$ and $H(n)$, $F(n)$ remains the same, meaning that the heuristic and cost function scales are a match. On the other hand, if the scales do not match, then either the heuristic or the cost function has too much weight on the final calculation [Pat11].

Figure 2.10 shows different search areas of exploration using different heuristics. With the same search map and same search algorithm applied, all of the heuristics returned a path length of 26 nodes, which is the optimal path. Using the Manhattan distance, however, leads to end the algorithm in 331 steps, whereas Euclidean distance takes 566 steps to finish and Chebyshev distance 648 steps. Figure 2.10 clearly shows the increase in nodes explored justifying the results obtained.

2.4.2.5 A* Variants

Over the years A* suffered many alterations in order to correct some aspects of the algorithm and to fulfill the needs of ever demanding commercial games, what gave birth to not only new algorithms such as Iterative Deepening A* (IDA*) [Kor85], Memory Enhanced IDA* [RM94], Fringe Search [Bjo05] but also new paradigms of pathfinding with hierarchical abstractions on grid maps and parallel searches.

IDA* [Kor85] is a depth-first search algorithm and works by searching the graph depth by depth via a threshold factor on the $F(n)$ value. When $F(n)$ is too large, the nodes are not be considered. At the beginning, IDA* expands a few nodes and at each subsequent iteration, the number of visited nodes increases. If a better path is found, the threshold is increased. IDA* revisits nodes when there are multiple, possibly non-optimal, paths to a goal node, which is often solved by using a transposition table as a cache of visited nodes [RM94]. When using a cache of visited nodes, IDA* is denoted as memory-enhanced IDA* (ME-IDA*). IDA* repeats all previous operations in a search when it iterates on a new threshold, what is necessary to operate when no storage is used. By storing the leaf nodes of a previous iteration and using them as the starting position of the next, IDA* efficiency improves significantly. These improvements gave birth to the fringe search algorithm [Bjo05], which is described in Section 2.4.4.

Weighted A* [Poh70] also provides significant speed-up at the price of a bounded solution sub-optimality [LMP⁺13]. There are many other search methods that build on A*, including methods for hierarchical pathfinding, such as, Hierarchical Pathfinding A* (HPA*) [BMS04], Hierarchical Annotated A* (HAA*) [SSGF11], Partial Refinement A* (PRA*) [SB05], Block A* [YBHS11] and meth-

ods for search on multiple CPU (Parallel Ripple Search [BB11]) [LMP⁺13].

2.4.3 Best-First Search

2.4.3.1 Leading Idea

Best-first search [Pea84] is a greedy algorithm that was inspired by A*, but only uses a heuristic to guide the search to the goal, not using the cost so far. Consequently, best-first search converges to the goal faster and explores a smaller amount of nodes, as illustrated in Figure 2.11a. However, best-first search does not guarantee to find the shortest path neither the optimal path as A*.

The main disadvantage of best-first search is that it is a greedy algorithm. A greedy algorithm will direct the search to local minima fairly easily, since it only considers the estimate value to the goal node, causing the exploration of nodes even when the search is not on the right path. The lack of knowledge on the traveled path so far may lead to long, non-optimal paths.

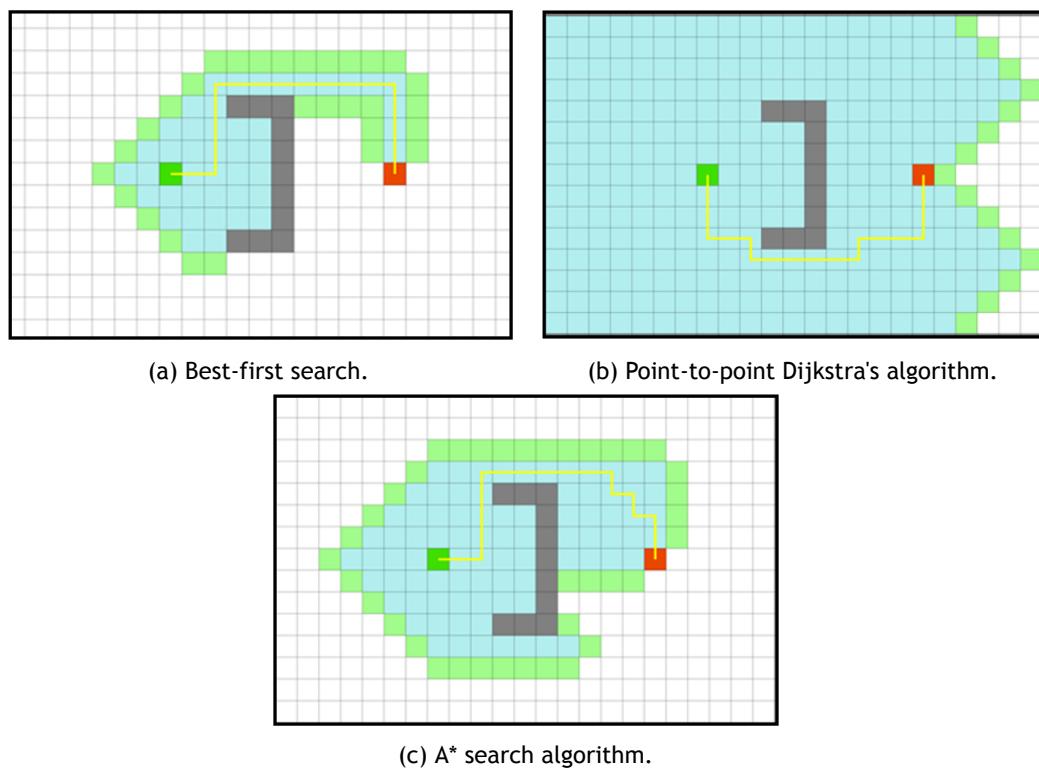


Figure 2.11: Search space explication comparison.

Figure 2.11 compares the different algorithm search results and search space explorations.

2.4.3.2 Detailed Description

The best-first search also works with an open list and a closed list (see Algorithm 4). Best-first search starts by adding the start node to the open list (line 1). At each iteration, while the open list is not empty, the node with the lowest $F(n)$ value, defined by $F(n) = H(n)$, from the list is popped out and set as the current node (lines 2-3). If the current node is the goal, the algorithm

returns the reconstructed path (lines 4-5). At this point, every neighbor of the current node is examined, added to the open list and their parent's reference is set to the current node (lines 6-9).

Algorithm 4 Best-First Search

```

1: push start node to open list
2: while open list is not empty do
3:   n = best node from open list
4:   if goal is found then
5:     Reconstruct path
6:   for all neighbor S in n do
7:     Explore S
8:     Push S into open list
9:     Set S's parent to n
    
```

2.4.4 Fringe Search Algorithm

2.4.4.1 Leading Idea

Fringe search algorithm [Bjo05] can be seen as a variant of A* or a variant of IDA*. As mentioned in Section 2.4.2.5, IDA* disadvantages are solved through the inclusion of cache memory to avoid repeated states and the *now list* and *after list* to remove unnecessary iterations over the same nodes, give birth to the fringe search algorithm.

Considering how IDA* works, there is a starting threshold H (root). The algorithm does a recursive left-to-right depth-first search, so that the recursion stops when either a goal is found or a node is reached that has an f value bigger than the threshold. If the search with the given threshold does not find a goal node, the threshold is increased and another search is performed.

2.4.4.2 Detailed Description

As said above, Fringe search uses two lists, the *now list* and the *later list* (see Algorithm 5). At each iteration, the leaf nodes are added to the *later list* to be explored in the next iteration. When the iteration completes, the *later list* becomes the *now list*, and when a new iteration begins, the search will resume from the *now list*, instead of re-iterating over the whole search tree. Fringe search algorithm does not need to keep the *later list* and *now list* ordered when compared to A*, which leads to improved computational costs. However, fringe search algorithm will revisit nodes. Fringe search explores on the limits of the graph, hence the term, fringe.

2.4.4.3 Algorithm

2.5 Final Remarks

Grid-based map representations (discussed in Section 2.3) have several drawbacks that may make them less useful in games in which environment objects don't have uniform sizes or shapes

Algorithm 5 Fringe Search Algorithm

```

1: fringe F = s
2: cache C[start] = (0, null)
3: found = false
4: while (found is false) AND (F not empty) do
5:   fmin = ∞
6:   for all N in F, from left to right do
7:     (g, parent) = C[node]
8:     f = g + h(node)
9:     if f is greater than flimit then
10:      fmin = min(f, fmin)
11:      Continue
12:      if node is goal then
13:        found = true
14:        break
15:      for all child in children(node), from left to right do
16:        child g = g + cost(node, child)
17:        if C[child] is not null then
18:          (g cached, parent) = C[child]
19:          if g child is greater than or equal to g cached then
20:            Continue
21:          if child in F then
22:            Remove child from F
23:            C[child] = (g child, node)
24:            Remove node from F
25:      flimit = fmin
26:    if reachedgoal is true then
27:      Reconstruct path

```

and where large areas of the map are empty leading to a waste of space [LMP⁺13]. Demeyen and Buro [DB06] present two pathfinding algorithms to address these problems, Triangulations A* (TA*), and Triangulation Reduction A* (TRA*). TA* uses dynamic constrained Delaunay triangulations (DCDT [KBT04]) to represent maps in which obstacles are defined by polygons, and finds optimal any-angle paths for circular objects by running an A* like algorithm on graphs induced by the triangulation.

Dijkstra's algorithm was designed to be a all-point search, finding all paths from a single node in a graph, exploring the entire graph. This leads to a very high computational cost, estimated around $O(n^2)$ where "n" is the number of nodes in the graph. Processing costs are a result of operations on open and closed lists. Each time a node is selected from the open list, the entire list is searched to find the node with the lowest G value. With each node explored, the open list grows in size and so the cost of performing this search. The open list search is the most computationally expensive operation in the algorithm and therefore has been a primary focus for optimization for both Dijkstra's algorithm and algorithms derived from it, such as A* and its variants.

A simple way of reducing the open list searching costs is to store the nodes in an ordered manner. If the open list is ordered in ascending order, the first node will always be the best node. Selecting the best node from the list becomes a trivial operation with respect to processing costs. Unfortunately, keeping the list ordered introduces additional costs during node exploration. When a node is inserted into the list, a search done to find the correct position to insert the node. When a node is updated, its C(n) or F(n) cost, depending on the algorithm used, will change and the node will change position inside the list, leading to extra operations by removing

and re-inserting the node into the list.

The closed list serves the purpose of checking whether a node has been visited or not. When examining successor nodes, a verification is done to verify if the node has already been visited, which would be indicated by being present in the closed list. When the closed list grows in size, searching for a particular node becomes quickly expensive. A simple solution around this issue is to have the nodes carrying a search state flag indicating whether they are *unseen*, *unvisited* or *visited*. When examining the nodes, changing its search state flag instead of inserting and removing it from lists allows the algorithms to perform better, both in time and memory. The closed list becomes obsolete and can be completely removed from the algorithm.

Uniform travel cost on maps lead to a considerable degree of symmetry in a pathfinding problem [LMP⁺13]. Symmetry reduction is based on empty rectangles identified on a map [HB10] [HBK11]. Given two nodes A and B on the border of an empty rectangle, all optimal paths connecting these are symmetrical to each other, and considering only one such a path is sufficient.

Jump Point search [HG11] takes a different approach to symmetry elimination. It allows pruning part of the successors of a node, ensuring that optimality and completeness are preserved [LMP⁺13].

Chapter 3

Influence Map-Based Pathfinding

Real-time strategy (RTS) games are a sub-genre of strategy games in which a player interacts in real time without awaiting for another player's turn [Bur03] [BF03]. RTS games such as for example, Starcraft by Blizzard Entertainment [Ent] and Age of Empires by Ensemble Studios [Stu] are usually set on a war theme [SDZ09] and require the player to gather resources, perform base management, unit coordination, and defeat enemies in real time [BF03].

In RTS games, all player's decisions are taken independently of other players, with no need to wait for a player's turn to finish [HJ08]. RTS games often require the player to control a great amount of units (i.e., troops, vehicles, planes, etc.) that is, units in movement. This movement of game units poses a challenge to pathfinding in games. In fact, pathfinding methods for these units may become slow when one need to calculate similar paths for hundreds of units that move in groups to the same location. To prevent these intensive calculations, one uses potential fields in path planning. Pathfinding specifically applied to the RTS genre focuses on alternatives, such as path planning using potential fields [BK89] [Bur03].

3.1 Potential Fields

One cannot talk about influence maps before first talking about potential fields. Potential fields (PF) or artificial potential fields (APF) are a technique used to avoid collisions and achieve human like movement of game agents in real-time [Joh12]. The leading idea of potential fields has some similarities with influence maps further discussed in Section 3.2. In 1985, Ossama Khatib came up with the idea of artificial potential fields in robotics, as a way to avoid obstacles and achieve human like movements [Kha86]. Later, Arkin proposed a new technique based on spatial navigation of vector fields that proved to be better than previous methods [Ark97]. The game's efficiency regarding object movement can be increased by applying different types of potential fields, simultaneously [HJ08] [HJ09a] [HJ09b].

3.1.1 Overview

Potential fields work by strategically placing attractive and repulsive force propagators in the game world. A propagator is a game unit that delivers a potential field. In fact, a propagator generates a potential field that gradually fades to zero [Hag08]. Generally, if its charge is attractive, other objects move towards it, if its charge is repulsive other objects move away from it.

Figure 3.1 (a) represents an attractive propagator, where the field of forces are represented by a vector field. Figure 3.1 (b) shows the repulsive potential field. The attractive field is uniform throughout the map, but fades near the propagator. The same does not happen with repulsive

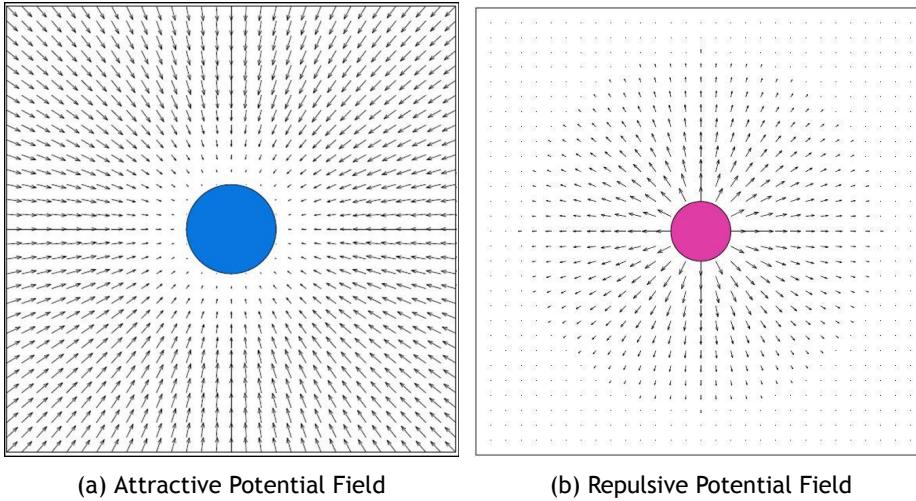


Figure 3.1: Attractive and repulsive force fields.

fields. Repulsive forces decay with distance to the propagator, having a higher force near the propagator. The origin of either potential field is the position of the propagator.

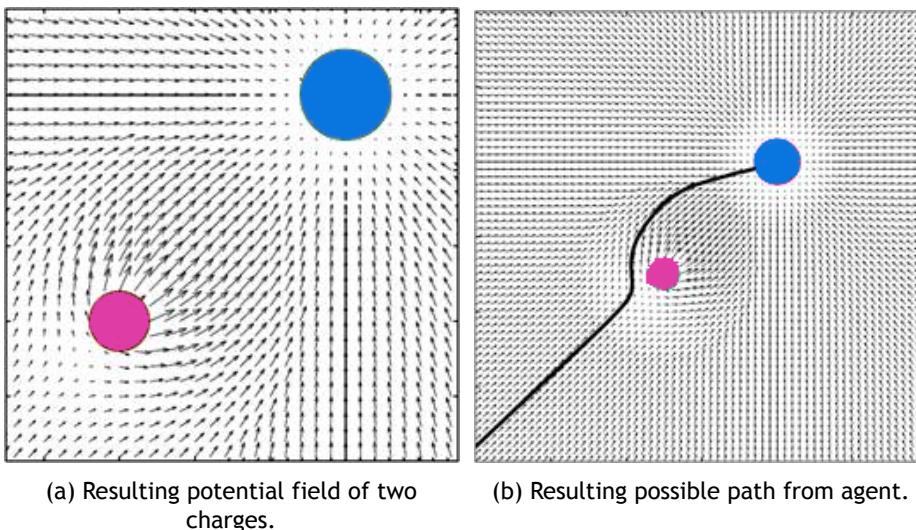


Figure 3.2: Opposing charges in same potential fields.

As illustrated in Figure 3.2 (a) and Figure 3.2 (b), the potential field spreads over the totality of the map. The attractive forces are present in most parts of the map, while the repulsive force change the attractive forces, and ultimately change the direction and intensity of the field near the repulsive propagators. Figure 3.2 (b) shows a possible path taken from an agent, which avoids the repulsive charge and continues to navigate to the goal.

3.1.2 Potential Field Types

There are two major types of potential fields namely: point fields and directional fields. The point fields converge/diverge to/from a point located at a finite distance. The directional fields assume that the propagator is in the infinite. There are three categories of point fields: attractive, repulsive, attractive-repulsive and two categories of directional fields: uniform, perpendicular.

Influence Map-Based Pathfinding Algorithms in Video Games

Attractive potential fields, as illustrated in Figure 3.1 (a), are used to represent the goal or any important location, and are used to lure the agents to their center [Hag08]. Repulsive potential fields, shown in Figure 3.1 (b), are the opposite of attractive potential fields, which pushes away agents from a game world zone, obstacles, or units [Hag08] so they can avoid each other.

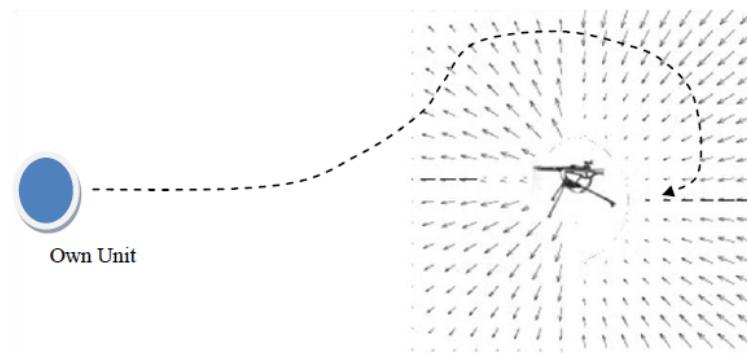


Figure 3.3: Asymmetric Potential Field. The path avoids facing the turret [Sul11].

Asymmetric potential fields, as illustrated in Figure 3.3, can have both attractive and repulsive forces coming from the same object. Figure 3.3 shows an example of an asymmetric potential field generated by a turret [Sul11]. This is useful when the agents need to act according to a strategy (e.g., flank the enemy). This way, the agent will not face the turret and risk being hit, instead, it will try to flank it and reach it from behind.

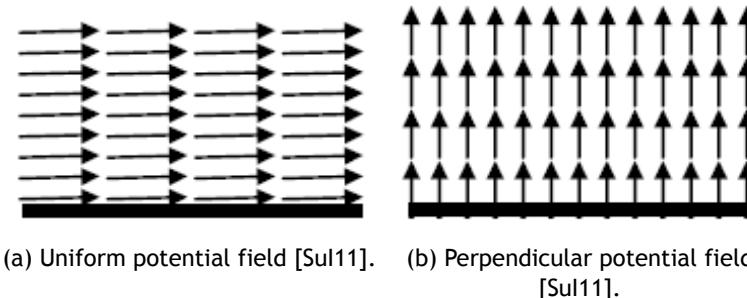


Figure 3.4: Different types of potential fields.

Uniform potential fields, shown in Figure 3.4 (a), guide agents in a specific direction and will not be able to guide them to the goal but are good at following walls [Goo10]. Perpendicular potential fields, shown in Figure 3.4 (b), are used to guide the agent away from a wall [Goo10].

3.1.3 Local Minima

Local minima likely is the most important problem that we find in artificial potential fields. Figure 3.5 shows an example of an agent stuck at a local minimum. If the goal is on the other side of the wall, the agent cannot go further, because the total sum of vector forces reaches zero, having no force nor direction to proceed any further.

One solution is to apply a random potential field to get the agent out of the local minimum [Goo10]. Another solution, as proposed by T. Laue [TL03], is to apply A* pathfinding search to find a suitable location to resume motion planning using potential fields when such location is reached. In short, in this scenario, A* is used to solve local minimum situations exclusively,

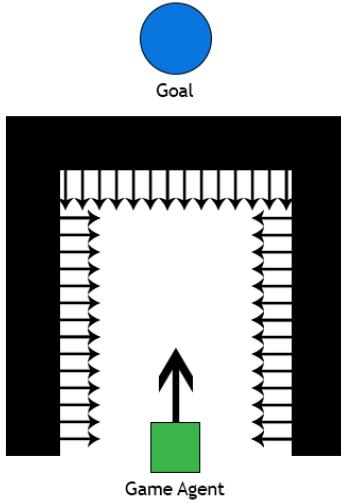


Figure 3.5: Local minima problem with U-shaped obstacle.

otherwise, one uses potential fields. That is, in the current state-of-the-art, A* and potential fields are never ran simultaneously.

In contrast, this dissertation is entirely based on combining existing search algorithms (i.e., Dijkstra's, A*) with the spatial reasoning provided by influence maps, in order to make pathfinding searches reactive to their environment (i.e., given updated information of the game environment changes), and find solutions faster (i.e., searching less nodes).

Yet, another solution is to avoid returning to the previous position in the potential field, as proposed by T. Balch and R. Arkin [TR93]. The solution consists in releasing repulsing potential force at regular intervals. To reduce interference with goal based potential fields, Thurau et al. [CTS04] proposed that the avoid-past potential field forces are reduced over time, resulting in a pheromone trail effect known from ant-algorithms. The pheromone-like repulsive potential forces push the agent from nearby nodes to avoid local minima.

3.2 Influence Maps

Influence maps evolved from work done on spatial reasoning for the GO game [Zob69]. Spatial and temporal reasoning has to do with the analysis of terrain as well as understanding events that occur over time. Influence maps have been used in video games [Toz01], more specifically for games such as Ms. PacMan [WG08] [SJ12], real-time strategy games [ML06a] [ML06b] [MQLL07] [Mil07] and turn based games [LM05].

Influence maps introduce two concepts into spatial reasoning: tactics and strategy. Tactics is a word from ancient Greek that means art of arrangement, which is used to refer specific actions or specific tasks by game agents; for instance, where a player should place a defense tower. Strategy is referred to high level planning to achieve goals under unknown conditions (i.e., adversarial troops, enemy location), which will impact the outcome of the game.

There are many possible representations for tactics information, such as artificial neural networks and knowledge based approaches. However, Avery et al. [ALA09] used influence maps due to their intuitive representation of spatial characteristics of game worlds.

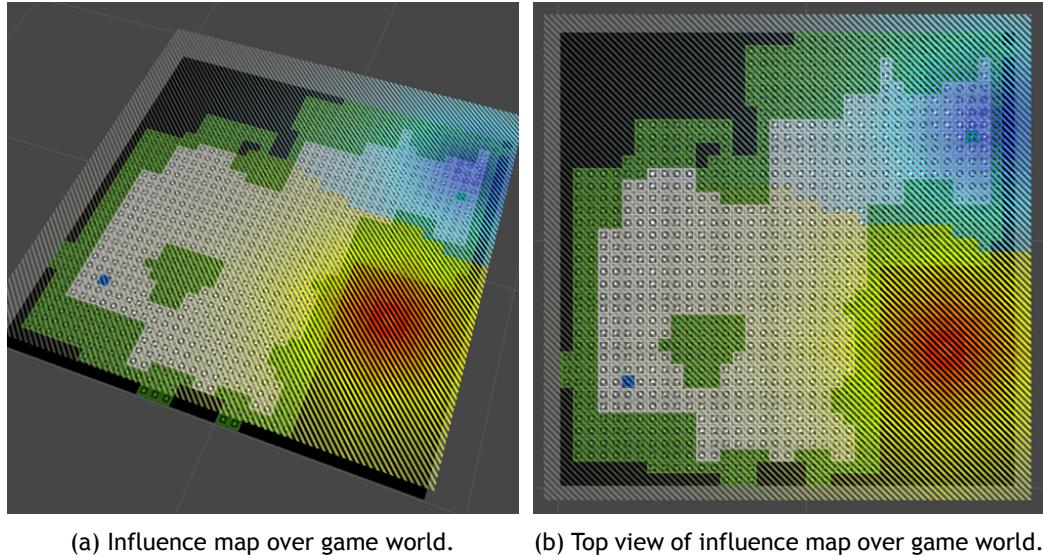


Figure 3.6: Influence map representations over a game world.

Strategy is used to make high-level decisions, namely, deciding which town to conquer next and why (i.e., precious resource). Tactics are used to make low-level decisions, at the game agent level, namely to decide which path to take, which enemy faction to attack and why (i.e., shortest path, weaker enemy). Influence maps provide information for both tactical and strategic decision making [Swe06].

Influence maps allow intelligent inferences about the game environment, namely, where the enemy has less troops, which country to invade, or other meaningful information that human players inherently would perceive through experience and intuition [Toz01] [WG00].

3.2.1 Overview

Similarly to potential fields (see section 3.1), influence maps have propagators carrying positive or negative influence. The main difference between potential field and influence maps is how the propagation is done. Whereas potential fields generate vector fields of forces, influence maps produce scalar fields of influence.

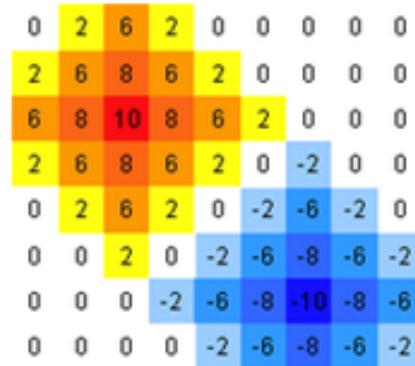


Figure 3.7: Visual representation of influence values.

Figure 3.7 shows an example of how the values are propagated through the map. Highest values

at the center of the influence are the source of the propagators. From there, the influence loses force and gradually fades to zero. The values range from 10 to 0 and -10 to 0, for an easy visualization and representation purposes only.

Repulsive and attractive propagators may carry negative or positive scalar values, which depends on implementation. On the contrary, from here on now, we assume that repulsive propagators carry a positive value and attractive propagators carry a negative value. As usual, propagators can be game units, game events, or any other game entity, as long as they are located in the game environment space.

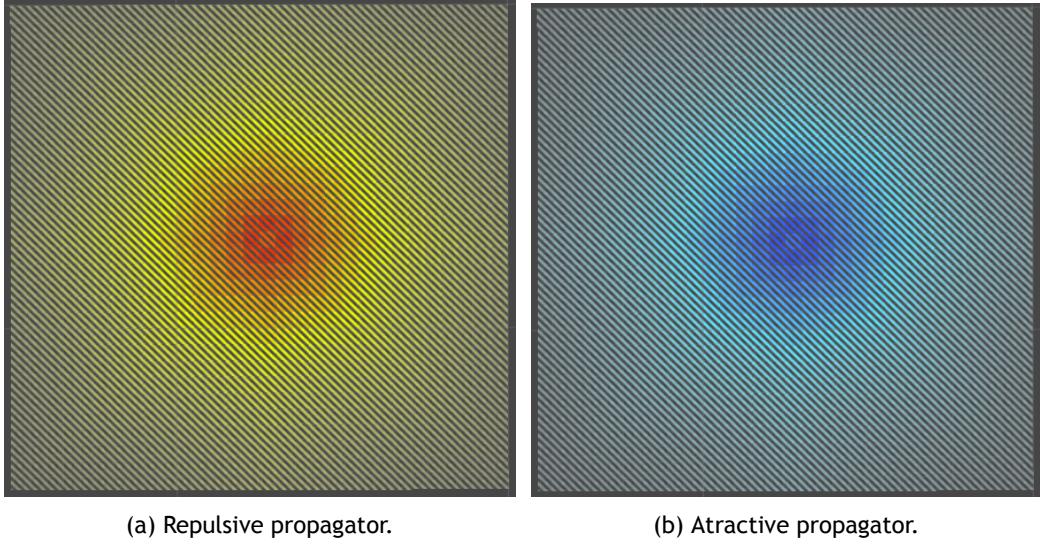


Figure 3.8: Repulsive and attractive propagators.

Figure 3.8 (a) shows a repulsive propagator carrying positive influence value, while Figure 3.8 (b) illustrates an attractive propagator carrying negative influence value. The darker colors at the center of the propagator represents the epicenter of the influence propagation, which is a minimum for attractive propagator and a maximum for repulsive propagator, fading gradually outwards to zero.

As discussed in Section 2.3, 2D grid-based representations are relatively easy to use and implement. Influence maps can be represented by 2D grids and also abstracted into 3D influence volumes [Pot00]. Influence map representation is overlapped on top of the game environment.

Influence maps can be layered to create new information by merging different influence map information, e.g., faction influence, town control influence, combat strength, visibility area, noise, frontier of war, unexplored areas, recent combats, and more [Toz01] [WG00].

Tozour [Toz04] proposed an approach in which each influence map layer, representing distinct information, are merged into a final influence map as a weighted sum, in which the weights are chosen arbitrarily.

Bergsma et al. [BS08] used a neural network (NN) for a layering algorithm to generate two influence maps, one indicating the trend of moving to a tile and the other indicating the trend of attacking a tile. Both influence maps are merged to maximize chances of moving and attacking a tile. Sweetser [Swe04] also used neural network and influence maps for strategic decision-making in games.

Figure 3.9 (a) shows opposite influences at opposing sides of a map, while Figure 3.9 (b) shows

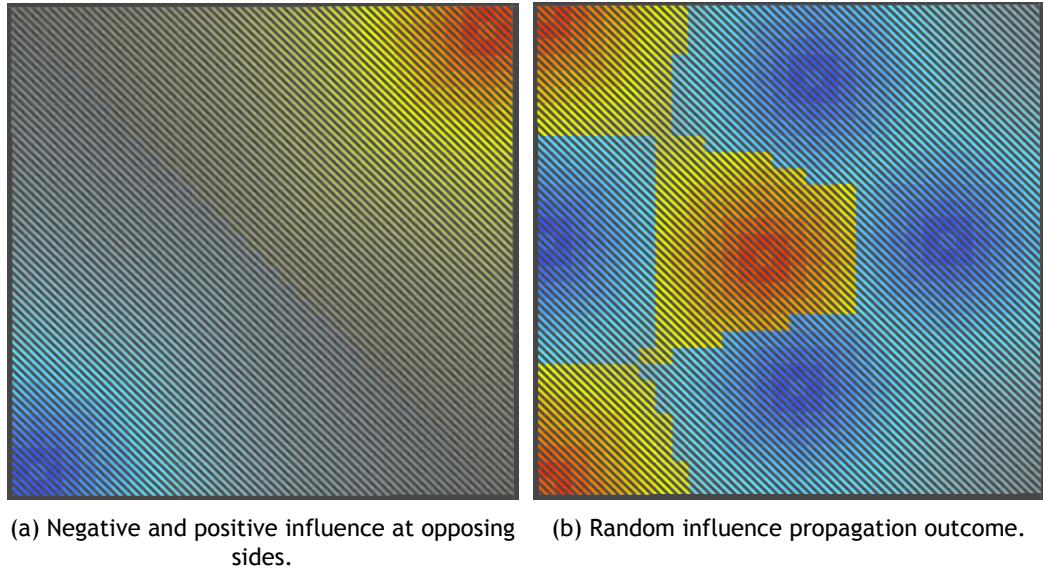


Figure 3.9: Possible final influence map representation.

how the influence propagates with randomly set propagators.

Influence maps are also used in non-violent game scenarios. Sim City [sim] series offer real-time maps that show the influence of police and fire departments around the city. These influence maps are used by the player, so high-level strategic decisions can be made, for instance, placing a police building that will help cover zones where there is low police influence.

Influence maps help the artificial intelligence to make better decisions, given useful information about the game world. They provide three types of information that are particularly useful for decision making [Cha11] namely:

- **Situation Summary:** Influence maps summarize the details in the world making them easy to understand at a glance. This is so because they provide visual feedback about where the influence is, who is in control of an area, and so on, are procedures that becomes trivial.
- **Historical Statistics:** Influence maps hold information for a limited amount of time, allowing the temporal analysis of recent events (i.e., recent war zones). This is controlled with the momentum parameter (see Section 3.2.3.1).
- **Future Predictions:** Influence maps predict upcoming events [LMAB⁺09]. Using terrain maps, enemy influence can be analyzed to know where they are headed.

On large and open game environments that have varying traversal costs between nodes, choke-points and large obstacles, influence maps provide interesting properties to help the game AI make better decisions, due to how the propagators influence is spread against with each other [Cha11].

3.2.2 Representations

Similarly to any game environment, influence maps needs to have two main components to be able to be represented, namely: spatial partitioning and connectivity.

Spatial partitioning is a way to divide the game environment in partition space (i.e., nodes) and store information in it (i.e., influence value). Connectivity dictates how the nodes are connected to their surrounding neighbors. This will directly affect how the influence is spread from the propagator to its neighbors.

Influence maps can be represented with 2D and 3D grids, as mentioned before, but other representation methods can be employed, as follows:

- fine-precision or coarse-precision grids
- area graphs and waypoint graphs

Depending on the game, coarser space partitioning will have a less precise resolution of the influence map and is better suited for high-level strategic decision making. Finer space partitioning will have a better resolution of the influence map and will be better suited for tactical low-level decision making:

- **Grids:** 2D grids are easy and fast to implement and depending on the level of detail the game requires. Fine grids will be have more resolution but consume more memory space (Figure 3.11 (a)), while coarse grids will have less resolution and consume less memory (Figure 3.11 (b)). Although coarse grids use less memory, the fact that they use a lower resolution may result in some cells to span over obstacles.

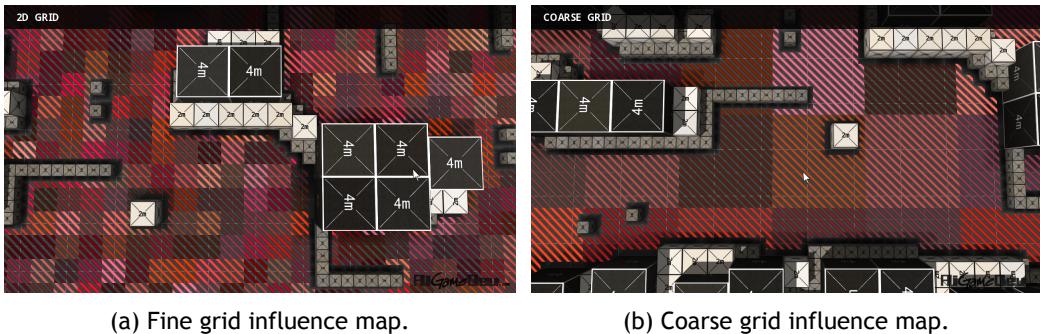


Figure 3.10: Influence maps grid representations [Cha11].

- **Area and Waypoint Graphs:** Area and waypoint graphs are used to solve some of 2D grid limitations, namely the high number of nodes that leads to high memory consumption. Also, area and waypoint graphs can be used in 3D environments, for instance, wrapping over multiple levels of a building. Although they bring benefits over grids, they may lack on precision where detail is needed for the decision making.

3.2.3 Parameters

Influence maps are relatively easy to set up and are very flexible. There are parameters that can change the behaviour of an influence map (e.g., momentum, decay and update frequency), as discussed bellow.

Influence Map-Based Pathfinding Algorithms in Video Games

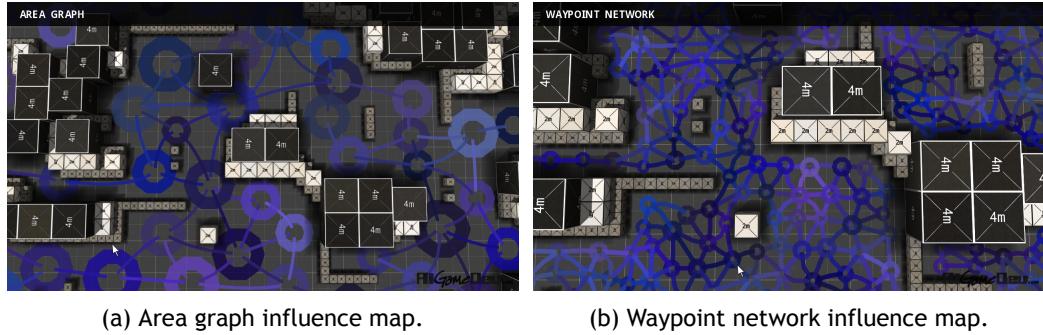


Figure 3.11: Influence maps area and waypoint graph representations [Cha11].

3.2.3.1 Momentum

In classical mechanics, linear momentum is the product of the mass and velocity of an object. Linear momentum states that a heavy object moving at high speed has a large momentum, whereas a lighter or slower object have a small momentum. This is somewhat applied in the propagation of the influence.

Linear interpolation is used to blend from the current influence value to the new value, and relies on the momentum parameter to control the result. If the momentum value is high (Figure 3.12 (a)), reaching 1.0, the algorithm will bias the results to old values, making it useful for historical values, since old values will tend to live longer. If the momentum value is low (Figure 3.12 (b)), next to 0.0, the algorithm will bias the results to newer values, so that the propagation happens quicker and the prediction is more accurate. Depending on the information layer (influence map), high-level strategic maps may benefit from high momentum, and low-level tactical maps from low momentum.

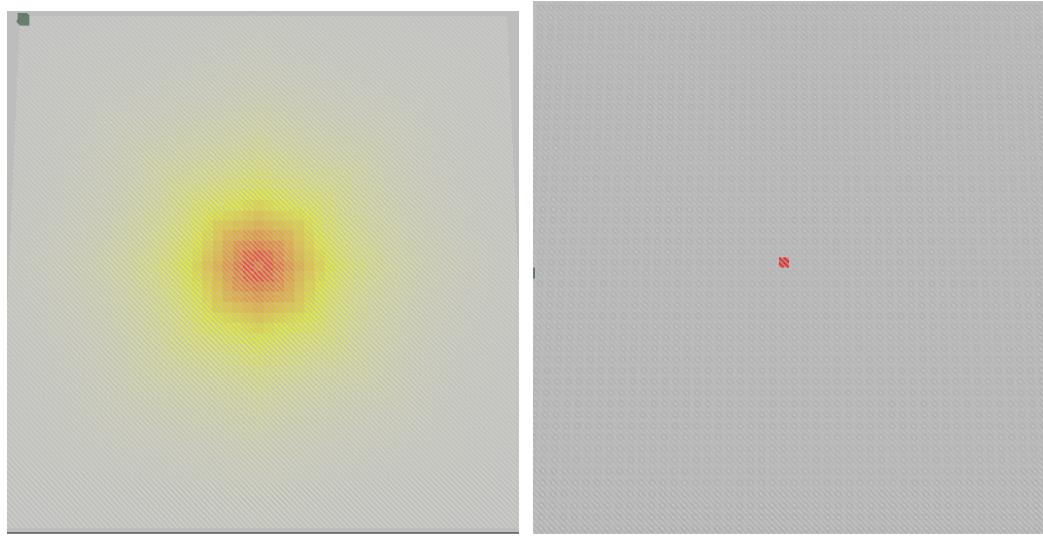


Figure 3.12: Momentum comparison.

3.2.3.2 Decay

While the momentum features the strength of the scalar influence field, the decay represents how quickly the influence fades. Larger, strategic maps tend to use slower decay, so the information spreads out to more area, while smaller, tactical influence maps tend to use faster decay.

Different decay rates can be used based on the size of the influence map (information layer) or based on propagators themselves (i.e., game agents, key items, events). This implies the use of different influence maps or layers for each type of propagator which have to be combined into the final layer of influence. The decay is controlled by multiplying it by the distance to the propagator before being passed to the exponential function (assuming exponential decay), as described in Algorithm 7.

In Algorithm 7, exponential decay is used, meaning that the decay rises exponentially over distance, expressed as:

$$\text{decay} = e^{-d*\delta} \quad (3.1)$$

where d is the distance to the propagator and δ is the decay parameter.

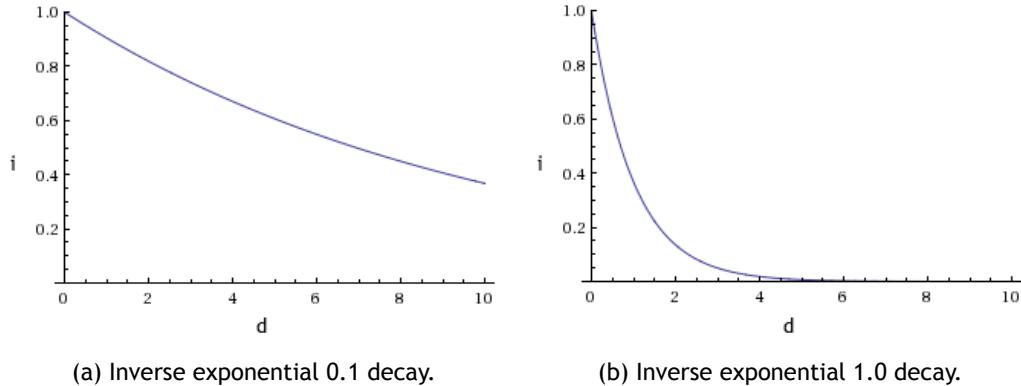


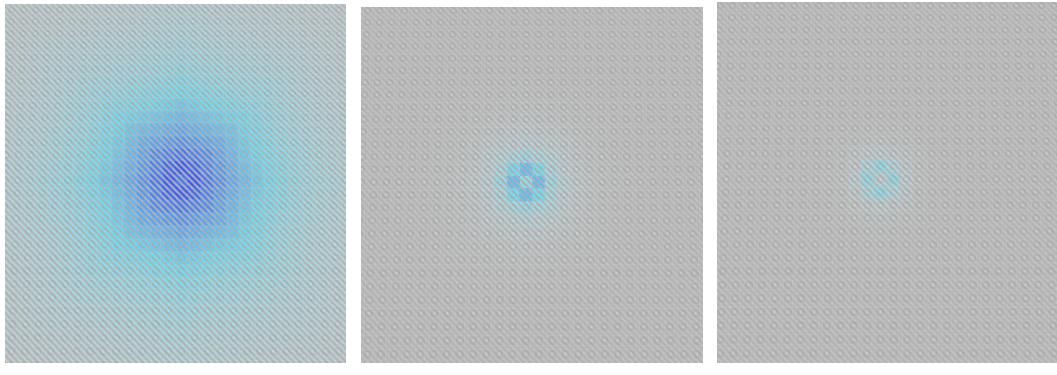
Figure 3.13: Comparison of exponential decay with different decay values.

Figure 3.13 shows the plot of different decay parameters, where "i" is the influence value and "d" is the distance. Figure 3.13 (a) plots an inverse exponential with 1.0 decay, while Figure 3.13 (b) plots an inverse exponential with 0.1 decay.

Figure 3.14 shows a comparison of different decay values used on a propagator. The higher the decay, less influence will be spread. The lower the decay, more influence will be spread. The next parameter to tune is the update frequency.

3.2.3.3 Update Frequency

Influence map update frequency depends on how many resources are to update in the game by the AI. Depending on the map, different update frequencies are used. For instance, high-level strategic influence maps need less updates per second than low-level tactical influence maps, since tactical maps are based on individual actions and tasks that occur more quickly. Strate-



(a) Propagator with 0.1 decay. (b) Propagator with 0.5 decay. (c) Propagator with 0.9 decay.

Figure 3.14: Comparison of different exponential decay values.

gic influence maps help make decisions that will define the outcome of the game, therefore, influence may spread slowly.

In either case, no influence map needs to be updated at the same refresh rate as the graphics engine of the game. One update per second (1Hz) is enough for high-level strategic maps, while up to 5 updates per second (5Hz) are sufficient for low level tactical maps [Cha11], which is good because this reduces the computation overhead needed for the influence map to work with other game subsystems (i.e., graphics, network, input, etc..).

3.2.4 Detailed Algorithm

The algorithm for influence mapping is similar to a blurring process that any photo editing software, such as, Adobe Photoshop or Gimp implements. Influence is set on the influence map and the algorithm repeatedly blurs the map to spread the influence from the source outward, to its neighboring nodes [Cha11]. The blurring process is a low-pass filter method as used in image processing to smooth images.

In order to function, influence mapping needs to have propagators. Recall that propagators can be walls, enemies, own units, etc., and since they can move or cease to exist (i.e., unit death), they have to be updated in the influence map. The creation of the propagators is done while loading a game level (e.g., obstacles, walls, etc.), or in-game (e.g., units, grenades, sound) and delivers influence while active. Then, the influence map spreads the influence.

Influence mapping algorithm consists of two steps:

- **Step 1:** While the propagator is active (i.e., enabled, alive), set influence field in the influence map.
- **Step 2:** Loop through all the cells in the influence map grid and spread influence to neighboring cells. This is done using a simple blurring method which decays with distance using an exponential fall off.

Each of these steps are periodically performed as ruled by the frequency update parameter, as described in Section 3.2.3.3.

Algorithm 6 shows the propagator updating step, which runs over all propagators (line 1), and set their influence in the influence map corresponding to their position in the game environment

Algorithm 6 Step 1: Update Propagators

```

1: for all propagators P do
2:   influence[x,y] = set propagator influence

```

(line 3).

Once the influence map has the correct influence value and influence position of each propagator (step 1), the spreading step occurs (step 2). Influence mapping requires the updating of a block of data, in this case a 2D array, and in order for the algorithm to work with a complete version of the data, double buffering is needed. Double buffering is done by using a secondary influence array to store new influence values. Once the calculations are done on the whole block of data, one copies the values into the main influences array.

Linear interpolation (Lerp) is a curve fitting method using linear polynomials, used in Algorithm 7 to find a value in between the provided initial value and final value. The momentum parameter (see Section 3.2.3.1) is used to weight the influence of the final value over the initial value.

$$lerp = initialValue + (finalValue - initialValue) * weight \quad (3.2)$$

Algorithm 7 Step 2: Update Propagation

```

1: for x = 0 to width do
2:   for y = 0 to height do
3:     maxInfluence = 0
4:     minInfluence = 0
5:     n = getNeighbors(x,y)
6:     for all neighbors C in n do
7:       tmpInfluence = influencesBuffer[x,y] * Math.Exp( -Decay * C.Distance )
8:       maxInfluence = Max( tmpInfluence, maxInfluence )
9:       minInfluence = Min( tmpInfluence, minInfluence )
10:      if Abs(minInfluence) > maxInfluence then
11:        influences[x,y] = Lerp( influencesBuffer[x,y], minInfluence, momentum )
12:      else
13:        influences[x, y] = Lerp( influencesBuffer[x,y], maxInfluence, momentum )

```

Algorithm 7 describes the propagation process. For each cell in the influence map, the algorithm resets the maximum and minimum influence to 0, and retrieves the neighboring cells of the current x and y position (lines 1-5). For all the neighbors of the current position, a temporary influence is calculated with the current cell influence from the influence buffer map, multiplied by the decay factor (see Section 3.2.3.2) over the distance of the neighbor (line 6). The influence buffer map holds the influence map information for double buffering. The maximum influence and minimum influence are updated (lines 7-8). If the absolute minimum influence is higher than the maximum influence, a linear interpolation of the influence value is calculated with the minimum influence and set to the influence map (lines 9-11). As discussed in Section 3.2.3.1, the momentum parameter will control how fast the linear interpolation fades, and it is set as the weight in the linear interpolation calculation. Otherwise, if the absolute minimum influence value is lower than or equal to the maximum influence, the influence map is updated with the linear interpolation of the influence buffer value, this time with the maximum influence (line 12-13).

Influence Map-Based Pathfinding Algorithms in Video Games

Influence mapping runs at the same time as the game, at a lower update frequency (see Section 3.2.3.3), and maintains influence from propagators up-to-date. Therefore, game agents are able to use influence map information and combine them with pathfinding algorithms.

3.3 Pathfinding with Influence Maps

Previous work has been done on using influence maps for reactive game agents in RTS-like games, either for a one-step look-ahead motion planning or for purely strategical or tactical purposes. But, as far as we are aware, there is no algorithm that combines pathfinding and influence maps.

3.3.1 Leading Idea

The main idea of this algorithm is to use existing search algorithms, namely Dijkstra's algorithm and A* algorithm, and combine their search with information from influence maps, in order to change the way the algorithms behave. Introducing the values from the influence maps search algorithms changes the resulting path, and therefore changes the algorithms behaviors. By having influence maps changing the cost to travel among nodes, Dijkstra's and A* algorithms find novel paths, avoiding or moving towards the repulsive or attractive influences.

From here on, we assume attractive influence has negative values, so the path finder will converge to node with more negative influence values. Repulsive influence has positive values, which, the path finder will avoid nodes with more positive influence values.

As described in Section 2.4, search algorithms choose the next node based on a cost function $F(n)$. Dijkstra's algorithm uses $F(n) = C(n)$, where $C(n)$ is the cost from the start node to the current node. A* algorithm uses $F(n) = C(n) + H(n)$, where $C(n)$ is the cost so far value, and $H(n)$ is the estimation to the goal, provided by a heuristic. So the two algorithms have specific cost metric that they follow, and they both try to optimize their respective metric, whether it is the shortest path they are finding in regard to distance, or the faster path in regard to time, both algorithms will search an optimal or near-optimal path.

For search minimization, it is clear where influence maps can change the rules of path finders. Integrating the influence map values into the $F(n)$ calculation of the search algorithms, the resulting cost will be either lower or higher depending on the influence in the evaluated node. Attractive influence will lower the cost, and repulsive influence will raise the cost. With that in mind, upon the selection of the best node to explore, the nodes influenced by attractive influence will be examined first, and nodes with repulsive influence will be examined later or not at all.

This results in guiding the search through the attractive influence (negative values) and tends to avoid the repulsive influence (positive values). A* and point-to-point version of Dijkstra's algorithm will visit less nodes, improving search times and memory consumption. As the searching algorithms avoid the exploration of nodes that have repulsive influence values, and favor the nodes that have attractive influence values, the search space is considerably reduced depending on the position of the influence map propagators.

Figure 3.15 shows how a game world map (Figure 3.15 (a)) can be combined with an influence map (Figure 3.15 (b)) into an overlay map (Figure 3.15 (c)).

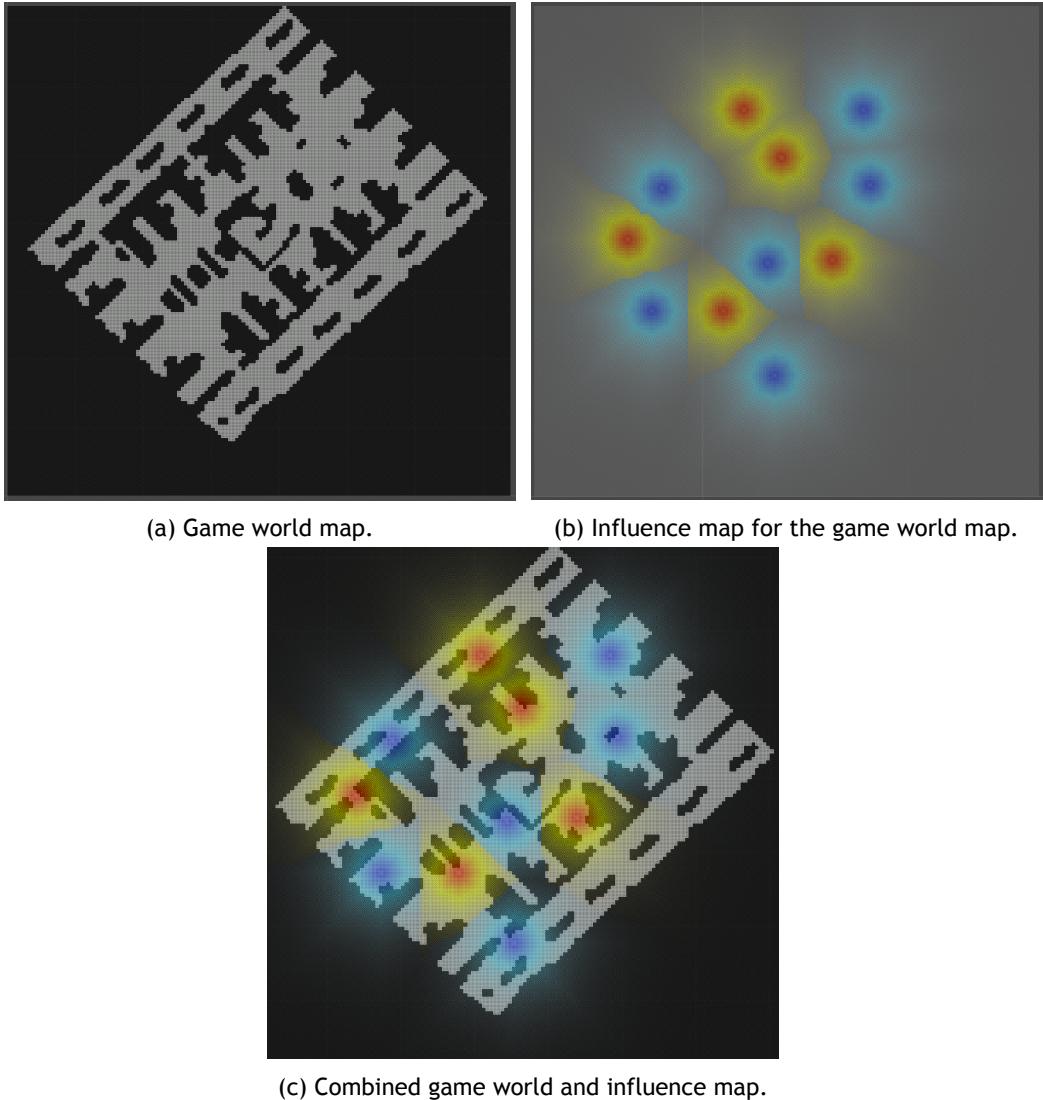


Figure 3.15: Representation of grid based game environment and grid based influence map.

For example, an influence map containing the information of recent events, such as, fires, in which the areas that contain fire will have positive influence (repulsive), the search algorithm will automatically avoid those areas, favoring areas where there are lower or no influences at all. On games where game agents are attracted to sound, an influence map could represent the sound made by human players, and whenever a sound is made, negative/attractive influence can be set on the map, and the game agent could search with that information, and would inherently be attracted to these areas.

Influence maps can be used to represent almost anything, depending on the game, and with properties such as momentum and decay, these allow the customization of the information and resulting searches, and ultimately altering the resulting path. For instance, high momentum can be used on influence maps for explosions where the influence would fade relatively fast, compared with an influence map representing the spreading of fire, in which the fire spreads slowly, therefore, having a slow momentum.

3.3.2 Dijkstra's Algorithm with Influence Maps

The main idea when combining influence maps with Dijkstra's algorithm is to change the final cost function from $F(n) = C(n)$ to $F(n) = C(n) + I(n)$, where $I(n)$ is the influence value of node n . Algorithm 8 describes the changes made to point-to-point version of Dijkstra's algorithm.

Algorithm 8 Dijkstra's search algorithm with influence maps.

```

1: Clear start node's parent and G cost.
2: Add start node to open list
3: while open list is not empty do
4:   n = best node from open list
5:   if n is goal then
6:     Reconstruct path
7:   for all Successor nodes S of N do
8:     Get I(n)
9:     Calculate new F cost ( c(n) + I(n) )
10:    if S is unexplored then
11:      Set S's F cost to new F cost
12:      Set S's Parent to n
13:      Add S to open list
14:    else if S is on the open list and S's F cost is greater than new F cost then
15:      Set S's G cost to new F cost
16:      Set S's Parent to n
17:    Add n to closed list
18: Reconstruct Path

```

The resulting changes to the algorithm are visible in visual aspect of the encountered paths.

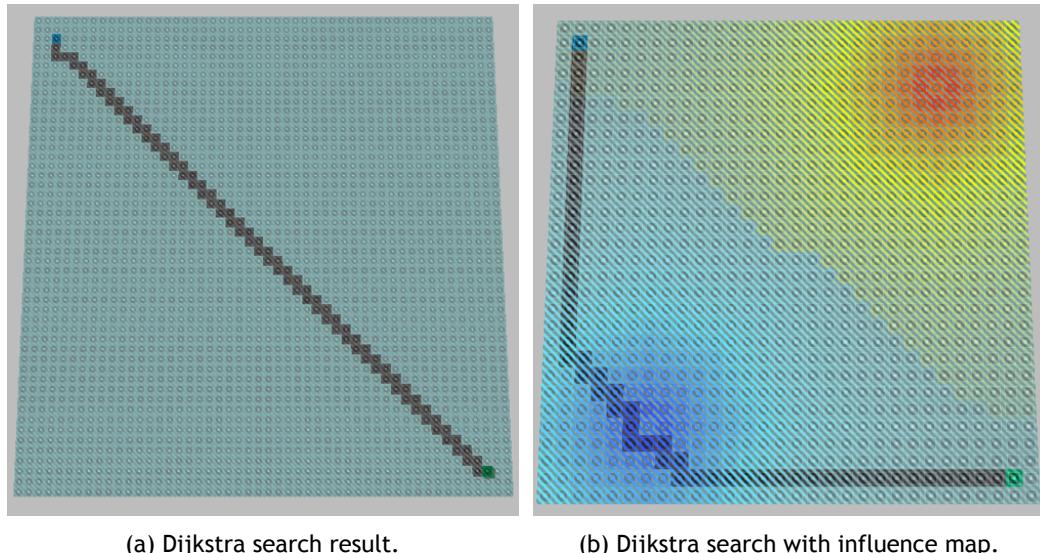


Figure 3.16: Dijkstra algorithm search comparison.

In Figure 3.16 (a) Dijkstra's illustrates the resulting search path from the start node (up left) to the goal node (down right), and in Figure 3.16 (b) illustrates the same search with influence maps applied, with an attractive propagator on the bottom left (in dark blue, more intense, to light blue, less intense, tones) and a negative propagator on the top right side of the map (in red, more intense, to yellow, less intense, tones). The grey blocks are resulting path and the light blue blocks are the visited nodes. The full map in Figure 3.16 (a) is blue because Dijkstra's algorithm searches the whole map as described in section 2.4.1.

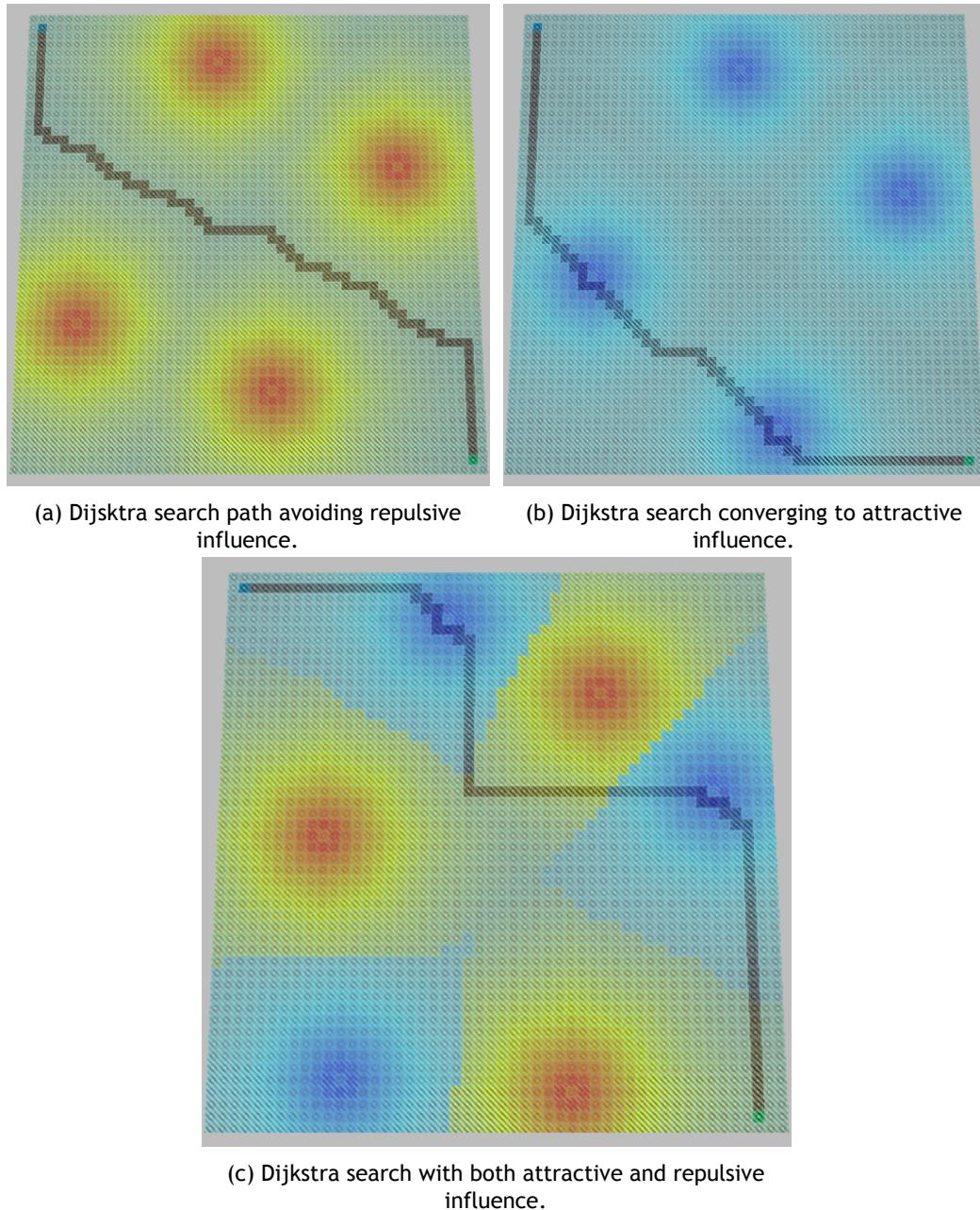


Figure 3.17: Dijkstra algorithm search comparison with influence maps.

Figure 3.17 shows a comparison of the same search with different influence maps. In Figure 3.17 (a), only repulsors are used; hence the path passing between them. In Figure 3.17 (b), only attractors are used, and the resulting path passes right inside of two of the attractive influence zones of propagators. In Figure 3.17 (c), the influence map has various influence sources, both attractive and repulsive, in which the resulting path passes through two attractive zones and avoids the repulsive zone in between.

3.3.3 A* Algorithm with Influence Maps

Combining influence maps with A* algorithm also allows results in smoother paths, faster search and less memory usage. However, in A* we also consider $H(n)$, thus the cost function becomes

Influence Map-Based Pathfinding Algorithms in Video Games

$F(n) = C(n) + H(n) + I(n)$, where $I(n)$ is the influence value of n . Algorithm 9 describes the changes made to the original A* algorithm described in Section 2.4.2.

Algorithm 9 A* search algorithm with influence map

```

1: Clear start node's Parent and cost values ( F, C, H )
2: Add start node to open list
3: while open list is not empty do
4:   n = pop best node from open list
5:   if n is goal then
6:     Reconstruct path
7:     Add n to closed list
8:   for all successor nodes S of n do
9:     if S is in open list then
10:      continue;
11:    Calculate new C cost
12:    if S is not in open list or new C cost is lower than S's G cost then
13:      Set S's Parent to n
14:      Set S's C cost to new C cost
15:      Set S's F cost to S's C cost + Heuristic estimate from S to Goal + Influence Value of n
16:      if S is not in open list then
17:        Add S into open list
18: Reconstruct No Path

```

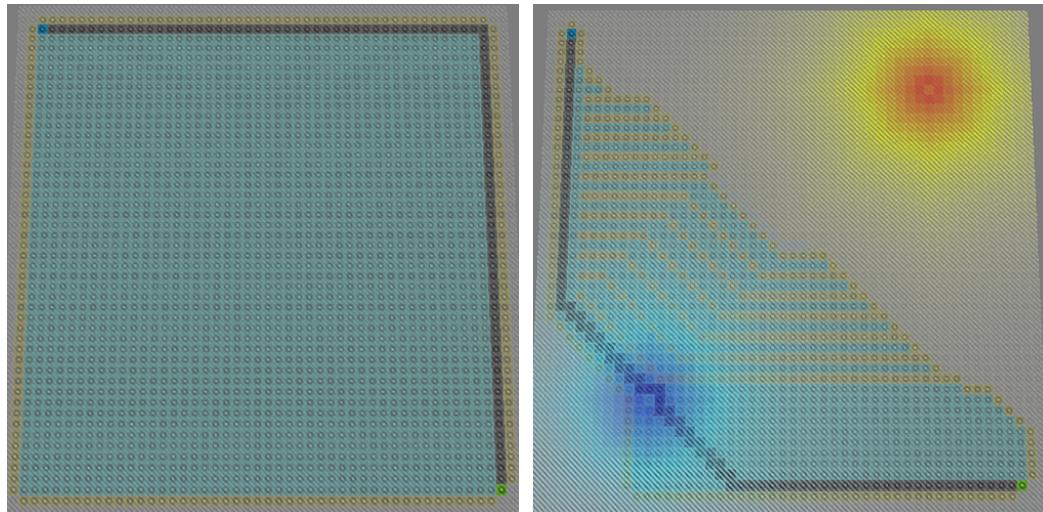


Figure 3.18: A* algorithm search comparison.

Figure 3.18 shows the resulting path of the A* algorithm without (Figure 3.18 (a)) and with influence maps (Figure 3.18 (b)). In Figure 3.18 (a) the path is found and the resulting search space shows a large area explored, recall that the light blue blocks represent the visited nodes (closed nodes). In Figure 3.18 (b) the path found passes through the attractor, being the resulting search space smaller when compared to the one in Figure 3.18 (a). Notice that there is also a great number of nodes that are left in the open list to be explored, more specifically those light yellow blocks, that represent nodes to be visited (open list).

Figure 3.19 shows the differences in the same search with different influence maps. In Figure 3.19 (a), only repulsive influence is used. The resulting path passes in between the repulsive propagators. Also, the resulting search space exploration is considerably smaller. In Figure 3.19 (b), only attractive influence is used and the resulting path will pass through one attractive

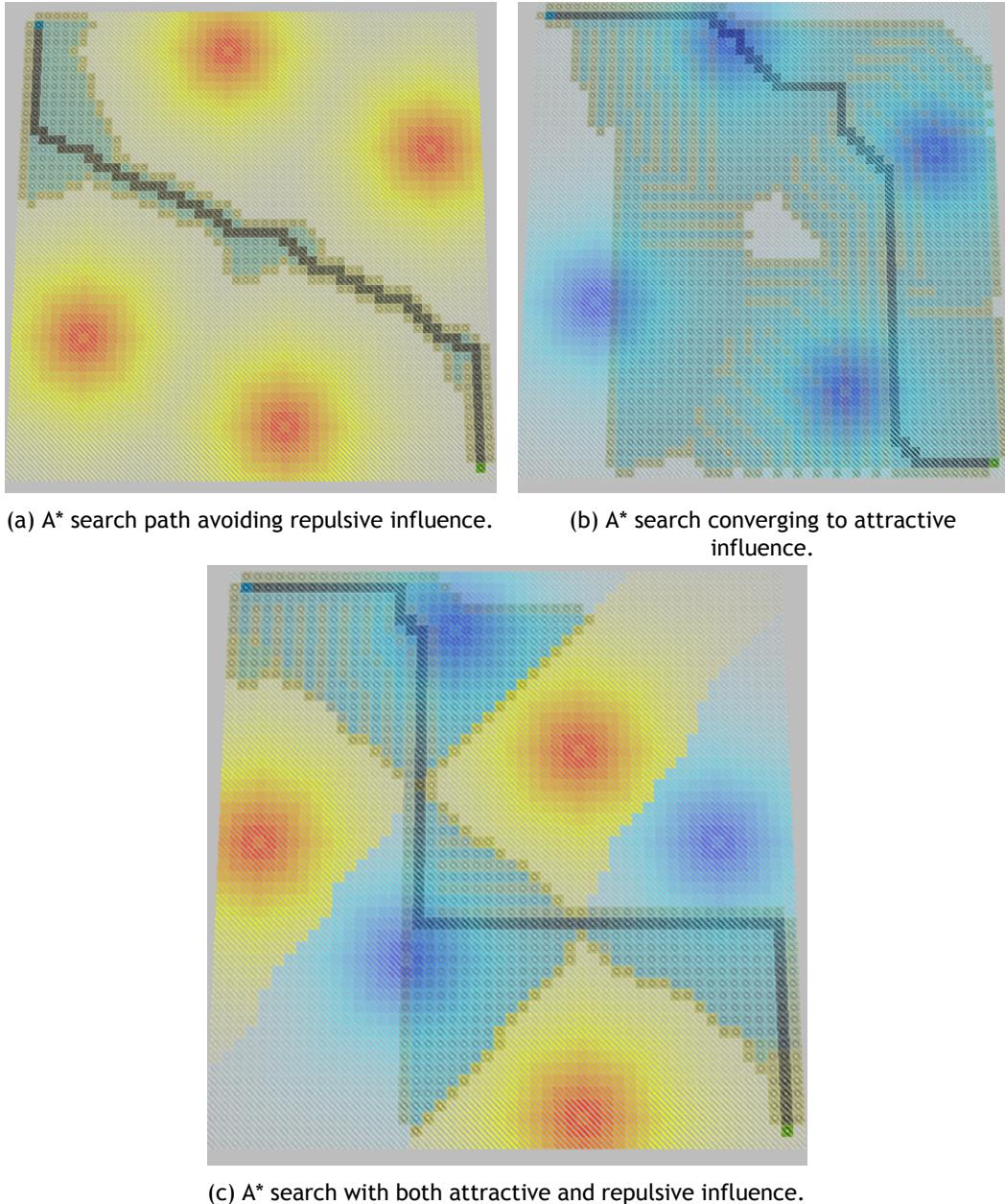


Figure 3.19: A* algorithm search comparison with influence maps.

zone. The resulting search space is also reduced with many nodes left to be visited. Figure 3.19 (c) illustrates various propagators, both attractive and repulsive. The resulting path tries to use as much attractive influence as it gets while avoiding the repulsive influences. The resulting search space is a mix between (a) and (b), but still less than (b). We conclude that repulsive propagators provide a better solution for path finding.

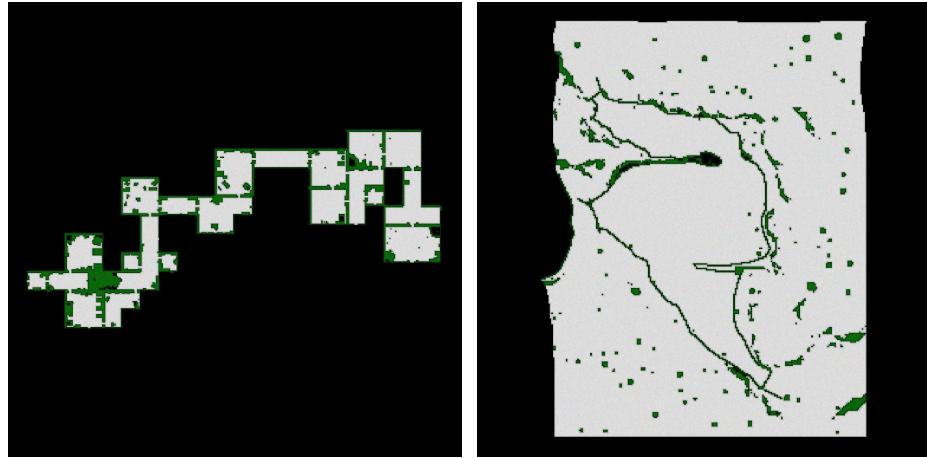
3.4 Tests and Discussion

This section describes individually and comparatively the analyses of Dijkstra and A* pathfinding algorithms and their counterparts combined with influence maps. These implementations were tested on a Intel(R) Core(TM)2 Duo E8400 at 3.00Ghz with 8192MBytes of DDR2 RAM and a XFX

Influence Map-Based Pathfinding Algorithms in Video Games

HD5700 graphics card with 1024 MBytes of GDDR5 RAM.

The tests were performed using a set of twenty maps from the HOG2 map repository [Stu12]. The maps chosen for the tests are from Dragon Age: Origin [dra] and Baldurs Gate II [bal] games, as illustrated in Figure 3.20 and Figure 3.22.

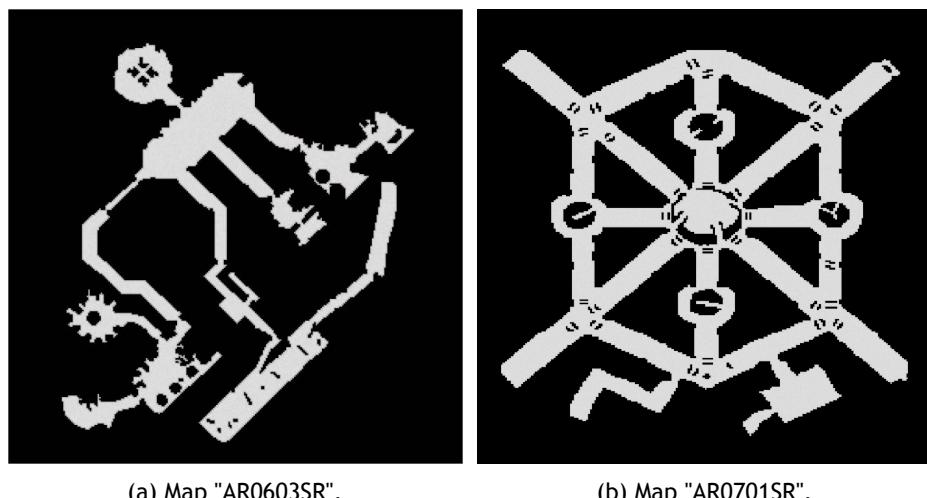


(a) Map "den600d".

(b) Map "brc501d".

Figure 3.20: Maps from Dragon Age: Origins

Figure 3.20 shows two maps used from the Dragon Age: Origin games, featuring a high number of rooms, and a high level of symmetry.



(a) Map "AR0603SR".

(b) Map "AR0701SR".

Figure 3.21: Maps from Baldur's Gate II.

Figure 3.22 shows two example maps used from the Baldur's Gate II game, featuring open areas, and some level of complexity.

Figure 3.22 show two maps used for the tests with propagators positions are random locations, with the goal and starting node at opposite locations of the map.

In order to test search algorithms combined with influence maps, the propagators were defined at pseudo-random locations inside the map, each having at least 10 propagators of both attractive and repulsive influences.

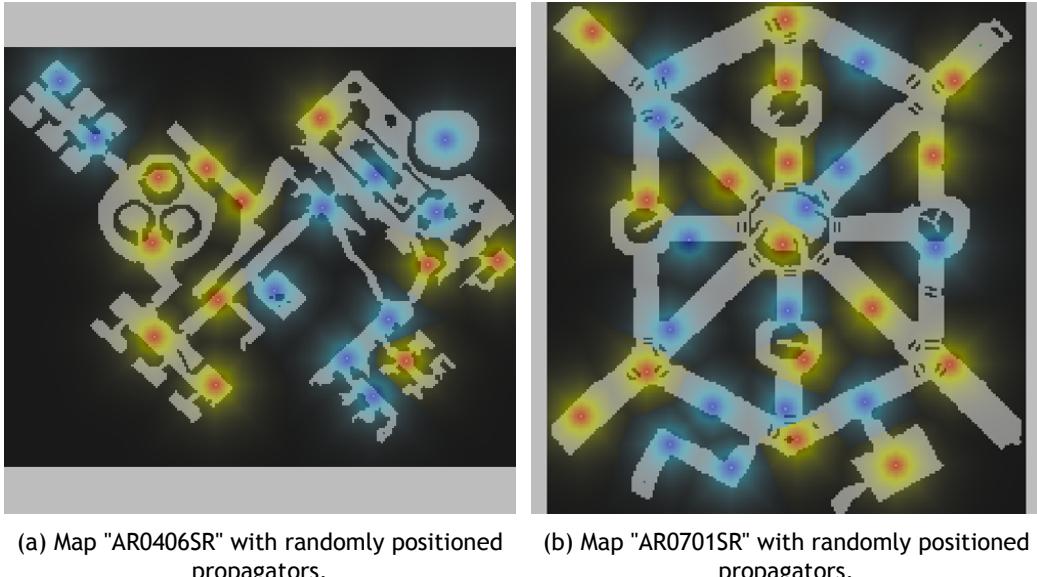


Figure 3.22: HOG2 game maps with influence maps.

3.5 Results

The algorithms tested were A*, A* combined with influence maps, point-to-point version of Dijkstra and point-to-point version of Dijkstra combined with influence maps. Original Dijkstra was not tested due to the fact that it is expected that influence maps aid to avoid node exploration and therefore achieve better results, and since Dijkstra's algorithm always searches the whole search space, the results would not be noticeable.

The following tables represent the results of the tests performed over twenty maps. In each table, the first column (**Algorithm**) indicates the algorithm used. The second column (**Average Memory**) represents the average memory spent of each iteration. The third column (**Peak Memory**) represents the maximum memory used in each iteration. The fourth column (**Iterations**) represents the number of steps the algorithm took to complete, which is the number of nodes explored in total. Finally, the last column (**Time**) represents the time spent by the algorithm to finish.

Table 3.1, shows the average result of all previous tests. It is used to calculate the overall gain in performance, demonstrated next in Table 3.2.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	4010.69	7971.63	248108.00	279206.00
A* with Influence Map	2722.18	5406.66	154528.00	101443.00
Dijkstra	5471.05	10887.96	463459.00	837854.00
Dijkstra with Influence Map	5230.28	10395.68	441942.00	741956.50

Table 3.1: Average of all test results.

In A* algorithm, the results show a 47% gain in average memory and peak memory usage. The number of iterations has a 1.60 gain, meaning a reduction in iterations by 60%, which ultimately affects time search, which has a gain over 2.75 indicating a overall speed up of 175%.

The results for Dijkstra's algorithm show a slight decrease in both average and peak memory usage by 4%. The number of iterations has a 5% also reduced factor, and the gain in time shows

Influence Map-Based Pathfinding Algorithms in Video Games

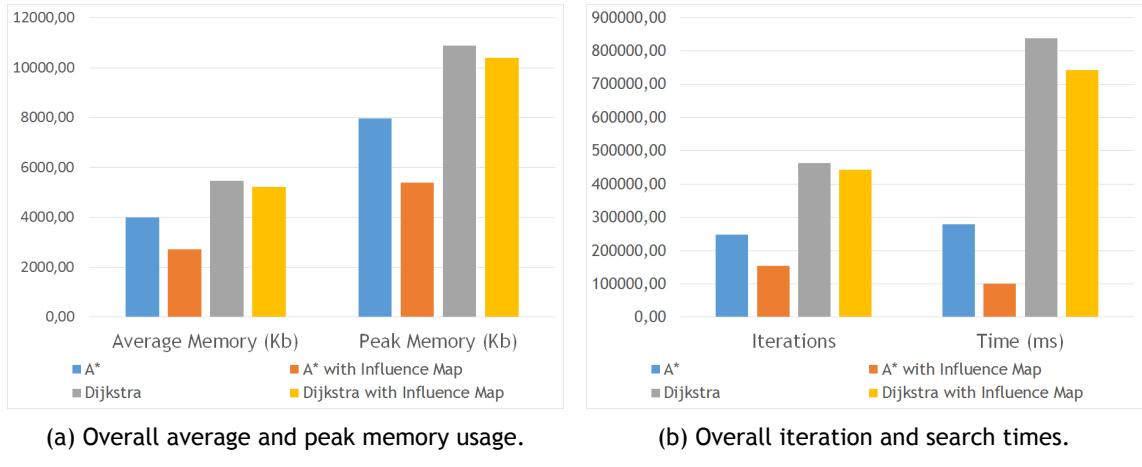


Figure 3.23: Overall results.

a 12% improvement over the original algorithm.

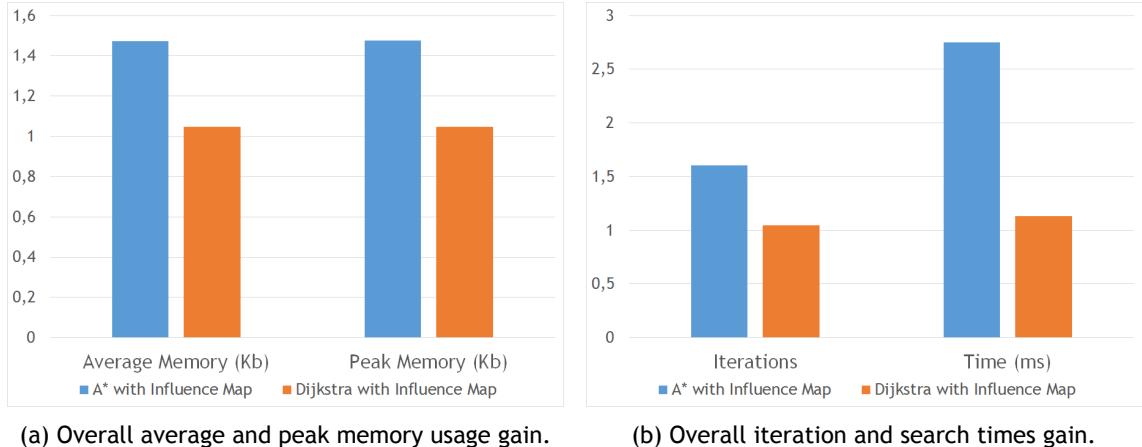


Figure 3.24: Overall gain results.

Algorithm	Average Memory (%)	Peak Memory (%)	Iterations %	Time (%)
A* with Influence Map	1.473336027	1.474409437	1.605586043	2.752343681
Dijkstra with Influence Map	1.046033725	1.047354407	1.048687384	1.129249491

Table 3.2: Overall Speedup.

Table 3.2 illustrates the overall gain in performance from Dijkstra's algorithm and A* algorithm combined with Influence maps. Note that the gain is the ratio of the old value to the new value.

The biggest contributing factor for the decreasing in search times is the number of nodes visited. A* profits greatly from influence maps due to its nature of using a heuristic function to guide its search. With influence maps, A*'s heuristic will be accelerated when the next node explore is near an attractive propagator. $F(n)$ value will decrease due to the influence of the propagator, therefore, $F(n)$ will under-estimate the distance to the goal. On the other hand, when the next node to explore is near a repulsive propagator, $F(n)$ will have a larger value, postponing the exploration of the node. Since A* chooses its next node to explore based on the $F(n)$ value, A* has better results than Dijkstra.

In short, nodes that are next to repulsive zones, will be examined later, and nodes that are next to attractive zones will be examined first, resulting in the search of less nodes, and increase in

search speed.

Dijkstra's algorithm combined with influence map does not show significant improvement as seen with A*, because Dijkstra's chooses the next node based on the $C(n)$ value, which is the node closest to the starting node. Due to the fact that there is no heuristic, influence maps only slightly change the cost to travel of each node.

The tests were performed using 1 and -1 for the repulsive and attractive propagators, respectively. Higher values can be used, but one must be aware of the scaling problem, as mentioned in Section 2.4.2.4. If the influence value is much higher than the cost function or the heuristic, the overall search may be biased to the influence cost only, and vice versa, if the influence value is much lower either $C(n)$ or $H(n)$ will have too much influence on $F(n)$.

These preliminary results show the benefits of using influence maps combined with A*, since search times, iterations and memory usage are greatly reduced.

3.6 Final Remarks

Influence map-based pathfinding opens a new possibility in pathfinding. Paths become aware of their surroundings, events or game specific characteristics (e.g., sound, faction location, line of sight, etc.). Game agents are therefore better informed at path planning, and changing the way they navigate through their environments.

For high-level strategic games, influence maps will expose general goals to the path finders, leading them to those goals faster. In low-level tactical decision making, influence maps expose recent events, i.e., wars, grenades exploding, sound propagation, and path finders will plan and re-plan their paths according to these changes.

In a scenario where a bomb explodes, the explosion causes a repulsive influence in its blast radius. The resulting influence map-based pathfinding searches will take the explosion into consideration when calculating a path, thus avoiding the blast radius (that might contain fire, or hazardous materials that would penalize the game agent).

Influence maps are easy to setup and integrate in any game engine, providing useful information for path finders, allowing game agents to react to changing environments as well as in-game events. Also, influence maps will allow for a faster search, with less memory consumption, and visually smoother paths.

Chapter 4

Conclusions

The use of influence maps combined with existing search algorithms has proven to be a successful approach. Point-to-point version of Dijkstra was slightly improved due to the fact that node exploration is not guided by an heuristic, so more nodes are searched in comparison to a heuristic-guided search algorithm (e.g., A* algorithm). We noticed that the influence map-based A* algorithm is less expensive than the A* in terms of time and memory space.

4.1 Comparative Remarks

In dynamic game environments (i.e., recent events, changing topologies, destructible environment), influence maps provide information to path finders so they may adapt in the same fashion. Path finders will calculate their paths with up to date information about its surrounding environment.

We have shown that influence maps are flexible, i.e., can represent any kind of relevant information for the game, they can be used in almost any context. In 3D environments, influence maps combined with pathfinding can also be used, either using a 2D grid representation for each level height, for instance, levels of a building, or using other representation such as waypoint networks or area graph as shown in Section 3.2.2.

Pathfinding was improved in terms of memory consumption, and search times. Depending on how dynamic a game environment is, more computation will be needed to keep the influence map updated, which is determined by the update frequency parameter as seen in Section 3.2.3.3. In static environments, influence maps can be pre-processed when loading levels, leading to no extra computation.

The use of influence map does not create local minima issues when combined with pathfinding algorithms. Since the original nature of path finders remain the same, they do not have that issue. Worst case scenario, if the influence map were to have a propagator attracting the search through a possible area with local minima, the path finder would search more nodes in that area, and find a way out, only slowing down the search but ultimately not getting stuck.

4.2 Future Work

Future work includes but is not restricted to testing the use of influence maps combined with other search algorithms (e.g., Fringe search). Since the implemented algorithms belong to the breath-first search family, it would be interesting to implement depth-first search algorithm such as IDA* and Fringe search combined with influence maps, to determine if they also show improvements over their original versions.

Influence Map-Based Pathfinding Algorithms in Video Games

Also, the implemented search algorithms (i.e., Dijkstra, A*), that were combined with influence maps were not optimized for performance, so a natural step towards optimizing them would improve their results surely.

Bibliography

- [AAG⁺85] Takao Asano, Tetsuo Asano, Leonidas Guibas, John Hershberger, and Hiroshi Imai. Visibility-polygon search and euclidean shortest paths. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*, pages 155--164. IEEE, 1985. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4568139. 7
- [AAG⁺86] Takao Asano, Tetsuo Asano, Leonidas Guibas, John Hershberger, and Hiroshi Imai. Visibility of disjoint polygons. *Algorithmica*, 1(1-4):49--63, 1986. Available from: <http://link.springer.com/article/10.1007/BF01840436>. 7
- [ALA09] Phillipa Avery, Sushil Louis, and Benjamin Avery. Evolving coordinated spatial tactics for autonomous entities using influence maps. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 341--348. IEEE, 2009. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5286457. 26
- [Ang12] Bobby Anguelov. Video game pathfinding and improvements to discrete search on grid-based maps. Master's thesis, University of Pretoria, 2012. 11, 12, 15
- [Ark97] R. C. Arkin. Motor schema based navigation for a mobile robot. In *IEEE International Conference on Robotics and Automation*, 1097. 23
- [bal] Baldur's gate official website. Available from: <http://www.baldursgateii.com/>. 41
- [BB11] Sandy Brand and Rafael Bidarra. Parallel ripple search--scalable and efficient pathfinding for multi-core architectures. In *Motion in Games*, pages 290--303. Springer, 2011. Available from: http://link.springer.com/chapter/10.1007/978-3-642-25090-3_25. 19
- [BEH⁺03] Yngvi Bjornsson, Markus Enzenberger, Robert Holte, Jonathan Schaejfer, and Peter Yap. Comparison of different grid abstractions for pathfinding on maps. In *Proceedings of the 18th international joint conference on Artificial intelligence, IJCAI'03*, pages 1511--1512, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc. 6
- [BF03] M. Buro and T. Furtak. Rts games as test-bed for real-time ai research. In *7th Joint Conference of Information Science, Department of Computing Science, University of Alberta, Canada*, 2003. 1, 23
- [BF04] Michael Buro and Timothy Furtak. Rts games and real-time ai research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS)*, volume 6370, 2004. 1
- [Bjo05] Yngvi Bjornsson. Fringe Search: Beating A* at Pathfinding on Game Maps. In *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05), Essex University, Colchester, Essex, UK, 4-6 April, 2005*. IEEE Computer Society, 2005. 18, 20
- [BK89] J. Borenstein and Y. Koren. Real-time obstacle avoidance for fast mobile robots. *IEEE Transactions on Systems, Man, and Cybernetics*, 1989. 23

Influence Map-Based Pathfinding Algorithms in Video Games

- [Bla04] Paul E. Black. Euclidean distance. Dictionary of Algorithms and Data Structures [online], Vreda Pieterse and Paul E. Black, 2004. Available from: <http://www.nist.gov/dads/HTML/euclidndstnc.html>. 16
- [Bla06] Paul E. Black. Manhattan distance. Dictionary of Algorithms and Data Structures [online], Vreda Pieterse and Paul E. Black, 2006. Available from: <http://www.nist.gov/dads/HTML/manhattanDistance.html>. 17
- [Bla08] Paul E. Black. L_m distance. Dictionary of Algorithms and Data Structures [online], Vreda Pieterse and Paul E. Black, 2008. Available from: <http://www.nist.gov/dads/HTML/lmdistance.html>. 17
- [BMS04] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical pathfinding. *Journal of game development*, 1(1):7--28, 2004. 18
- [Boa02] Ducker Mike. Board, Ben. *Game Programming Gems 3*, chapter Area Navigation: Expanding the Path-Finding Paradigm. Charles River Media, 2002. 9
- [BS08] Maurice HJ Bergsma and Pieter Spronck. Adaptive spatial reasoning for turn-based strategy games. In *AIIDE*, 2008. Available from: <http://www.aaai.org/Papers/AIIDE/2008/AIIDE08-027.pdf>. 28
- [Buc05] Mat Buckland. *Programming Game AI by Example*. Wordware Publishing, Inc., 2005. 2
- [Bur03] M. Buro. Real-time strategy games: A new ai research challenge. In *Eighteenth International Joint Conference on Artificial Intelligence*, 2003. 23
- [Cha11] Alex J. Champandard. The mechanincs of influence mapping: Representation, algorithm & parameters, May 2011. Available from: <http://aigamedev.com/open/tutorial/influence-map-mechanics/> [cited March 2nd, 2014]. xv, 29, 30, 31, 33
- [CTS04] C. Bauckhage C. Thurau and G. Sagerer. Learning human-like movement behavior for computer games. 2004. 26
- [DB06] Douglas Demyen and Michael Buro. Efficient triangulation-based pathfinding. In *AAAI*, volume 6, pages 942--947, 2006. Available from: <http://www.aaai.org/Papers/AAAI/2006/AAAI06-148.pdf>. 21
- [Dij59] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *NUMERISCHE MATHEMATIK*, 1(1):269--271, 1959. 10
- [DP85] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of a*. *Journal of the ACM (JACM)*, 32(3):505--536, 1985. Available from: <http://dl.acm.org/citation.cfm?id=3830>. 14
- [dra] Dragon age official website. Available from: <http://www.dragonage.com/>. 41
- [Ent] Blizzard Entertainment. Starcraft official website. Available from: <http://us.blizzard.com/en-us/games/sc/>. 23
- [GMS03] Ross Graham, Hugh McCabe, and Stephen Sheridan. Pathfinding in computer games. *ITB Journal Issue Number*, 8, 2003. 9

Influence Map-Based Pathfinding Algorithms in Video Games

- [Goo10] M.A. Goodrich. Potential fields tutorial. Internet, Dec 2010. Available from: <http://students.cs.byu.edu/~cs470ta/readings/Pfields.pdf>. 25
- [Hag08] J Hagelback. Dealing with fog of war in a real time strategy game environment. In *IEEE Symposium On Computational Intelligence and Games*, 2008. 23, 25
- [HB10] Daniel Harabor and Adi Botea. Breaking path symmetries on 4-connected grid maps. In *AIIDE*, 2010. Available from: <http://www.aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/viewPDFInterstitial/2122/2541>. 22
- [HBK11] Daniel Damir Harabor, Adi Botea, and Philip Kilby. Path symmetries in undirected uniform-cost grids. In *SARA*, 2011. Available from: <http://www.aaai.org/ocs/index.php/SARA/SARA11/paper/viewPDFInterstitial/4243/4589>. 22
- [HG11] Daniel Damir Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In *AAAI*, 2011. Available from: <http://www.aaai.org/ocs/index.php/aaai/aaai11/paper/download/3761/4007>. 22
- [HJ08] J. Hagelback and S.J. Johansson. The rise of potential fields in real time strategy bots. *Artificial Intelligence and Interactive Digital Entertainment*, 2008. 23
- [HJ09a] Johan Hagelbäck and Stefan J Johansson. A multi-agent potential field-based bot for a full rts game scenario. In *AIIDE*, 2009. Available from: <http://www.aaai.org/ocs/index.php/AIIDE/AIIDE09/paper/download/838/1071>. 23
- [HJ09b] Johan Hagelbäck and Stefan J Johansson. A multiagent potential field-based bot for real-time strategy games. *International Journal of Computer Games Technology*, 2009:4, 2009. Available from: <http://dl.acm.org/citation.cfm?id=1512637>. 23
- [HNR68] P.E. Hart, N.J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100--107, 1968. 6, 14, 16, 18
- [Joh12] *Multi-Agent Potential Field Based Architectures for Real-Time Strategy Game Bots*. PhD thesis, School of Computing Blekinge Institute of Technology, 2012. 23
- [JTB11] K Stanley Julian Togelius, G Yannakakis and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions*, 2011. 5
- [KBT04] Marcelo Kallmann, Hanspeter Bieri, and Daniel Thalmann. Fully dynamic constrained delaunay triangulations. In *Geometric modeling for scientific visualization*, pages 241--257. Springer, 2004. Available from: http://link.springer.com/chapter/10.1007/978-3-662-07443-5_15. 21
- [Kha86] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*, 1986. 23
- [Kor85] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97--109, September 1985. 18
- [LM05] Sushil J Louis and Chris Miles. Playing to learn: Case-injected genetic algorithms for learning to play computer games. *Evolutionary Computation, IEEE Transactions on*, 9(6):669--681, 2005. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1545942. 26

- [LMAB⁺09] Hua Li, Héctor Muñoz-Avila, Diane Bramsen, Chad Hogg, and Rafael Alonso. Spatial event prediction by combining value function approximation and case-based reasoning. In *Case-Based Reasoning Research and Development*, pages 465--478. Springer, 2009. Available from: http://link.springer.com/chapter/10.1007/978-3-642-02998-1_33. 29
- [LMP⁺13] Simon M Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, Julian Togelius, Peter I Cowling, Michael Buro, Michal Bida, Adi Botea, Bruno Bouzy, et al. *Artificial and Computational Intelligence in Games*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik GmbH, 2013. 2, 5, 14, 18, 19, 21, 22
- [Mil07] Christopher E Miles. *Co-evolving real-time strategy game players*. ProQuest, 2007. 26
- [ML06a] Chris Miles and Sushil J Louis. Co-evolving real-time strategy game playing influence map trees with genetic algorithms. In *Proceedings of the International Congress on Evolutionary Computation, Portland, Oregon*, pages 0--999, 2006. 26
- [ML06b] Chris Miles and Sushil J Louis. Towards the co-evolution of influence map tree based strategy game players. In *Computational Intelligence and Games, 2006 IEEE Symposium on*, pages 75--82. IEEE, 2006. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4100111. 26
- [MQLL07] Chris Miles, Juan Quiroz, Ryan Leigh, and Sushil J Louis. Co-evolving influence map tree based strategy game players. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 88--95. IEEE, 2007. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4219028. 26
- [Nil98] Nils J Nilsson. *Artificial Intelligence: A New Synthesis*. Elsevier, 1998. 1
- [OP11] Ramon Oliva and Nuria Pelechano. Automatic generation of suboptimal navmeshes. In *Motion in Games*, pages 328--339. Springer, 2011. Available from: http://link.springer.com/chapter/10.1007/978-3-642-25090-3_28. 9
- [Pat11] Amit Patel. Amit's notes about path-finding, September 2011. Available from: <http://theory.stanford.edu/~amitp/GameProgramming/> [cited March 2nd, 2014]. 16, 18
- [Pea84] Judea Pearl. Heuristics: intelligent search strategies for computer problem solving. 1984. Available from: <http://www.osti.gov/scitech/biblio/5127296>. 14, 19
- [PMG98] David Poole, Alan Mackworth, and Randy Goebel. *Computational Intelligence*. Oxford University Press Oxford, 1998. 1
- [Poh70] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3):193--204, 1970. Available from: <http://www.sciencedirect.com/science/article/pii/000437027090007X>. 18
- [Pot00] Dave C Pottinger. Terrain analysis in realtime strategy games. In *2000 Game Developer Conference Proceedings*, 2000. 28
- [RM94] A. Reinefeld and T. A. Marsland. Enhanced Iterative-Deepening Search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(7):701--710, July 1994. 18

Influence Map-Based Pathfinding Algorithms in Video Games

- [RNC⁺95] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Englewood Cliffs, 1995. 1
- [S00] Rabin S. *Game Programming Gems*, chapter A* Aesthetic Optimizations. Charles River Media, 2000. 6
- [SB05] Nathan Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *AAAI*, volume 5, pages 1392--1397, 2005. Available from: <http://www.aaai.org/Papers/AAAI/2005/AAAI05-221.pdf>. 18
- [SDZ09] T. Lin S. Ding, N. Tang and S. Zhao. Rts-gameflow: A new evaluation framework for rts games. *International Conference on Computational Intelligence and Software Engineering*, 2009. 23
- [sim] Simcity official website. Available from: <http://www.simcity.com>. 29
- [SJ12] Johan Svensson and Stefan J Johansson. Influence map-based controllers for ms. pacman and the ghosts. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 257--264. IEEE, 2012. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6374164. 26
- [SSGF11] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume One*, pages 662--667. AAAI Press, 2011. Available from: <http://dl.acm.org/citation.cfm?id=2283504>. 18
- [Stu] Ensemble Studios. Age of empires. Available from: <http://www.ensemblestudios.com/>. 23
- [Stu12] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 -- 148, 2012. Available from: <http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf>. 41, 53
- [Sul11] Muhammad Sajjad and Muhammad Mansur ul Islam. Asymmetric potential fields. Master's thesis, School of Computing Blekinge Institute of Technology, 2011. xv, 25
- [Swe04] PM Sweetser. Strategic decision-making with neural networks and influence maps. 2004. 28
- [Swe06] Penelope Sweetser. *An Emergent Approach to Game Design - Development and Play*. PhD thesis, School of Information Technology and Electrical Engineering, The University of Queensland, 2006. 27
- [TL03] T. Rofer T. Laue. A behavior architecture for autonomous mobile robots based on potential fields. 2003. 25
- [Ton10] Matt Tonks. Navigation mesh reference. *Epic Games, Inc.) Retrieved July, 23:2010, 2010.* 9
- [Toz01] Paul Tozour. Influence mapping. *Game programming gems*, 2:287--297, 2001. 26, 27, 28

Influence Map-Based Pathfinding Algorithms in Video Games

- [Toz04] Paul Tozour. Using a spatial database for runtime spatial analysis. *AI Game Programming Wisdom*, 2:381--390, 2004. 28
- [TR93] T.Balch and R.Arkin. Avoiding the past: A simple but effeteve strategy for reactive navigation. *IEEE International Conference on Robotics and Automation*, 1993. 26
- [WG00] Steve Woodcock and AI Game. The state of the industry. *Gamasutra*, November, 1, 2000. 27, 28
- [WG08] N. Wirth and M. Gallagher. An influence map model for playing ms. pac-man. In *IEEE Symposium on Computational INtelligence and Games*, 2008. 26
- [Whi02] Christensen Christopher White, Stephen. *Game Programming Gems 3*, chapter A Fast Approach to Navigation Meshes. Charles River Media, 2002. 9
- [Yap02] Peter Yap. Grid-based path-finding. In *15th Canadian Conference on Artificial Intelligence*, 2002. 6
- [YBHS11] Peter Yap, Neil Burch, Robert C Holte, and Jonathan Schaeffer. Block a*: Database-driven search with applications in any-angle path-planning. In *AAAI*, 2011. 18
- [Zob69] Albert L Zobrist. A model of visual organization for the game of go. In *Proceedings of the May 14-16, 1969, spring joint computer conference*, pages 103--112. ACM, 1969. Available from: <http://dl.acm.org/citation.cfm?id=1476819>. 26

Appendix A

Map Dataset and Results

This appendix shows the performance results (memory space and time) concerning twenty game maps taken from HOG2 repository [Stu12].

Influence Map-Based Pathfinding Algorithms in Video Games

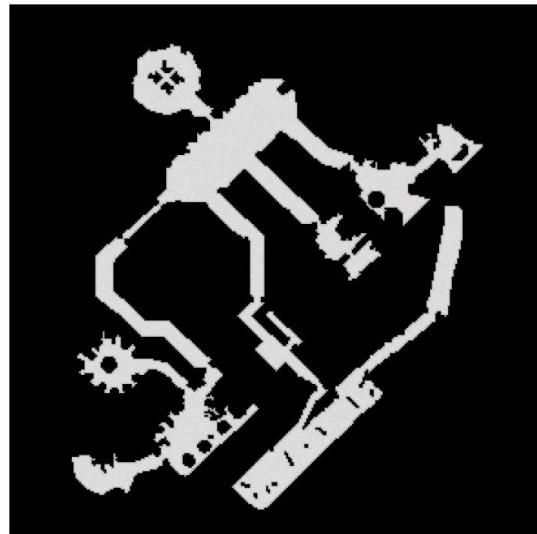
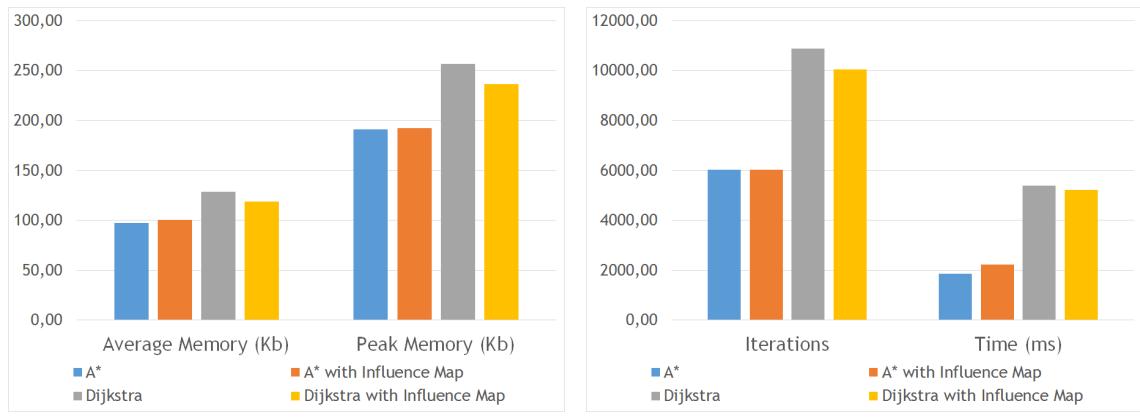


Figure A.1: Map "AR0603SR".



(a) Map "AR0603SR" - Average and peak memory usage.

(b) Map "AR0603SR" - Iterations and search time.

Figure A.2: Map "AR0603SR" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	97.36	191.06	6015.00	1866.50
A* with Influence Map	100.51	192.41	6020.00	2236.00
Dijkstra	128.51	256.73	10886.00	5394.50
Dijkstra with Influence Map	118.69	236.13	10047.00	5218.50

Table A.1: Map "AR0603SR" test results.

Influence Map-Based Pathfinding Algorithms in Video Games



Figure A.3: Map "orz200d".

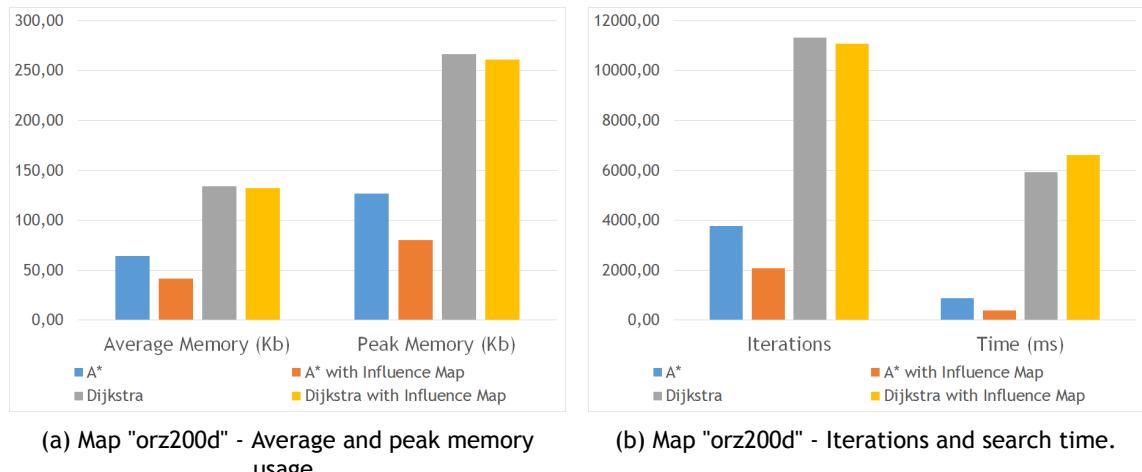


Figure A.4: Map "orz200d" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	64.34	126.75	3770.00	879.50
A* with Influence Map	41.76	80.06	2072.00	385.00
Dijkstra	134.27	266.34	11317.00	5933.50
Dijkstra with Influence Map	132.13	260.98	11079.00	6602.00

Table A.2: Map "orz200d" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

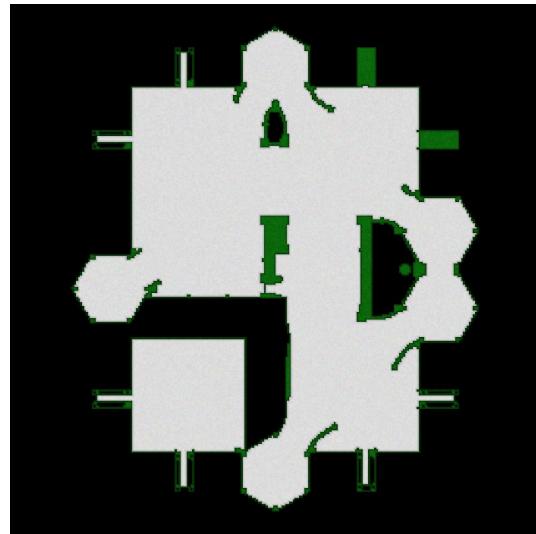
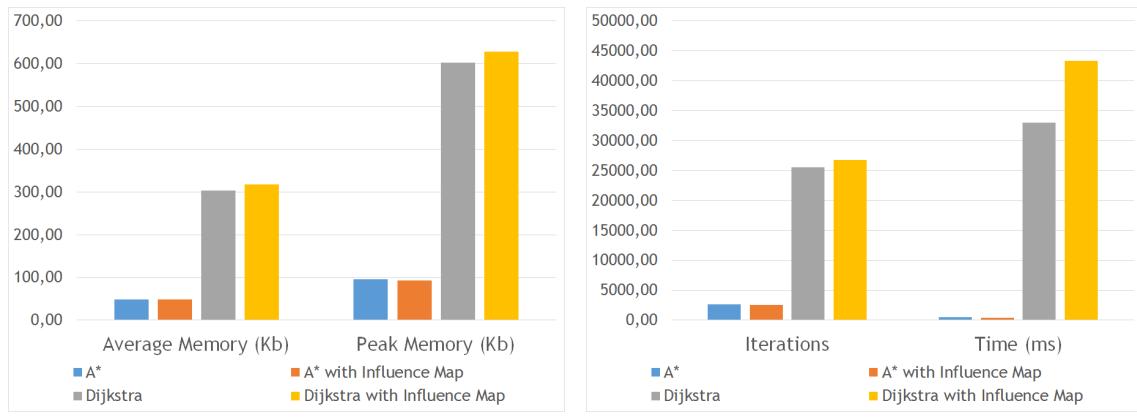


Figure A.5: Map "den502d".



(a) Map "den502d" - Average and peak memory usage.

(b) Map "den502d" - Iterations and search time.

Figure A.6: Map "den502d" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	632.55	1260.13	39255.00	69275.00
A* with Influence Map	200.79	402.09	9999.00	5662.00
Dijkstra	556.24	1106.55	47191.00	103646.00
Dijkstra with Influence Map	556.77	1106.34	47182.00	94161.00

Table A.3: Map "den502d" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

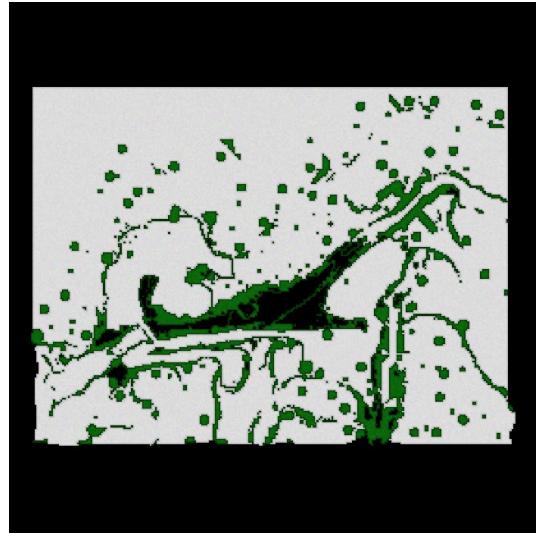
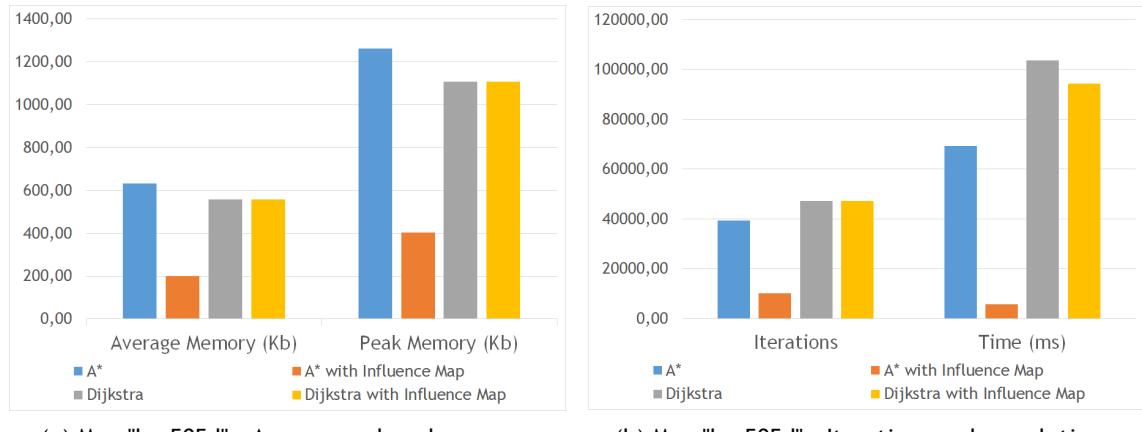


Figure A.7: Map "bcr505d".



(a) Map "bcr505d" - Average and peak memory usage.

(b) Map "bcr505d" - Iterations and search time.

Figure A.8: Map "bcr505d" test results.

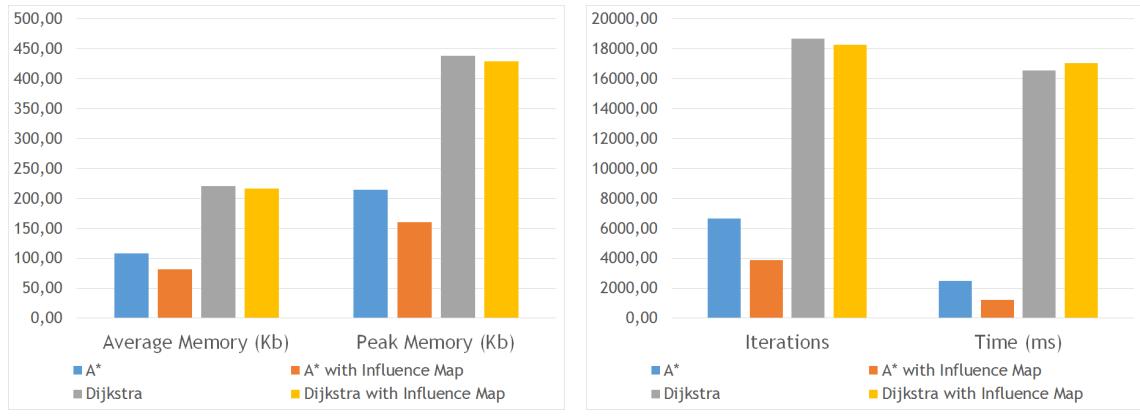
Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	164.88	331.53	10290.00	5629.50
A* with Influence Map	137.27	283.53	8348.00	3616.50
Dijkstra	236.38	471.66	20063.00	18494.50
Dijkstra with Influence Map	232.84	463.73	19740.00	20418.50

Table A.4: Map "bcr505d" test results.

Influence Map-Based Pathfinding Algorithms in Video Games



Figure A.9: Map "lgt604d".



(a) Map "lgt604d" - Average and peak memory usage.

(b) Map "lgt604d" - Iterations and search time.

Figure A.10: Map "lgt604d" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	108.34	214.38	6637.00	2484.50
A* with Influence Map	82.07	160.06	3895.00	1228.00
Dijkstra	220.33	437.44	18645.00	16519.00
Dijkstra with Influence Map	216.24	428.63	18263.00	17021.00

Table A.5: Map "lgt604d" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

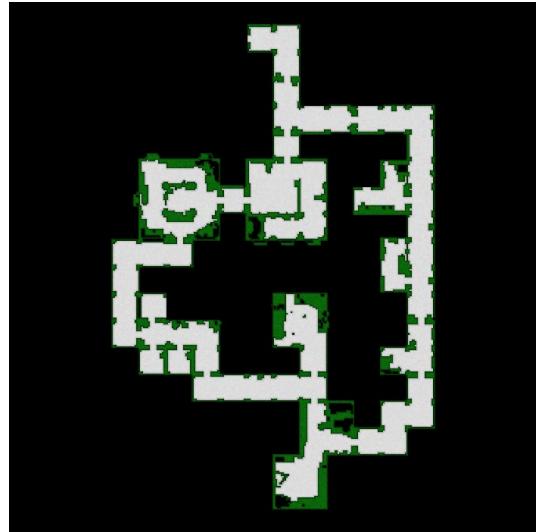


Figure A.11: Map "lak302d".

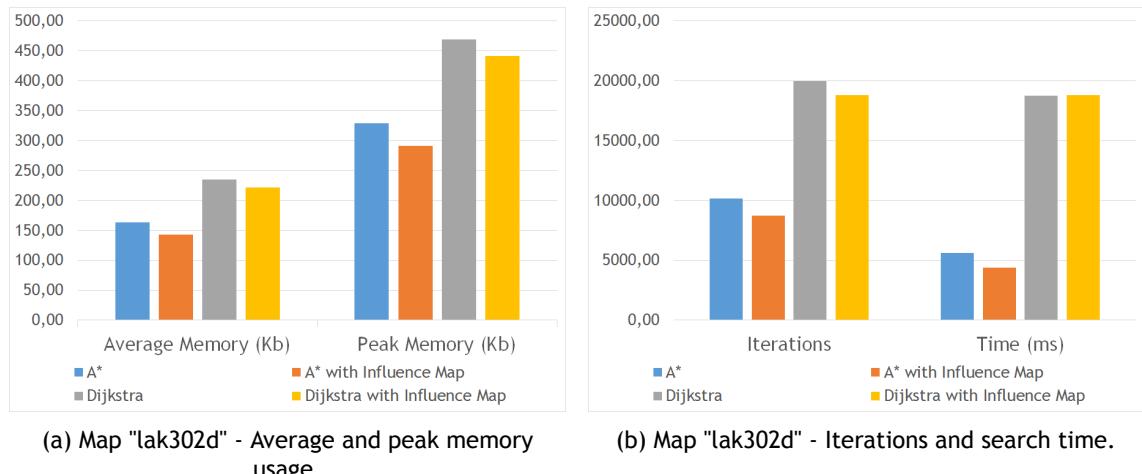


Figure A.12: Map "lak302d" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	163.13	328.44	10180.00	5624.00
A* with Influence Map	142.80	291.28	8726.00	4412.50
Dijkstra	234.99	468.77	19944.00	18746.50
Dijkstra with Influence Map	221.35	441.21	18763.00	18795.50

Table A.6: Map "lak302d" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

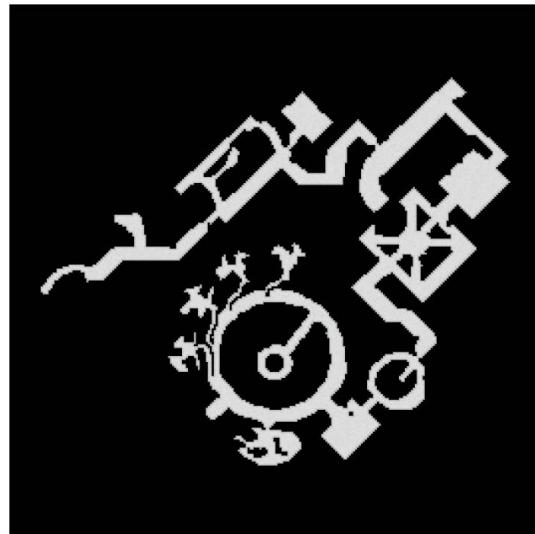
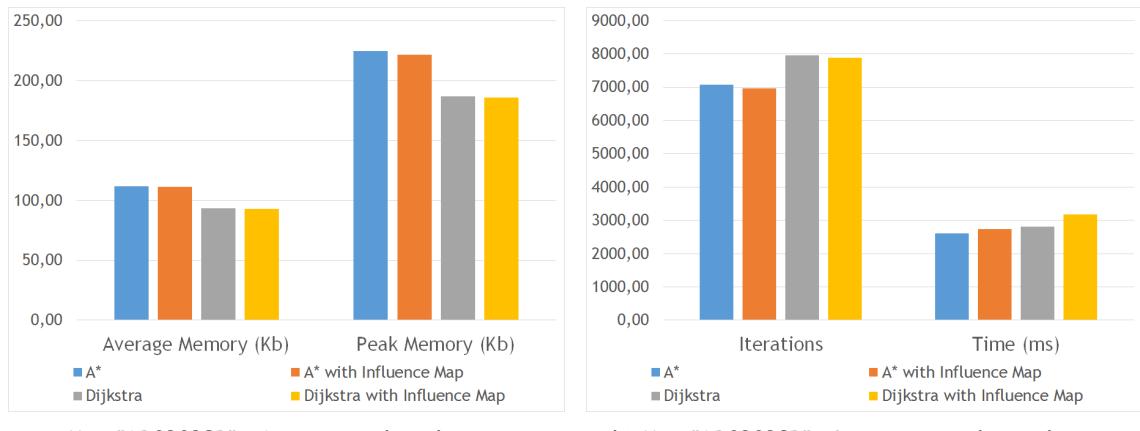


Figure A.13: Map "AR0202SR".



(a) Map "AR0202SR" - Average and peak memory usage.

(b) Map "AR0202SR" - Iterations and search time.

Figure A.14: Map "AR0202SR" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	112.02	224.72	7070.00	2609.00
A* with Influence Map	111.06	221.31	6962.00	2745.00
Dijkstra	93.57	186.94	7947.00	2816.50
Dijkstra with Influence Map	92.95	185.86	7891.00	3181.00

Table A.7: Map "AR0202SR" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

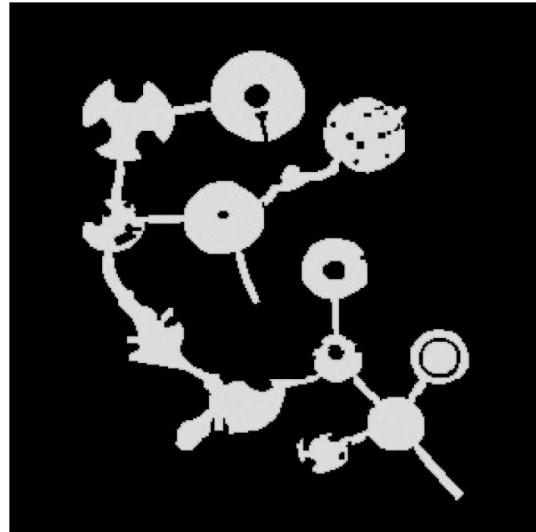
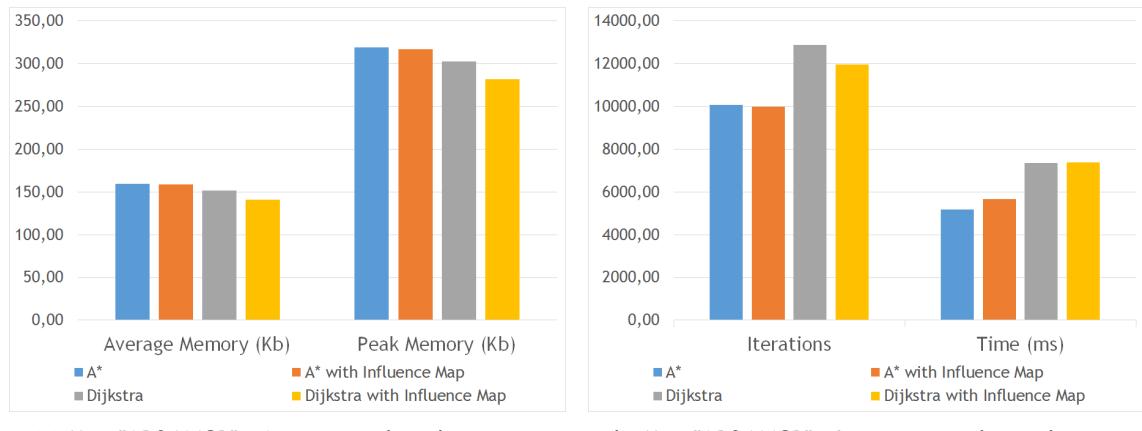


Figure A.15: Map "AR0411SR".



(a) Map "AR0411SR" - Average and peak memory usage.

(b) Map "AR0411SR" - Iterations and search time.

Figure A.16: Map "AR0411SR" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	159.47	318.63	10069.00	5167.50
A* with Influence Map	158.95	316.31	9987.00	5655.00
Dijkstra	151.57	302.32	12861.00	7346.00
Dijkstra with Influence Map	140.87	281.51	11934.00	7365.00

Table A.8: Map "AR0411SR" test results.



Figure A.17: Map "AR0011SR".

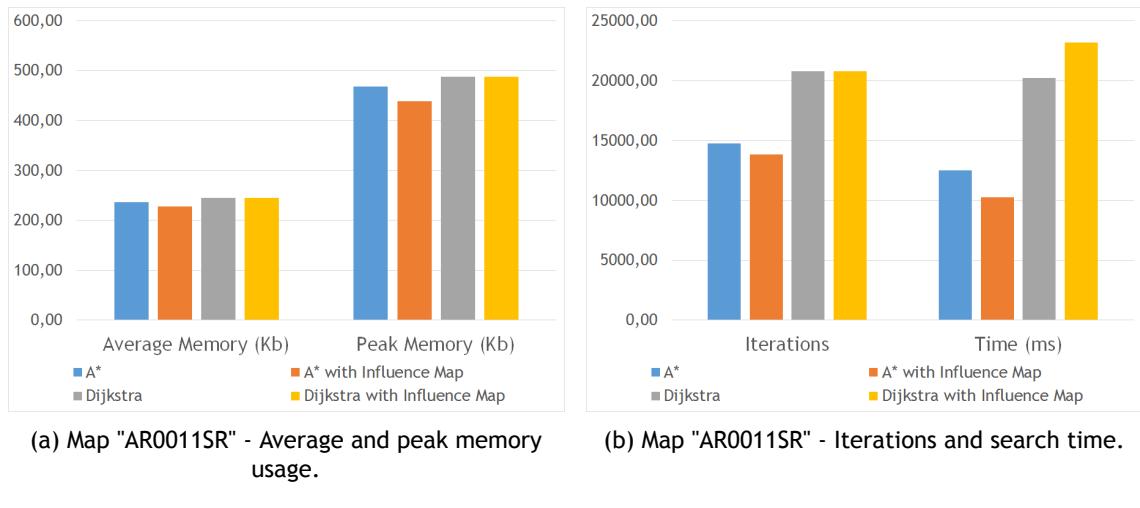


Figure A.18: Map "AR0011SR" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	236.75	468.03	14760.00	12497.50
A* with Influence Map	227.22	438.19	13828.00	10269.50
Dijkstra	245.23	487.34	20779.00	20223.00
Dijkstra with Influence Map	245.43	487.17	20772.00	23171.00

Table A.9: Map "AR0011SR" test results.

Influence Map-Based Pathfinding Algorithms in Video Games



Figure A.19: Map "AR0400SR".

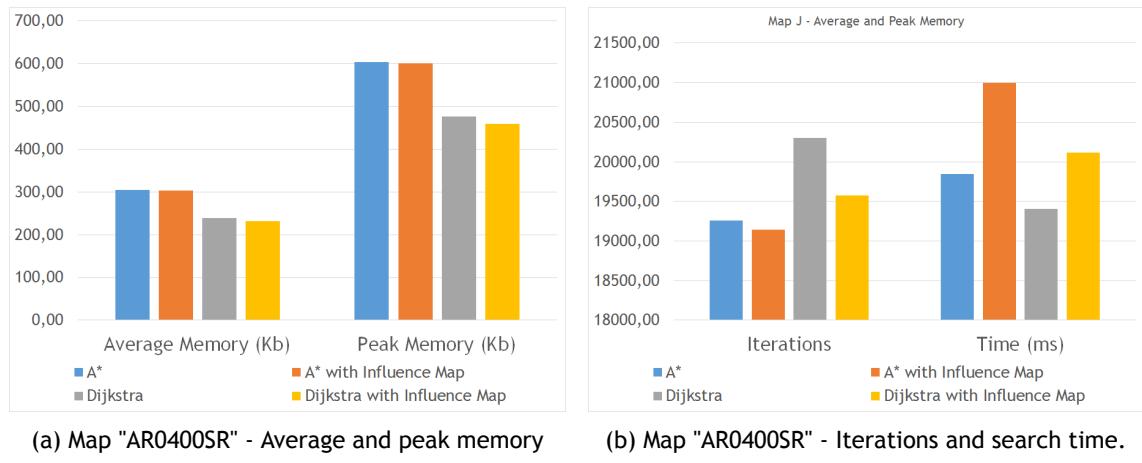


Figure A.20: Map "AR0400SR" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	305.01	603.16	19262.00	19843.00
A* with Influence Map	303.60	600.88	19147.00	20996.50
Dijkstra	239.04	476.20	20300.00	19403.50
Dijkstra with Influence Map	230.80	459.02	19579.00	20115.50

Table A.10: Map "AR0400SR" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

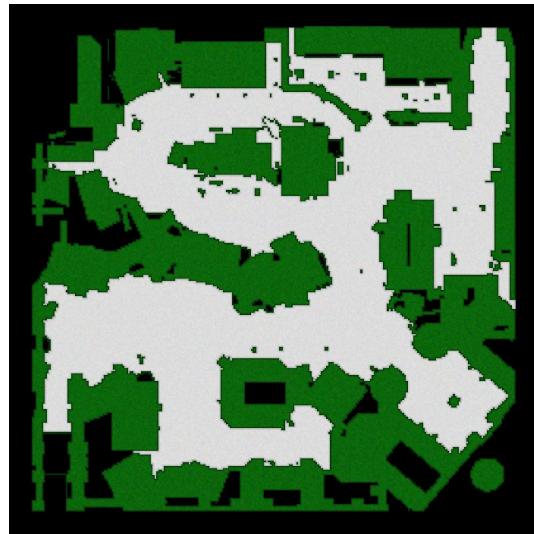
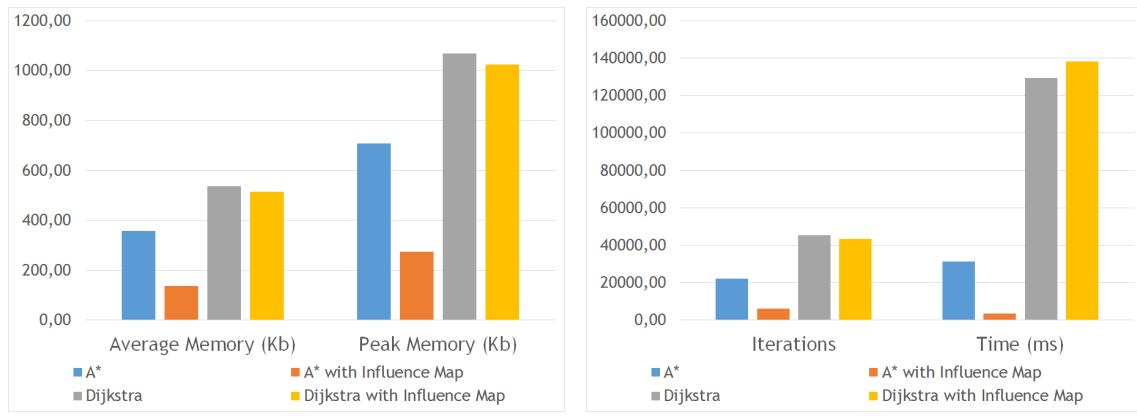


Figure A.21: Map "den520d".



(a) Map "den520d" - Average and peak memory usage.

(b) Map "den520d" - Iterations and search time.

Figure A.22: Map "den520d" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	357.45	709.13	22046.00	31299.50
A* with Influence Map	135.98	274.53	6321.00	3612.00
Dijkstra	536.00	1067.65	45415.00	129490.00
Dijkstra with Influence Map	514.52	1025.18	43446.00	138272.50

Table A.11: Map "den520d" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

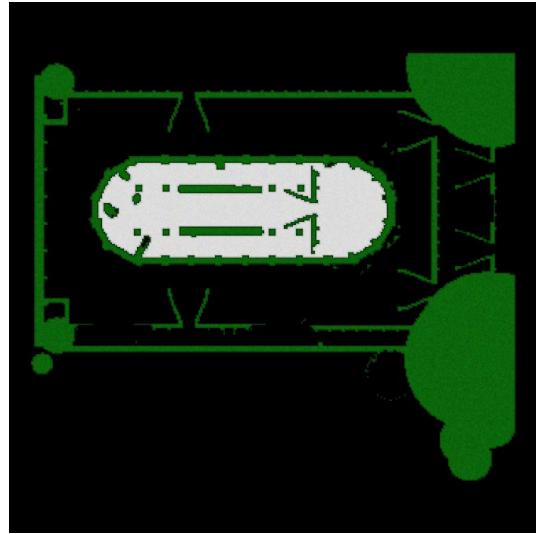


Figure A.23: Map "lak510d".

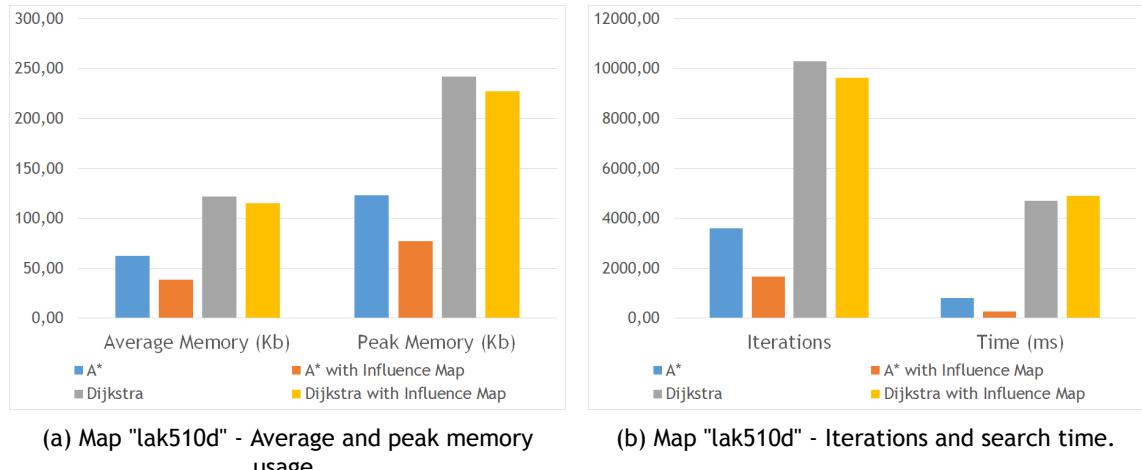


Figure A.24: Map "lak510d" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	62.34	123.22	3598.00	807.00
A* with Influence Map	38.35	77.03	1671.00	258.00
Dijkstra	121.91	242.04	10281.00	4700.50
Dijkstra with Influence Map	114.84	227.16	9629.00	4894.00

Table A.12: Map "lak510d" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

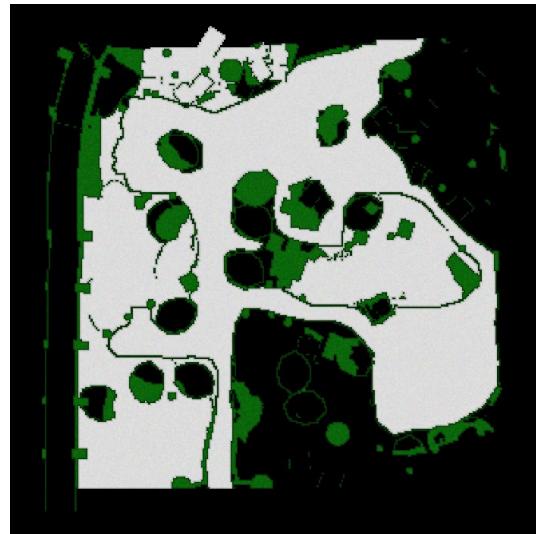
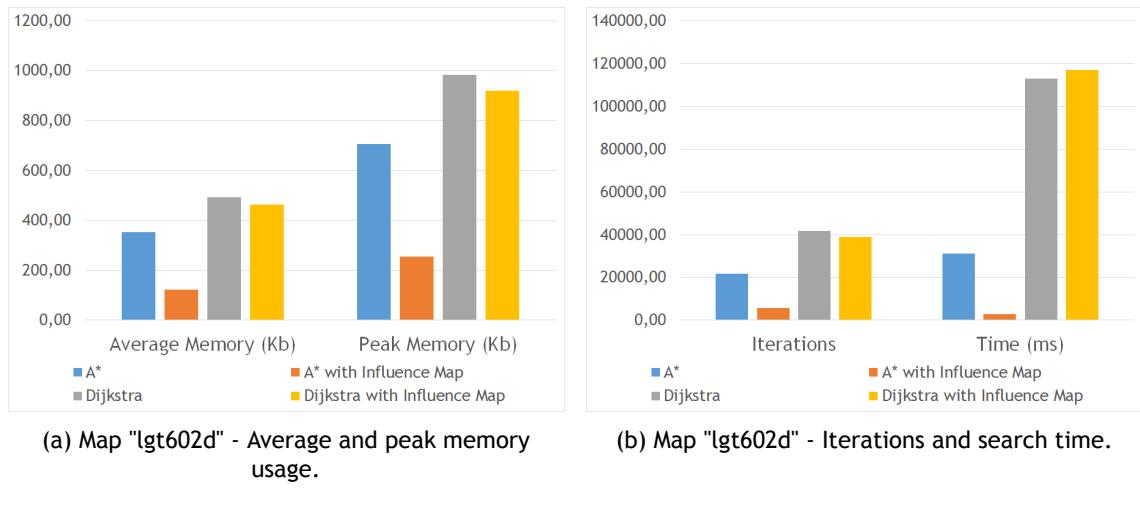


Figure A.25: Map "lgt602d".



(a) Map "lgt602d" - Average and peak memory usage.

(b) Map "lgt602d" - Iterations and search time.

Figure A.26: Map "lgt602d" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	353.46	704.94	21797.00	31038.00
A* with Influence Map	123.23	254.19	5818.00	2875.50
Dijkstra	493.46	981.73	41840.00	112964.00
Dijkstra with Influence Map	461.85	917.88	38998.00	116906.50

Table A.13: Map "lgt602d" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

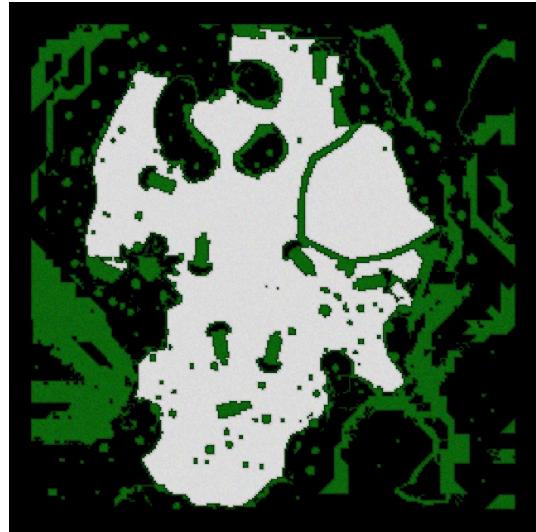


Figure A.27: Map "brc997d".

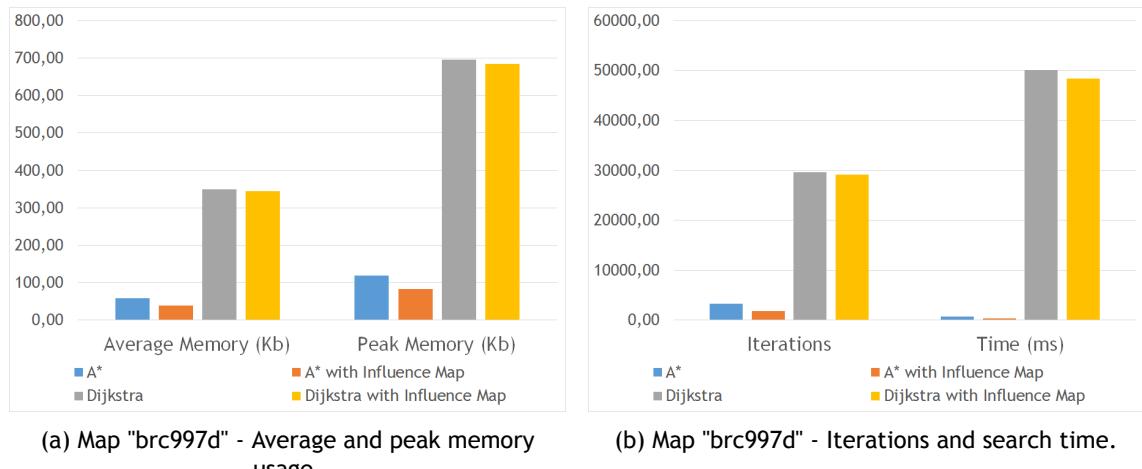


Figure A.28: Map "brc997d" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	59.18	118.69	3329.00	760.50
A* with Influence Map	39.56	83.25	1848.00	315.00
Dijkstra	350.20	696.28	29615.00	50137.50
Dijkstra with Influence Map	344.82	684.82	29095.00	48321.50

Table A.14: Map "brc997d" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

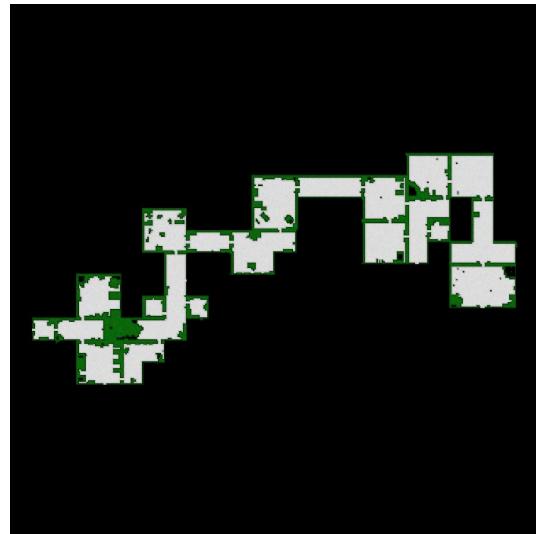
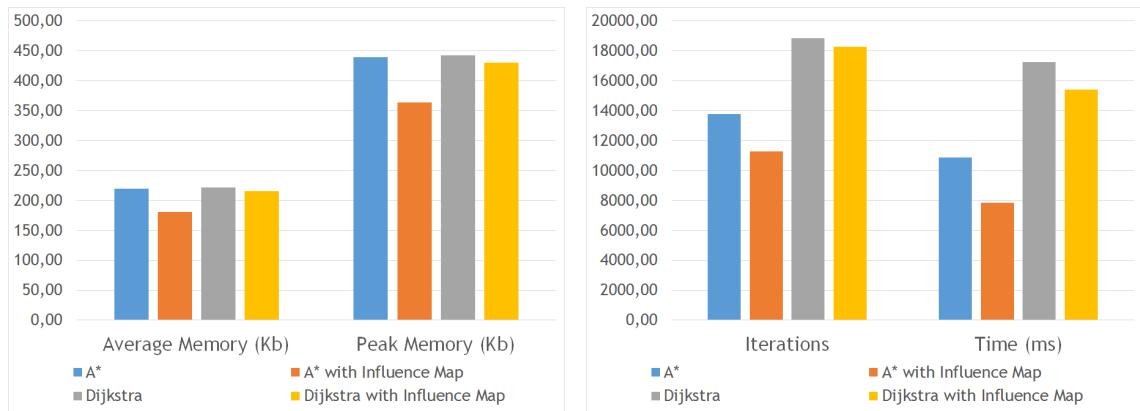


Figure A.29: Map "den600d".



(a) Map "den600d" - Average and peak memory usage.

(b) Map "den600d" - Iterations and search time.

Figure A.30: Map "den600d" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	219.42	438.63	13748.00	10840.00
A* with Influence Map	180.92	363.13	11284.00	7852.50
Dijkstra	221.90	442.24	18833.00	17210.00
Dijkstra with Influence Map	215.31	429.30	18243.00	15379.00

Table A.15: Map "den600d" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

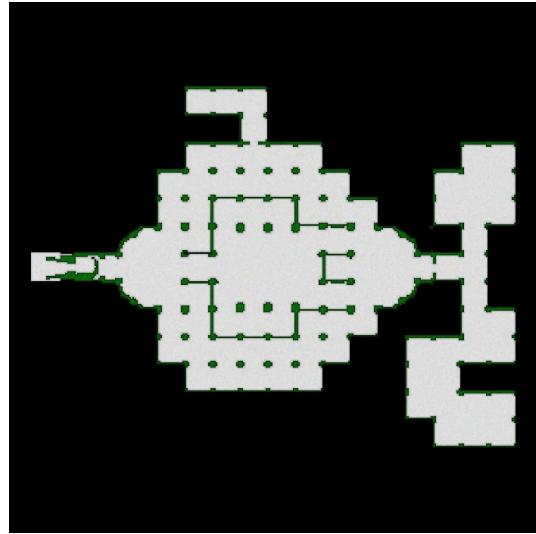
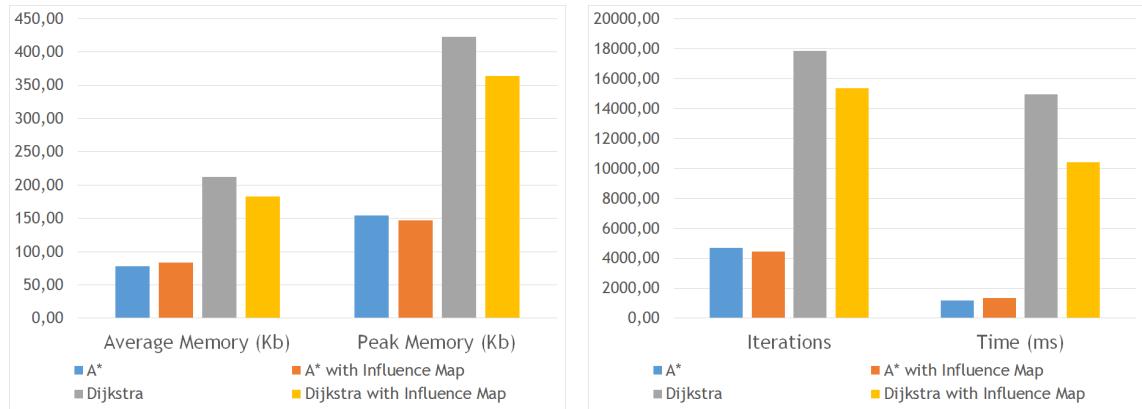


Figure A.31: Map "arena2".



(a) Map "arena2" - Average and peak memory usage.

(b) Map "arena2" - Iterations and search time.

Figure A.32: Map "arena2" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	78.20	154.13	4699.00	1185.50
A* with Influence Map	83.62	147.34	4449.00	1358.00
Dijkstra	212.07	422.55	17858.00	14945.50
Dijkstra with Influence Map	182.85	363.33	15358.00	10392.50

Table A.16: Map "arena2" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

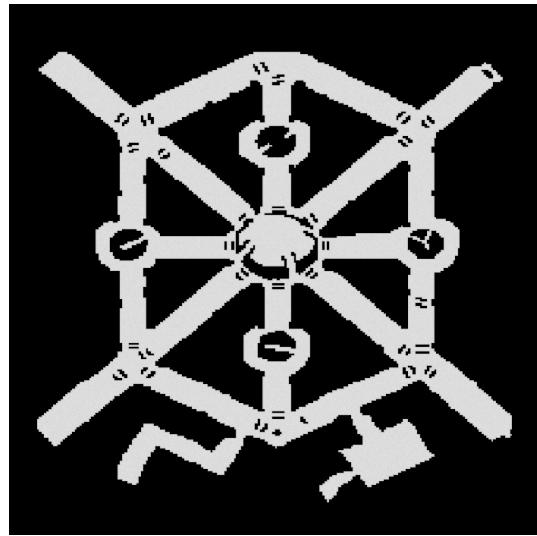
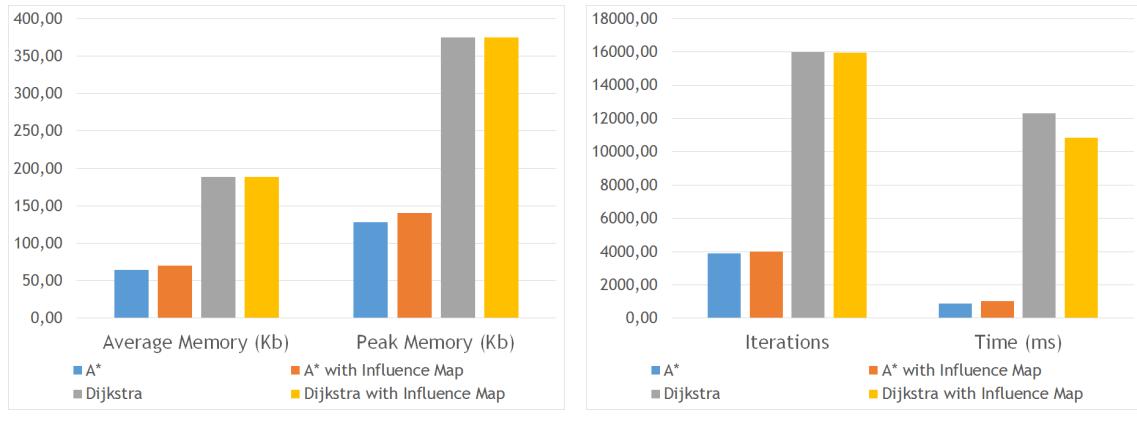


Figure A.33: Map "AR0701SR".



(a) Map "AR0701SR" - Average and peak memory usage.

(b) Map "AR0701SR" - Iterations and search time.

Figure A.34: Map "AR0701SR" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	64.91	128.34	3898.00	867.00
A* with Influence Map	69.99	140.75	3997.00	1033.50
Dijkstra	189.01	375.12	15997.00	12320.00
Dijkstra with Influence Map	188.80	374.41	15947.00	10831.00

Table A.17: Map "AR0701SR" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

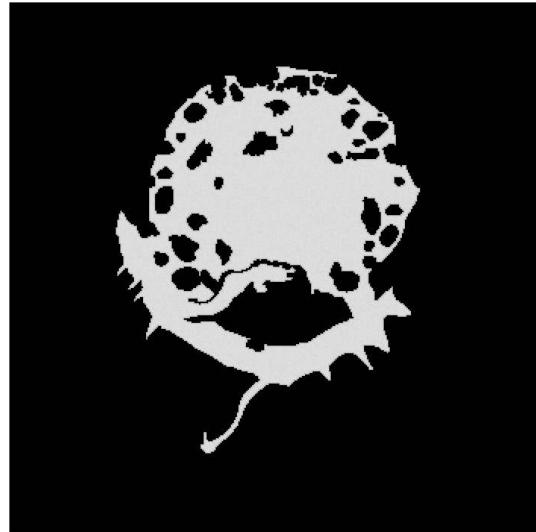
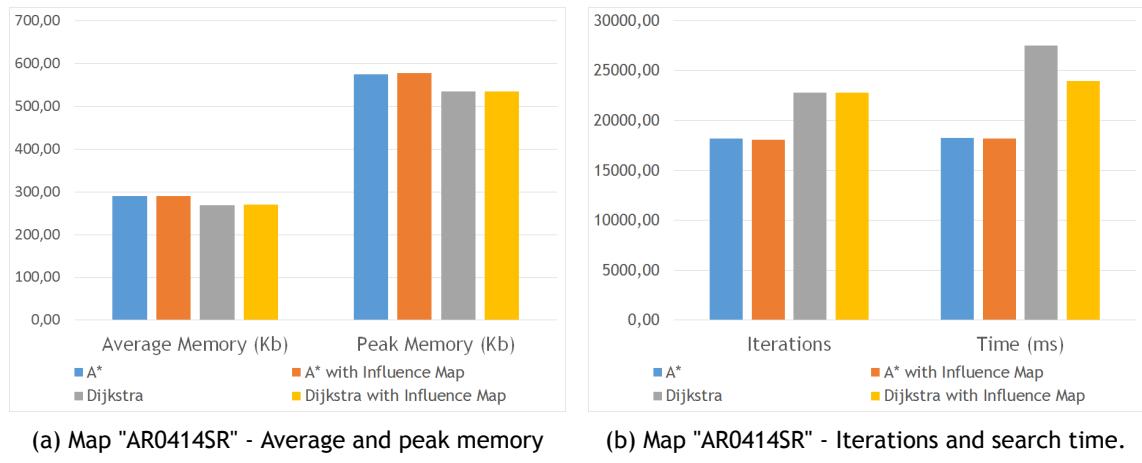


Figure A.35: Map "AR0414SR".



(a) Map "AR0414SR" - Average and peak memory usage.

(b) Map "AR0414SR" - Iterations and search time.

Figure A.36: Map "AR0414SR" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	290.40	574.81	18180.00	18267.00
A* with Influence Map	291.26	578.75	18056.00	18200.00
Dijkstra	269.43	534.61	22804.00	27504.50
Dijkstra with Influence Map	269.81	534.61	22804.00	23930.50

Table A.18: Map "AR0414SR" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

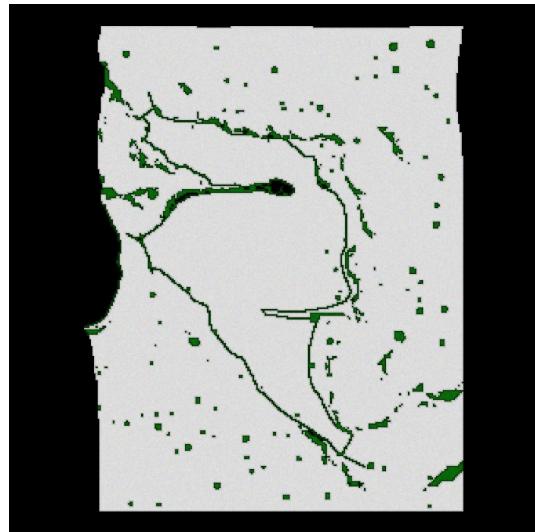
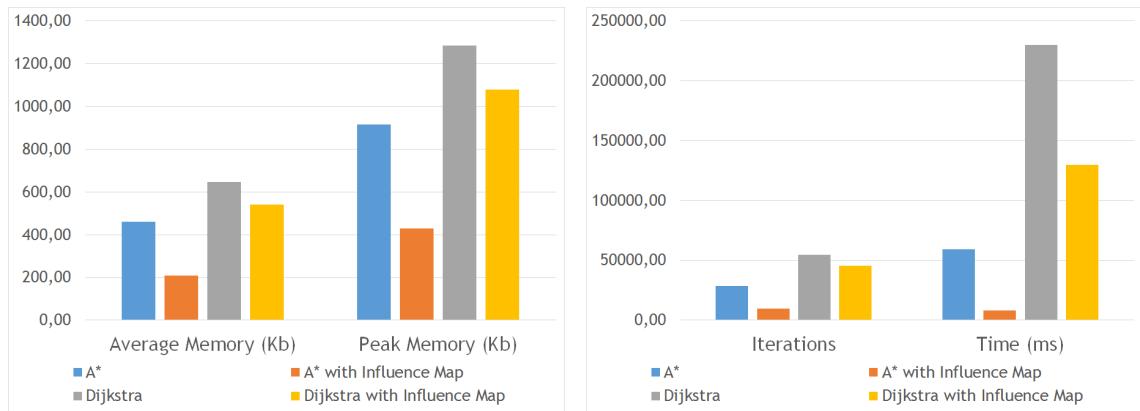


Figure A.37: Map "brc501d".



(a) Map "brc501d" - Average and peak memory usage.

(b) Map "brc501d" - Iterations and search time.

Figure A.38: Map "brc501d" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	460.80	915.94	28560.00	59066.50
A* with Influence Map	209.85	428.19	9613.00	8321.50
Dijkstra	644.83	1284.33	54685.00	229648.50
Dijkstra with Influence Map	540.50	1076.20	45597.00	129618.00

Table A.19: Map "brc501d" test results.

Influence Map-Based Pathfinding Algorithms in Video Games

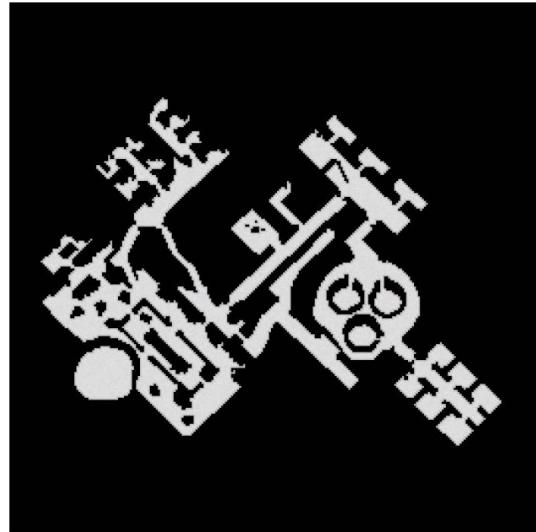


Figure A.39: Map "AR0406SR".

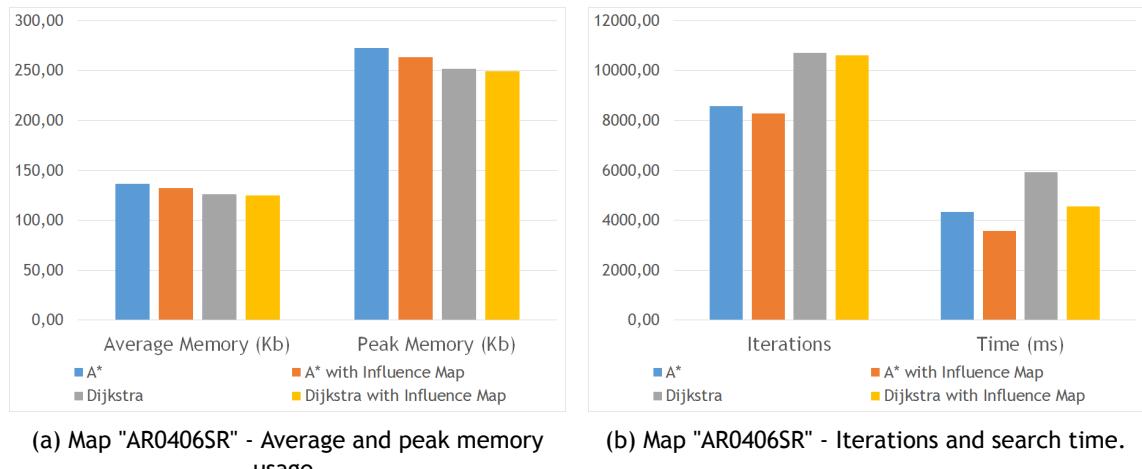


Figure A.40: Map "AR0406SR" test results.

Algorithm	Average Memory (Kb)	Peak Memory (Kb)	Iterations	Time (ms)
A*	136.30	272.50	8579.00	4344.50
A* with Influence Map	132.34	263.50	8268.00	3569.00
Dijkstra	126.15	251.53	10700.00	5933.50
Dijkstra with Influence Map	125.12	249.12	10603.00	4550.00

Table A.20: Map "AR0406SR" test results.

