# Pathfinding and Map Feature Learning in RTS Games with Partial Observability

**Hao Pan**
QOMPLX, Inc.
1775 Tysons Boulevard
McLean, Virginia, 22102
hao.pan@qomplx.com

**Abstract**

In this paper we deal with the pathfinding problem in Real Time Strategy (RTS) games with partial observability for Artificial Intelligence (AI) agents. We first propose a Bootstrap JPS algorithm to perform pathfinding efficiently alongside a routine to preprocess terrain features. We then establish a grid system to learn map features systematically considering both mobile and immobile units. Utilizing the learned map features, we employ a waypoint-based pathfinding technique to navigate pathfinding agents away from threats efficiently. Using maps from a few established RTS games, we demonstrate the performance of our pathfinding framework and compare it with a few alternative approaches.

## 1 Introduction

Pathfinding is the process of searching a graph and finding a path between the start and the destination nodes. There can be map features such as obstacles along the way, making the process potentially challenging to solve. Pathfinding is not a new topic, and various algorithms have already been developed. There are mainly two schools of such algorithms. Breadth-first search and depth-first search approach this problem by examining all possible paths exhaustively. The other school of algorithms focus more on finding the optimal path. Dijkstra's algorithm (Dijkstra 1959) maintains an "open set" and examines the node in the set with the shortest distance from the start. Once the destination is marked as visited, the shortest path is determined by tracing back all the visited nodes. (Wagner, Willhalm, and Zaroliagis 2005) reduced the search space of Dijkstra's algorithm by extracting geometric information from a given layout of the graph and by encapsulating precomputed shortest-path information in the so-called geometric containers. A* (Doran and Michie 1966) is an extension of Dijkstra's algorithm. The main difference is that A* assigns a weight to each node in the open set equal to the cost from the starting node to that node plus the approximate cost between that node and the destination. The approximate distance is found by heuristic. (Björnsson and Halldórsson 2006) used the dead-end heuristic to eliminate certain map areas that are provably irrelevant, and the so-called gateways to improve estimates for the current query. The authors further demonstrated that both heuristics reduced the exploration and time complexity of A* search significantly over a standard distance metric. Jump Point Search (JPS) (Harabor and Grastien 2011) is considered an improvement over the A* algorithm as it facilitates the search procedure by means of graph pruning, eliminating certain nodes in the grid based on assumptions that can be made about the current node's neighbors. This means the algorithm can consider long jumps from the current node, while traditional A* only considers adjacent nodes. In the following years, the authors further optimized JPS by considering a preprocessing grid and stronger jump rules (Harabor and Grastien 2012, 2014).

It can be demanding when applying pathfinding in games, as one has to consider extra factors such as map properties and algorithm efficiency. (Björnsson and Halldórsson 2006; Sturtevant and Buro 2005) used maps from different games and of different sizes. They evaluated the performance of pathfinding methods based on multiple criteria such as the number of nodes expanded, path-length, and runtime. (Zarembo and Kodors 2015) studied pathfinding algorithms including A*, BFS, Dijkstra's algorithm, HPA* and LPA* (Koenig, Likhachev, and Furcy 2004), and compared them based on execution time and the number of traversed nodes. (Sturtevant et al. 2019) improved pathfinding efficiency by planning in an abstract space and then refining abstract paths to traversable paths. The authors took into account terrain costs and dynamic terrain, and achieved a good balance between memory usage, path quality, and pathfinding speed. (Hatem, Stern, and Ruml 2013) studied the suboptimal search algorithms and proposed a linear space analogue of Explicit Estimation Search (EES) to alleviate the problem of the memory being overrun on large problems.

Additional considerations were needed when applying pathfinding in Real Time Strategy (RTS) games such as StarCraft. On a general level, the prevalent approach is to couple terrain learning/analysis methods and pathfinding. (Hagelbäck 2015) coupled pathfinding with two techniques: 1) Potential Field (PF); and 2) Flocking. They discovered that PF techniques require much more computational resources than the counterpart which is relatively simpler. (Amador and Gomes 2019) combined Influence Maps and pathfinding in a way that the influence costs replaced the traditional distance-based costs. Utilizing repulsors and at-

tractors in the game world, the authors constructed bounded-search pathfinders that avoid obstacles (repulsors) and follow check points (attractors). (Ninomiya et al. 2014) introduced hybrid environment representations that balance computational efficiency and search space density to provide a minimal, yet sufficient, discretization of the search graph for constraint-aware navigation.

There are still challenges present for pathfinding in RTS games with partial observability. Even for a well established algorithm such as the A*, it can be improved much further. An AI agent with poorly performing pathfinding routines slows down the game and hurts a human's experience of playing against it. Additionally, in tournaments there are limits on how much time an agent can spend within each logical step (also known as frame) for doing everything including pathfinding. For example, a bot will be given a game loss if it spends more than one second in at least ten game frames in the AIIDE StarCraft AI competition[1]. Last but not least, the Fog of War (FoW) obscures the position information of opponent units, making pathfinding difficult. FoW is the common way partial observability presents itself in RTS games. (Hagelback and Johansson 2008) illustrated that a bot using potential fields can deal with imperfect information equally well as one with the perfect information, while being computationally more efficient, using the game Open Real Time Strategy (ORTS). However, ORTS is a much simpler game than StarCraft and these authors did not consider unit movement under the FoW. Furthermore, potential field based methods are prone to get stuck in local minima.

In this paper, we propose a pathfinding framework to tackle the aforementioned challenges by means of: 1) performing preprocessing to eliminate nodes deemed unnecessary to explore; 2) devising the Bootstrap JPS to further improve the efficiency of the exploration; 3) establishing a grid system which is capable of both efficiently storing the learned position of immobile units and predicting the potential position a mobile unit may move to under the FoW; and 4) employing a waypoint-based pathfinding technique to navigate the pathfinding agents out of danger efficiently.

# 2 Methodology

## 2.1 Problem Definition and Assumptions

First we describe some details of the pathfinding problem we are dealing with here. The input of the pathfinding problem is a map where some parts are covered by the FoW which can be lifted once the influenced area is explored. The presence of the FoW conceals certain information such as the position of an enemy unit and the shape of the terrain. Even with perfect information, the built-in pathfinding algorithm can sometimes fail to find a valid path, causing units to get stuck indefinitely when being issued a move command. Furthermore, certain information such as the start and destination nodes can be potentially changing. We utilize the 8-connected grid map representation. We further assume there is no traveling cost (e.g., it is equally easy to traverse grasslands and deserts, and there is no extra difficulty when moving downhill or uphill). We deal with single-agent pathfinding problems and the pathfinding agent does not have size. We also assume that the unit collision does not exist. The solution to the problem is expected to yield an optimal path safe for pathfinding agents to travel on from enemy threats which can damage or even kill the agents.

[1]AIIDE StarCraft AI tournament has rules on frame times: https://www.cs.mun.ca/ dchurchill/starcraftaicomp/rules.shtml

## 2.2 Terrain Preprocessing

We believe terrain preprocessing is a necessary step to perform before the actual pathfinding process begins. More specifically, this is about performing obstacle filling which is achieved by completing the following sub-tasks:

$i$ Identify U-/L- shaped islands which are adjacent obstacle nodes connected to each other either horizontally or vertically. We do not consider diagonally connected nodes as doing so could potentially identify two different islands as a single one.

$ii$ Mark the nodes bounded by those islands identified previously as obstacles.

The motivation here is that the U-/L-shaped islands could lead to the exploration of nodes which are not on the optimal path and cause an algorithm to get stuck. Subtask $i$ can be accomplished by sweeping through every node on the map and conducting a depth-first search (DFS). In each DFS traversal we recursively call for the 4 neighbours. We keep track of the visited nodes so that they are not visited again. Subtask $ii$ is fulfilled by examining every node on the map to see if they are bounded by the obstacles in at least three out of the four directions (up, down, left, right). We only care about the L-shaped islands when at least one of the ends touches the boundary of the map. If that is the case, the map boundary effectively acts as an obstacle, and combined with the L-shaped island, they together form a U-shaped island. To ease the understanding, one can consider the map boundary as an artificially added obstacle which is to be welded to the L-shaped island (see fig. 1 for a visual demonstration). Note that for cases where the destination node or nodes from another island are bounded by the U-/L-shaped islands, obstacle filling is not to be performed, otherwise one runs the risk of blocking the agent from finding the destination by adding obstacles unnecessarily. All the procedures discussed here are summarized in algorithm 1.

In algorithm 1, $grid$ is a matrix storing the status of each node. A value of 0 means the current node is not an obstacle, 1 means it is an obstacle, and 2 means the current node is already visited for island counting purposes. $count$ is the number of the islands identified. The size of a particular island, $Island_k$, is the number of nodes constituting this island. There are three blocks of code in algorithm 1. The first defines the $MarkIsland()$ function which is to be recursively called by the second block of code. The execution of the first two blocks of code identifies all the islands if they exist. The last block of code examines if a node is surrounded by a U-/L-shaped island and artificially marks the node as an obstacle if so.

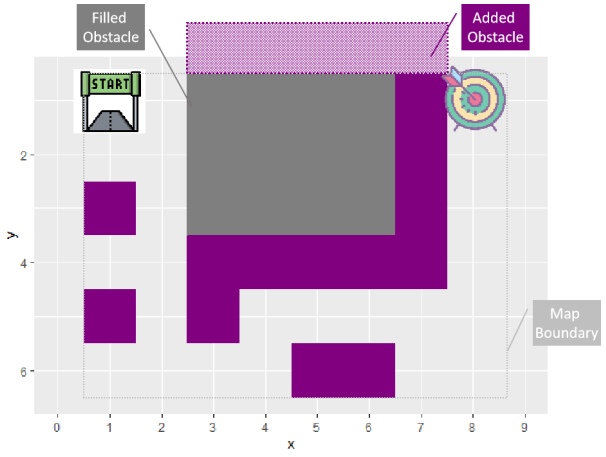


Figure 1: Illustration of filling an L-shaped island

**Algorithm 1** Obstacle Filling

```
MarkIsland(grid, p, q, ROWS, COLS)
if Node(p, q) is within the map boundaries then
    Mark current cell as visited: grid[p][q] = 2
    MarkIsland(grid, p, q-1, ROWS, COLS)
    MarkIsland(grid, p, q+1, ROWS, COLS)
    MarkIsland(grid, p-1, q, ROWS, COLS)
    MarkIsland(grid, p+1, q, ROWS, COLS)
end if

for i' in 1:ROWS do
    for j' in 1:COLS do
        if grid[i'][j'] == 1 then
            count++
            MarkIsland(grid, i', j', ROWS, COLS)
        end if
    end for
end for

for i in 1:ROWS do
    for j in 1:COLS do
        if Node(i, j) is surrounded by an island or a map
        boundary in any three of the four directions (left,
        right, top, bottom) then
            Mark Node(i, j) as an obstacle
        end if
    end for
end for
```

## 2.3 Bootstrap Jump Point Search

Jump Point Search (JPS) speeds up pathfinding on uniform-cost grid maps by "jumping over" many locations that would otherwise need to be explicitly considered. Here we aim at expediting the exploration further by using JPS itself (hence the term bootstrap) to compute the cost function between a node and the destination instead of calculating the heuristics using traditional means such as the Euclidean or Manhattan distance. Depending on the complexity of the terrain, Euclidean or Manhattan distance may sometimes provide an overly optimistic estimation because neither metric considers obstacles along the way. The Bootstrap JPS is to be used after the completion of the terrain preprocessing as discussed previously. The time complexity associated with the bootstrap JPS is $\mathcal{O}(n^2)$ considering the worst case scenario where every possible (source, destination) pairing must be evaluated. However, this process is easily parallelizable.

**Memoization routine for Bootstrap JPS** We store the path found and its length for each node, hoping to further speed up the search process. At each iteration of Bootstrap JPS, a few sub-pathfinding problems are solved and potentially the solution to one of them is a part of the optimal path. The solution to each of these sub problems consists of a series of nodes leading to the destination from the current node. Storing such nodes can potentially allow us to terminate the Bootstrap JPS at a much earlier point and further improve the efficiency.

To determine when to terminate the Bootstrap JPS, we rely on two simple conditions:

$i$ Terminate the Bootstrap JPS when an optimal path from the current node to the destination is found.

$ii$ Terminate the Bootstrap JPS when an optimal path from the next node (a jump point) to the destination is found.

**Optimality of Bootstrap JPS** Often in a pathfinding process, a heuristic is used to estimate the cost of reaching the goal state. A heuristic function is said to be admissible if it never overestimates the cost of reaching the goal. That is, the cost it estimates to reach the goal is not higher than the lowest possible cost from the current point in the path.

An algorithm will terminate on the shortest path if an admissible heuristic is used and the algorithm progresses per iteration only along the path that has the lowest total expected cost of several candidate paths and terminates the moment any path reaches the goal. In short, using an admissible heuristic guarantees the optimality of the path found if such a path exists.

Our proposed approach computes the heuristic by calculating the lengths of the paths found by JPS and memorizes them for all following intermediate calculations. It is already proven that the JPS yields optimal paths (Harabor and Grastien 2011, 2012). This means our heuristic can never overestimate the cost and consequently the heuristic is indeed admissible. At this point we can say our proposed approach also guarantees the optimality of the paths.

## 2.4 Unit movement estimation under the FoW

When one applies pathfinding methods in an RTS game environment where the Fog of War is present, the optimal path may sometimes backfire, as the obstacles/threats may have changed their position without being detected, and as a result hinder/damage the pathfinding agent(s). The corrrect estimation of unit movement under the FoW becomes vital to solving the pathfinding problem in this setting.

The core of the solution to address the estimation of unit movement with partial observability is the velocity of the

unit. Such quantity is available via BWAPI[2] which enables an AI bot to interact with the game StarCraft. BWAPI provides the kind of information a human player would have access to. The velocity information is not always available, as it would become inaccessible if the unit in question is under the FoW.

In order to estimate the unit movement, the question to consider is, how far into the future would we like to perform the forecast? Our approach is to set the time window to be the duration $t_{dur}$ it takes for a unit to cover its sight range $d_{sight}$ while traveling at its speed $v$:

$$t_{dur} = \frac{d_{sight}}{v} \tag{1}$$

In StarCraft, a unit has both a sight range and an attack range, with the sight range being oftentimes greater than the attack range. If an enemy unit were to attack ours, it needs to travel to a point where the target is within the attack range. By having it to cover the entire sight range, we ensure that the estimation serves as the upper ceiling.

What happens when we do not have access to a unit's velocity (i.e., the unit is covered by the FoW)? We turn to heuristics by first acquiring the last known position and the velocity of the unit. Using such information, the velocity is then modeled as one that decays over time. This means that as time goes by, we grow more and more uncertain about the velocity, and as a result, the magnitude of the velocity will gradually reach zero, indicating that any obscured area in the contiguous FoW is possible for the unit to travel to. The interpretation in this context is that, given enough time, if a unit is still under the FoW, it becomes reasonable to assume that it is not actively trying to come out of FoW/chasing our unit(s). As a result, it would be safe to assume that the magnitude of its velocity is zero as the unit can be traveling in any direction. This approach is similar to the particle model (Weber, Mateas, and Jhala 2011). However, our decay model has the advantage of being flexible to account for the threat level perceived, and this is controlled by the power parameter $p$ in the decay model:

$$v = v_0 - \frac{v_0}{t_{dur}^p} \cdot t^p \tag{2}$$

Here $v_0$ is the last known velocity of the unit before it entered the FoW and became inaccessible. The model is further illustrated in fig. 2.

## 2.5 Map Feature Learning

Having the correct unit movement estimation alone is not enough to solve the pathfinding problem in an RTS game environment. There are not only the usual terrain obstacles we see, but also enemy units and buildings which can spawn or move throughout the game, and affect the traversability of nodes. We deemed the following features necessary to learn to facilitate the pathfinding process.

[2]The GitHub page of BWAPI: https://github.com/bwapi/bwapi

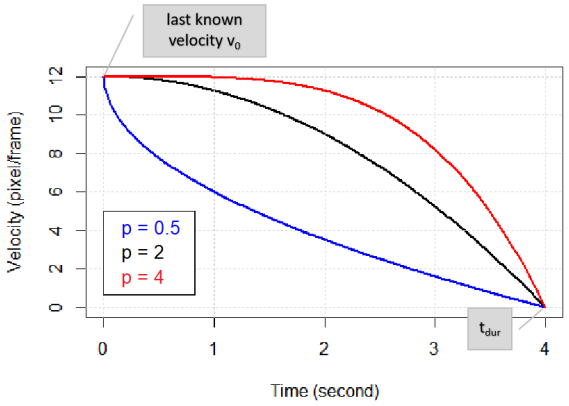


Figure 2: Illustration of the decay model with varying decay rate

**Terrain Features (Altitudes)** Here we identify terrain features such as a plateau which cannot be moved/destroyed at all times. The identification of such features needs to be done only once. Depending on the destination, ground units will have to find either the way around the plateau, or the ramp leading up to the plateau, thus posing a challenge for pathfinding.

**Enemy Units' Position** It is easy to mark a unit's position if we can see the unit, but otherwise this matter becomes difficult, as a unit can move, and under the FoW, we would not be able to know the direction of the movement. This is exactly the place where section 2.4 can become useful.

Next, we categorize all mobile units as follows:

$i$ Workers: these units are the primary targets for us to harass as much as possible, so that we can disrupt the opponent's economy. After all, "having more units" remains a fundamental winning condition in many RTS games.

$ii$ Units that can attack ground targets

$iii$ Units that can attack air targets

Separating items $ii$ and $iii$ is essential because pathfinding for ground units and pathfinding for air units are very different: they depend on the opponent's ground and air threats.

Having the position of the units is not enough. We also care about the attack range a unit has. This means the attack range of both the opponent's units and our own. This kind of information together with the position information would help a pathfinding routine to discover the position that would allow us to fire on the opponent's units at our maximum attack range while staying out of theirs.

**Enemy Buildings' Position** We learn only the opponent's defensive buildings here. An immobile defensive structure has its own attack range. Together with its position, all the areas under the influence of such buildings are marked as red grids (as in fig. 3) indicating positions to avoid. Unlike a unit, a building cannot move, so what we once discovered remains there unless the building gets destroyed. There are

opponent's workers alongside the defensive buildings, complicating the learning process. However, we were still able to find the few green grids which are safe places for the attackers to harass the workers.

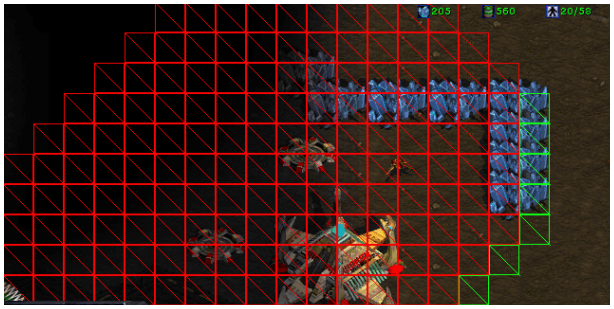

Figure 3: Marking the optimal positions to harass the enemy's worker line while avoiding defensive buildings

### 2.6 Waypoint-based Pathfinding Technique

Waypoint-based pathfinding techniques (Zhu et al. 2015) offer an easy way to specify the space and handle obstacles. The motivation of us employing it here is to establish waypoints around any identified threat, and in turn facilitate the pathfinding process. The simplest way to do so is to set up a few corner points around the threat. fig. 4 illustrates a simple example. The pathfinding agent first identifies a threat which was previously unknown (potentially due to the Fog of War). At the time of the discovery, the agent is already within the radius of the effect exerted by the threat. The agent must choose a set of waypoints in order to return to safety. First path #1 is chosen because waypoint D is the closest to the agent. It is desirable to evade the threat as soon as possible. Then path #2 is chosen because waypoint C is closer to the destination than waypoint B. Waypoint A is out of question as an agent is only allowed to travel between adjacent waypoints. Finally path #3 is chosen since it leads to the destination directly and is deemed safe to travel upon.

Compared to conventional pathfinding techniques such as A*, the waypoint-based one has the advantage of exploring much fewer nodes at the cost of the optimality of the path. This is an acceptable tradeoff in an RTS game environment where speedy reactions are important.

There are a few small caveats here: 1) the agent is forbidden from traveling through all the waypoints as that would mean the agent is stuck travelling around the threat indefinitely; 2) it is forbidden to set waypoints (illustrated as the grey circles on fig. 4) directly on the edge of the threat radius. This would endanger the agent as it travels between such waypoints; and 3) the number of waypoints established around the threat can have an impact on the agent's evasion from the threat as well as the computation efficiency. In practice we found that eight waypoints around each threat usually suffice to ensure that there are enough safe nodes/options for the agent to choose from while not inducing too much additional computation burden.

We couple the waypoint-based pathfinding technique with the proposed Bootstrap JPS based on the state of the pathfinding agent. In the normal state where the agent has no threat nearby, it proceeds to the destination using the Bootstrap JPS. We switch immediately to the waypoint-based pathfinding technique once there are threats close to the agent, in order to minimize the potential damage dealt to the agent. The waypoint-based technique also pairs with the unit movement estimation smoothly as the position of the waypoints can be readily updated based on the estimated position of the threats.

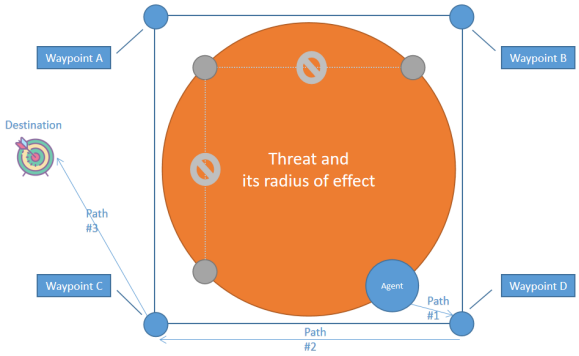


Figure 4: Demonstration of the waypoint-based pathfinding technique

## 3 Case Study

The purpose of the case study is to compare the proposed pathfinding framework with a few alternative approaches, namely, A* and JPS. We applied all three pathfinding techniques on two maps which are illustrated by figs. 5 and 6, respectively. The maps were taken from a benchmark map set[3] created by (Sturtevant 2012). The two maps are from the RTS Game WarCraft III (published in July, 2002) and StarCraft (published in March, 1998), respectively. On each map we randomly chose 1000 pairs of starting and destination nodes. Throughout the case study we assumed partial observability. For each pair of starting and destination nodes, we insert either a static or mobile enemy unit at a random location along the optimal path which was pre-computed by A*. Both the static and mobile enemy units were initially unknown to the computer-controlled pathfinding agent due to the partial observability. We programmed a set of simple rules to govern the behavior of the mobile enemy unit. This enemy unit always tries to move to or stay at its starting position if the pathfinding agent is outside of its sight range. Once the pathfinding agent is within the sight range but outside of its attack range, the enemy unit will begin to chase the agent. The enemy unit will attack the agent if the agent is within its attack range. When using A* and JPS, the enemy unit is handled by marking all the nodes within its sight range as obstacles. Whenever a certain amount of nodes are

[3]The map set is available from here: http://movingai.com

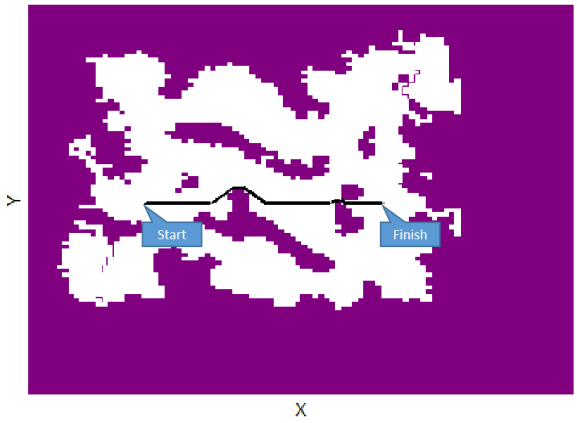


Figure 5: Map #1 (hillsofglory.map) and a sample optimal path

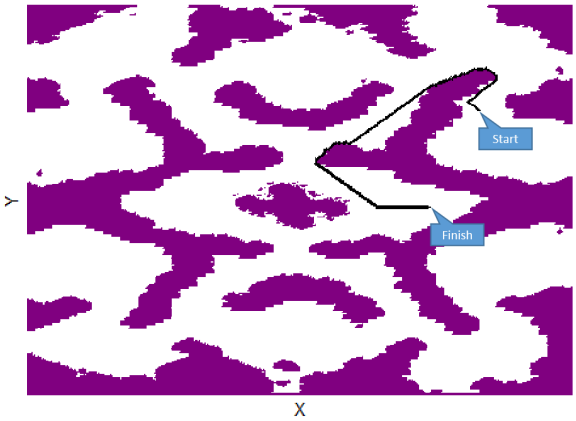


Figure 6: Map #2 (Sanctuary.map) and a sample optimal path

newly marked as obstacles, every pathfinding technique is going to be rerun as the map they operate on has effectively changed. All experiments were run on a 2.2 GHz CPU computer.

Due to the complexity of the environment for pathfinding, we use a few criteria to evaluate the performance of pathfinding routines quantitatively: 1) number of nodes explored/evaluated (NE); 2) runtime (RT) measured in seconds; and 3) remaining hit points (RHP) of the pathfinding agent after reaching the destination. The RHP is expressed as a percentage of the agent's total hit points. The results for all the scenarios are summarized in table 1. We can see that A* explored many more nodes than the other two techniques and the RT is much longer as a result. Both the JPS and our proposed pathfinding framework greatly reduced the NE values. The proposed framework explored a few more nodes on average than JPS mainly due to the additional waypoints around the threats it established. As these nodes are easily obtained, the resulting RT is slightly lower than that of JPS. Both the A* and JPS scored poorly in terms of RHP as they are not dedicated to this particular pathfinding problem where partial observability is present. The matter becomes worse when the enemy threat is mobile. While by design, the proposed framework handles this problem more sophisticately and significantly reduced the amount of damage received from the enemy threat. Overall, the proposed pathfinding framework performed the best here.

Table 1: Peformance of various pathfinding routines

| CRITERION | A* | JPS | PROPOSED |
|---|---|---|---|
| MAP #1 (391 × 388, AVG PATH LENGTH = 270) | | | |
| NE | 17803 | 213 | 216 |
| RT(S) | 3.04 | 1.72 | 1.68 |
| RHP | 52% | 51% | 82% |
| MAP #2 (512 × 512, AVG PATH LENGTH = 446) | | | |
| NE | 42346 | 220 | 224 |
| RT(S) | 8.89 | 1.97 | 1.94 |
| RHP | 68% | 69% | 92% |

## 4 Conclusions and Future Work

In this paper we proposed a pathfinding framework to address the challenges encountered when performing pathfinding in RTS games with partial observability by means of: 1) preprocessing terrain/map features to prevent pathfinding routines to get stuck locally; 2) creating the Bootstrap JPS to compute the cost function more accurately and in turn further eliminate nodes deemed unnecessary to explore; 3) constructing the velocity decay model to predict unit movement under the FoW; and 4) employing a waypoint-based pathfinding technique to help pathfinding agents to maneuver around threats efficiently. We demonstrated that the proposed framework was superior than a few mainstream pathfinding routines such as A* and JPS.

In the future we would like to augment this work by: 1) solidifying an effective way to learn the decay rate used in the decay model either in-game on the fly or from past data; 2) considering other possibilities of velocity development under the FoW. Currently we assume the worst case where an enemy unit would immediately begin to chase our units at all costs. This can be highly situational and depends on several things such as the opponent's stance (defensive or offensive); 3) optimizing the resolution of the grid system. We have noticed that occasionally units can still catch enemy's fire while maneuvering around using Bootstrap JPS and the grid system. This was mainly caused by a grid falsefully being marked as safe due to the relatively coarse resolution and not accounting for unit sizes; 4) parallelizing the preprocessing procedure to further speed up pathfinding; and 5) comparing results with those from D* (Stentz 1994) and/or D* Lite (Koenig and Likhachev 2005), especially for scenarios where the obstacles are changing their position.

## 5 Acknowledgment

The authors would like to specifically thank Nathan Roth (MSc) for his dedicated effort in helping to construct our own StarCraft bot. The authors would also like to thank the SSCAIT community for their kind support. The authors are equally thankful for Dennis Waldherr and his BASIL ladder, as it provided an invaluable test-bed like environment to see replays in a timely fashion, ensuring our StarCraft bot is performant and bug-free.

## References

Amador, G.; and Gomes, A. J. 2019. Bounded-Search Pathfinders Based on Influence Maps Generated by Attractors and Repulsors. *IEEE Transactions on Games* 99.

Björnsson, Y.; and Halldórsson, K. 2006. Improved Heuristics for Optimal Pathfinding on Game Maps. In *Artificial Intelligence and Interactive Digital Entertainment Conference*, 9–14.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1: 269–271. doi:10.1007/BF01386390.

Doran, J. E.; and Michie, D. 1966. Experiments with the Graph Traverser program. In Lockwood, M., ed., *Proceedings of the Royal Society A*, 235–259. London, United Kingdom: Royal Society.

Hagelback, J.; and Johansson, S. J. 2008. Dealing with Fog of War in a Real Time Strategy game environment. In *2008 IEEE Symposium On Computational Intelligence and Games*, 55–62. doi:10.1109/CIG.2008.5035621.

Hagelbäck, J. 2015. Hybrid Pathfinding in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games* 8: 1–1. doi:10.1109/TCIAIG.2015.2414447.

Harabor, D.; and Grastien, A. 2011. Online Graph Pruning for Pathfinding on Grid Maps. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 1114–1119. San Francisco, CA, USA: AAAI.

Harabor, D.; and Grastien, A. 2014. Improving Jump Point Search. In Chien, S.; and Fern, A., eds., *Proceedings International Conference on Automated Planning and Scheduling, ICAPS*, 128–135. Portsmouth, NH, USA: AAAI.

Harabor, D. D.; and Grastien, A. 2012. The JPS Pathfinding System. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search*, 207–208. Niagara Falls, Canada: AAAI.

Hatem, M.; Stern, R.; and Ruml, W. 2013. Bounded Suboptimal Heuristic Search in Linear Space. In *Proceedings of the Symposium on Combinatorial Search (SoCS-13)*.

Koenig, S.; and Likhachev, M. 2005. Fast Replanning for Navigation in Unknown Terrain. *Transactions on Robotics* 21: 254–363. doi:10.1109/tro.2004.838026.

Koenig, S.; Likhachev, M.; and Furcy, D. 2004. Lifelong Pathplanning A*. *Artificial Intelligence* 155: 93–146.

Ninomiya, K.; Kapadia, M.; Shoulson, A.; Garcia, F.; and Badler, N. 2014. Planning approaches to constraint-aware navigation in dynamic environments. *Computer Animation and Virtual Worlds* .

Stentz, A. 1994. Optimal and Efficient Path Planning for Partially-Known Environments. In *Proceedings of the International Conference on Robotics and Automation*, 3310–3317.

Sturtevant, N.; and Buro, M. 2005. Partial Pathfinding Using Map Abstraction and Refinement. In *Proceedings of the 20th National Conference on Artificial intelligence*, volume 3, 1392–1397.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games* 4(2): 144–148. doi:10.1109/TCIAIG.2012.2197681.

Sturtevant, N. R.; Sigurdson, D.; Taylor, B.; and Gibson, T. 2019. Pathfinding and Abstraction with Dynamic Terrain Costs. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, 80–86.

Wagner, D.; Willhalm, T.; and Zaroliagis, C. 2005. Geometric Containers for Efficient Shortest-Path Computation. *ACM Journal of Experimental Algorithmics* 10(1.3): 1–30.

Weber, B. G.; Mateas, M.; and Jhala, A. 2011. A Particle Model for State Estimation in Real-Time Strategy Games. In *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Zarembo, I.; and Kodors, S. 2015. Pathfinding Algorithm Efficiency Analysis in 2D Grid. In *Proceedings of the International Scientific and Practical Conference*. doi:10.17770/etr2013vol2.868.

Zhu, W.; Jia, D.; Wan, H.; Yang, T.; Hu, C.; Qin, K.; and Cui, X. 2015. Waypoint Graph Based Fast Pathfinding in Dynamic Environment. *International Journal of Distributed Sensor Networks* 11. doi:10.1155/2015/238727.