Map Marker: a Multi-Agent Pathfinder for Cohesive Groups in Real-Time Strategy Games

Enrique Wicks Rivas
Instituto de Computação
Universidade Federal da Bahia
Salvador, Brasil
enrique.wicks@ufba.br

Rodrigo Souza
Instituto de Computação
Universidade Federal da Bahia
Salvador, Brasil
rodrigorgs@ufba.br

***Abstract*—Multi-agent pathfinding algorithms are essential to studies on graph navigation with more than one agent. Their importance is especially evident in Real-Time Strategy (RTS) games, as even modern implementations can still lead to unintended agent behaviors that cause frustration. One such problem occurs if a group of agents breaks military formation when given the order around a big obstacle, becoming more vulnerable to enemies. To mitigate this problem, we propose Map Marker, an extension of the A* algorithm that takes in concepts from ant colony optimization to improve group cohesion. To evaluate the new algorithm we constructed three scenarios that stress the separation of groups. These scenarios were simulated with different parameters and the best results were then recreated inside the engine of the game StarCraft II and compared to A*. As a result, we found that the new algorithm effectively maintained group cohesion in all scenarios, resulting in fewer casualties and less time needed to reach the destination when the group is ambushed by enemies, in addition to being 2.8 faster than A* when computing multiple paths.**

Index Terms*—pathfinding, RTS, A

I. Introduction

Real-Time Strategy (RTS) games in their classic form are military simulators where players collect resources, build infrastructure, train soldiers, and control their army's movement and tactics.When the game's pathfinder is well implemented, the army follows the player's command with precision and efficiency in a responsive manner that is critical to competitive RTS games.

When looking at how polished modern implementations are, pathfinding in RTS games might appear as a solved problem. There are still cases, even in the market leader StarCraft II (Blizzard Entertainment, 2010), in which player's agency is disrupted by the pathfinder, leading to a lack of responsiveness that could even cause the loss of a match and frustrate the player [1], [2].

One of such cases is what we call the separation problem, which happens when commanding a group of agents around a large obstacle and the group splits because each agent computed a different optimal path. While this behavior may result in the shortest time for the whole group to reach the destination, it also means that the army moves in a vulnerable and uncoordinated fashion, making it a weaker force in numbers if an engagement with the enemy army occurs.

In this paper, we propose a multi-agent pathfinding algorithm called Map Marker. It computes paths for groups of agents in a way that units tend to stick together when moving around large obstacles while allowing some separation when moving around small obstacles, so that path lengths are not dramatically increased. We believe this behavior is more in line with players' intents when playing RTS games.

To evaluate the Map Marker algorithm, we built three scenarios that stress the separation problem and computed paths using both Map Marker and the traditional A* algorithm. After that, we recreated the scenarios and paths inside the engine of StarCraft II. Finally, for each algorithm and scenario, we measured two variables: time to reach destination and number of casualties in combat.

The remainder of this paper is structured as follows. In Section II we discuss movement and pathfinding in RTS games. In Section III we describe the Map Marker algorithm. In Section IV we present the scenarios that are used to evaluate the performance of the Map Marker with different combinations of parameters, while in Section V we compare Map Marker and A* through multiple simulations in StarCraft II. In Section VI we discuss related work and, finally, in Section VII we present concluding remarks and ideas to further improve the new algorithm.

II. Movement in RTS Games

In an RTS game, players indirectly control agents, called *units*, by giving them orders. Such orders include moving to a specific point in the map or attacking enemy units and buildings [3].

The first step of agent control is the selection, i.e., how the player decides to which agent to give each order. In Dune II (Virgin Games, 1992), the player could only select one unit at a time. As years passed by, new mechanics for selection were developed, and players could now select multiple agents and delegate orders on the whole group instead of having to select one by one.

After selecting a unit group, the player can delegate orders to the group. The most basic order an agent can receive is the one to move to a destination. This is where pathfinders come in. The second most common order is the one to attack; the unit will move towards its target and shoot them or whack

them if they are a melee combatant. Modern RTS games are also able to mix attack and movement into one action commonly referred to as "attack-move" or "a-move", in which the unit will move towards the target point and engage in battle with any enemy they find along the way.

A. Pathfinding in RTS games

One of the most commonly used pathfinder solutions in games, A* [4], is an extension of Dijkstra's shortest-path algorithm [5]. The algorithm efficiently finds the optimal path from node A to node B of a graph, just like Dijkstra's algorithm, but uses heuristics during its calculations to speed up processing.

The core concept in A* is the exploration of nodes based on their real traveled distance plus their heuristic distance to the target, making up the total distance. Nodes are explored based on this total distance so that the next node to be explored is always the one with the shortest total distance among the nodes on the found/open nodes list.

Moving the algorithm inside an RTS engine requires a bit of care. Games built on a grid, like Dune II, could assign each node to a tile and run A* with a graph that is a perfect representation of the battlefield with blocked nodes that the units cannot pass through and open nodes that are traversable. More modern approaches use navigation meshes to reduce the number of nodes in the graph, significantly speeding up the exploration in A*.

A common approach to deal with pathfinding of groups is to compute a path for each agent [6] using A*. This is the approach used in StarCraft II[1], and it allows each agent to find the shortest path considering its characteristics, such as size and ability to climb ledges. Although multiple A* runs would seem computationally expensive, they are feasible in real games due to optimizations such as using navigation meshes, choosing suitable data structures, and performing hierarchical pathfinding [6]–[8].

Even after integrating graphs into the RTS maps to be able to use A*, a developer's job is not over. Making an agent mindlessly follow the output path of A* will lead to rigid and clunky movement; if multiple units are present they might stumble upon one another or even get stuck if they ram directly into each other. Modern RTS games have very sophisticated algorithms doing the motion planning that drives the agents through the paths they receive from A*. Algorithms such as flocking [9] and flow fields [10] can be used to make units move around other units and through chokes fluidly.

B. Responsiveness and Frustration

Disruption in the player input and expected outcome is a common source of frustration found in games. The defining example of a disruptive agent causing frustration is latency created by a bad internet connection and input delay [1]. Other examples include elements that unintentionally mess with the player's ability to properly control the game, such as a camera that gets stuck or is blocked by the scenario in a weird angle [2].

In a competitive RTS match, an agent moving in dissonance to the player's intent could lead to the death of said agent or make it give away information of its whereabouts to opponents, which could lead to the loss of a match. In highly competitive games, losing to seemingly unresponsive controls is bound to cause great frustration.

In an email to StarCraft II's Lead Engineer, James Anhalt III told us that the game prioritized predictability in the behavior of its agents. A player wanting to keep their agents close to one another could do so with direct control, as their inputs would reliably follow the same rules. This philosophy necessitates less developer overreach with complex game logic to control agent behavior as this task is given to the player as another venue of expression and mastery.

Another philosophy could say that the predictability of agent behavior is not clear during gameplay. In other words, the player has expectations towards how the agents should behave when given an order that are not always met by the pathfinder. When moving an army around a large obstacle, which player would expect a couple of their units to disband and follow the opposite path the army took?

In this work we will venture into this second philosophy, exploring game logic that tries to add a communication layer between agents so that they can better consider situations that would leave the army vulnerable without direct input from the player.

C. The Separation Problem

The concept of army *cohesion* is related to the vulnerability of a group of agents mid-maneuver; if the army can quickly regroup or it is already unified, it has cohesion. If an obstacle is too large and the split army cannot re-unify rapidly to face a potential threat, then the army's cohesion is broken. The *separation problem* is the unexpected separation caused by the game's pathfinder that breaks an army's cohesion.

When moving around a small object, the army may safely split as it can quickly reassemble to face a threat. A pathfinder that chooses to split around small obstacles does not cause a separation problem; instead, this may contribute to making the agents reach their destination earlier since it avoids unnecessary detours.

A good pathfinder should avoid the separation problem without causing extensive increases in path lengths. The case of the small obstacle that does not break cohesion when splitting around is important because a good pathfinder should be flexible enough to split the agent group in such cases and not overcompensate by moving the entire army through one side just to maintain cohesion.

III. The Map Marker Algorithm

To solve the separation problem we experimented with variations on the A* algorithm. Our idea was to equip the pathfinder with the ability to figure if the obstacles it finds while mapping a path will break the group's cohesion or not.

[1]Source: private email from StarCraft II's Lead Engineer, James Anhalt III.

We came up with the Map Marker algorithm; it uses a system of path markers that each agent leaves on its found path so that the next agent of the group that is calculating its path will consider the marked nodes as more favorable by applying a cost reduction on its perceived length. We hope that this duality of independently calculated paths that factors in the previous path calculations makes the agents more likely to keep cohesion while maneuvering around large obstacles, but not overcompensate around smaller ones to the point of significantly reducing the army's speed.

The Map Marker is an extension to A* that records the path of each agent through markers in the nodes so that paths that keep the group's cohesion are favored. Although the idea of path markers is inspired by the virtual pheromones used in ant colony optimization [11], the markers are used in a group's pathfinding and then discarded, so they leave no impact in the graph for future agents.

The Map Marker algorithm has two parameters: *marker factor* and *marker cap*. The *marker factor*, between 0.0 and 1.0, is the value that is multiplied by the perceived distance of each node in selected paths; that way, lower values cause agents to favor nodes in paths that have been selected by other agents. The *marker cap*, also between 0.0 and 1.0, defines the lowest marker value that can be assigned to a node. Limiting the lowest marker value prevents that successive applications of the marker factor result in values that are so attractive that groups choose to stick together even in the face of small obstacles.

The algorithm has three core parts: the main body, the pathfinding step, and the node update function. The main body (Algorithm 1) controls the whole process. It starts by ordering the group by their straight-line distance to the target position. Then it iterates over the now ordered group calling the pathfinding function and the node updating function for each agent.

The pathfinding step (Algorithm 2) works mostly as A* would, taking the starting position the agent is in and finding the nodes that lead to the target position. The difference is that the node's marker value is multiplied by the total distance.

And, finally, the node update function (Algorithm 3) takes in the mapped path in the previous step and updates the markers on its nodes and neighboring nodes by multiplying the marker value by the marker factor. Nodes that are adjacent to the path have the marker value updated too; however, the marker factor, in this case, is added 0.02 before the multiplication, so neighbors are less attractive than the nodes that actually belong in the path.

A. Discussion

The Map Marker algorithm starts by taking one of the agents of the group and calculating its path. At this point, all nodes have the initial marker value, so the path will be exactly the output an A* algorithm would give. After that, it updates the path's nodes and their neighbors. Then, it computes the path for the second unit; at that moment, each node that was marked by the first agent now presents a cost reduction, which skews the pathfinder to select the nodes with markers if they do not drastically increase the path's length. The algorithm then keeps iterating over the remaining agents and updating the node's marker values until all paths are calculated.

The nodes start with a marker value of 1.0, meaning that the first time a path crosses a node, this multiplication will not change the total distance the pathfinder sees. Once the node gets marked, its marker value is multiplied by our *marker factor* (between 0.0 and 1.0), decreasing further every time it is selected. The lower the marker, the stronger its effects in warping paths.

This algorithm, although heavily inspired by ant colony optimization (ACO), differs from that algorithm in that ACO is using multiple agents to find an optimal path from a point to another; here, we have a starting position for each agent and are not interested in the best path for each agent, but in keeping all paths close enough to not break the group's cohesion.

It is worth noting that the markers used in the pathfinder should impact only the movement of the group in that movement instance and are not meant to stay permanently in the node.

Another crucial point is that Map Marker is order-dependent, in that each agent affects all the others coming after it. We are using a heuristic that prioritizes agents that are the closest to the target (using Euclidean distance), so they will influence the farthest agents. In preliminary experiments, we also tried sorting agents by their distance to the group's gravity center. The distance to target heuristic tended to yield smaller average path lengths, although this cannot be guaranteed in all scenarios.

As the path computed for the first agent is not influenced by marker values, it is guaranteed to be optimal (i.e., with minimal length), since the algorithm is reduced to A*. For subsequent paths, Map Marker does not guarantee optimality as it will favor paths that increase the overall group's cohesion over individual agent's path lengths.

IV. Preliminary Evaluation and Parameter Selection

We have conducted a preliminary evaluation of the Map Marker algorithm to find suitable parameter configurations (for *marker factor* and *marker cap*) and assess whether path lengths are significantly increased when compared to A*. To this end, we have come up with scenarios where the standard pathfinding solutions fail, and then implemented the algorithm and the scenarios in a prototype inside the Unity game engine[2]. The prototype was run with multiple parameter configurations for each scenario, including the configuration that makes it equivalent to A*.

A. Scenarios

We have built three distinct scenarios that stress the separation problem, shown in Fig. 1. In the images, green squares represent empty space, red squares represent obstacles, the flag

[2]Available at https://unity.com/

Algorithm 1: Base of the Map Marker algorithm

Input : A group of agents, a target position, a marker factor, and a marker cap
Output: Nothing (a path is assigned for each agent)

```
1  foreach node in map nodes do
2  |  node.markerValue ← 1.0;
3  end
4  Sort the group of agents in increasing order of straight line distance to target position ;
5  foreach agent in the group of agents do
6  |  path ← MapMarkerPathfinder(agent, target position);
7  |  agent.SetPath(path);
8  |  UpdateNodes(path, marker factor, marker cap)
9  end
```

Algorithm 2: The MapMarkerPathfinder function. In red (line 16) you can see the step that integrates the markers into what would otherwise be an implementation of A*.

Input : An agent and a target position
Output: The path for the agent

```
1  openList ← new List();
2  closedList ← new List();
3  NodeAux ← agent.GetNode();
4  NodeAux.totalValue ← 0;
5  NodeAux.realValue ← 0;
6  openList.Add(NodeAux);
7  while openList.NotEmpty() do
8  |  currentNode ← node with lowest totalValue from openList;
9  |  if currentNode == target position then
10 |  |  Return currentNode and its parents recursively as the output path;
11 |  end
12 |  neighbors ← currentNode.Getneighbors();
13 |  foreach NodeAux in neighbors do
14 |  |  realDistance ← currentNode.GetRealDistance() + Distance(currentNode, NodeAux);
15 |  |  heuristicDistance ← Distance(target position, NodeAux);
16 |  |  totalDistance ← (realDistance + heuristicDistance) * GetMarker(NodeAux);
17 |  |  if neither openList nor closedList have an instance of NodeAux with totalValue < totalDistance then
18 |  |  |  NodeAux.parent ← currentNode;
19 |  |  |  NodeAux.realValue ← realDistance;
20 |  |  |  NodeAux.totalValue ← totalDistance;
21 |  |  |  openList.Add(NodeAux);
22 |  |  end
23 |  end
24 |  closedList.Add(currentNode);
25 end
26 Return Failure;
```

Algorithm 3: The UpdateNodes function

Input : a path, a marker factor, and a marker cap
Output: Nothing (the nodes get updated)

```
neighborNodes ← new Set;
foreach node in the path do
    neighborNodes.Add(node.GetNeighbors());
    node.markerValue ← node.markerValue * marker factor;
    if node.markerValue < marker cap then
        node.markerValue ← marker cap;
    end
end
Since the nodes in a path are adjacent and have already been updated, we will remove them from our set;
neighborNodes.Remove(the sequence of nodes in the path);
The adjacent nodes have a softer marker update;
foreach node in neighborNodes do
    node.markerValue ← node.markerValue * (marker factor + 0.02);
    if node.markerValue < marker cap then
        node.markerValue ← marker cap;
    end
end
```

(on top) represents the destination, and the red squares with rounded corners (below) represent units. These scenarios are not meant to be extreme edge cases, since similar scenarios can be easily found in RTS games.

Lonely Tree. In our first scenario, the Lonely Tree consists of a single blocked node between the agents and the target node (Fig. 1a). This solution is not stressing the separation problem directly; it is, instead, a benchmark to test if our pathfinder is overcompensating to guarantee cohesion and will try to move the whole agent group around one side of the obstacle instead of ignoring it and splitting the group around it. For our tests, succeeding in this scenario means dividing the group around the obstacle, since the units can quickly regroup due to the small obstacle size.

Chinese Wall. The Chinese Wall scenario plays directly into the separation problem; the obstacle is a straight line of blocked nodes (Fig. 1b) that completely breaks the group's cohesion if the army does not converge fully to only one of its sides. To succeed in this scenario, our algorithm has to make sure all paths calculated are on only one side of the wall.

Boulder. The Boulder scenario consists of a single massive obstacle in the way of the agent group (Fig. 1c), it is similar to the Chinese Wall in the criteria to break cohesion, but its larger width impacts on the marker distribution, possibly splitting a group that would remain together in the Chinese Wall scenario. Succeeding in this scenario is again to pick a side and have the whole group stick to it.

B. Parameter Selection

Inside Map Marker two parameters need to be tweaked to yield the best results: the *marker factor*, which controls the impact and how fast the marker lowers, and the *marker cap*, which is how low the marker can get. We ran different values in these parameters for each scenario and recorded the mean path length in the calculated paths.

Other parameters that could also be tweaked but were left constant for our tests are the heuristic function used (distance to target in our case), the ordering of the group (we used the heuristic function to order them), the neighbor marker increment, and propagation (we increase the marker value by 0.02 and propagate only to the adjacent neighbors).

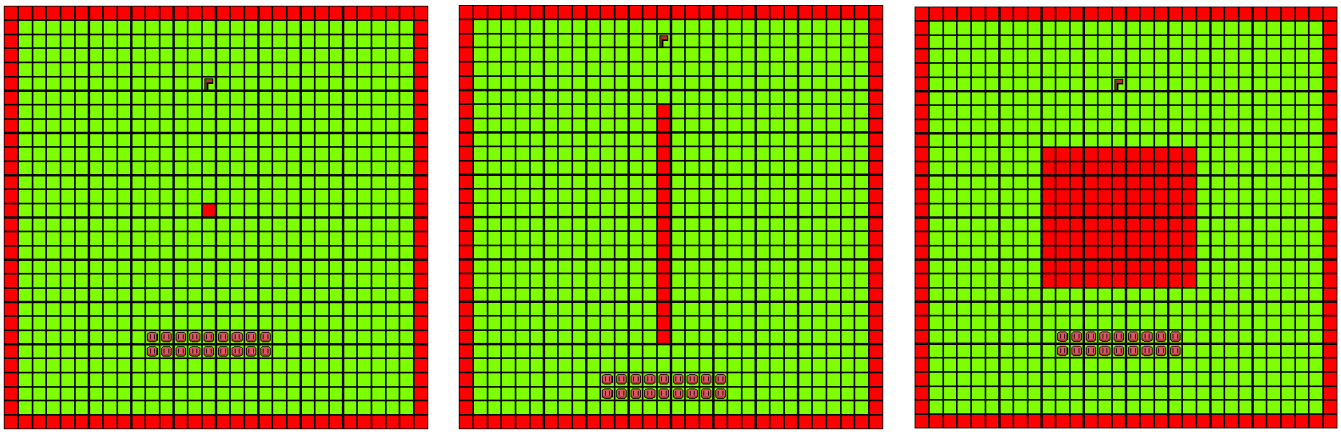
The balance of the marker factor and marker cap is fundamental to make the Map Marker pathfinder work. With a low marker value, the algorithm skews too rapidly making the few first agents define the path of the rest of the group while leaving it too high might not be enough to form a trend that keeps cohesion. A low cap can create super deep "valleys" in the graph that forces all future paths to gravitate towards it, while leaving it too high may not impact the pathfinding enough.

We have tested the following values for the marker factor: 0.50, 0.60, 0.75, 0.90, 0.95, 0.99. As for the marker cap, we tried 0.01, 0.50, 0.75, resulting in 18 possible parameter configurations. We also added the configuration with marker factor and marker cap both equal to 1.0, which reduces the algorithm to A*.

C. Results

In Table I, we present the average path length for all configurations, highlighting those that met the success criteria of all three scenarios. It is worth noting that the control group (A*) failed on both the Chinese Wall and the Boulder scenarios, but have the shortest path lengths.

Out of the tested combinations that passed our three test scenarios, the one that had the shortest average path length,



(a) Lonely Tree (b) Chinese Wall (c) Boulder

Fig. 1: Scenarios used to evaluate the Map Marker algorithm.

TABLE I: Average path for different parameter configurations of the Map Marker algorithm. The line in bold represents the best configuration; the last line represents a configuration that reduces the algorithm to A*. Asterisks in column *Pass* mark configurations that pass all scenarios.

			Average path length		
Pass	**Factor**	**Cap**	**Lonely Tree**	**Ch. Wall**	**Boulder**
	0.50	0.01	20.88	26.88	26.48
*	0.50	0.50	20.08	26.34	25.37
*	0.50	0.75	19.73	26.12	25.23
	0.60	0.01	20.88	26.88	26.37
*	0.60	0.50	20.08	26.34	25.37
*	0.60	0.75	19.73	26.12	25.23
	0.75	0.01	20.88	26.88	26.37
*	0.75	0.50	20.08	26.34	25.37
*	0.75	0.75	19.73	26.12	25.23
*	0.90	0.01	20.25	26.41	25.70
*	0.90	0.50	19.93	26.14	25.43
*	**0.90**	**0.75**	**19.64**	**26.01**	**25.23**
	0.95	0.01	20.18	26.41	24.31
	0.95	0.50	20.18	26.41	24.31
	0.95	0.75	19.85	26.14	23.30
	0.99	0.01	19.51	25.56	23.30
	0.99	0.50	19.51	25.56	23.30
	0.99	0.75	19.51	25.56	23.30
	1.00	1.00	19.51	25.51	23.30

and therefore the best results, was the 0.90 factor with a 0.75 cap. This combination led to a meager 0.6% and 1.9% increase on path lengths in the Lonely Three and Chinese Wall scenarios respectively, but a more substantial 8.2% increase in the Boulder scenario—which is expected, since in this scenario the obstacle is wider.

The visualization of the best results of the Map Marker algorithm can be seen in Fig. 2d, Fig. 2e, and Fig. 2f. The control group, running A*, is shown in Fig. 2a, Fig. 2b, and Fig. 2c.

D. Complex scenarios

The three scenarios presented previously, although useful for parameter selection, do not represent realistic scenarios that would be found in a game match. To evaluate the Map Marker algorithm concerning its ability to prevent the separation problem, as well as its run time performance, we have run both A* and Map Marker on all 75 StarCraft maps available on Moving AI Lab's benchmarks for grid-based pathfinding [12].

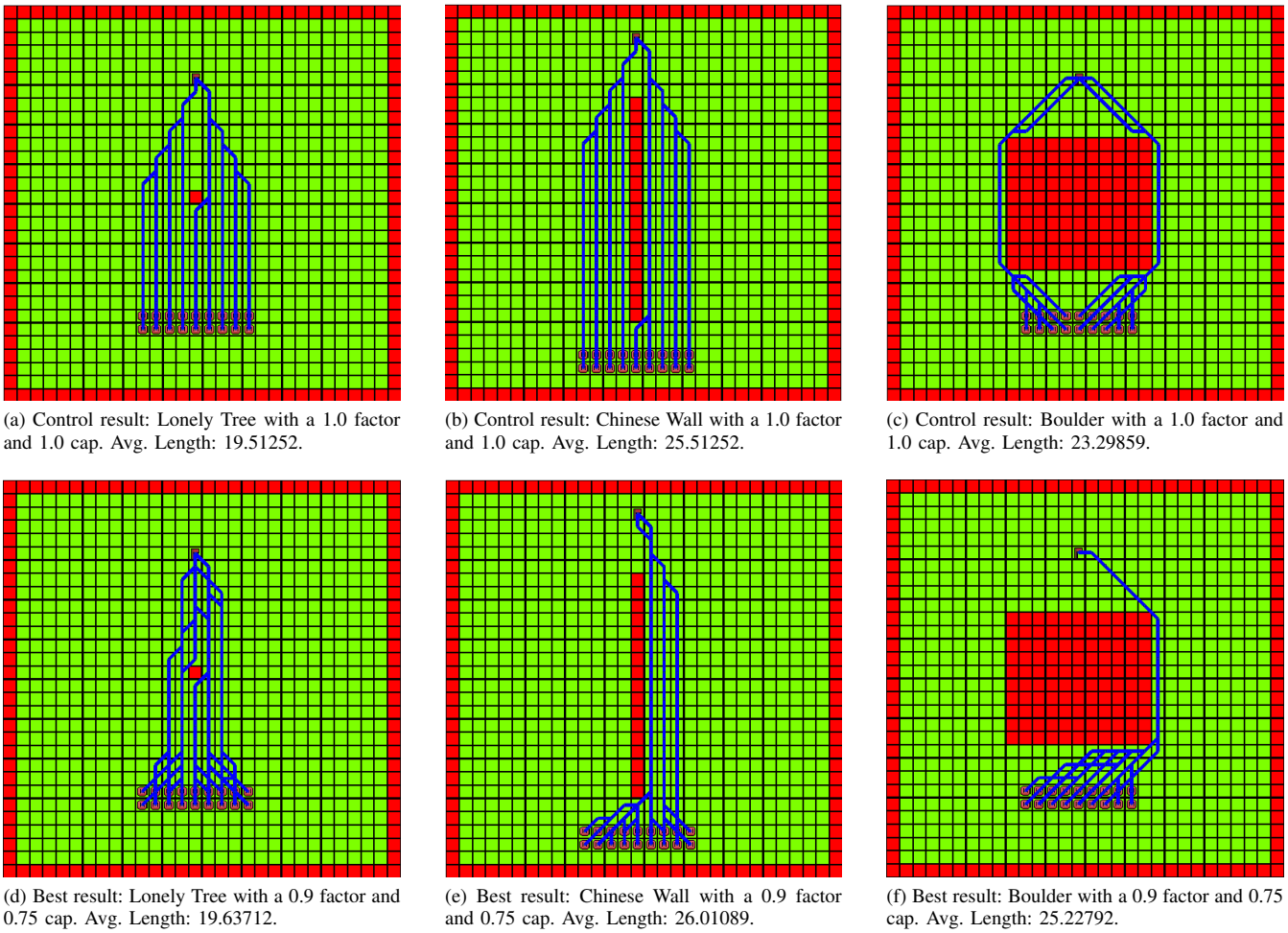
For each map, we have created 10 possible scenarios by randomly choosing a start and a target point. The start point was a reference used to position 50 agents: the first agent occupied the start point, and each additional agent occupied a point adjacent to the previous agent. All agents had the same target point.

For each scenario, we plotted the paths found by both A* and Map Marker, and inspected visually to determine whether the separation problem occurred; each scenario was first inspected by one person and, in case of doubt, all the authors discussed to reach a consensus. This happened to 12 scenarios.

Of all 750 scenarios, 53 ($\tilde{7}$%) presented the separation problem. Among those, the separation occurred only in A* in 48 scenarios ($\tilde{9}$1%), and in both algorithms in the remaining 5 scenarios ($\tilde{9}$%).

The results suggest that A* and Map Marker yield similar results in most scenarios, as exemplified in Fig. 3a. In a significant number of cases, however, A* suffers from the separation problem. In the vast majority of these cases, Map Marker can prevent the problem (see Fig. 3b). Even in the few cases in which Map Marker suffers from the separation problem (e.g., Fig. 3c), the agents tend to regroup shortly after the obstacle.

Regarding run time, A* took on average 8.2 seconds to compute paths for all agents, while Map Marker took 3.0 seconds. It should be noted that these long times are due to unoptimized algorithm implementations in a script language. Using navigation meshes would greatly improve performance [7], as



(a) Control result: Lonely Tree with a 1.0 factor and 1.0 cap. Avg. Length: 19.51252.

(b) Control result: Chinese Wall with a 1.0 factor and 1.0 cap. Avg. Length: 25.51252.

(c) Control result: Boulder with a 1.0 factor and 1.0 cap. Avg. Length: 23.29859.

(d) Best result: Lonely Tree with a 0.9 factor and 0.75 cap. Avg. Length: 19.63712.

(e) Best result: Chinese Wall with a 0.9 factor and 0.75 cap. Avg. Length: 26.01089.

(f) Best result: Boulder with a 0.9 factor and 0.75 cap. Avg. Length: 25.22792.

Fig. 2: Best paths found by A* and Map Marker.

well as using Hierarchical Path-Finding A*, which is expected to be about 10 times faster than traditional A* [8].

Therefore, in addition to mitigating the separation problem, Map Marker improved run time performance by a factor of 2.8. This can be explained by the fact that markers laid when computing a path restrict the number of nodes that need to be explored when computing subsequent paths.

V. Evaluation using StarCraft II

Although the preliminary results involving path lengths are encouraging, they are incomplete for two reasons. First, in a realistic simulation, the time taken to reach the destination depends not only on path length but also on possible collisions between agents. Since agents normally cannot pass through each other, if the computer paths cram our agents together we run into the risk of having various collisions that slow down the real movement of the group.

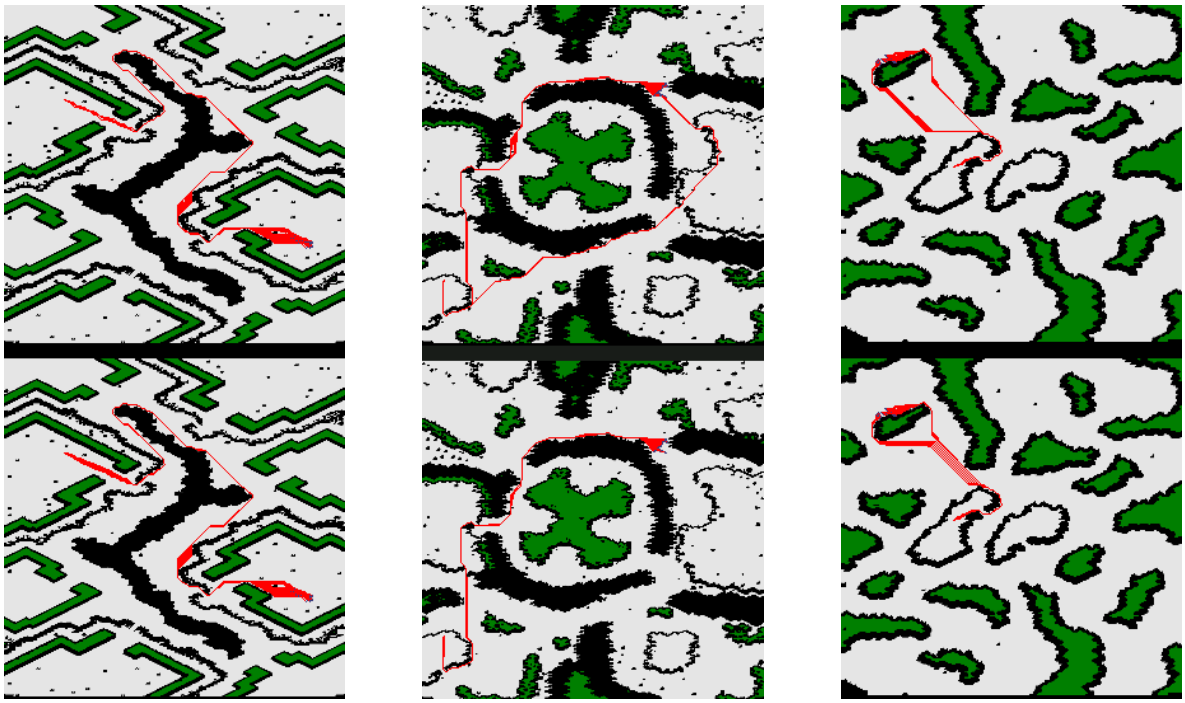
Second, the results do not account for combats, which not only affect the time to destination but can also result in casualties. The preliminary evaluation fails to highlight how important cohesion is in an engagement.

To measure how well Map Marker fares in a real case, we recreated our test scenarios inside the game StarCraft II with our best case's waypoints and checked how well they performed in comparison to A*. We adopted StarCraft II as testing grounds because it is a competitive RTS that, although published in 2010, still has a large player base, with around 1 million players [13].

A. Methods

Inside StarCraft II we were able to create game maps similar to our scenarios and simulate the orders the agents would be given if they used our algorithm. We have selected the classic Zergling as our test agent; this unit was selected due to it being a fast close combat fighter that heavily relies on the positioning of its engagement to succeed in battle. In virtually all competitive matches the faction is present in large quantities, demanding an efficient multi-agent pathfinder.

The tests were run with either Map Marker or A* paths in two different categories: without combat, to test the pure difference of speed, and with combat, to test the impact of each pathfinder in battle results. In the second category of



(a) No separation problem (b) Separation problem with A* (c) Separation problem on both algorithms

Fig. 3: Real scenarios from StarCraft. Green and black are obstacles. Paths are in red (A* on top, Map Marker on the bottom).

tests, we spawn nine enemy combatants (in contrast to our 18) in specific points (represented by red circles in Fig. 4) and measured how well each pathfinder performed in friendly casualties. The chosen spawn points are unfavorable either for armies with broken cohesion or, in Lonely Tree's case, for armies that take too long to regain cohesion.

Once the tester selected the test type by typing a command, the agent group would receive their orders to attack-move through a series of waypoints. Each unit would have its own set of waypoints that corresponds to the path found for the agent back either by A* or by the best configuration of Map Marker.

Each test ended when all allied Zergling finished their final order by reaching the target point or dying; at that point, a script would inform us of how many Zerglings survived the encounter and how long did it take to reach the target destination (in seconds).

B. Results

The results are summarized in Table II, which shows the average time spent by the group to reach the destination (both with and without combat), as well as the number of units that died in combat. The results are presented separately for A* and Map Marker.

The average times are shown in Fig. 5, with the standard deviation shown as a vertical error bar. The difference in time between the algorithms in the Lonely Tree scenario is small and not statistically significant (which is expected since both algorithms split the group around the obstacle). In the other two scenarios, however, the difference is statistically significant at the 95% level (using Wilcoxon Rank Sum Test with Bonferroni correction).

In simulations without combat, groups guided by Map Marker in the Chinese Wall and Boulder scenarios took about 13% longer, on average, to complete the path. This represents a large effect size, as measured by Wilcoxon effect size ($r > 0.5$).

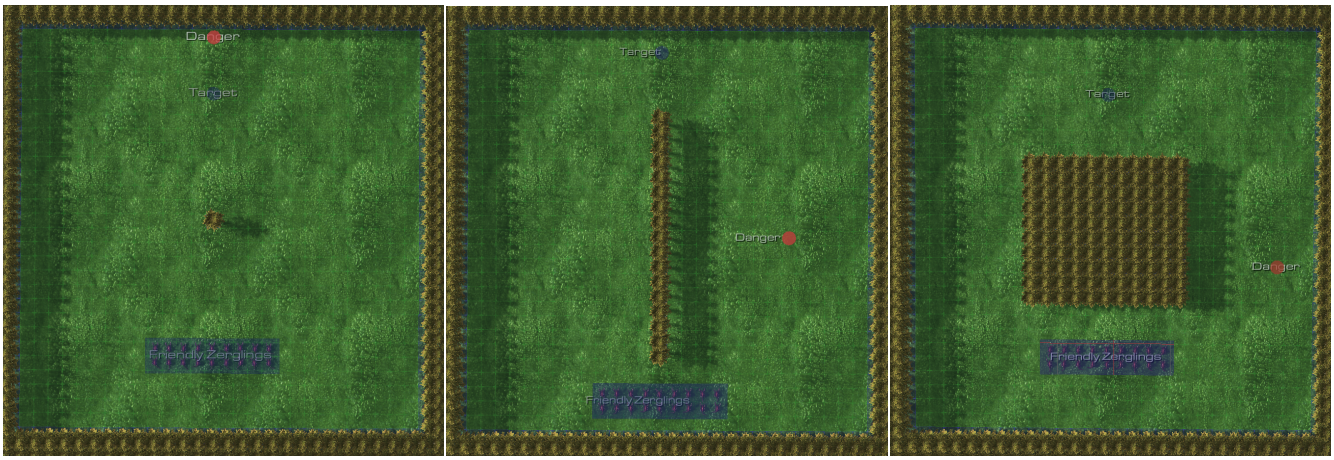
The opposite holds for simulations involving combat. In these cases, Map Marker took 8.2% and 12.7% less time (Chinese Wall and Builder, respectively). The difference is statistically significant at the 95% level, and the effect size is large.

Fig. 6 shows the average number of casualties, i.e., player units that died after combat. Although the average is lower for Map Marker in all scenarios, the difference is statistically significant only in the Boulder scenario, with large effect size, in which the number of casualties was reduced by 47.8%.

In summary, in the event of a combat, groups led by the Map Marker algorithm tend to present less casualties and to reach the destination earlier than groups led by A*.

C. Discussion

During testing, we observed that the waypoints method used to replicate the paths found in Map Marker did not integrate perfectly into the engine; on multiple occasions, the agents went back to a point they had already passed by, either after



(a) Lonely Tree (b) Chinese Wall (c) Boulder

Fig. 4: Scenarios used to evaluate the Map Marker algorithm in StarCraft II.

TABLE II: Results of the simulations performed in StarCraft II with a group of 18 units, averaged over 10 simulations for each configuration. Times are given in seconds.

Scenario	Avg. Time (without combat) A*	Avg. Time (without combat) Map Marker	Avg. Time (with combat) A*	Avg. Time (with combat) Map Marker	Avg. Casualties A*	Avg. Casualties Map Marker
Lonely Tree	5.0	5.3	12.2	12.1	2.1	1.6
Chinese Wall	6.0	6.8	14.9	13.0	3.9	2.9
Boulder	6.0	6.8	15.8	14.5	4.6	2.4

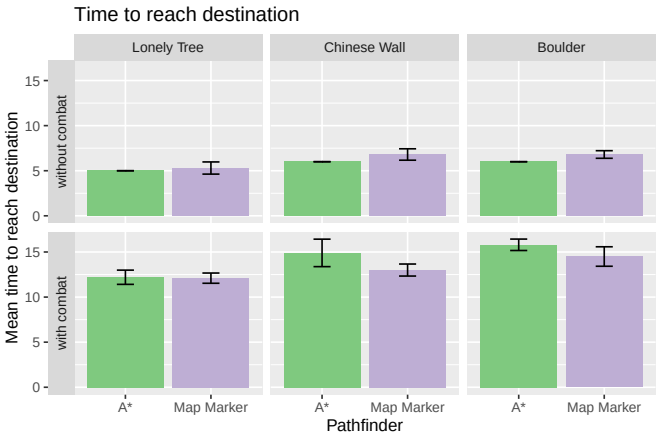


Fig. 5: Time (in seconds) groups needed to reach their destination, averaged over 10 simulations for each configuration.

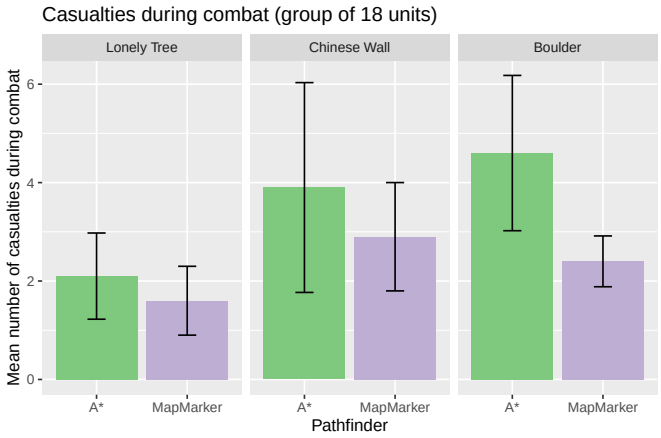


Fig. 6: Number of units that died in combat while following their paths, averaged over 10 simulations for each configuration.

being dragged by other units in the group or after killing enemy combatants.

This problem is caused by the fact that our pathfinder runs offline and, thus, it does not recompute paths during the simulation. This is in contrast with StarCraft II's native pathfinder, which seems to recalculate paths when combat is over.

In preliminary evaluations, we measured the time it took for units to reach their destination using StarCraft II's native pathfinder. Since this pathfinder is not subject to the problems caused by the use of waypoints to represent static paths, direct comparison with Map Marker would not be fair. Thus, we decided to switch to precomputed paths, using A*. We verified that these paths are very similar to those found by StarCraft II.

When moving around large obstacles, such as the Boulder,

the difference between pure A* and Map Marker is evident. In Map Marker, some units inevitably choose longer paths in order to keep cohesion, and this results in increased time for the whole group to reach the target position.

On the other hand, if the player expects their group to be attacked while following the path, the paths chosen by Map Marker allow units to reach the destination earlier, given the group survives the attack. This can be explained by the fact that using Map Marker, the whole group engages in combat, leading to a quick victory and also fewer casualties.

A solution to have both speed and cohesion resilience would be to implement a dual behavior solution: use Map Marker only while attack-moving and A* in common movement. That way, the agent group keeps cohesion when the player is expecting an engagement with enemy troops, but still has the best average path lengths to move in situations where speed is more important than cohesion, such as moving in reinforcements or retreating. Map Marker can easily integrate this change of behavior concept as its output when set to a marker value of 1.0 is A* paths. A function could easily switch marker values depending on the type of order the player is issuing.

A counterargument to this dualistic approach is that it could become an obscure game mechanic that the players will not understand or be able to replicate during competitive gameplay which itself could lead to frustrating situations. We argue, however, that this mechanic allows players to gain control over their troops without having to resort to micromanagement such as controlling individual agents or subgroups.

VI. Related Work

The idea of customizing the distance calculation in A* for use in RTS games is not new. In many cases, the goal is not to find the shortest paths, but to find reasonably short paths while avoiding common problems found during gameplay. To the best of our knowledge, however, no pathfinder in the literature focuses on individual path length performance while avoiding the separation problem.

Critch and Churcill [14] proposed a pathfinder that avoids areas visible by enemies units, so that player units can move closer to enemy buildings without being intercepted in the way. To this end, they build influence maps around enemies and use them in the cost calculation of A*. The new pathfinder is evaluated in StarCraft.

Geramifard *et al.* [15] noticed that sometimes separated groups controlled by the player take paths that intersect at some point, effectively slowing them down. To mitigate this problem, they proposed a pathfinder that assigns a higher cost to intersection points, and then run A* considering these costs.

Hagelbäck [16] evaluated two approaches for positioning units in a group when attacking: flocking and potential fields. They ran simulations in StarCraft and found potential fields to be much more computationally expensive.

VII. Conclusion

This work presents Map Marker, a multi-agent pathfinder algorithm that aims to increase agent cohesiveness to avoid group splits in situations that would leave the group vulnerable. The algorithm guaranteed cohesion in the tests imposed by our work, significantly reducing casualties in combat. The time needed to reach the destination decreased when units engaged in combat and increased otherwise.

Our results indicate that the new algorithm without additional components is not a direct upgrade to the A* algorithm in RTS games; however, a developer that is more interested in cohesive group movement or is willing to add game logic that switches movement behaviors on different contexts can use the new Map Marker algorithm to reach their goals as we presented.

In future experiments it would be interesting to change the test map structure to incorporate navigation meshes. Also, further research is needed to understand how to leverage other algorithms used in RTS movement, such as flocking and potential fields, to mitigate the separation problem.

References

[1] S. Jorg, A. Normoyle, and A. Safonova, "How responsiveness affects players' perception in digital games," in *Proceedings of the ACM Symposium on Applied Perception (SAP'12)*, Jan. 2012, pp. 33-38.

[2] A. Nylund and O. Landfors, "Frustration and its effect on immersion in games: A developer viewpoint on the good and bad aspects of frustration," M.S. thesis, Dept. Informatics, Umea Univ., Umea, Sweden, 2015.

[3] R. Lara-Cabrera, C. Cotta, and A. J. Fernández-Leiva, "A review of computational intelligence in RTS games," in *2013 IEEE Symposium on Foundations of Computational Intelligence (FOCI)*, 2013, pp. 114–121.

[4] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, Dec. 1959.

[6] F. Pentikäinen and A. Sahlbom, "Combining influence maps and potential fields for AI pathfinding," M.S. thesis, Dept. Computer Science, Blekinge Institute of Technology, Karlskrona, Sweden, 2019.

[7] X. Cui and H. Shi, "A*-based pathfinding in modern computer games," Int. Journal of Computer Science and Network Security, vol 11, no. 1, pp. 125–130, Jan. 2011.

[8] A. Botea, M. Müller, and J. Schaeffer, "Near optimal hierarchical path-finding," *J. Game Dev.*, vol. 1, no. 1, pp. 1-30, 2004.

[9] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *ACM SIGGRAPH Computer Graphics*, vol. 21, no. 4, July 1987, pp. 25-34.

[10] E. Emerson, "Crowd pathfinding and steering using flow field tiles," in *Game AI Pro 360*, S. Rabin, Ed., Boca Raton, FL, USA: h. 23, pp. CRC Press, 2019, pp. 307-316.

[11] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29–41, 1996.

[12] N. Sturtevant, "Benchmarks for grid-based pathfinding," *Transactions on Computational Intelligence and AI in Games*, vol 4, no. 2, pp. 144-148, 2012.

[13] M. D. B. S. Supriyadi, S. M. S. Nugroho, and M. Hariadi, "Fuzzy coordinator based AI for dynamic difficulty adjustment in StarCraft 2," in *2019 Int. Conf. Artif. Intell. and Inf. Technol. (ICAIIT)*, 2019, pp. 322–326.

[14] L. Critch and D. Churchill, "Combining influence maps with heuristic search for executing sneak-attacks in RTS games," in *2020 IEEE Conference on Games (CoG)*, pp. 740–743.

[15] A. Geramifard, P. Chubak, and V. Bulitko, "Biased cost pathfinding," in *Proceedings of the Second AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'06)*, 2006, pp. 112–114.

[16] J. Hagelbäck, "Hybrid pathfinding in StarCraft," *IEEE Transactions on Computational Intelligence and AI in Games*, vol.8, no. 4, pp. 319–324, 2015.