Multithreaded prioritized planning for multiple tailed-agents

Alejandro Gleason, *CS Student at Tecnológico de Monterrey*

Abstract

Multi-agent path finding (MAPF) is the problem of finding paths for multiple agents from their individual start locations to their individual goal locations without collisions. This problem has hundreds of practical applications, including video games, robotics, self-driving vehicles, component routing inside engines, and so on.

In most MAPF algorithms, the conflict resolution mechanism is based on the fact that agents vacate spaces over time, as well as the consideration that the agents have pre-defined shapes and sizes. However, in some environments, we want to route agents that leave a tail/trail in their path, therefore, blocking other agents' paths.

In this paper I present an algorithm that tries to route two "tailed" agents at the same time. The algorithm makes use of multiple threads to run in parallel two agents with A* as an underlying planner; it also makes use of a conflict resolution mechanism based on priority as well as backtracking to save time on path replanning. The algorithm has an optimistic assumption that at each iteration, it is likely for more than one agent to be routed without collisions.

In my experiments I compare my approach with running A* sequentially; the second proves to be better on small grids rather than on large grids, where my solution performs better.

I. Introduction

MAPF algorithms have become incredibly popular due to the range of its appliances in real-world problems, including video games, traffic control, aviation and more [1]. Therefore, computer scientists, roboticists and AI learners have long studied this problem resulting in a lot of efficient algorithms that depending on the specific situation you are involved with, can suit great to provide solution; in section II, I will be describing some relevant well-known algorithms for your consideration.

Formally, the problem considers a graph consisting of vertices and edges, and a set of agents occupying vertices. An agent can only move to an unoccupied, neighboring vertex (defined by a movement policy, which is defined by the underlying planner), and even though finding the minimal sequence of moves to transfer each agent from its start location to its destination is an NP-hard problem, [2] there are many algorithms that present suboptimal but complete solutions, and as stated before, some of them will be briefly presented.

MAPF algorithms tend to assume that agents can occupy only a single location/cell/vertex/node at any time step, and as expressed in the prior paragraphs, resolve conflicts by basically "waiting" until the agent empties that specific cell. And even in some proposals, where agents have a "tail" to emulate agents such as trains, buses, planes or snakes, once they get to their goal node, they start retracting their tail and therefore, releasing their occupied cells (see figure 1), making them available for other agents. [3]

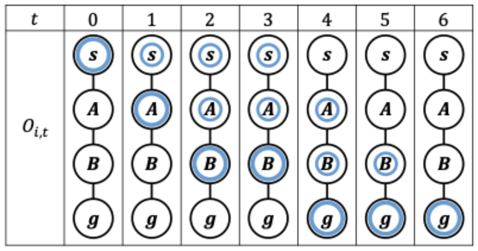


Figure 1: Agent "retracting" its tail [4]

So, for my proposal, agents are treated as tailed agents that once they have found a valid path will potentially mark its composing nodes as occupied. Further details to be described.

On the next sections you will first find related algorithms for further analysis, followed by the context of the problem, my proposed solution and results obtained. Concluding with final thoughts, future work, set up instructions and some evidence/reference to back up what my solution declares.

II. Related Work

In this section, a few related algorithms will be briefly introduced, describing how they intend to solve the MAPF problem and why they got discarded for this specific, tailed agent problem.

Previous approaches can be generally divided into two groups: coupled and decoupled planners. Coupled ones combine the features and constraints of each robot into a single high-dimensional composite robot, and then, plan in this joint space. On the other hand, decoupled approaches plan for each robot independently then adjust the plans in various ways to take into consideration other agents' paths. [3]

Multi-Train Path Finding [4]. While in standard MAPF each agent occupies its current location, here each agent occupies a series of locations, due to them having different sizes and shapes. Each agent starts by occupying only its start location, growing its size of *k + 1* as it leaves, and shrinking back to the size of 1 as it enters its goal location. The core of this algorithm is on the *get-occupation* function, that keeps track at each time frame the occupied nodes for each agent, as well as updating them. At a first glance, it might seem as adaptable to the tailed agent problem, but it's not, since it does not consider fixed positions.

Meta-agent conflict-based search for optimal multiple-agent path finding [5]. This algorithm uses the same waiting function as collision solving mechanism many other algorithms use. What is a bit different is that it presents a constraint-based search, which aims to solve internal conflicts first, to then move to a higher level. This proposal was not only discarded due to the collision resolution mechanism, but also because it has proven to perform poorly on some types of in groups of agents with many internal conflicts. [6]

Decentralized prioritized planning in large multirobot teams [7]. This scheme actually suited quite good; though it is presented in a very abstract way, many of its concepts were useful for this algorithm development. It basically presents a distributed planner that has a message passing mechanism for individual agents to plan their individual paths and if they encounter collisions, broadcast a message to agents with lower priority, so they can replan their path. It could not be implemented directly since it also uses windows of time to solve collisions, but as stated before, multiple ideas were retrieved from here.

The objective of analyzing above algorithms was to mix some of their concepts, functions and data structures on a new solution, rather than taking one of them as it is. It would be naïve to expect to find an idea that suited perfectly from the beginning.

Up to this point, there is no existing solution, at least found on my two-month research, for agents that not only do they occupy multiple cells at the same time, but also have to mark those cells as occupied for the rest of the execution, reconfiguring the grid. This specific problem actually occurs in real world applications, such as when routing multiple cables at the same time within a certain component, among other scenarios.

III. Context of the problem

Consider a two-dimensional grid made up empty spaces as well as occupied spaces by external components. This grid can be translated into a connected graph, where each cell is mapped to a node and vertices are defined according to the local movement policy (e.g., north, south, east, west). We then have a set of n agents with a start node and end node, and for each agent ai a plan π is required in order for them to reach their goal nodes. A path πi is valid if there are no conflicts between agents' paths, and is optimal if it has the lowest cost, among all valid plans.

A conflict between agent ai and agent aj can be defined as the following: any of the graph's nodes in the exploration space is included in paths πi and πj at any point in time of the program's execution. The induvial paths πi and πj can be seen as a set that contains a collection of nodes, therefore, no node can of πi set can be contained in πj set.

The problem is to find an algorithm that in most cases routes all agents in a faster and equally optimal way than running a MAPF algorithm sequentially, and that in its worst-case scenario, provides results not significantly worse than the sequential run, regarding time complexity. The solution should also be highly scalable, allowing to get involved more than two agents, which is the starting problem for this specific paper.

Examples. In figure two you will see what is expected as an average case scenario for the algorithm, which consists of routing multiple agents to their goal node in just one run. Differently, in figure 3, you will see the worst-case scenario flux, where agents end up being routed in a sequential manner. In both examples, black cells are considered as blocks, while white cells are available spaces; colored cells represent the agents. “s” stands for start and “g” for goal.

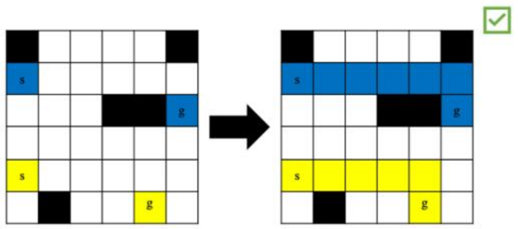


Figure 2: Expected average case scenario

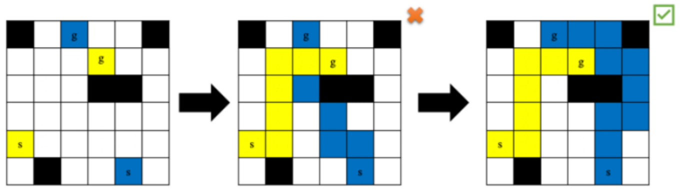


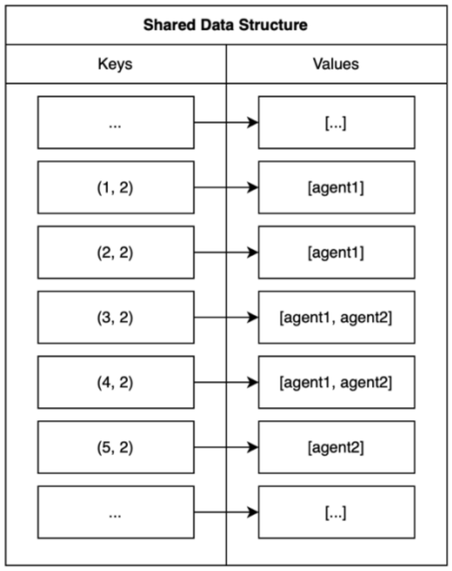
Figure 3: Expected worst case scenario

IV. Solution

As for any problem in computer science, there are multiple ways to go around it. This specific solution has three main features that make it work: firstly, multithreading, so that each thread handles independently an agents’ search operations; secondly, a collision resolution mechanism, that is scalable for hundreds of agents; and lastly, backtracking, which allows agents to return to valid states once they collided. In the next paragraphs, each feature will be described in detail, as well as the general algorithm with a brief introduction to A*.

Multithreading. The use of multiple threads in the algorithm is easy to understand. As we discussed in section II, this algorithm uses a decoupled approach, which means that each agent independently plans its path and subsequently adjusts it to take into account the other agents, avoiding invalid solutions. With this description in mind, let us just map it to the use of threads: each agent launches a thread in parallel with a specific task, to find an optimal route for its assigned start and end location, using A* as the underlying planner; all agents search in the same exploration space, or grid, so no new copies of it are needed. It is important to note that a small modification was made to the traditional A* algorithm, which consists of adding a step to update a shared structure holding the paths for all agents, which are basically composed of coordinates, having this information accessible will help on collision resolution, more details of this shared data structure as well as the mechanism are shared in the next paragraphs.

Shared Data Structure. This data structure is the core for conflict resolution, in this section it is only intended to show how it is structured, as well as its update function, to refer to it in the next paragraph. Attention is requested. The structure is simple. This consists of a list of pairs, where in the first position of the pair we find a pair of coordinates, which can be interpreted as keys, since they will never be repeated according to our update function, and the second part of the pair is composed by a list of agents, which will have a size of 1 to n, where n is the total number of agents to route.


Shared Data Structure

Keys	Values
...	[...]
(1, 2)	[agent1]
(2, 2)	[agent1]
(3, 2)	[agent1, agent2]
(4, 2)	[agent1, agent2]
(5, 2)	[agent2]
...	[...]

Figure 4: Shared Data Structure representation

Each time a new agent gets added to this data structure, a POSIX semaphore makes sure to block it until the accessing thread finishes associating the agent to a specific pair of coordinates. This avoids memory consistency errors, which are prone to happen when two agents try to access the same pair of coordinates at the same time. This appending operation is simple: once an agent has decided to add determined pair of coordinates to its solution path, its thread access and locks the data structure, which goes and checks if the coordinates already exists, if they do, the agent gets appended to the earlier created list of associated agents, otherwise, a new list of agents is generated, initializing it with the current agent. In the next two sections I will refer to this data structure as the "shared vector". Here is the algorithm for updating the shared vector.

Algorithm 1: *update-shared-vector* function

```
1  update-shared-vector(sharedVector sv
   coordinates c, agent a)
2    Lock(vector_mutex)
3    if c in sv then
4        sv.At(c).Append(a)
5    else
6        sv.Add(Make_Pair(c, a))
7    Unlock(vector_mutex)
```

Finally, it is important to say that after each routing iteration, the data structure gets cleared because we will have finished routing at least one agent, and for the agents that were not routed, a new path will be calculated, so it does not make sense to keep past iterations information.

As I stated earlier, the shared vector helps to find collisions, below you will find this algorithm.

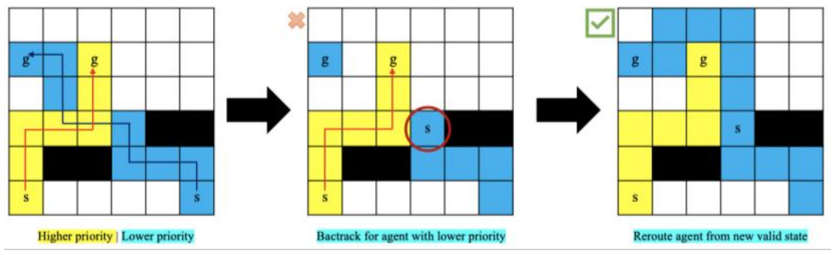
Algorithm 2: *find-collision* function

```
1  find-collision(sharedVector sv, grid g)
2   collisionFound ← false
3   if sv.count() > 0 then
4     for pair in sv do
5       if just one agent a on pair.coordinates then
6         g[a.xCoordinate][a.yCoordinate] ← a.id
7         if AgentReachedGoal(a) then
8             a.completed ← true
9       else
10          collisionFound ← true
11          alp ← GetAgentWithLowerPriority()
12          ahp ← GetAgentWithHighestPriority()
13          removeFromMap(alp.id)
14          ahp.completed ← true
15  return collisionFound
```

As a side note, it is important to say that it would have made sense to use a hash map, since we are mapping coordinates to a list of agents, and access time would be a lot faster, but there is a main reason why a map was not used. STL maps are ordered based on the key, using a "less than" operation, which will structure its content based on the sorted order (numerical or alphabetical), rather than on its insertion order, [9] that is why a vector is used. Now, why would we care about insertion order? Fundamentally, to perform a backtracking operation by iterating through this map, and since this operation cares about the insertion order, to emulate the path travelled by an agent, a vector was used. More detail will be provided on the next paragraph.

Backtracking. Let us first define what backtracking is: according to the National Institute of Standards and Technology, backtracking is a useful technique to find a solution by trying one of several options; if a specific option proves to be incorrect, the idea is to backtrack or restart at a valid point, to try another option. This technique appears commonly on tree related problems, depth-first search, and more. [10]

Now let us map it to my solution, here is the idea: when we find a collision between two agents on the map, we want to make use of the previously determined information to save computation time, specifically, we want to return the agent to a valid position, that is, to a position where there were no collisions. This will mean that at the next iteration part of the agent's path will be already calculated, so it will start from a new beginning point which will be closer to its target node, this will save time to calculate whatever path is remaining to get the agent to its goal node. Below you will find an image that illustrates this concept as well as the backtracking algorithm.



g
g
s
s
Higher priority | Lower priority
Bactrack for agent with lower priority
Reroute agent from new valid state

Figure 5: Backtracking illustration

Algorithm 3: *backtrack-agent* function

```
1 backtrack-agent(sharedVector sv, start s, agentId aid)
2   newStart ← ∅
3   for pair in sv starting in s do
4       if !collision in pair and pair.agentId == aid then
5           newStart ← pair.coordinates
6       else
7           break
8   return newStart
```

A*. A lot has been said about A* throughout this paper. In order to focus on the present proposal for the MTPF problem, only an overview of A* will be provided, contemplating its main concepts. After all, my proposal works with any deterministic search algorithm, such as Dijkstra or D*. A* is a well-known best first search algorithm, used in pathfinding, this algorithm efficiently plans a walkable path between two points in a graph or grid. This algorithm can be considered as an extension to Dijkstra, the key is that it adds a heuristic to calculate the cost of each node, which helps to give direction to the search agent, and to arrive at an optimal solution faster. In the next image you will see how Dijkstra is more expensive when compared to A*, in both scenarios, agents have the same start point and end point, in purple, you will see the explored nodes required by each algorithm to find the solution. [8]

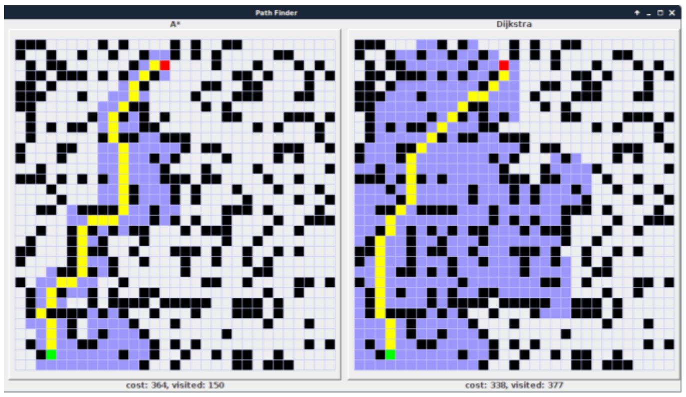


Figure 6: A* (left) vs Dijkstra (right) exploration [12]

To continue, let's discuss the role of heuristics in the algorithm. A * only expands its search either to nodes that have not been explored, or that have a lower cost than was previously calculated arriving from another node. The cost function looks like this $f(n) = g(n) + h(n)$, where $f(n)$ is the total cost for that node n, g(n) the accumulated cost until that node and $h(n)$ is the estimated cost from n to the goal node, calculated with the heuristic function. And that is basically what a heuristic is, a function to estimate the cost to the goal node, according to our movement policy. [8] In my A* implementation I used Manhattan distance as the admissible heuristic, since it is widely used for when allowed movements are in four directions. This results in paths that have as much straight lines as possible.

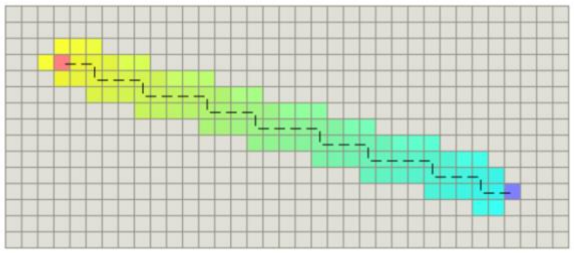


Figure 7: Planning with Manhattan distance [11]

This is the most relevant information about A*, for more, I recommend going to [8] and [11].

Multithreaded prioritized planning. Finally, we will see how previous algorithms and data structures converge into this proposal. The solution will be firstly introduced along with important considerations and towards the end you will see the final algorithm. With the previous information I am sure that understanding the algorithm will be quite simple. It is important to remember that only two agents were launched in this proposal for simplicity, but surely the algorithm is scalable to more agents, it will simply be necessary to add structures to manage the agents as a team, as well as their information.

Suppose we have a team of n agents A = {a_1, ... a_n} which are traveling through a grid with intermediate obstacles; each agent a_i has a pair of starting coordinates s_i and another one ending g_i. Our goal is to find a set of trajectories $\pi = \{\pi_1, \ldots, \pi_n\}$ such that the first point of each corresponds to its agent's initial point, and the last point corresponds to its end point. This set of paths should not allow collision between robots, for that, we will use the previously discussed *find-collision* function to check that a pair of paths π_1

and π_2 do not contain the same node or coordinate, returning a boolean value for this.

Now, each agent must independently plan its path using the given underlying planner, in this case, A*, which is called in the *run-A** function, that takes as an agent object containing all its information, as well the search grid. It will return a collision-free trajectory for each agent, if exists.

With this information as input, let us proceed to describe the algorithm. We are going to start a cycle that only ends when all the agents have found a valid path π_i, inside the cycle we will be creating several threads with the task of launching concurrently the *run-A** function to find the potential path for each agent and updating the shared data structure with the *update-shared-vector* function. The next step is to call the *find-collision* function and if it returns true, the lower-priority agents will have been discarded because they collided with other higher-priority agents, which will be marked as completed. Before launching another thread for the missing agents, we will call the *backtrack-agent* function to reassign each start point and finally, clean the shared-vector; on the other hand, if the find-collision function returns false, we will mark all agents as completed and finish the execution. If the individual threads do not return any valid path at all, the algorithm halts its execution, since not all agents were routed (A* did not find solution).

Algorithm 4: *multithread-prior-plan* function

```
1  multithread-prior-plan(agentSet $a_s$, sharedVector $sv$, grid $g$)
2    repeat
3      for $a_i$ in $a_s$ do
4        $a_i$.LaunchThread(run-A*($a_i$))
5        update-shared-vector($sv, c, a_i$)
5      JoinThreads( )
6      collisionFound ← find-collision($sv, g$)
7      if !collisionFound then
8        for $a_i$ in $a_s$ do
9          $a_i$.completed ← true
10     else
11       for $a_i$ in $a_s$ do
12         if !$a_i$.completed then
14           backtrack-agent($sv$, $a_i$.start, $a_i$.id)
10     if noRoutedAgents then
11       Print("Not all agents could be routed")
12       return
13     $sv$.Clear( )
14   until allAgentsCompleted
```

V. Results

Let us recall what the goal of this algorithm was, which we can divide into two main aspects: 1st objective: To route multiple "tailed" agents in a seamless way, which means that once the user has entered the information for all agents, the routing gets finished without any additional input. 2nd objective: To have a better performance on the routing operations, at least in the majority of the cases (> 50%), when compared to running A* sequentially. It is to say that not all tests were documented, just the most significative ones.

Let us start by tackling the first issue. To verify that this algorithm correctly routed multiple search agents, an extensive set of tests was run in different scenarios, which consisted of modifying parameters such as priorities, starting and ending points, and even the grid configuration. The developed algorithm finds solutions, which consists in routing efficiently all agents, in an average of 90% of the. Below you will see the success rate graph, where the trend line (in red) indicates that the wider the grid, the more probable the algorithm can find solutions for all agents. Tests can be found over here.

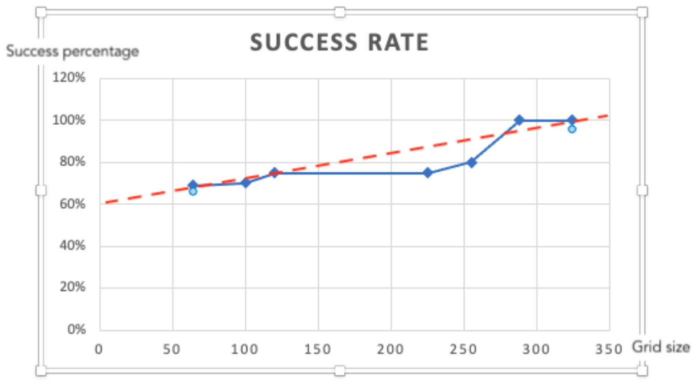


Figure 8: Success rate graph

It is important to consider that when it does not find a solution, it does not mean that it is fault of the present algorithm. In some cases, due to the configuration of the map, solutions cannot be found. There are also some cases where there are solutions for both agents, but the algorithm does not have enough intelligence to find them. This is where the priority assigned to each agent comes in, in section VI we will talk about how this priority could be manipulated to avoid scenarios where the route from one agent blocks another, when there existed another non-blocking path.

Now, regarding the precision of the algorithm, which is important to verify if A* was implemented correctly, another set of tests was run which can be found here. To test them, the map was extracted, and the routes of the agents were traced by hand, following an optimal path. The results were positive since in 100% of the cases the algorithm presented a similar behavior, which means that the so-called underlying planner was implemented correctly. Below you will find examples of expected routes (left side) versus obtained (right side), being quite similar to each other.

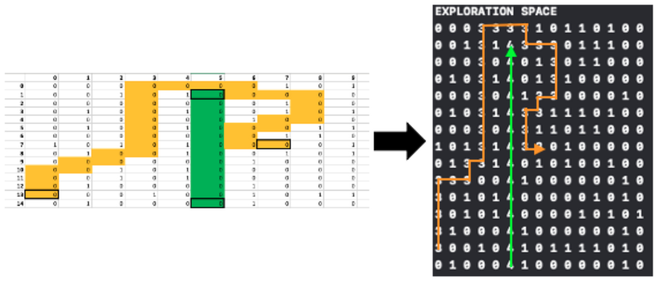


Figure 9: Expected and obtained behavior (1)

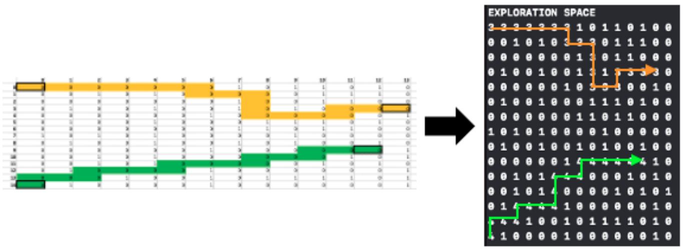


Figure 10: Expected and obtained behavior (2)

Finally, we want to test the second objective, that has to do with the performance of the algorithm. To do this, several runs were carried out on grids of different sizes, the observed result is clear but does not comply with the hypothesis initially raised: the proposed algorithm finds solutions in less or almost the same time in wide grids, while executing traditional A * sequentially takes less time small grids or those with too many obstacles. This makes sense for the following reason, the key to our algorithm is to drop more than one agent in each iteration, and that is not prone to happening on a small grid or with too many obstacles due to collisions. On the other hand, what saves time in my proposal is to route multiple agents in the same iteration, since their planning is done concurrently, and this is more likely to happen in sparse grids. A conclusion will be provided in section VI, for now, graphs are shown comparing the performance of the two implementations on different grid size.

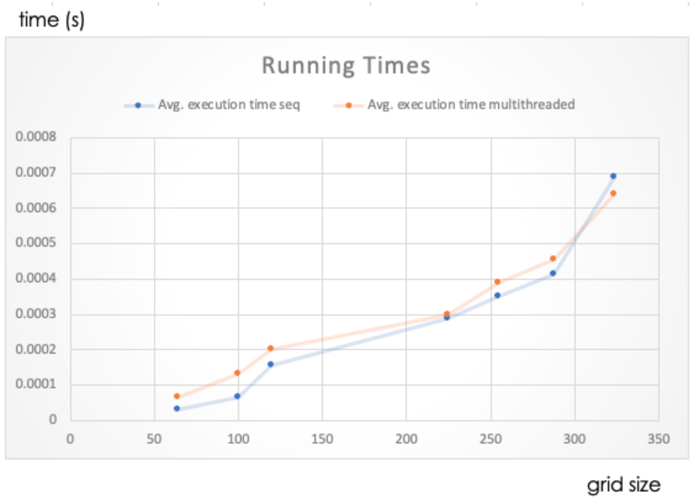


Figure 11: Comparing execution times

As it can be seen in the above graph, the execution times are quite similar but there are two characteristics that are worth highlighting: the first one is that in small grids the sequential algorithm behaves better, while in larger grids my proposal performs equally or slightly better, and the second characteristic and equally important is that the new algorithmic proposal did not show an exponential growth in time, so it can be explored with this assumption, that it is not significantly more expensive than the sequential version.

Finally, it is to say the results obtained cannot be considered as definitive for two main reasons: the first is that this will not be the definitive version of the algorithm, it just served as a proof of concept to reach the conclusion of whether it is appropriate to move forward with this proposition or not, also, it is expected to add many more search agents and then evaluate execution times again. Large grids, with dimensions above a thousand units, will deliver more insight into time complexity, but for time reasons it was only tested in small grids, this being another of the tasks to be carried out in next steps.

VI. Conclusions

People might be used to think that simply by using threads the overall computation time of X program will be reduced, but not always, there is a lot behind the scenes, and things like thread creation and message passing between threads is expensive. The objective of this paper was not to reduce the time of some algorithm, simply because that algorithm did not exist; instead, its goal was come up with a new algorithm that could route multiple tailed agents in a way at least a bit smarter than doing it sequentially, and that is could scale to multiple agents, which boils down to a proof of concept. Analyzing Figure 11, the new multithreaded algorithm does not present an exponential increase in the computation time for the solution, which makes it feasible to continue with its scaling and adaptation; being able to conclude that the initial objective was met.

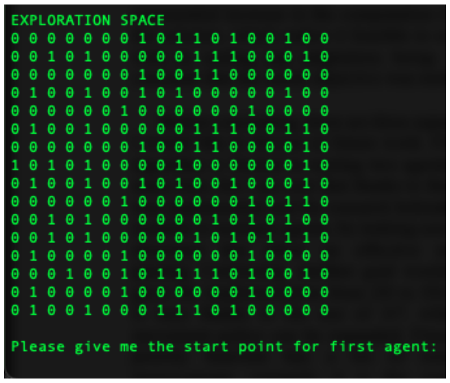
Before concluding, there are three aspects that I would like to discuss as future work. First, the algorithm is limited to routing two agents at the same time, but I am sure that thanks to the way it was implemented and the research behind it, it is scalable to multiple agents, by making use of data structures that allow an effective message passing. Continuing, another goal would be to grow the implementation from 2D to 3D, which is easy thanks to the use of A*, where the movement policy can be expanded. Finally, the priority function has a lot of room for improvement, currently it is the user who determines the priority, but in the future, heuristics could be explored to prevent future collisions e.g., if the longest agents were routed first, the probability of collisions between agents decreases in future iterations, since shorter routes collide less.

A real-world problem, that for several months of research I found no answer to, is being addressed. Due to confidentiality reasons, I cannot share the specific intended application for this algorithm, but the impact it can have is enormous. Processes of hours have been reduced to minutes with the implementation of routing one agent at a time, if we can route multiple agents at some point, this will increase the popularity of the current tool and time/cost savings will increase. The work developed here will have an immediate follow-up and serves as the first step for something greater; the group in charge of approving next steps regarding my work has already given green light on taking this algorithm as a basis and work on it.

VII. Set up instructions

Setting up the project is quite easy, just follow the next steps:

1. Clone the repository from github.com/alegleason/MAPFAstar.
2. If you have XCode installed, you can use it to open it as a Command Line Application, otherwise, just navigate to the route MAPFAstar/PBNA*/PBNA/.
3. On that route, you are going to open a terminal and run the command `g++ -std=c++11 main.cpp -lpthread -o` *program*, where *program* is the name of the executable file.
4. Now, on that same terminal, just run the command `./program` and you will be prompted with the next screen.



```
EXPLORATION SPACE
0 0 0 0 0 0 0 1 0 1 1 0 1 0 0 1 0 0
0 0 1 0 1 0 0 0 0 0 1 1 1 0 0 0 1 0
0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0
0 1 0 0 1 0 0 1 0 1 0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0
0 1 0 0 1 0 0 0 0 0 1 1 1 0 0 1 1 0
0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 1 0
1 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0
0 1 0 0 1 0 0 1 0 1 0 0 1 0 0 0 1 0
0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1 0
0 0 1 0 1 0 0 0 0 0 0 1 0 1 0 1 0 0
0 0 1 0 1 0 0 0 0 0 1 0 1 0 1 1 1 0
0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0
0 0 0 1 0 0 1 0 1 1 1 1 0 1 0 0 1 0
0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0
0 1 0 0 1 0 0 0 1 1 1 0 1 0 0 0 0 0

Please give me the start point for first agent:
```

Figure 12: Main program window

5. In that screen, you will enter the following information from the two agents: start and end coordinate, priority and an identifier.
6. The program will search for a solution and display it, if exists.
7. If you want to modify the grid, you can do it by opening the file main.cpp and modifying the *explorationSpace*

variable, as well as the ROW and COL constants, which are at the top of the file.

VIII. References

[1] Felner, A. Et all. 'Search-Based Optimal Solvers for the Multi-Agent Pathfinding Problem: Summary and Challenges', *Tenth International Symposium on Combinatorial Search (SoCS 2017)*, 2017. Link.

[2] La Valle, S. Jingjin Y. 'Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs∗', *s.f* Link.

[3] Velgapudi, P. Sycara, K. Scerri, P. 'Search-Based Optimal Solvers for the Multi-Agent Pathfinding Problem: Summary and Challenges', *s.f* Link.

[4] Atzmon, D. Diei, A. Rave, D. 'Multi-Train Path Finding', *Twelfth International Symposium on Combinatorial Search (SoCS 2019)*, 2019. Link.

[5] Sharon, G. Stern, R. Felner, A. Sturtevant, N. 'Conflict-based search for optimal multi-agent pathfinding', *Artificial Intelligence, 2015.* Link.

[6] Wilde, B. Mors, A. Witteveen, C. 'Push and Rotate: a Complete Multi-agent Pathfinding Algorithm', *Journal of Artificial Intelligence, 2014.* Link.

[7] Velgapudi, P. Sycara, K. Scerri, P. 'Decentralized prioritized planning in large multirobot teams', *s.f* Link.

[8] Abiy, T., 2020. *A* Search | Brilliant Math & Science Wiki*. [online] Brilliant.org. Available at: <https://brilliant.org/wiki/a-star-search/> [Accessed 24 November 2020].

[9] Charlesworth, O., 2020. *Are C++ Std::Map<String,String> Ordered?*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/11274978/are-c-stdmapstring-string-ordered#:~:text=Yes%2C%20a%20std%3A%3Amap,order%20that%20you%20inserted%20elements.> [Accessed 24 November 2020].

[10] Black, P., 2020. *Backtracking*. [online] Xlinux.nist.gov. Available at: <https://xlinux.nist.gov/dads/HTML/backtrack.html> [Accessed 24 November 2020].

[11] Patel, A., 2020. *Heuristics*. [online] Theory.stanford.edu. Available at: <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html> [Accessed 24 November 2020].

[12] Wang, K., 2020. Compare A* With Dijkstra Algorithm. [online] Plainaslife.blogspot.com. Available at: <http://plainaslife.blogspot.com/2015/02/compare-with-dijkstra-algorithm.html> [Accessed 24 November 2020].