Military Unit Path Finding Problem

DANIEL MUNDELL, University of KwaZulu-Natal, South Africa

The Military Unit Path Finding Problem is concerned with finding the lowest cost safe path across a potentially changing or hostile environment. When previously unknown hostile units are detected by the military unit during path traversal, the path finding algorithms must efficiently plan the new lowest cost safe path. This paper developed a web-based hexagonal grid simulation tool to inspect, analyze and compare the behaviour and performance of various static and dynamic path finding algorithms, including Dijkstra's algorithm, the A* search, Focused D*, and D* Lite. As expected, the dynamic algorithms had very fast replanning times, making them ideal for situations where multiple replanning processes are expected. However, a specific type of scenario was identified in which the static A* search algorithm was able to consistently outperform the dynamic algorithms. A* was also able to consistently find the initial solution path in the smallest amount of time, due to its guiding heuristic and simple implementation. Unfortunately, the D* Lite algorithm implementation did not work in all circumstances, potentially invalidating its outstanding performance in most testing scenarios.

CCS Concepts: • **Computing methodologies** → **Artificial Intelligence**; *Search methodologies*; • **Mathematics of computing** → Mathematical software.

Additional Key Words and Phrases: dynamic path finding, hexagonal grid, military simulation tool

1 INTRODUCTION AND BACKGROUND

The problem of path finding has received considerable attention over the years, due to the large range of applications which benefit from path finding algorithms that compute faster and are less memory intensive. Some uses of path finding are found in robotics [11], video games [1], GPS systems [25], layout design [27], and crowd simulation [10]. While general path finding deals with finding the lowest cost path between two locations, the Military Unit Path Finding Problem (MUPFP) introduces an additional objective of avoiding potential dangers to keep the military unit save.

Lives are on the line when a military unit navigates a hostile environment. While determining the shortest or fastest path is important, the safety of the military unit is paramount. Avoiding hostile units is an easy task if the environment and hostile units' positions are known ahead of time, however it becomes more difficult when there is imperfect information to start with, and hostile units are only detected after the military unit begins moving. Any software used to provide a military unit with navigation instruction should therefore be capable of providing real-time updates to the planned path to stay safe by avoiding newly detected hostile units. This paper presents a web-based simulation tool to assist in military unit navigation in dynamic environments. The simulation tool allows a user to create different map configurations, with various options including multiple grid sizes, obstacles, terrain types and unknown hostile units, to match real-life or fabricated environments. The tool includes two static algorithms, namely Dijkstra's algorithm and the A* search algorithm, and two dynamic algorithms, namely Focused D* and D* Lite. Any of these four path finding algorithms can be executed on a map configuration to test their performance and visualize their behaviour. The following performance metrics are recorded for comparison: overall time (OT), initial searching time (IST), updating time (UT), number of nodes expanded (NE), and path cost (PC).

The included path finding algorithms were executed over a range of different map configurations, and their performance assessed and compared to provide insights into the best algorithms to use for different types of scenarios. The paper questions whether dynamic algorithms always provide the best performance, or if there are some map configurations where repeated use of a static algorithm results in equal or superior performance.

In section 2, a literature review is provided, showing the origin and progression of various static and dynamic path finding algorithms, as well as some previous comparisons conducted between them. Some other methods for solving the MUPFP are also discussed. Section 3 describes how the problem was approached, discusses the four algorithms included, and explains the choices made for creating the different types of map configurations used for testing. Section 4 provides and analyses the results obtained, while discussing the problems encountered. In section 5, a final conclusion is drawn, critiquing the work done and providing direction for future work.

2 LITERATURE REVIEW

Research has been conducted into various path finding search space representations, including Voronoi diagrams [4], visibility graphs [6], and navigation meshes [8, 20]. However, the most simple and therefore most commonly used are regular grids [2, 3]. Various different grid representations are compared in [31], where it is concluded that the best grid-based topological representation of the path finding search space is provided by hexagonal grids.

Dijkstra's algorithm was proposed in 1959 [9] as a technique capable of finding the shortest path between two nodes on a directed graph with non-negative weights. It was able to considerably reduce the amount of work required to calculate a shortest path than the other methods used at the time, as well as lowering the memory requirements by only storing the nodes which had been evaluated or which were expected to be evaluated next. While even today it is one of the most widely recognised graph traversal algorithms, there are many newer algorithms which provide superior performance with lower memory requirements.

The A* search algorithm, proposed in 1968 [12], with corrections in 1972 [13], builds on Dijkstra's algorithm by incorporating an evaluation function and a heuristic function which provide cost estimates to narrow its search and guide it towards the goal node. Using these cost estimates to order the nodes in the expansion queue allows the algorithm to require far fewer node expansions than previous path finding techniques. The papers prove that A* is admissible and that an optimal path is guaranteed if a suitable evaluation function is chosen. However, there is no functionality built into A* to enable it to efficiently deal with real-time changes to the environment. If changes to the environment are discovered, the algorithm discards its prior knowledge and recalculates the new optimal path from the current node to the goal node.

A dynamic variant of A* proposed in 1994 [29], was named D* because of its resemblance to A* while allowing for dynamic environment changes during path traversal. D* attempts to improve on the performance of A* in real-time replanning scenarios by incrementally repairing the path between the current and goal nodes when changes to the environment are discovered. An extension of D*, called Focused D*, was later proposed in 1995 [30], in order to reduce the overall computational time required by focusing on repairing the nodes most likely to be part of the optimal path based on a suitable heuristic function. This allows Focused D* to outperform the original D* based on consumption time, as well as having a lower memory requirement, the benefits of which are realized when dealing with larger map configurations. Unfortunately, D* and Focused D* are commonly thought to be very complex and hard to implement.

Lifelong Planning A* (LPA*) is an incremental version of A* proposed in 2002 [16]. It uses the same actual cost and heuristic functions from A*, but incorporates one-step lookahead values to potentially improve the cost estimate for the start distances. While LPA* was able to greatly outperform the breadth-first search, DynamicSWSF-FP and A* in terms of total number of vertex expansions, vertex accesses and heap percolates, it always recalculates the shortest path from the start node to the goal node. As such, it is not designed for use with a robot or military unit that is already traversing the shortest path when a change to the environment is discovered, as seen in D* and Focused D*.

D* Lite is an incremental heuristic search algorithm proposed in 2002 [15] as a modification of the LPA* algorithm, but with the same behaviour as the Focused D* algorithms, i.e. when discovering environment changes, it updates the shortest path between the current node and the goal node, rather than always from the start node as in LPA*. It has a simpler program flow, due to requiring only one datapoint for priority comparison, as opposed to the complex nested-if statements used by D*, making it simpler to understand and easier to implement. The paper provides proof that D* Lite always performs at least as fast as D*. Due to its grounding in LPA* and similarity to A*, it provides a strong algorithmic foundation for future dynamic path planning research.

In a survey on widespread path planning algorithms, [28] categorized and discussed various algorithms, including D*, Focused D*, LPA* and D* Lite, which formed the "classical dynamic" A* variants category. The other A* variant categories included any-angle movement, moving target, and sub optimal. The article also discusses the various environment modeling options, including regular and irregular grids, navigation meshes, visibility graphs and Voronoi diagrams.

The use of path finding algorithms for games is explored in 2011 [7], where a number of different A*-based algorithms are reviewed. There is also discussion around the available A* optimization techniques which include search space representations, heuristic functions, and data structures. The paper concludes that determining how to use path finding algorithms to solve tricky problems is the most important part of game artificial intelligence. While good analysis is made, very little original experimentation was performed on the topics discussed.

The static MUPFP was formulated as a constraint-based problem in 2010 [18], with the authors concluding that the ease of modelling new environments and flexibility to add additional constraints was the main strength of their approach. Three custom path finding algorithms were implemented and tested, including constraint-based versions of A*, branch and bound, and hill-climbing algorithms. The focus was then targeted towards the dynamic MUPFP in 2012 [17], wherein the D* algorithm was used to solve a similar problem as in this paper. And finally in 2013 [19], a heuristic function similar to that of Focused D* was introduced to their original D* implementation to focus the algorithm in the direction of the goal, thereby improving its performance.

Various researchers have also approached the problem of robot path planning using population based optimization techniques, as shown in 2010 [5]. The ant colony optimization (ACO) technique is used to plan paths for a robot through an environment which is modified after convergence is found. The technique is successfully demonstrated, but its performance is not compared to any other known techniques.

A Multi-Objective Ant Colony Optimization (MOACO) was proposed in 2006 [22] to solve the MUPFP. Initially it used a square grid and provided good solution paths faster than an expert human could. The algorithm was extended in 2007 [24] with a hexagonal grid and other improvements, and then compared with various other ACO systems in 2007 [23]. It concluded that instead of focusing on one of the objectives (speed or safety), the best performance came from approaches which maintained a good balance between the two. However, only two map configurations were considered, with one or zero hostile units, and with no unknown information or dynamic environments. A final paper in 2013 [21] tests a variety of MOACO algorithms in a larger range of more realistic scenarios, with very good military behaviour being produced throughout.

While researchers are quick to promote the performance of their new algorithms, none of the listed papers investigated whether there are some map configurations in which simple static algorithms are able to consistently outperform dynamic ones, even for environments in which changes occur and replanning is necessary.

3 METHODOLOGY

3.1 Simulation Tool

The simulation tool was developed in Visual Studio Code 1.52.1 with the JavaScript programming language to allow for execution in a web browser with no need for prior installation and setup. As grids are the most common search space representation for learning about path finding algorithms, and due to the superior representation of a hexagonal grid over other grid shapes shown in [31], the decision was made to start with a hexagonal grid in the first version of the simulation tool and move on to a more complex representation in future work. A prebuilt hexagonal grid JavaScript library called "Honeycomb" [14] was used as a basis for the simulation tool. Honeycomb allows for custom hexagonal grids to be created and manipulated, has both Cartesian and cube coordinate systems available (which is important for the heuristics used in path finding algorithms), and has a large range of useful functions.

3.1.1 Functionality. The tool allows the user to translate an environment into a hexagonal grid representation by selecting various node types, including the start and goal nodes, obstacles, different terrain types, and known and unknown hostile units, and applying them to nodes in the grid. Obstacles can also be created in random nodes with a density of 30%, using the "Random Walls" button. Existing map configurations, as well as different grid sizes, can be loaded with the provided on-screen options. The user can then select one of the four implemented path finding algorithms, and start the search by selecting the "Search!" button. The search will then be animated in the grid, showing how the selected path finding algorithm incrementally expands more nodes until an optimal path is found. The military unit attempts to traverse the path until any hostile units are found, where it uses this new information to update the optimal path and continue traversing. This process repeats until the goal node is reached or it has been determined that no path is available. There is also an option to turn off animations, in which case only the final path will be shown after the path finding process takes place. Once a search is complete, the recorded performance metrics discussed earlier are displayed. To edit the map configuration or select a different path finding algorithm, the "Clear Search" button must be used to first remove the search information from the grid. The whole grid can be cleared using the "Clear Grid" button.

Two sample hexagonal grids from the implemented simulation tool are shown in figure 1. The yellow nodes show the start node and the path travelled by the military unit as it navigates the environment, avoiding the black obstacle nodes and the red hostile unit to reach the green goal node. In the case of figure 1a, the hostile was previously unknown, but when the military unit discovered it, a new path was calculated around it. The light grey nodes have been evaluated, while the dark grey nodes are those which are in the queue to be evaluated next, according to the priority ranking of the used algorithm. The white nodes are empty and have a travel cost of one, while blue nodes shown in figure 1b represent water and have a travel cost of 4, and orange nodes (not pictured here) represent sand and have travel costs of 2.

3.2 Path Finding Algorithms

In total, four path finding algorithms were implemented and tested: Dijkstra's algorithm, A*, Focused D*, and D* Lite. While some of these algorithms are intended for dynamic scenarios, others needed slight modification to repeat their search process when changes to the environment were detected.

3.2.1 Dijkstra. Dijkstra's algorithm starts at the start node and repeatedly expands the lowest cost node which has not yet been expanded until it reaches the goal node. It does not make use of any sort of heuristic function to guide the search. This makes it the simplest of the included algorithms to implement, but in many cases a large number of nodes is expanded unnecessarily due to the

(a) Example of Completed Path (b) Forced Backtrack Map Configuration

Fig. 1. Sample Hexagonal Grids from Simulation Tool

high branching factor. While Dijkstra's algorithm is unlikely to be the best performer in any map configuration, it was included as a baseline algorithm to show the worst-case scenario performance. The implemented Dijkstra's algorithm, with modifications to work in dynamic environments, is identical to the A* algorithm discussed next, except that for Dijkstra's algorithm every heuristic value h is always equal to 0.

3.2.2 A.* The A* search algorithm repeatedly expands the highest priority discovered node according to the evaluation function:

$$f(n) = g(n) + h(n), \tag{1}$$

where g is the actual cost from the start node to the current node, and h is the actual cost from the current node to the goal node. The evaluation function f is defined in such a way that the node with the smallest value of f is considered the best option to be expanded next. In order to use the values of f to determine which nodes should be expanded next, nodes must be evaluated before the actual costs $g(n)$ and $h(n)$ of a node are known. This requires an estimation evaluation function defined as:

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n), \tag{2}$$

where $\hat{g}$ is the current best cost from the start node to the current node, and $\hat{h}$ is a heuristic function which estimates the cost from the current node to the goal node. Careful consideration is required when choosing the heuristic function, as they have mixed results if applied incorrectly or to the wrong search space representations. The Euclidean distance and the Manhattan distance are common heuristic functions used when the search space is represented in 4- and 8-connected square grids, respectively. However, these do not work well with hexagonal grid search space representations, as costs are not accurately estimated. Because of this, the admissible heuristic function $\hat{h}$ for the implemented A* algorithm made use of the cube coordinate system with the maximum absolute difference between one of the three coordinate axis values [26], as defined in the following equation:

$$\hat{h}(n) = max(|a.x - b.x|, |a.y - b.y|, |a.z - b.z|), \tag{3}$$

where a and b are the two nodes, and x, y, and z are the three cube axis. Using these estimation functions, A* is able to order the priority of discovered nodes in a smarter way and therefore requires fewer node expansions than Dijkstra's algorithm to reach the goal node.

There are also some useful properties of A* mathematically proved by the original author [12]:

(1) Guaranteed to find a path if there are no negative node-to-node travel costs and a valid path exists

(2) Optimal if the heuristic is admissible, i.e. if it does not overestimate the cost between two nodes
(3) Makes the most efficient use of the heuristic

The implementation of A* was modified slightly to work with dynamic environment changes. When a change to the environment is detected, the algorithm clears all known node information and starts a new search from the current node to the same goal node. However, in this new search, the more accurate updated representation of the environment is used.

3.2.3 Focused D.* The basic D* algorithm starts at the goal node and searches for an optimal path back to the military unit's current node, which is the start node during the first iteration. Nodes from the OPEN set are iteratively evaluated and moved to the CLOSED set, and their neighbours added to the OPEN set, just as in Dijkstra's algorithm. Each node in the CLOSED set has a backpointer b which helps determine the optimal path from any node back to the goal node. While with Dijkstra's algorithm and A*, estimates are required for the cost from the current node to the goal node, in the case of D* the actual cost is known for all expanded nodes. Once the current node is found, the backpointers can then be followed to show the optimal path.

D* builds on Dijkstra's algorithm and A* by introducing the concept of RAISE and LOWER states, over and above the OPEN and CLOSED states used by the previous algorithms. If an obstacle node is discovered while the military unit is moving along the planned path, the affected nodes are moved to the OPEN set, but are also marked as being in a RAISE state. Each node in the RAISED state checks with its neighbouring nodes whether its cost can be lowered. Each affected node is then evaluated with the RAISED state being passed to all nodes which backpointers point towards it, which forms a wave of RAISE state nodes until nodes which can be reduced are found, and they pass on the LOWER state preventing further nodes from being processed unnecessarily as a new valid path of backpointers has been found to the goal.

The Focused D* algorithm improves on D* in a similar way that A* improves on Dijkstra's algorithm, i.e. it uses a suitable heuristic function to guide the search and the replanning process when changes to the environment are detected. This allows the algorithm to maintain the majority of its calculated values when a change to the environment is discovered after the military unit begins moving, as fewer nodes will become inconsistent and need to be reprocessed compared to the basic D* algorithm and the previous static algorithms discussed.

Some minor changes were made to the Focused D* pseudocode provided in [30], due to the use of the JavaScript programming language, and the corresponding animation requirements. In essence, the while loops must be replaced with iterative function calls, to allow the animation to take place after each function completes. The same cube-based heuristic function shown in the A* section was used here too. The implementation of Focused D* was notably far more complex than any of the other implemented path finding algorithms.

3.2.4 D Lite.* The D* Lite algorithm was created in [15] to challenge the performance of the D* and Focused D* algorithms. It is a modification of the LPA* algorithm, making it simpler to understand and easier to implement than the other D* variants.

LPA* is an incremental version of A* which uses the same heuristics to estimate the cost between a node and the goal node, but incorporates one-step lookahead values to potentially improve the cost estimate for the start distances. It retains its knowledge and can update the value of nodes when it discovers changes to the environment, but always calculates the smallest cost between the start node and the goal node. Because of this, it cannot be directly applied to the MUPFP which needs to find the optimal path from the military unit's current position to the goal as the military unit moves through the environment.

D* Lite modifies the LPA* algorithm by searching from the goal node to the military unit's current node, which makes it suitable for the MUPFP. During the initial search, its behaviour is the same as A*, except for the opposite search direction. When discovering a change in the search space, D* Lite updates only the goal distances which are relevant to recalculate the shortest path between the military unit's current node and the goal node.

LPA* and D* Lite implementation pseudocode is available in [15]. Some minor changes were made to allow for JavaScript animations as described in the Focused D* section.

3.3 Testing Map Configurations

The testing map configurations were designed to isolate certain scenarios and highlight the strengths and weaknesses of the implemented algorithms. Low obstacle density maps were created to highlight the heuristic guided algorithms, as the branching factor of nodes would remain high. High density maps were created to investigate whether Dijkstra's algorithm (the only implemented algorithm which doesn't use a heuristic) was able to produce competitive performance when the branching factor of nodes was considerably reduced. Maps had one or three unknown hostile units in direct obstruction of the optimal path. This is expected to show the strengths of the dynamic algorithms when dealing with changes in the environment and the replanning process. A special "forced backtrack" map configuration was created, as shown previously in figure 1b. This map configuration mimics a real-life situation such as a bridge across a river wherein the optimal lowest-cost path is across the bridge. However, an unknown hostile unit is blocking the bridge, and thus requires the military unit to backtrack and find a new path to the goal. This test was used to investigate what algorithms would perform well when a very large change to the optimal path was required, as all previous test maps only require relatively small changes. Additionally, each type of scenario discussed above will have three variants of grid size (10x10, 30x30, and 50x50). It is expected that all algorithms will perform similarly at small grid sizes, but the gap in performance will increase quickly as the grid size increases. Altogether, this creates a total of 15 map configurations for the algorithms to be tested on.

All tests were conducted with the animation setting turned off. The animations were however used to determine the behaviour of the algorithms, and to provide insight into why they performed how they did.

4 RESULTS AND DISCUSSION

By looking at the results shown in table 1 - 5 and figure 2 - 6, it is clear that in the smaller grid sizes the difference between algorithms is minimal. Any of the implemented algorithms could be used on a 10x10 grid without any human-noticeably slow performance. When the grid size increases to 30x30 and then to 50x50, the strengths of the dynamic algorithms start to show more clearly. For even larger grid sizes, which would likely be used in the real world, it is expected that this trend would continue, with the dynamic algorithms' strength of being able to reuse calculated data being more beneficial the greater the grid size. Additionally, as the number of hostile units increased, thus leading to an equally large number of replanning processes required, the dynamic algorithms again showed off their replanning strengths, as expected.

After running all 15 test map configurations, it is clear that Dijkstra's algorithm provides the worst performance. With regard to the initial search times, Dijkstra's algorithm was reasonably competitive (but still the worst performer) in the high obstacle density map configurations shown in figure 4 and 5. In these configurations, the potential branching factor of nodes is limited due to the large number of obstacles, and this lessens Dijkstra's weakness of having no heuristic function to guide it. However, this is the algorithm's only positive. As the grid size increased, Dijkstra's performance in all other performance metrics was exponentially worse than the other algorithms,

and as such the vertical axis on many figures was limited to focus on the difference between the other algorithms. Based on these results, Dijkstra's algorithm should only be considered in small high density map configurations where it is known that no changes to the environment will be found. Even under these circumstances, it is highly unlikely that it will outperform the other algorithms.

A* provided the lowest initial search times throughout the test map configurations, due to its efficient use of the heuristic function and very simple overall implementation. However, even though it expanded a similar number of nodes, it struggled to compete with the fast replanning performance of the dynamic algorithms, especially as the grid size and the number of unknown hostiles increased. Figure 6 shows A*'s excellent all-round performance when dealing with a forced backtrack situation. A* was able to find the initial optimal path the fastest, and then when the unknown hostile was detected in the trap, it was able to escape in the shortest time and find a new optimal path around while expanding the fewest overall nodes. Based on this, A* would be a good choice of algorithm in scenarios where a large backtrack may be required, or in which limited real-time environment changes are expected.

Focused D* produced mixed results for initial search times in the larger grid sizes. This could be due to the randomness of maps which may have caused an uneven distribution of complexity, combined with the backwards direction of the search compared to A*. It managed to find an optimal path faster than D* Lite in all larger tests, but could not compete with the inital searching speed of the simple A* implementation. The updating times of Focused D* were better than A* and Dijkstra's in the normal tests, but were still roughly double that of D* Lite, with this difference increasing as the grid size and number of unknown hostiles increased. This must be due to the high complexity level of the algorithm implementation, as in many cases it expanded approximately the same number of nodes as D* Lite. There is notably worse performance of Focused D* in the 50x50 grid size of figure 2, with it performing more in line with A* than D* Lite, except for the initial search time. On inspection of this map configuration, it was found that this bad performance was due to a unique portion of the grid surrounding the unknown hostile unit which caused a large number of unnecessary nodes to be prioritized highly, which in turn increased the updating time taken.

The initial search times of D* Lite were not impressive, often being closer to Dijkstra's than the other algorithms. This is surprising since its performance was expected to be similar to that of A* in the initial searching phase, but might have be caused by the backwards searching direction required, as discussed previously. On the other hand, D* Lite provided exceptionally low updating times compared to the other algorithms, and routinely expanded the fewest nodes. This meant that, despite its high initial search times, it completed its path finding and updating process in the lowest overall time in almost all tests. However, certain problems with the D* Lite implementation could not be solved before this paper was required, and as such it was not able to navigate out of forced backtrack situations. This is an important note, as whatever problems the implementation is facing could be causing the impressively low updating times and these results could therefore be invalidated. The performance of D* Lite was expected to be good, based on other research shown in the literature review, and so it is unfortunate that the problems our D* Lite implementation faced could not be solved in time.

When a change to the environment is detected that results in only a small change to the optimal path, the dynamic algorithms excel. Focused D* and D* Lite are able to seamlessly replan the new path, making use of large quantities of previously calculated node costs and other information. The static algorithms require a full replanning from the current node to the goal node, while not retaining any previously calculated costs, and so they performed more poorly both in terms of updating time and number of nodes expanded. However, when there was a very large change to

the optimal path as demonstrated in the forced backtrack tests, A* provided the best performance in all metrics.

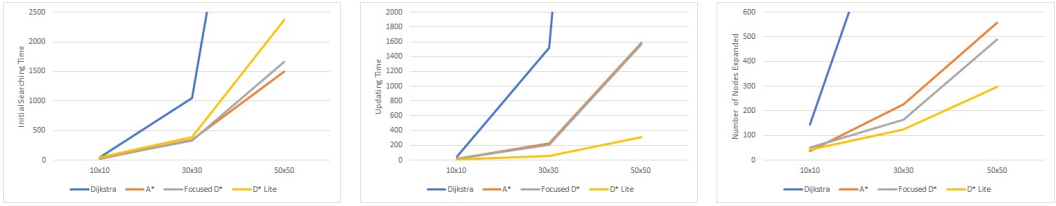

Fig. 2. IST, UT, and NE for Low Obstacle Density with 1 Unknown Hostile

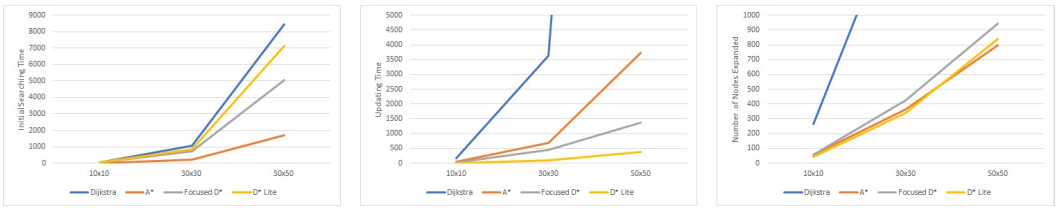

Fig. 3. IST, UT, and NE for Low Obstacle Density with 3 Unknown Hostiles

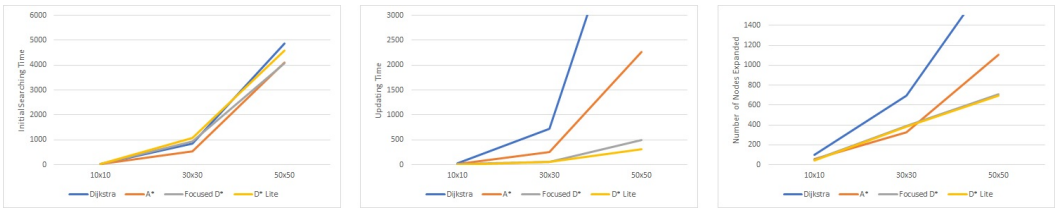

Fig. 4. IST, UT, and NE for High Obstacle Density with 1 Unknown Hostile

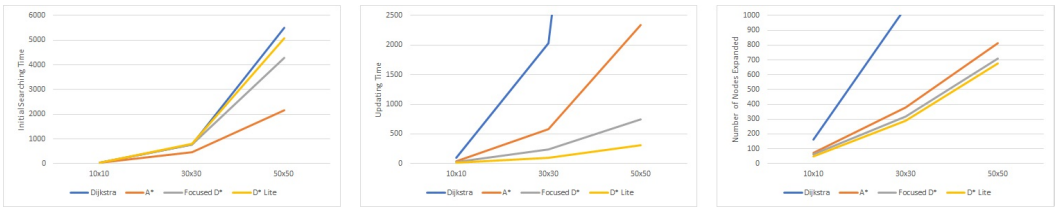

Fig. 5. IST, UT, and NE for High Obstacle Density with 3 Unknown Hostiles

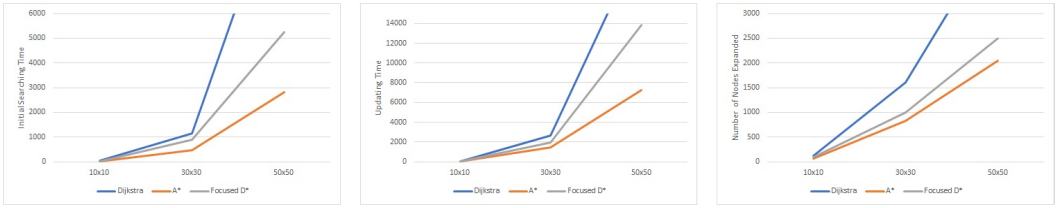

Fig. 6. IST, UT, and NE for Forced Backtrack

Table 1. Low Obstacle Density with 1 Hostile

	Dijkstra			A*			Focused D*			D* Lite		
	10x10	30x30	50x50	10x10	30x30	50x50	10x10	30x30	50x50	10x10	30x30	50x50
OT	98	2572	25408	43	567	3081	60	539	3227	61	449	2689
IST	46	1048	10051	20	344	1496	41	331	1660	53	394	2378
UT	52	1524	15357	23	223	1585	19	208	1567	8	55	311
NE	144	1238	3599	37	227	559	51	164	490	42	125	297
PC	15	45	76	15	45	76	15	45	76	16	46	78

Table 2. Low Obstacle Density with 3 Hostiles

	Dijkstra			A*			Focused D*			D* Lite		
	10x10	30x30	50x50	10x10	30x30	50x50	10x10	30x30	50x50	10x10	30x30	50x50
OT	202	4698	52331	68	923	5428	70	1165	6431	51	960	7512
IST	46	1066	8460	16	226	1695	37	722	5071	43	859	7132
UT	156	3632	43871	52	697	3733	33	443	1360	8	101	380
NE	263	1837	5429	59	363	796	56	425	943	45	339	841
PC	17	56	90	17	56	90	17	56	91	17	59	101

Table 3. High Obstacle Density with 1 Hostile

	Dijkstra			A*			Focused D*			D* Lite		
	10x10	30x30	50x50	10x10	30x30	50x50	10x10	30x30	50x50	10x10	30x30	50x50
OT	75	1579	11141	52	786	6363	53	987	4556	49	1137	4908
IST	40	861	4862	30	537	4100	39	924	4066	44	1086	4595
UT	35	718	6279	22	249	2263	14	63	490	5	51	313
NE	103	693	2003	60	325	1107	52	390	706	42	386	694
PC	18	68	102	18	68	102	18	68	102	18	68	102

5 CONCLUSIONS AND FUTURE WORK

A directly proportional relationship was found between the grid size, and the updating time performance difference between the static and dynamic algorithms. The larger grid sizes caused an exponentially larger slowdown in the updating time and nodes expanded for the static algorithms. While the dynamic algorithms experienced a much more linear increase, it was still large, giving a clear indication that a different search space representation is required for larger environments. While the hexagonal grid is able to capture real life movement more accurately than a square

Table 4. High Obstacle Density with 3 Hostiles

	Dijkstra			A*			Focused D*			D* Lite		
	10x10	30x30	50x50	10x10	30x30	50x50	10x10	30x30	50x50	10x10	30x30	50x50
OT	131	2797	20386	71	1050	4699	64	1006	5043	55	884	5393
IST	34	766	5489	30	474	2156	41	767	4292	45	790	5081
UT	97	2031	14897	41	576	2343	23	239	751	10	94	312
NE	164	1055	2862	74	381	813	62	319	708	48	287	675
PC	27	64	95	26	64	95	26	64	95	26	66	96

Table 5. Forced Backtrack

	Dijkstra			A*			Focused D*		
	10x10	30x30	50x50	10x10	30x30	50x50	10x10	30x30	50x50
OT	79	3807	34141	48	1917	10070	86	2846	19097
IST	30	1133	11788	10	455	2811	15	897	5244
UT	49	2674	22353	38	1462	7259	71	1949	13853
NE	124	1610	4658	59	828	2040	93	1001	2503
PC	24	138	145	24	138	145	24	138	145

grid, it was also determined through the extensive testing conducted for this paper that in many cases a more direct path could have been taken if the military unit was not limited to grid-based movement. One type of scenario was identified wherein the static A* search algorithm was able to consistently outperform the dynamic algorithms in all performance metrics, showing that under certain conditions this static algorithm still provides competitive performance due to its very simple implementation. While A* also provided the best all-round initial searching times, its static nature made the replanning process very slow, and as such it did not perform well in scenarios with many changes to the environment. Focused D* provided great performance in terms of updating time and nodes expanded, but struggled to compete with the simpler A* implementation for initial searching times. While D* Lite showed good promise when only small changes were required to the optimal path, an error-free implementation would be needed to make final conclusions between this algorithm and the others. Future work could include an error-free version of D* Lite to verify the results found here, as well as implementing more recent algorithms, albeit in a more efficient search space representation. It would also be interesting to investigate whether a "smart" algorithm could be developed, in which the search space is analyzed before hand, to mark potentially dangerous zones based on known hostile units and narrow paths which could be blocked easily. The performance of this could be compared to always taking the optimal path. To expand the usability of the simulation tool to a larger audience, improvements would be needed in terms of search space and memory usage optimization and potentially exploring alternative data structures, to ensure the software works to its full potential even on low memory devices.

REFERENCES

[1] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. 2015. A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. *International Journal of Computer Games Technology* 2015 (apr 2015), 1–11. http://dx.doi.org/10.1155/2015/736138

[2] Anton Andreychuk and Konstantin Yakovlev. 2017. Applying MAPP Algorithm for Cooperative Path Finding in Urban Environments. *Interactive Collaborative Robotics (ICR) Lecture Notes in Computer Science* 10459 (2017), 1–10.

https://doi.org/10.1007/978-3-319-66471-2_1

[3] Anton Andreychuk and Konstantin Yakovlev. 2018. Path Finding for the Coalition of Co-operative Agents Acting in the Environment with Destructible Obstacles. *Interactive Collaborative Robotics (ICR) Lecture Notes in Computer Science* 11097 (jul 2018), 13–22. https://doi.org/10.1007/978-3-319-99582-3_2

[4] Priyadarshi Bhattacharya and Marina L. Gavrilova. 2008. "Roadmap-Based Path Planning - Using Voronoi Diagram for a Clearance-Based Shortest Path. *IEEE Robotics & Automation Magazine* 15 (jun 2008), 58–66. https://doi.org/10.1109/MRA.2008.921540

[5] Michael Brand, Michael Masuda, Nicole Wehner, and Xiao-Hua Yu. 2010. Ant Colony Optimization Algorithm for Robot Path Planning. *International Conference On Computer Design And Appliations (ICCDA 2010)* 3 (2010), 436–440. https://doi.org/10.1109/ICCDA.2010.5541300

[6] Pei Cao, Zhaoyan Fan, Robert X. Gao, and J. Tang. 2018. "Design for Additive Manufacturing: Optimization of Piping Network in Compact System With Enhanced Path-Finding Approach. *Journal of Manufacturing Science and Engineering* 140, 8 (aug 2018), 58–66. https://doi.org/10.1115/1.4040320

[7] Xiao Cui and Hao Shi. 2011. A*-based Pathfinding in Modern Computer Games. *International Journal of Computer Science and Network Security (IJCSNS)* 11, 1 (Jan. 2011), 125–130. https://www.semanticscholar.org/paper/A*-based-Pathfinding-in-Modern-Computer-Games-Cui-Shi/c45b0fc413542d19256dd494129e43ab97e31ba5

[8] Xiao Cui and Hao Shi. 2012. An Overview of Pathfinding in Navigation Mesh. *International Journal of Computer Science and Network Security* 12, 12 (dec 2012), 48–51. http://paper.ijcsns.org/07_book/201212/20121208.pdf

[9] Edsger W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (Dec. 1959), 269–271. https://doi.org/10.1007/BF01386390

[10] Cherif Foudil, Djedi Noureddine, Cedric Sanza, and Yves Duthen. 2009. Path Finding and Collision Avoidance in Crowd Simulation. *Journal of Computing and Information Technology* 17 (jan 2009), 217–228. https://doi.org/10.2498/cit.1000873

[11] Enric Galceran and Marc Carreras. 2013. A survey on coverage path planning for robotics. *Robotics and Autonomous Systems* 61, 12 (sep 2013), 1258–1276. https://doi.org/10.1016/j.robot.2013.09.004

[12] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (July 1968), 100–107. https://doi.org/10.1109/TSSC.1968.300136

[13] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1972. Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *SIGART Bull.* 37, 1 (Dec. 1972), 28–29. https://doi.org/10.1145/1056777.1056779

[14] Abbe Keultjies. 2020. *Honeycomb - Another hex grid library made in JavaScript.* Retrieved November 2, 2020 from https://github.com/flauwekeul/honeycomb

[15] Sven Koenig and Maxim Likhachev. 2002. D* Lite. *Eighteenth national conference on Artificial intelligence* (July 2002), 476–483. https://dl.acm.org/doi/10.5555/777092.777167

[16] Sven Koenig and Maxim Likhachev. 2002. Incremental A*. In *Advances in Neural Information Processing Systems*, Thomas Dietterich, Suzanna Becker, and Zoubin Ghahramani (Eds.), Vol. 14. MIT Press, 1539–1546. https://proceedings.neurips.cc/paper/2001/file/a591024321c5e2bdbd23ed35f0574dde-Paper.pdf

[17] Louise Leenen, Hermanus le Roux, and Alexander Terlunen. 2012. A Constraint Programming Solution for the Military Unit Path Finding Problem. *Mobile Intelligent Autonomous Systems: Recent Advances* 1 (2012), 225–240. https://doi.org/10.1201/b12690-14

[18] Louise Leenen, Hermanus le Roux, and Johannes S Vorster. 2010. A Constraint-based Solver for the Military Unit Path Finding Problem. *Proceedings of the 2010 Spring Simulation Multiconference* (apr 2010), 1–8. https://doi.org/10.1145/1878537.1878564

[19] Louise Leenen and Alexander Terlunen. 2013. A Focussed Dynamic Path Finding Algorithm with Constraints. *International Conference on Adaptive Science and Technology* (nov 2013), 1–8. https://doi.org/10.1109/ICASTech.2013.6707501

[20] Matheus R.F. Mendonça, Heder S. Bernardino, and Raul F. Neto. 2015. Stealthy Path Planning Using Navigation Meshes. *2015 Brazilian Conference on Intelligent Systems (BRACIS)* (nov 2015), 31–36. https://doi.org/10.1109/BRACIS.2015.49

[21] Antonio Miguel Mora, Juan Julian Merelo, Pedro A. Castillo, and M.G. Arenas. 2013. hCHAC: A family of MOACO algorithms for the resolution of the bi-criteria military unit pathfinding problem. *Computers & Operations Research* 40, 6 (2013), 1524–1551. https://doi.org/10.1016/j.cor.2011.11.015

[22] Antonio Miguel Mora, Juan Julian Merelo, Cristian Millan, Juan Torrecillas, and Juan Luís Jiménez Laredo. 2006. CHAC. A MOACO Algorithm for Computation of Bi-Criteria Military Unit Path in the Battlefield. *Proceedings of the Workshop on Nature Inspired Cooperative Strategies for Optimization (NICSO'06)* (jun 2006), 85–96. https://arxiv.org/abs/cs/0610113v1

[23] Antonio Miguel Mora, Juan Julian Merelo, Cristian Millan, Juan Torrecillas, Juan Luís Jiménez Laredo, and Pedro A. Castillo. 2007. Balancing Safety and Speed in the Military Path Finding Problem: Analysis of Different ACO Algorithms. *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO'07)* (jul 2007), 2859–2864. https://dl.acm.org/doi/10.1145/1274000.1274032

[24] Antonio Miguel Mora, Juan Julian Merelo, Cristian Millan, Juan Torrecillas, Juan Luís Jiménez Laredo, and Pedro A. Castillo. 2007. Enhancing a MOACO for Solving the Bi-criteria Pathfinding Problem for a Military Unit in a Realistic Battlefield. *Applications of Evolutionary Computing, Evo Workshops 2007. Lecture Notes in Computer Science* (apr 2007), 712–721. https://doi.org/10.1007/978-3-540-71805-5_77

[25] Dustin Ostrowski, Iwona Pozniak-Koszalka, Leszek Koszalka, and Andrzej Kasprzak. 2015. Comparative Analysis of the Algorithms for Pathfinding in GPS Systems. *ICN 2015: The Fourteenth International Conference on Networks* 2015 (apr 2015), 102–108. http://www.academia.edu/download/37948903/icn_2015_full.pdf

[26] Amit Patel. 2020. *Hexagonal Grids - Red Blob Games.* Retrieved November 2, 2020 from https://www.redblobgames.com/grids/hexagons/

[27] Siyuan Song and Eric Marks. 2019. Construction Site Path Planning Optimization through BIM. *ASCE International Conference on Computing in Civil Engineering* (2019), 369–376. https://doi.org/10.1061/9780784482421.047

[28] Omar Souissi, Rabie Benatitallah, David Duvivier, and AbedlHakim Artiba. 2013. Path Planning: A 2013 Survey. *Proceedings of 2013 International Conference on Industrial Engineering and Systems Management (IESM)* (Oct. 2013), 1–8. https://ieeexplore.ieee.org/abstract/document/6761521

[29] Anthony Stentz. 1994. The D* Algorithm for Real-Time Planning of Optimal Traverses. *Technical Report CMU-RITR-94-37, Carnegie Mellon University Robotics Institute* (1994). https://doi.org/10.1007/BF01386390

[30] Anthony Stentz. 1995. The Focussed D* Algorithm for Real-Time Replanning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence* (Montreal, Quebec, Canada) *(IJCAI'95, Vol. 2)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1652–1659. https://dl.acm.org/doi/10.5555/1643031.1643113

[31] Peter Yap. 2002. Grid-Based Path-Finding. In *Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence* (Berlin, Heidelberg) *(AI '02, Vol. 2338)*. Springer-Verlag, Berlin, Heidelberg, 44–55. https://dl.acm.org/doi/10.5555/647463.726444