A Comparison of A* Search Variants

Nathan Comer
University of Minnesota
comer028@umn.edu
Computer Science 4511W

***Abstract*—A* is a widely used algorithm for solving sliding puzzles and path planning problems. This work experiments with various alternative forms of the A* algorithm to see if they may be better equipped to solve these problems. The results of these experiments show that variations of the A* algorithm such as Bidirectional A* and Weighted Bidirectional A* are more effective for solving path planning problems.**

I. Introduction

A* is often used to solve path planning problems, and other similar problems such as sliding puzzles. The experiments that follow will evaluate whether variations of A* may be better suited for solving these problems.

The first variant being examined is Bidirectional A*, which instead of evaluating the distance between the current state and the goal state, its heuristic evaluates the distance between the the search from the forward direction and the search from the backward direction. The second variant being examined is Weighted Bidirectional A* which functions similar to Bidirectional A* but uses multiple weighted heuristics instead of a single heuristic function.

A* is often used to solve path planning problems, so a more efficient alternative would be beneficial. The experiment results reveal that Bidirectional A* performs equivalently or better than A* with respect to run time and nodes generated. However, it does under perform when a near optimal solution is desired. Weighted Bidirectional A* is an excellent compromise between the two, providing greater efficiency than A*, with improved optimality over Bidirectional A*.

II. The Algorithms

Three search algorithms will be examined: Standard A*, Bidirectional A*, and Weighted Bidirectional A*. All three of these algorithms use evaluation functions, which consist of the distance required to get to the current state plus a heuristic function, which approximates the distance from the current state to the goal state. The heuristic used by each algorithm will be discussed below.

A. *Standard A**

Standard A* is a best-first search algorithm. It uses an evaluation function to traverse a graph of nodes. The evaluation function consists of $g(n)$ the cost to reach the current node, plus $h(n)$ the estimated cost of reaching the goal from the current node.

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

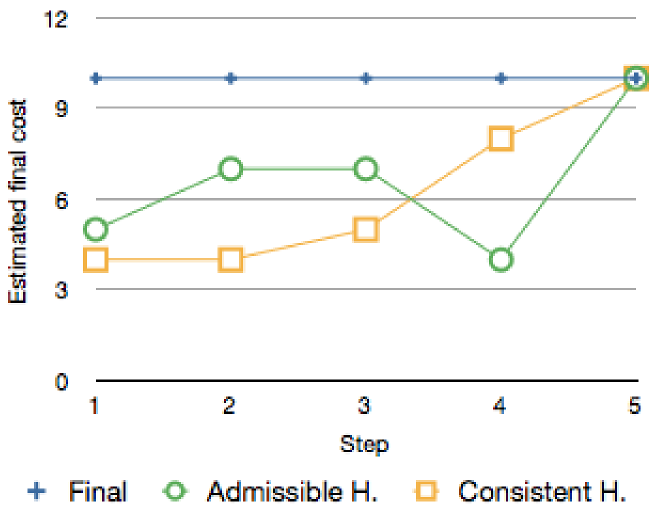


Fig. 1. A strong heuristic function is admissible and consistent

A strong heuristic has two characteristics, it is admissible and consistent. **Figure 1** demonstrates these characteristics. An admissible heuristic never over-estimates the cost of reaching the goal state, thus the heuristic is at most the lowest possible cost of reaching the goal state from the current state. A heuristic is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n': [2]

$$h(n) \leq c(n, a, n') + h(n') \tag{2}$$

The Manhattan distance heuristic implemented here is admissible, but it is not consistent. For some problems the state may need to move further from the goal state in order to correctly traverse the graph to the goal state. Manhattan distance is also the heuristic function used for Sliding Puzzle. The heuristic function calculates the sum of the distances, $d(c, g)$, each piece must travel to get from its current location, c, to its goal location, g.

$$h(n) = \sum_{c=1}^{s-1} d(c, g) \tag{3}$$

For the path planning experiments the heuristic function finds the Manhattan distance from the current coordinate to the goal coordinate.

Standard A* has a worst-case performance of:

$$O(b^d) \tag{4}$$

Where b is the breadth factor, and d is the depth required to find a solution. This is derived because at every depth, d, the algorithm will expand, b nodes until it finds a solution. For binary maps and sliding puzzle b equals three. Each node can be expanded in four directions with one of them being the direction the search just came from.

One of the negative characteristics of A* is that if a problem requires the algorithm to go against its heuristic to find the solution, A* becomes less efficient. This is why a consistent heuristic is vital to A*'s performance. When A* reaches a depth in the graph where it must go against its heuristic, A* will instead choose to expand other nodes. An example of this is in sliding puzzle when a piece needs to move away from its goal position for the puzzle to be solved, which goes against the Manhattan distance heuristic function. When this happens A* begins to function similar to an Iteratively Deepening Search algorithm. Where each iterative depth is the depth at which a state needs to go against the heuristic in order to reach the goal state. This often occurs in both sliding puzzle and path planning problems. The optimal way to prevent this would be to use a consistent heuristic; however, consistent heuristics for path planning and sliding puzzle are not easily developed. Another way to combat this is through the use of a Bidirectional Search Algorithm.

B. Bidirectional A*

Bidirectional A* adds the evaluation function of A* to a Bidirectional Search algorithm. It has two separate instances of A*, one that starts at the initial state and another that begins at the goal state. The two instances search attempting to meet each other. Once they meet, the solution has been found. Bidirectional A* also uses Manhattan distance as a heuristic function, it calculates the distance between the states of the two searches. For the sliding puzzle the heuristic calculates the sum of the Manhattan distances, $d(c, o)$, between a piece's location in the current search, c, and its location in the opposing search, o.

$$h(n) = \sum_{c=1}^{s-1} d(c, o) \tag{5}$$

As was discussed above, Bidirectional A* is less affected by a state needing to move away from the goal state because of it's heuristic. Being Bidirectional A*'s heuristic is calculated with respect to the opposing search, a nodes heuristic value can be less than the previous node's heuristic value, even if it moved further away from the goal state. For example, assume that in a sliding puzzles the forward directional search is moving a piece away from its goal location but the backward directional search just moved a piece closer to its intended location. In this case the forward directional search's heuristic value can decrease (or remain the same), even though it moving a piece further away from its goal location.

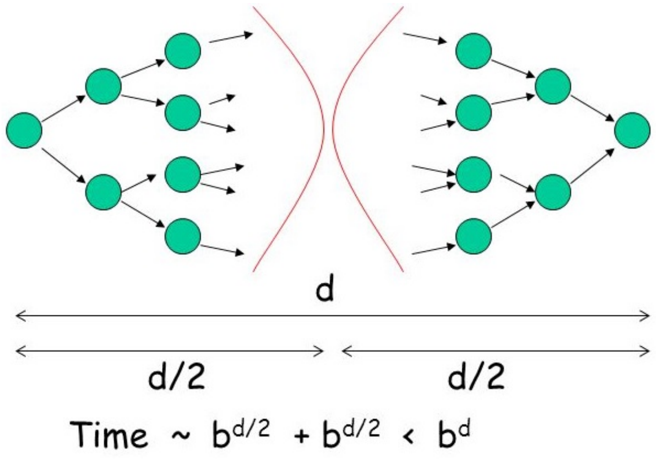


Fig. 2. Visual representation of Bidirectional A*

One key problem does arise with Bidirectional A*, its inability to efficiently update each nodes value after the value has been initially calculated. In Standard A*, a nodes value is always calculated with respect to the goal state, and the goal state never moves. In the case of Bidirectional A*, where the searches are attempting to meet each other, the "goal" state of the search from the other direction is always moving. Thus, it would be expensive to recalculate the values for every node after every iteration. So in the implementation of Bidirectional A* being examined here, nodes' values are not updated after they're initially calculated. Also a node's heuristic is calculated with respect to the currently active node from the opposing direction, not the closest node.

This gives a worst case of:

$$2 * O(b^{d/2}) \tag{6}$$

Where b is the number of nodes expanded at each depth, and d is the depth required to find a solution. This is derived from the fact that Bidirectional A* is composed of two instances of A*. The instances have b values equivalent to A*, but only half of the depth, see **figure 2**.

One consequence of Bidirectional A* is that it tends to stray away from the optimal solution. This is caused by its heuristic function which directs the search towards the search from the opposing direction, even if a faster route to the goal state is available. One way to use Bidirectional A* to find an optimal solution is to search with Bidirectional A* then switch to A* once a solution has been found. A* can then be used to find the optimal solution [3]. Another way of improving the sub-optimal solutions generated by Bidirectional A* is to develop a similar algorithm that doesn't stray as far from the optimal solution.

C. Weighted Bidirectional A*

Weighted Bidirectional A* has the efficiency of Bidirectional A* while reducing how much it strays from the optimal solution. Weighted Bidirectional A* is different from A* and Bidirectional A* in that its evaluation function contains two weighted heuristics instead of one:

$$f(n) = g(n) + w_1 * h_1(n) + w_2 * h_2(n) \tag{7}$$

where $g(n)$ is the number of moves required to get to its current state, $h_1(n)$ is the first heuristic function which is given weight w_1, and $h_2(n)$ is the second heuristic function which is given weight w_2. In order to maintain an admissible heuristic the following must hold true:

$$w_1 + w_2 = 1 \tag{8}$$

Weighted Bidirectional A* has the same worst case as Bidirectional A*:

$$2 * O(b^{d/2}) \tag{9}$$

This worst case is derived from two A* searches each with the same b-value but only half of the depth, being each will only be required to go half of the distance from the start to the goal, see **figure 2**.

The first heuristic used is the heuristic used by A*, the Manhattan distance to the goal **(3)**. This heuristic is an excellent choice because it tends to find solutions that are optimal or nearly optimal. The second Heuristic used is the heuristic used by Bidirectional A*, the Manhattan distance to the opposing search **(5)**. A combination of these two heuristics should provide a strong evaluation function, that is almost as efficient as Bidirectional A*, with near optimal solutions. In the following experiments four weighting combinations will be used to get a wide scope of Weighted Bidirectional Searches performance compared to the other two algorithms.

III. Experiment Methods

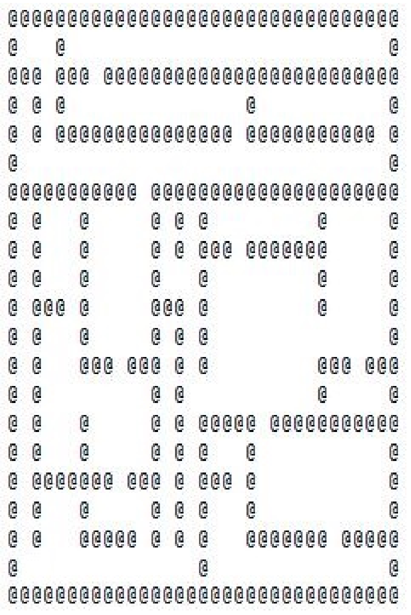
A. Map Generators

Three map generators are used in the following experiments. This allows for extensive testing of the algorithms in a variety of path planning environments. A brief description of each generator is provided.

- **Prim's Algorithm Generator** - generates maps made up of many small rooms, mazes are less complex and have many solutions
- **Depth First Search Generator** - maps with one unit wide corridors, mazes are of average complexity with only one solution
- **Repeated Division Generator** - maps can be generated with corridors and/or rooms, mazes are more complex and have only one solution *(example in **Fig.3**)*.

B. Path Planning on Binary Maps

A Binary map is a grid containing only two characters. One character represents passable terrain, terrain that can be moved across or through. The second character represents impassable terrain, marking somewhere the character or entity cannot move. In this case, the passable character is represented by



```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@   @                           @
@@@ @@@ @@@@@@@@@@@@@@@@@@@@@@@@@
@ @ @               @           @
@ @ @@@@@@@@@@@@@@@ @@@@@@@@@@@ @
@                               @
@@@@@@@@@@@ @@@@@@@@@@@@@@@@@@@@@
@ @   @     @ @ @         @     @
@ @   @     @ @ @@@ @@@@@@@     @
@ @   @     @   @         @     @
@ @@@ @     @@@ @         @     @
@ @   @     @ @ @               @
@ @   @@@ @@@ @ @         @@@ @@@
@ @         @ @           @     @
@ @   @     @ @ @@@@@ @@@@@@@@@@@
@ @   @     @ @ @   @           @
@ @@@@@@@ @@@ @ @@@ @           @
@ @   @     @ @ @   @           @
@ @   @@@@@ @ @ @   @@@@@@@ @@@@@
@               @               @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Fig. 3. Binary map with rooms and corridors generated by the Repeated Division Generator *(map size: 33 by 17)*

a space, while the impassable character is represented by the '@' character, see **figure 3**. For Path Planning, two metrics are collected: the time it takes the algorithm to find a solution, and the number of nodes generated. A third metric is added for maps with multiple solutions, the distance required to traverse from the starting location to the ending location following the solution path. This distance will provide an excellent measure of relative optimality given that all of the algorithms solve the same maps.

Prim's Generator allows for a comparison of the algorithms on maps featuring many solutions and various paths. This experiment is important because it will indicate which algorithms find the best solutions. For this part of the experiment each algorithm, and weighting, solved 25 maps of each size. Map sizes were 50x50, 100x100, 150x150, 200x200, 250x250 and 300x300.

The DFS Generator creates maps where optimality is not a concern given that there's only one solution. These maps only have one unit wide corridors which means only one node is expanded per iteration unless its at an intersection point of two hallways in the map. In the experiment each algorithm solved 100 mazes for each size. Sizes were 20x20, 40x40, 60x60, 80x80 and 100x100, and the results for each size were averaged.

The Recursive Division Generator generates maps with both rooms and corridors, and only one solution. They're also more complex than the maps generated by the other two generators, most of the solutions involve traversing most areas of the map in order to find a solution. In this experiment each algorithm solved 250 mazes of each size. Sizes were 33x33, 65x65, 129x129, 257x257 and 513x513.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

1	2	3	4
5	6	7	8
9	10	11	
13	14	15	12

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

1	2	3	4
5	6	7	8
9		10	11
13	14	15	12

Fig. 4. A demonstration of Sliding Puzzle scrambling process. The scrambling algorithm insures that it doesn't immediately undo the previous move by never moving the same piece twice in a row.

C. Sliding Puzzle

A sliding puzzle is a square grid containing $w * h - 1$ tiles, with w being the width of the puzzles and h being the height. The tiles are numbered and one space is empty. To complete the puzzle, tiles are sequentially slid into the empty space. Once all tiles are in number order, and the empty space is in the lower right, the puzzle is complete, see **figure 5**.

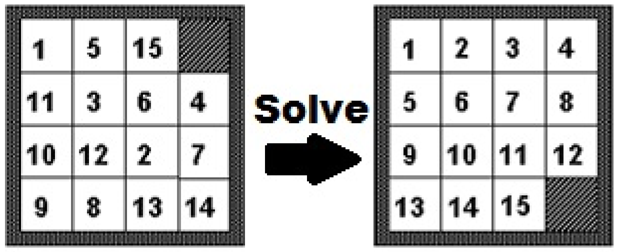


Fig. 5. Example of a Sliding Puzzle

In this experiment each algorithm solved 150 sliding puzzles. Ten for each size, 3x3, 4x4 and 5x5, and for each number of moves used to scramble, 10, 20, 30, 40 and 50. Results were averaged for each combination. Analyzing the functionality of both algorithms leads to the hypothesis that Bidirectional A* will likely be more effective at sliding puzzles, the biggest flaw with using A* to solve sliding puzzles is that in order to find a solution some puzzles require pieces to be moved further away from their goal state which goes against the heuristic being used. However, being the heuristic in Bidirectional A* is the distance between the two currently active states, it's less affected by this problem. A* is often ineffective when pieces need to be moved further away from their goal states in order to find the solution.

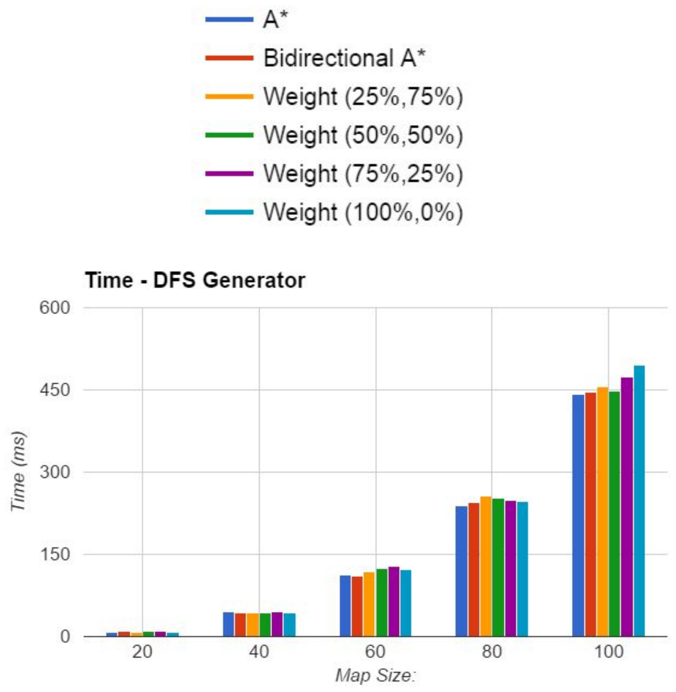


Fig. 6. Graph comparing run time on maps generated using DFS generator

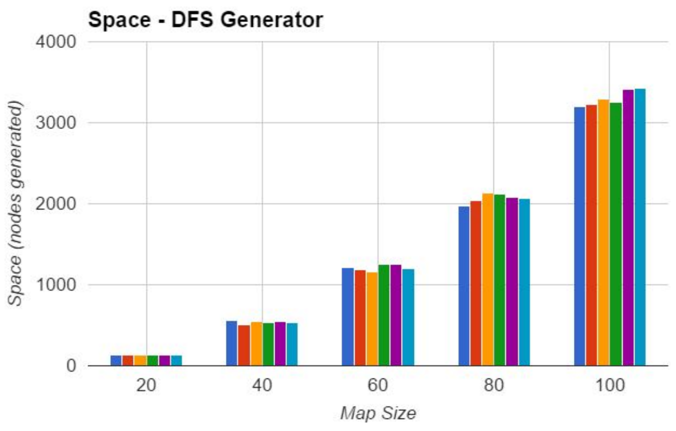


Fig. 7. Graph comparing space on maps generated using DFS generator

IV. Experiment Results and Analysis

A. Comparing A*, Bidirectional A* and Weighted Bidirectional A* for Path Planning

The Depth First Search Generator allowed for a comparison of the algorithms on maps where optimality is not a concern given that there's only one solution. It's also worth noting that maps generated using Depth First Search only have corridors of width one. **Figure 6** shows Weighted Bidirectional A* slightly under performing relative to the other algorithms for the largest maps. Other than that, all of the algorithms seem to compare similarly with respect to run time. In **Figure 7** it can be seen that the difference between the number of nodes generated by the various algorithms is insignificant. This is to be expected being the maps tested have only width one corridors, so the Manhattan distance heuristic function is less effective than in the other experiments. A* and Bidirectional A* do appear to do slightly better than all of the weightings for Weighted Bidirectional A*, but not by a significant amount.

Prim's Generator allowed for a comparison of the algorithms on maps featuring many solutions. **Figure 8** shows that Bidirectional A* and Weighted Bidirectional A* performed exponentially better than A* with respect to run time. Bidirectional A*, on average, solved 300x300 puzzles 7.4 times faster than A*. The best weighting for Bidirectional A* with respect to run time was $w_1 = .25$ and $w_2 = .75$ which performed only slightly worse than Bidirectional A*. **Figure 9** shows that Bidirectional A* and Weighted Bidirectional A* also generated fewer nodes before finding a solution than A*.

Figure 10 demonstrates Bidirectional A*'s greatest flaw, it often finds a sub-optimal solution for problems with multiple solutions. Given that finding an optimal or near optimal solution is often a priority in path planning, Bidirectional A* is likely not a good algorithm for path planning unless techniques are used to improve the optimality of the solutions that it finds.

Weighted Bidirectional A* performs extremely well, it's significantly more efficient than Standard A*, and also found solutions closer to being optimal than those found by Bidirectional A*. Comparing **figure 8** and **figure 10**, it can be seen that as the weighting for h_1 increases, the solutions became closer to being optimal, but the efficiency declines. This reinforces the idea that Weighted Bidirectional A* is an excellent compromise between the efficiency provided by Bidirectional A*, and the optimality of A*. Considering that Standard A* is also not guaranteed to return an optimal result, Weighted Bidirectional A* appears to be a better alternative to Standard A* for path planning.

The Recursive Division Generator allowed for a comparison of the algorithms on complex maps with rooms and corridors. **Figure 11** shows Weighted Bidirectional A*'s efficiency is comparable with the efficiency of Bidirectional A*, with both finding solutions 1.3 times faster then A* on average. In **Figure 12** it can be seen that Bidirectional A* scales significantly better than Standard A*. However, the greatest weakness of Bidirectional A* is that it often strays away from the optimal solution, which is not shown in this experiment given that maps generated with the recursive division generator only have a single solution.

A research paper by Pulido discusses how Bidirectional heuristic searches often yield sub-optimal solutions. The findings in his paper are in line with the results found here. When Bidirectional A* is used for path planning, it finds a solution faster than A*; however, the solution is often sub-optimal [1]. Dr. Pulido also investigates techniques to try and improve the optimality of solutions found by Bidirectional search. When adding the time and space costs of these techniques to the cost of Bidirectional A* it usually yields a higher overall cost than Standard A*, except for extremely large map sizes.

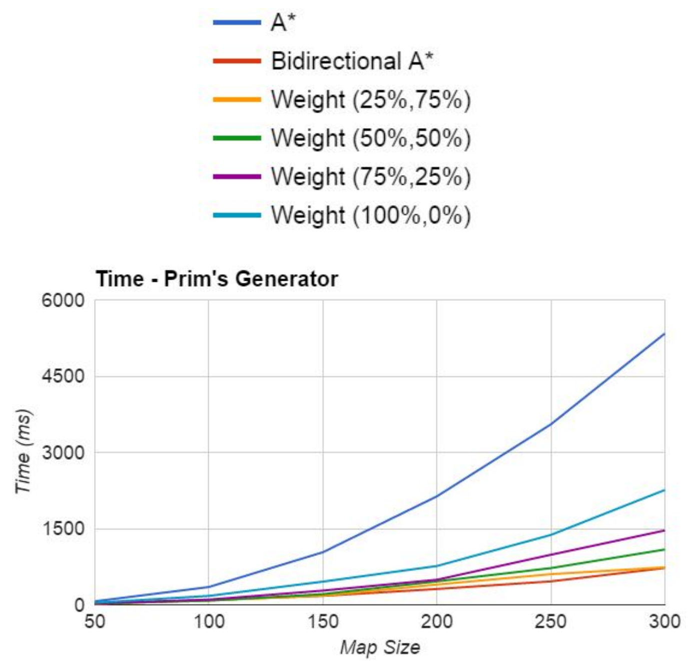


Fig. 8. Graph comparing run time on maps generated using Prim's generator

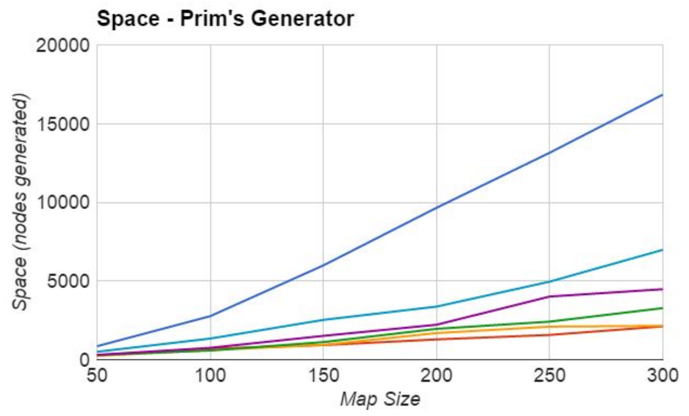


Fig. 9. Graph comparing space on maps generated using Prim's generator

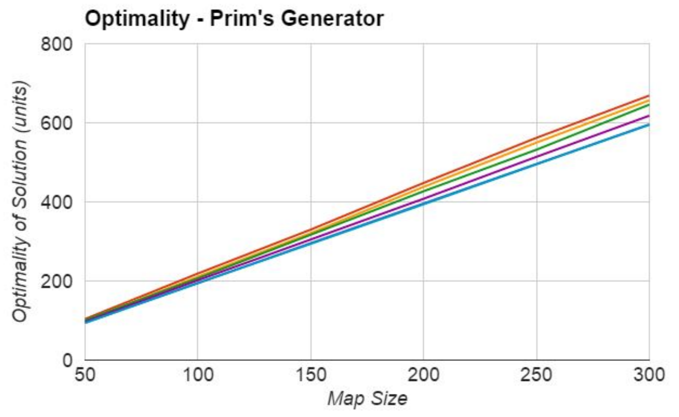


Fig. 10. Graph showing the average solutions found by the algorithms *(less is better)*

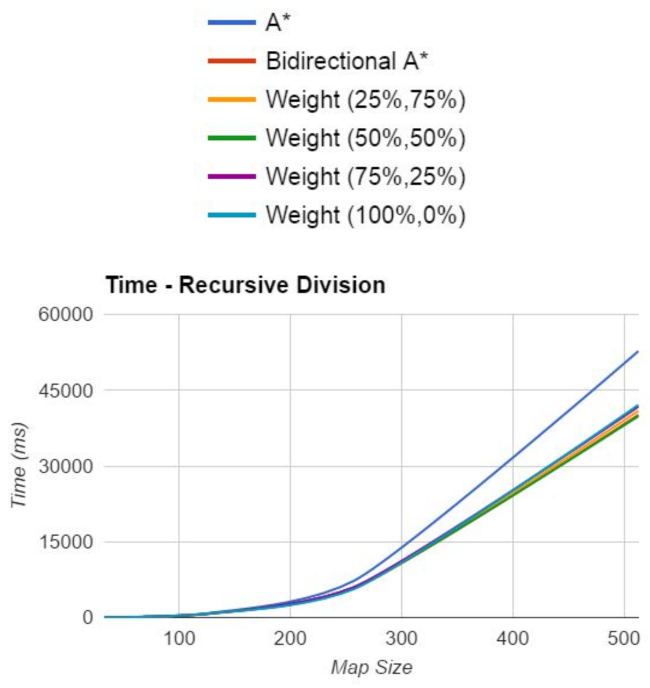


Fig. 11. Graph comparing run time required for maps generated using the Recursive Generator

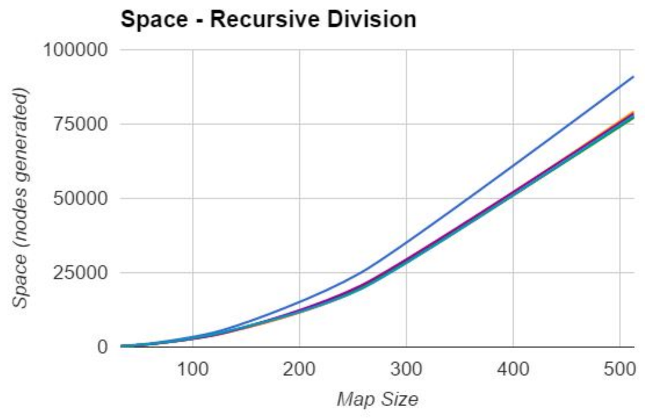


Fig. 12. Graph comparing space complexity for maps generated using the Recursive Generator

So in summary A* is superior to Bidirectional A* when finding an optimal or near optimal solution is a priority. When finding any solution is desired, regardless of how optimal it is, Bidirectional A* thrives and significantly outperforms A*. If finding a near optimal solution is important but not required, Weighted Bidirectional A* combines the efficiency of Bidirectional A*, with the optimality of A*.

B. Comparing A and Bidirectional A* on Sliding Puzzle*

A* is commonly used to solve sliding puzzles. Therefore comparing Bidirectional A*'s performance to A* on the sliding puzzle will explore how it preforms outside of path planning. **Figure 13** shows the run times versus the number of moves used to scramble the puzzles. The numbers following

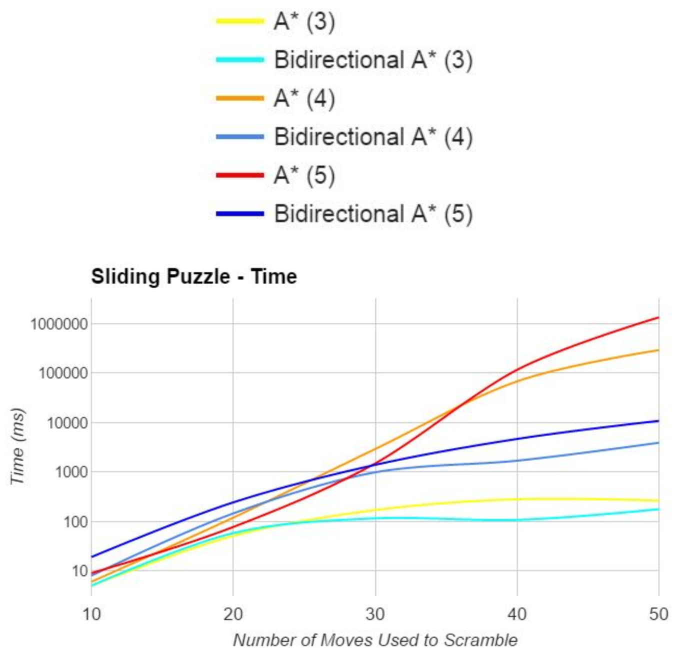


Fig. 13. Graph comparing run time on sliding puzzle

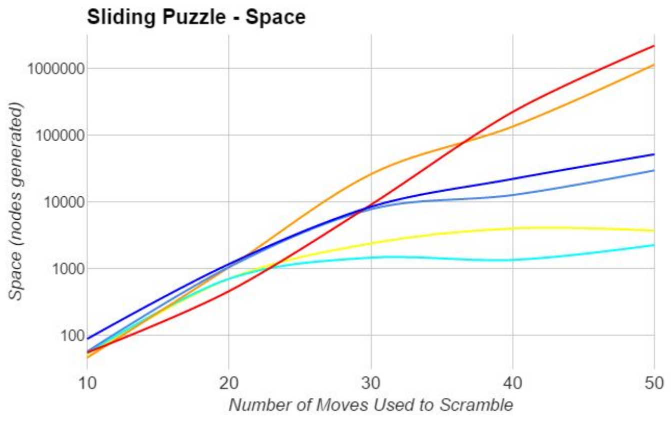


Fig. 14. Graph comparing space required for sliding puzzle

the Algorithms indicates the puzzle size, i.e. A* (5) is A* solving a 5x5 puzzle.

Figure 13 and **figure 14**, show that Bidirectional A* is an exponential improvement over A* for sliding puzzle. A* is superior when less moves are used to scramble. It finds a solution approximately 1.2 times faster than Bidirectional A* for all sizes, with 10 moves used to scramble. This is likely caused by the over-head caused by Bidirectional A* using two instances of A*, and two priority queues. However, Bidirectional A* scales better than A*. Bidirectional A* scales at a linear rate with respect to 'moves used to scramble', while A* scales exponentially. Bidirectional A* solved the ten 5x5 puzzles, with 50 moves used to scramble, 42 times faster than A*. This appears to reinforce the initial analysis about the differences between A* and Bidirectional A*. A* has trouble with sliding puzzles because it becomes ineffective when pieces need to be moved further from their goal locations.

Bidirectional A* is less affected by this being its searching from both directions.

Bidirectional A* is also capable of solving much larger puzzles than Standard A*. Standard A* consistently solved 7x7 puzzles that were scrambled with 40 moves and 5x5 puzzles scrambled with 50 moves but could only occasionally solve larger puzzles. Bidirectional A* consistently solved 7x7 puzzles that were scrambled with 70 moves and 5x5 puzzles scrambled with 80 moves.

V. Conclusion

It can be seen from these experiments that Bidirectional A* performs similarly or better with respect to time and space than Standard A* in all problems tested. However, Bidirectional A* is less desirable in problem where there are multiple solutions and finding an optimal solution is a priority. Bidirectional A* also vastly outperforms Standard A* in problems that occasionally go against the heuristic, such as sliding puzzle.

Given that in path planning finding an optimal solution is often a priority, Bidirectional A* would generally not be beneficial in path planning. For Sliding Puzzle Bidirectional Search is extremely beneficial, scaling exponentially better than Standard A*. Weighted Bidirectional A* is a good balance between efficiency and optimality, significantly more efficient than Standard A* while yielding solutions almost as optimal as solutions generated by Standard A*. Weighted Bidirectional Search is a good choice for solving path planning problems if finding a near optimal solution is important without requiring the optimal solution.

A. Future Work

In future works I will be creating an incrementally updating version of Bidirectional A* to explore how Bidirectional A*s inability to update nodes' values after they've been created affects its efficiency. Another potential feature of this algorithm will be to calculate a node's heuristic value compared to the closest node from the opposing direction instead of the currently active node. Both of these changes will likely improve efficiency on memory constrained systems but reduce performance on CPU constrained systems.

Future work will include investigating techniques for improving the optimality of solutions found by Bidirectional A* to make it more of a viable algorithm for path planning. Weighted Bidirectional A* seems to improve the solutions found by Bidirectional A*, but other techniques will be required in order to yield solutions with that are as optimal as those found by Standard A*. Finding a consistent heuristic for path planning and sliding puzzle would also help with the performance for all of the algorithms.

Acknowledgment

Parts of the testing framework were made by Noah Wong. Prim's Algorithm Maze generator made by Benjamen Weber. DFS Maze generator and Repeated Division Generator made by Nathan Comer.

References

[1] Pulido, Francisco Javier, L. Mandow, and J.L. Prez De La Cruz. *"An Analysis of Bidirectional Heuristic Search in Game maps."* Research Gate. Universidad De Mlaga, n.d.

[2] Russell, Stuart; Norvig, Peter (2003) [1995]. Artificial Intelligence: A Modern Approach (2nd ed.). Prentice Hall. pp. 97104. ISBN 978-0137903955

[3] Kaindl, Hermann, Gerhard Kainz, Roland Steiner, Andreas Auer, and Klaus Radda. Switching from Bidirectional to Unidirectional Search. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA 1999, n.d. Morgan Kaufmann Publishers Inc, 1999. 1 Dec. ISBN 1-55860-613-0.