Search and Pathfinding Algorithms in Trees and Graphs

Gordon, Owen
Kirchner, Seth
Trimm, Shaun

Abstract

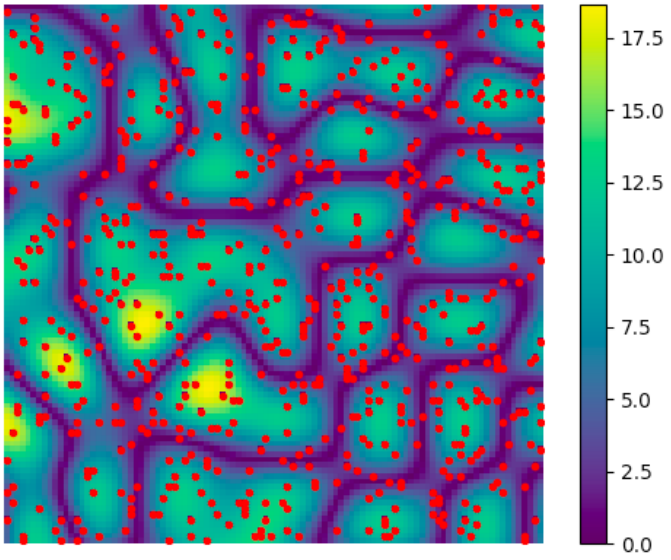
There exist many different pathfinding algorithms, each of which has different methods of traversing across a graph. The objective of this project is to analyze different search algorithms to understand which algorithms are better in terms of time and space complexity, and accuracy. After thorough analysis, it was discovered that informed searches, regardless of all possible board attributes, performed better. A* and Dijkstra's algorithm both found the optimal path in the graph every single time, but they usually found it without looking at nodes in the same order. Beam search could find a nearly optimal path, but suffered from too much information at a full length of 4 because it was expanding far more nodes, and this had a major time disadvantage. Bidirectional search always expands far fewer nodes than any other algorithm, and also finds a nearly optimal if not the optimal path. UCS and Greedy search each have their own single heuristic, and A* combines both of them, and when the heuristics are combined A* optimizes both of these searches. Both uninformed searches, BFS and DFS were always able to find the goal, but never found an optimal path. Since these uninformed algorithms are less complex than some of the informed searches, both BFS and DFS were still quite fast, even though they looked at many more nodes. Overall, A* and Dijkstra's were the best algorithms for this problem of pathfinding, but each algorithm could be adapted for different data and find their own niche.

Introduction

There exist many different algorithms for finding optimal paths from one point to another in a given data set. There are many to choose from, simply because there does not exist one algorithm that is better than the rest in every situation. The goal of this project is to analyze different search algorithms to understand which algorithms are better in terms of time and space complexity, accuracy, as well as identifying which algorithms work better on different parameters such as size, number of obstacles, and maximum cost, for a NxN grid of nodes.

This project's hypothesis is the following: If an algorithm is given more information, then that algorithm will find a more optimal path because it will have the necessary tools to discover the best path. On any board state, the result should be the same independent of any factors like board size, maximum board weight, obstacle density or start and goal locations.

The search space of this project was an NxN board of nodes, where each node was associated with a particular value that represented the cost to visit the position. Randomness was critical to visualize and test different algorithm behavior, so to assign random values to each location the perlin noise algorithm was used, supplied by the 'Noise' module within Python3. This is simply a way to create smooth randomness, where the change in a number isn't entirely random, but can be thought of as stepping through a random field and the values slowly changing. This was necessary because it creates a sudo maze that the algorithms have to find their way through. The figure below is an example of a board.



On the right, the color bar shows the varying cost. The purple is the lowest cost, so if an algorithm has found a cheap path, it will most likely be along a continuous purple path. Likewise, if an algorithm finds a suboptimal solution, it will most likely traverse over green or even yellow nodes, which will add greatly to that path's cost. The red dots are the obstacles in the way of the algorithm. These are locations that cannot be visited, and thus presents a slight challenge for an algorithm to path find around.

Algorithms will be analyzed using three different factors. Firstly and most important, cost. If an algorithm finds the optimal cost it will have met the project goal. Second, the length of time it took for the algorithm to find a solution. This is an important metric because it represents the time complexity, as well as shows how fast the algorithm is with the current board set up. Third, the number of nodes analyzed. Just like the speed, this metric shows the space complexity of the algorithm, and provides differentiation between algorithms that find the same cost of path. These 3 factors then go into the formula to calculate the algorithm's score (lowest cost is the lowest cost found by all algorithms who tested the same board):

*Score(algorithm) = ((algorithm_cost - lowest_cost) + number_of_nodes_visited) * algorithm_time*

This was necessary because if multiple algorithms find the same cost, then the number of nodes they expanded, as well as how long this took, will show which algorithms were ultimately more powerful than others.

Individual Analysis of Algorithms

There exists many different pathfinding and search algorithms. This project focuses on only a handful of the more popular ones.

The algorithms used in this project vary in a few important categories. First, there are two main types of algorithms being tested here: uninformed and informed algorithms. Uninformed algorithms, such as Breadth-First Search, Depth-First Search, and Iterative Deepening Depth-First Search, approach searching the data set in a brute force style. Uninformed algorithms, also called blind searches, only expand the next node given a primitive predefined rule. These searches also rarely find an optimal path. On the other hand, informed algorithms use a heuristic to make an informed decision when traversing a graph and are designed to reach an optimal solution. The informed algorithms used in this project are: A* Search, Greedy Search, Dijkstra's, Beam Search, Uniform Cost Search, and Bi-Directional Search. The heuristics used by these algorithms are either Manhattan distance, or Euclidean distance. Both of these give an estimated distance from the goal, but since they assume no cost, the heuristic will always be optimistic and underestimate the real cost to the goal. This means both heuristics are admissible.

For each of these algorithms, a board showing the path found, as well as the cost of the path, the time taken, the number of nodes expanded and the overall score will be used. The following table is a brief overview of this information, and each piece will be elaborated on further in each algorithm's specific section.

Algorithm	**Cost**	**Time (s)**	**Nodes**	**Score**
BFS	1398.907	0.0985	9296	1140.285
DFS	7405.015	0.0857	8114	6141.648
UCS	250.766	0.1638	9062	14.842
Greedy Manhattan	411.195	0.1784	5608	296.180
Greedy Euclidean	825.960	0.1390	4919	806.317
A* Manhattan	250.766	0.2015	9017	18.174

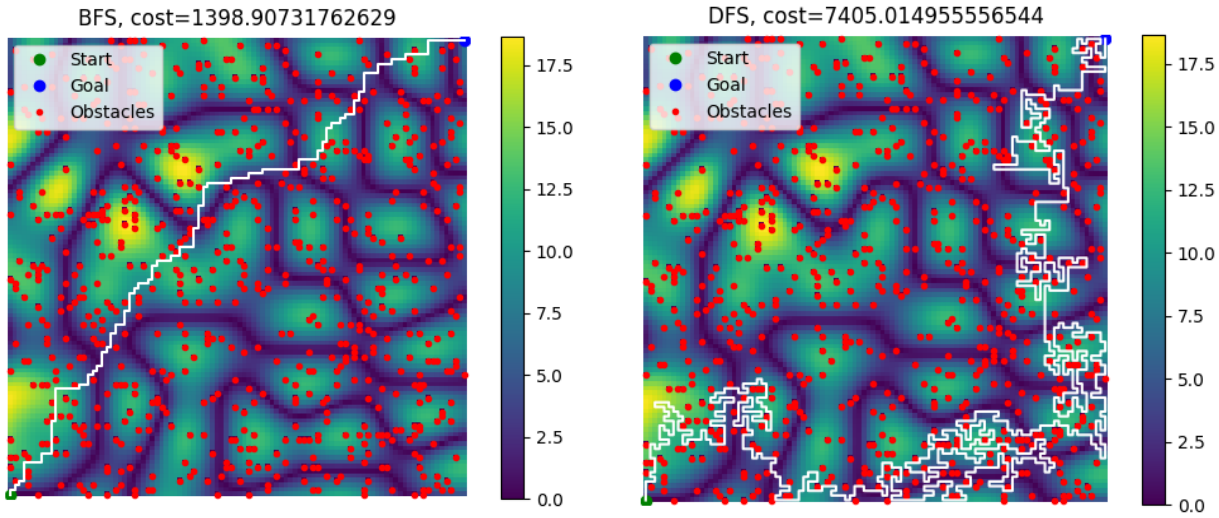
A* Euclidean	250.766	0.2401	9010	21.636
Beam, length=1	255.826	0.4190	9040	59.079
Beam, length=2	253.041	0.2429	8863	27.058
Beam, length=3	251.587	0.2287	9010	22.479
Beam, length=4	251.587	0.2635	9010	25.906
Dijkstra	250.766	0.0822	9167	7.535
Dijkstra Full	250.766	0.1520	9296	14.133
Bidirectional Manhattan	255.712	0.0971	2164	6.901
Bidirectional Euclidean	252.578	0.1637	3204	8.212

Breadth First Search (BFS) is attributed to Edward Moore in 1957, but was also discovered by Konrad Zuse in 1945 (Erickson). BFS works by expanding the starting node, and adds each successive node to a queue data structure. In a queue, the first item added will also be the first removed when an item is retrieved, like a first-come first-serve line. Then to keep exploring BFS retrieves the next node in the queue. This style of fringe expansion is useful to keep the space used by the algorithm to a minimum because if the goal is near the start, not many nodes will have to be expanded around the start. But in the case where the goal is far from the starting position, this will take a long time. Worst case, this algorithm will have an exponential time and space complexity of $O(b^d)$ where b is the branching factor, and d is the depth. In this project, the branching factor will average to 4, since each node in a grid has 4 neighbors, and d will relate to how far the goal node is from the starting node.

Just like BFS, Depth-First Search (DFS) is an uninformed search algorithm. DFS is attributed to Charles Pierre Tremaux in the 19th century as a strategy for solving mazes (Even). DFS works by expanding the starting node, and pushes each successive node to a stack data structure. A stack works like a pile of dishes, the first dish you clean is on the bottom of the pile. Similarly, the first item in the list that you look at was the last item added to the list. This algorithm continuously pushes newly expanded nodes to the stack and pops the top item off until the goal is eventually reached. One large concern with DFS is that there is no guarantee any plunge down the search tree is finite, so DFS could potentially expand infinitely many nodes and never reach a goal. This means that unlike BFS, there is the possibility that DFS never reaches a goal, even after a long search time. One way to prevent this is to set a depth limit, this algorithm is called Iterative Deepening DFS (IDDFS). The problem with setting a depth limit is that the ideal cutoff for the depth limit is rarely known before solving the problem, leading repetitive expansion as

the algorithm continually tries to search deeper. This will lead to a longer run time, not finding a non-optimal solution, or even still not finding a solution. The only improvement over BFS that DFS gains is that DFS has to store many less nodes in memory, and thus the space complexity of DFS is not exponential, and is instead O(bd), linear time. DFS time complexity is still $O(b^d)$. IDDFS tries to capitalize on the advantages gained by BFS, but the repetition in node expansion still leads to the same time and space complexity as DFS.

Uninformed searches can have their time and place, but in this project, the goal of the algorithm was to find an optimal path through a graph, and neither BFS, DFS, or IDDFS are optimized to find an optimal path, and rarely ever find an optimal path. The figure below shows both BFS and DFS on the same puzzle. The background of the image is a visualization of the cost to visit each node in the 100x100 field.



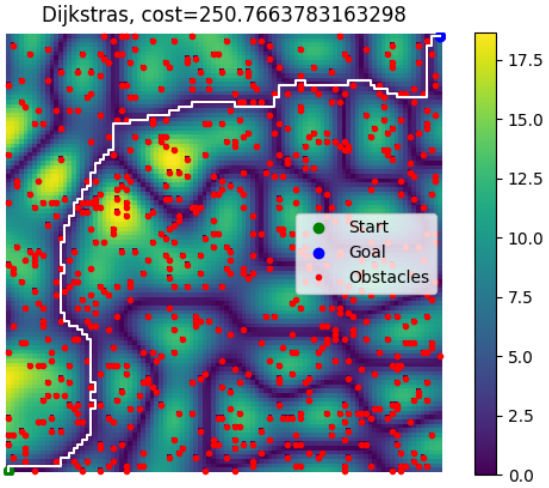
BFS Performance
Cost: 1398.907
Time: 0.0985
Number of Nodes Searched: 9296
Performance Score: 1140.285

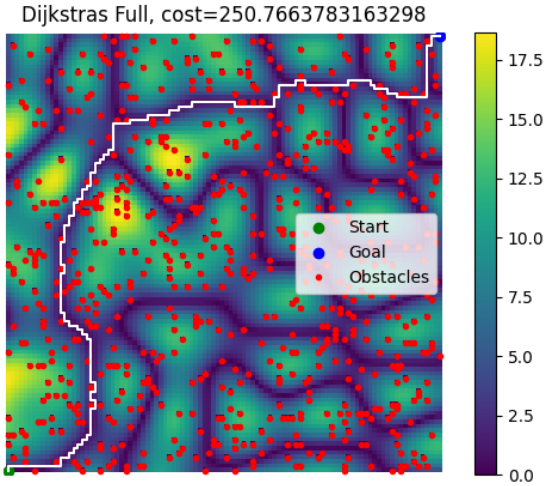
DFS Performance
Cost: 7405.015
Time: 0.0857
Number of Nodes Searched: 8114
Performance Score: 6141.748

Given that the optimal path cost for this board was 250.766, both of these algorithms were unsuccessful. Even though these aren't the best algorithms, it shows that even with a primitive rule for fringe expansion, a complex space can be navigated.

Dijkstra's Algorithm was published by Edsger W. Dijkstra in 1959 (Frana). Dijkstra can be implemented two ways. In one method, Dijkstra finds the shortest path from the start to the goal node. The second method allows the algorithm to continue and find the shortest path from the starting node to every node in the graph. The way this project implements Dijkstra, using a heap,

the time complexity is O((|E|+|V|)log|V|). Dijkstra's algorithm was by far one of the best algorithms for path finding due to the fast search times, but Djikstra's algorithm must look at many vertices and edges, expanding many nodes, and using lots of space. The figure below shows the performance of the two Dijkstra variants.



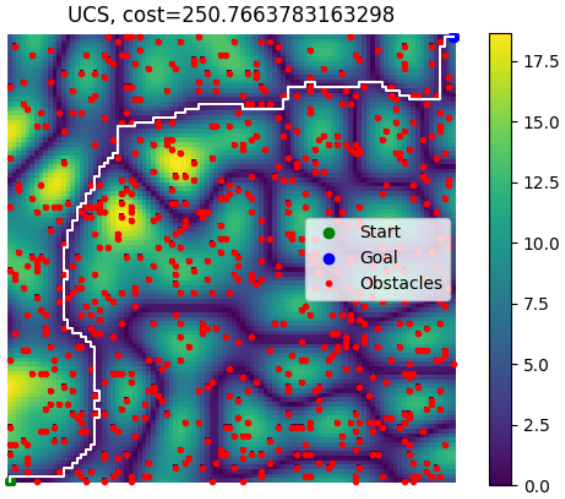


Dijkstra ending when found
Cost: 250.766
Time: 0.0822
Number of nodes Expanded: 9167
Performance Score: 7.535

Dijkstra exploring full graph
Cost: 250.766
Time: 0.1520
Number of nodes Expanded: 9296
Performance Score: 14.133

Both of these algorithms found the optimal path, and so their performance score was very low. Even though the algorithm that explored the whole board only looked at about 130 more nodes, these all had to be thoroughly expanded, and so it took just under twice the length of time compared to the algorithm that ended when the goal was reached.

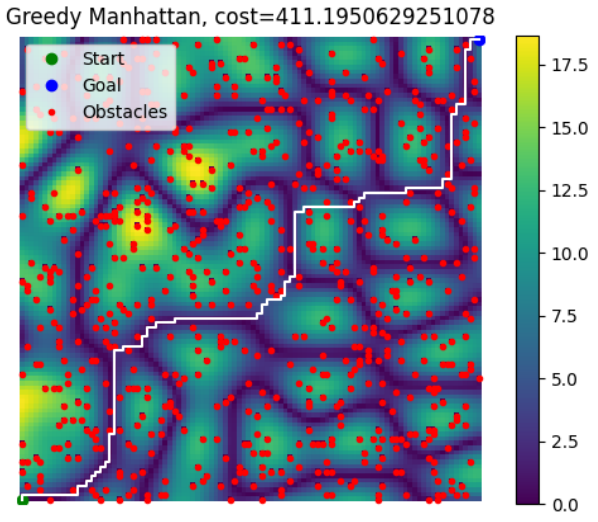
The next algorithm looked at was Uniform-Cost Search (UCS). This algorithm is also attributed to Dijkstra in 1959 (Coppin). UCS works much in the same way as BFS, however, instead of finding a path with the least amount of edges, UCS finds the path with the lowest weight. There are a few disadvantages, such as the need to update the cost of a node if a lower cost path is found. UCS also requires larger and larger amounts of storage as the graph that it is searching becomes larger. Just like Dijkstra's algorithm, UCS performs very well. The figure below shows an example of a path found using UCS.

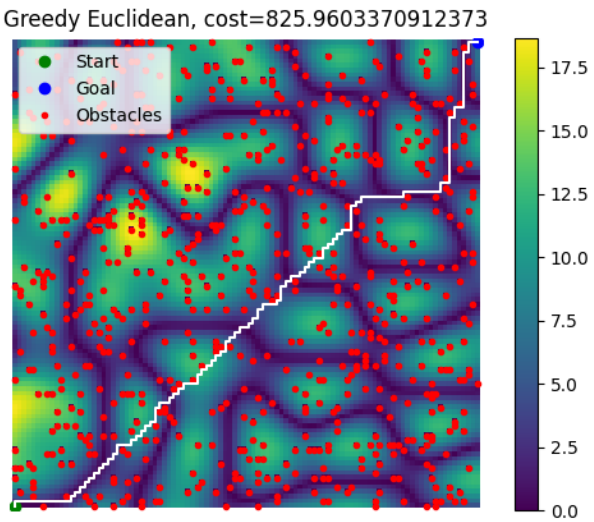


UCS Performance
Cost: 250.766
Time: 0.1638
Number of nodes Expanded: 9062
Performance Score: 14.842

UCS found the optimal path, but it didn't do it quite as fast as either Dijkstra algorithm, but it did explore slightly fewer nodes. UCS is a great option, and was especially easy to implement. Human time isn't always prioritized over machine time, but for this algorithm, the cost for humans was very low, and the results are impressive for that.

Greedy algorithms are any generic algorithm that makes a locally optimal choice at each step, this is where the term greedy comes from. Using different heuristics, such as manhattan distance or euclidean distance, the greedy algorithm can move across a grid in different ways. Manhattan distance heuristics allow for 4 directions and euclidean distance heuristics allow for movement in any direction. A greedy algorithm selects the lowest cost that is immediately available. Only makes decisions based only on information that it has at any one step. This may lead to not finding a solution. The figure below shows the two different types of greedy searches used, manhattan distance and euclidean distance.



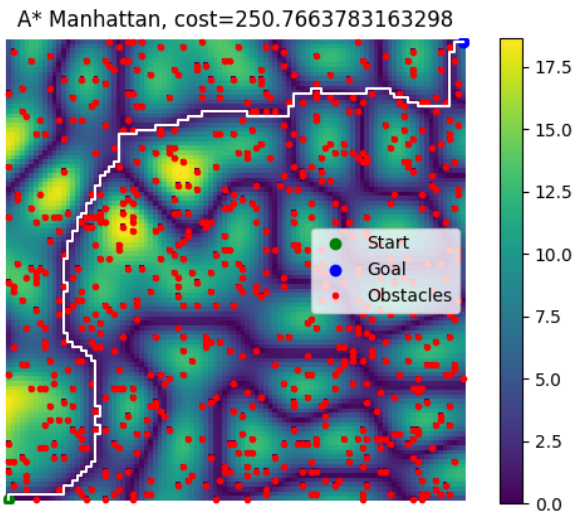


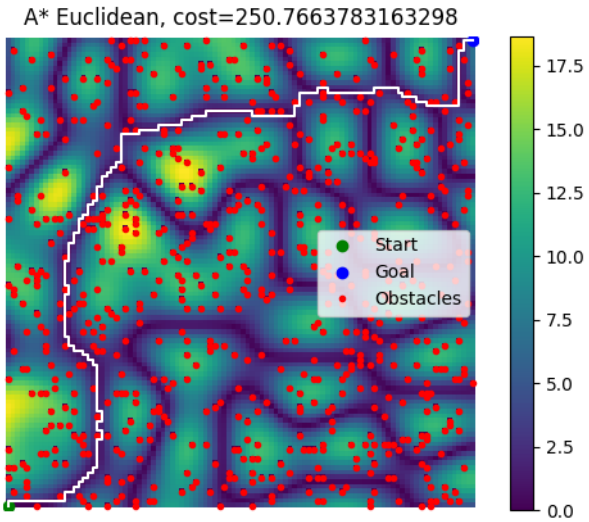
Greedy Manhattan Performance
Cost: 411.195
Time: 0.1784
Number of nodes Expanded: 5608
Performance Score: 296.180

Greedy Euclidean Performance
Cost: 825.960
Time: 0.1390
Number of nodes Expanded: 4919
Performance Score: 806.317

While neither of these found the optimal goal, the number of nodes expanded was significantly lower than any other algorithm, saving on space. Now taking a moment to focus on the actual paths found, these are by far the most visually striking. The manhattan distance comparison can be seen so clearly in the manhattan distance path because the straight lines and right angles are prefered. In euclidean distance, where the straight line between two points is the shortest, the algorithm shows this too by finding a path that is almost completely diagonal, only diverging from this when obstacles are in the way.

A* algorithm is attributed to Peter Hart, Nils Nilsson, and Bertram Raphael of Stanford Research Institute in 1968. A* can change its performance by implementing different heuristics. A* takes the UCS cost function, and combines it with Greedy's heuristic to find an optimal solution, and expanding a minimal set of nodes to do this. As with most of the algorithms, the time and space complexity is $O(b^d)$ where b is the branching factor and d is the depth, but unlike most algorithms, A* takes in much more information to try to avoid this worst case scenario. The figure below shows A* using manhattan distance and A* using euclidean distance.





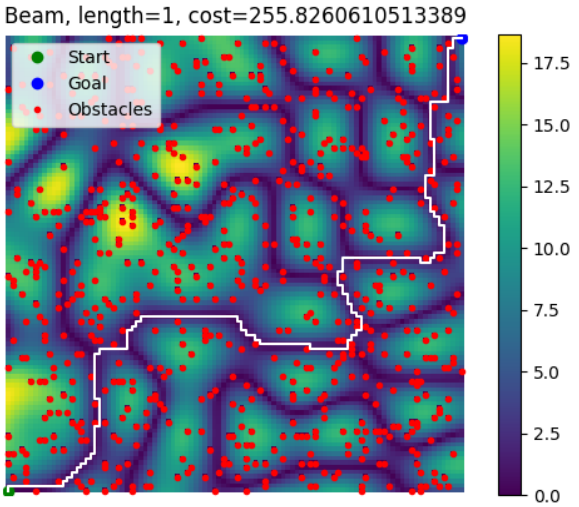
A* Manhattan Performance
Cost: 250.766
Time: 0.2015
Number of nodes Expanded: 9017
Performance Score: 18.174

A* Manhattan Performance
Cost: 250.766
Time: 0.2401
Number of nodes Expanded: 9010
Performance Score: 21.636

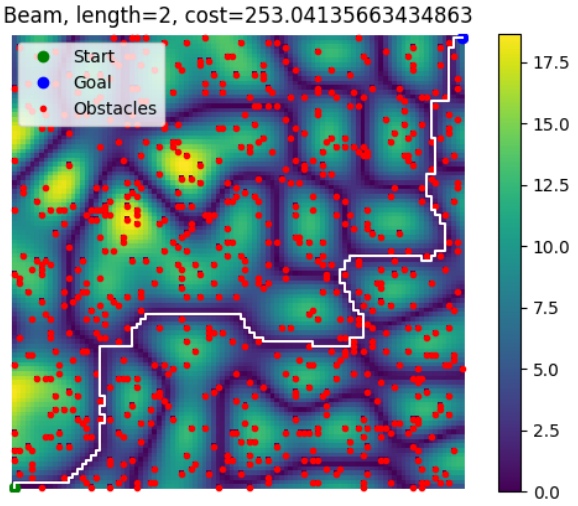
Both variants of A* found the optimal solution, and they both did it exploring less nodes than UCS. Since A* was able to combine UCS and Greedy, an optimal solution was found and it needed less expansion to do it. But there is a downside to A*, and the time taken reveals it. A* is a much more complex algorithm, and thus lowering the number of iterations it takes is a good thing, it doesn't quite make up for the longer single algorithm iteration. This is the primary tradeoff with A*. Since the heuristics used are admissible A* will always find the optimal path.

Moving along, the next algorithm is called Beam search and is attributed to Raj Reddy in 1977. Beam search operates very similarly to a UCS. The main difference is that you can specify the number of children that are added to the fringe each time you progress to a new node. These new nodes are added based on the heuristic value given. At each new node the fringe is sorted, resulting in the best options being expanded first, and an increase in time complexity at large graph sizes due to the large size of the fringe. The time complexity of Beam search is equal to O(B * m), where B is equal to the beam width and m is the maximum depth of any path in the graph/tree you are searching in. In the tests completed regarding beam search, the time complexity decreased dramatically when going from a beam width of 1 to a beam width of 2. From 2 to 3, and later to 4, the time complexity was consistent. The differences between beam widths may be more pronounced with trees and graphs with a bigger max beam width. A beam length of 1 means that only 1 node is ever added to the fringe. This dramatically reduces the ability for an algorithm to look ahead, and must instead only ever traverse the next best node. Increasing the beam length gives the algorithm more look-ahead, and the algorithm may even

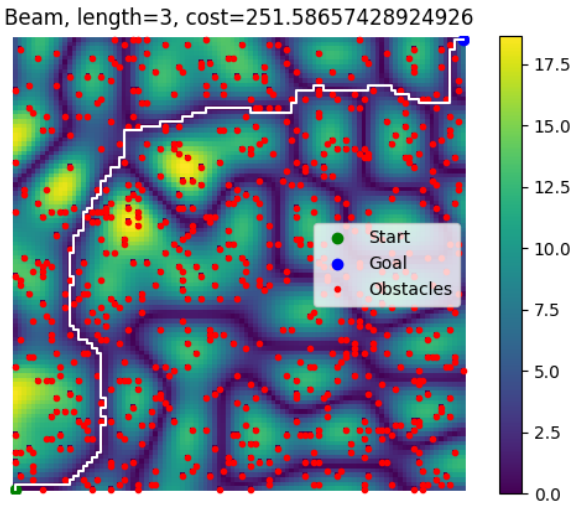
find an optimal path before increasing the beam length all the way up to the max. For the purpose of this project, an example of each of the 4 beam lengths was completed on the same board and the figure below shows the resulting paths.



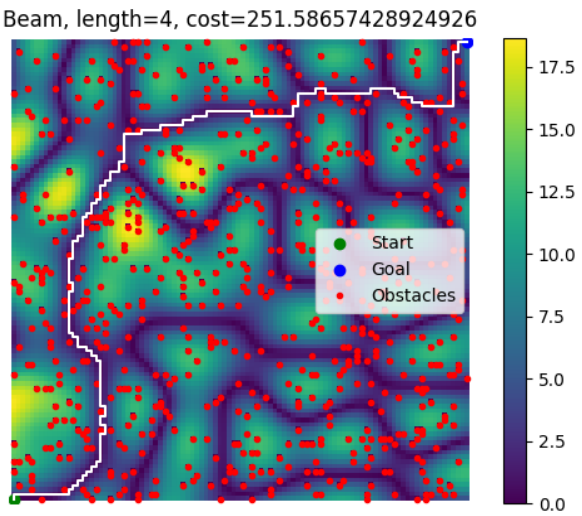
Beam, Length = 1
Cost: 255.826
Time: 0.4190
Number of Nodes Expanded: 9040
Performance Score: 59.079



Beam, Length = 2
Cost: 253.041
Time: 0.2429
Number of Nodes Expanded: 8863
Performance Score: 27.058



Beam, Length = 3
Cost: 251.587
Time: 0.2287



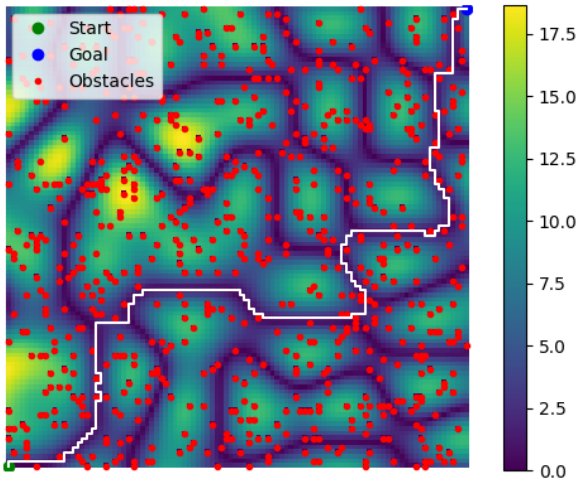
Beam, Length = 4
Cost: 251.587
Time: 0.2635

Number of Nodes Expanded: 9010
Performance Score: 22.479

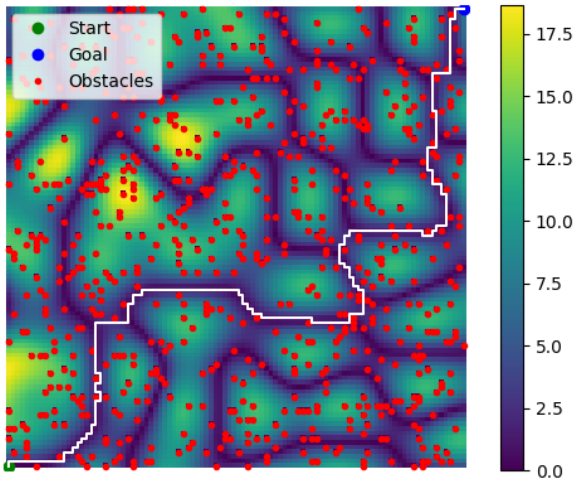
Number of Nodes Expanded: 9010
Performance Score: 25.906

Beam searches between length 1 and 3 all improve, and then at length 4, the search doesn't perform as well. Since a length of 4 doesn't improve the speed of the search, a length of 3 is optimal. Interestingly, all four algorithms find a nearly optimal path, much better than any uninformed search, and also they expand fewer nodes than the algorithms that find the optimal path. Another notable mention with Beam search is that usually the shorter lengths find an entirely different path than the long lengths, and it ends up being almost optimal. This could be implemented to store potential shortest paths and any of these paths could be accessed if the board were to shift while the algorithm was searching. This would build in flexibility to the algorithm's performance, and allow for potentially better and more adaptable results.

Bi-Directional was designed by Ira Pohl in 1971. Bidirectional search works when the start and goal are known. This search runs two simultaneous searches, one forward from the starting node and another in reverse from the goal node. The search is finished when both searches meet in the middle. In this project, bidirectional search was implemented as expanding two different sets of nodes around the two ends. The nodes are expanded just like the other informed searches using a heuristic, and the best node is the one expanded first. This expansion takes place until the sets are no longer disjoint, and at that point the node where the sets overlap is taken as the midpoint and the path from each starting point to the middle is joined into one path and the search is over. Bidirectional search gains the unique benefit of exploring far fewer total nodes. The figure depicting the performance of Bidirectional search is below.



Bidirectional Manhattan
Cost: 255.712



Bidirectional Euclidean
Cost: 252.578

Time: 0.0971	Time: 0.1637
Number of Nodes Expanded: 2164	Number of Nodes Expanded: 3204
Performance Score: 6.901	Performance Score: 8.212

Neither search found the optimal solution, but the number of nodes expanded is between a quarter and a third of what most algorithms have to expand. This extremely reduced number of iterations means that the algorithm is able to complete much faster, but it also is what allows the search to happen from both ends and still reach a conclusion in a respectable time. Since the number of nodes expanded is so much fewer, the space required is significantly reduced and the performance score of each Bidirectional search is low, thus these algorithms perform exceptionally.

The two algorithms that were implemented but not thoroughly discussed or even mentioned were Iterative Deepening DFS and Iterative Deepening A*. Both of these algorithms were too slow to test on a larger scale, and so they had to be excluded. These were created as variations of the tried and true algorithms as a way to gain marginal benefits, if the conditions were right. If these algorithms were to be implemented correctly they would potentially save lots of space as they find the optimal path, while expanding the fewest nodes, but on modern computers, this space requirement is hardly a concern, and thus the popularity of these algorithms has decreased significantly.

Comparison Analysis of Algorithms

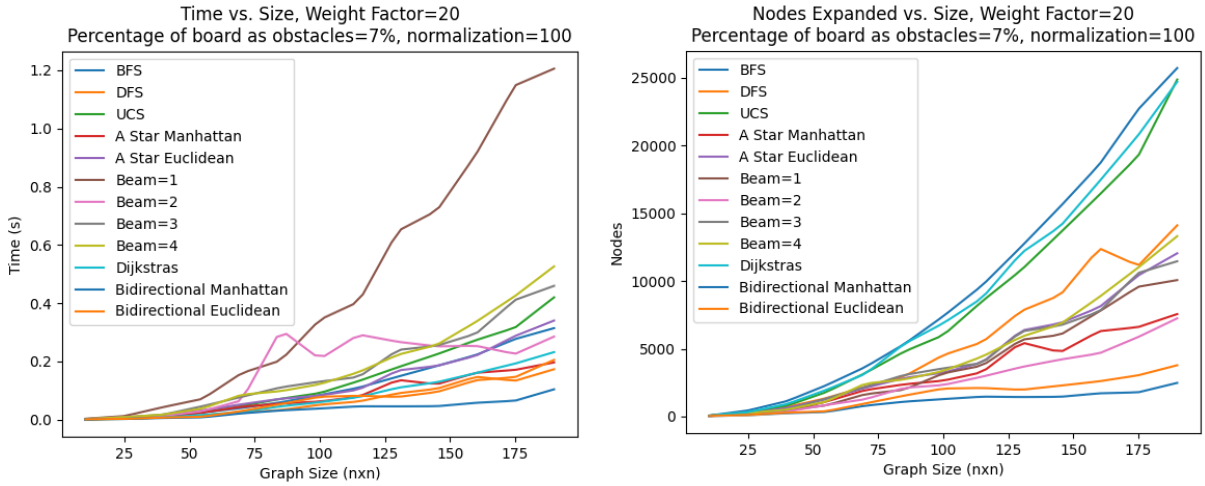
It is extremely important to find optimized algorithms for specific tasks, since finding the correct path quickly and reliably can be quite important. One significant example of why optimization is so necessary is in wifi/cell signals. Routers cell towers, phones, computers, satellites, and nearly everything talk to each other, and as messages are sent from receiver to receiver it is important for the minimum distance to be covered in order to handle the plethora of messages being sent every second. Thus to ensure minimum distance is being traveled the machines are constantly checking to make sure they are still privy to their shortest and fastest path. This requires constant pathfinding, and if a slow algorithm is used, it could potentially cause backups and holdups down the entire chain, resulting in catastrophic failures.

To compare these algorithm's effectiveness and performance, each algorithm was tested many times for specific sizes and under different parameters, and the speed, number of nodes visited, and cost of each test was recorded and then averaged out to gain an insight into how well algorithms perform in different scenarios.

One note, these tests were performed on personal computers. Results should only be taken as a comparison between algorithms, and may not accurately show the actual time taken in different

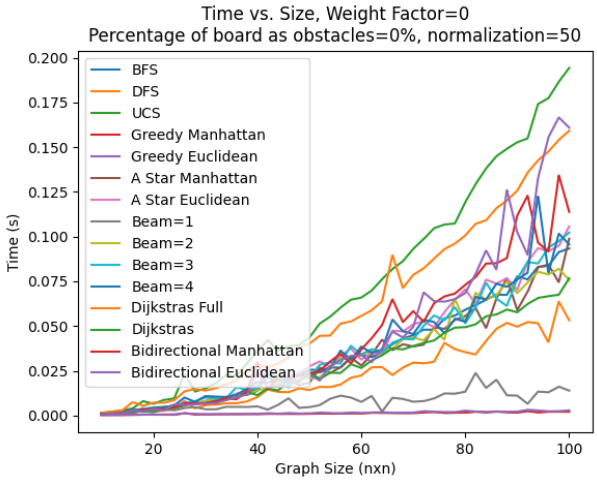
conditions. All of the comparisons are still valid though, because the only the constants in the algorithm running time are affected by the speed of the computer executing the code.

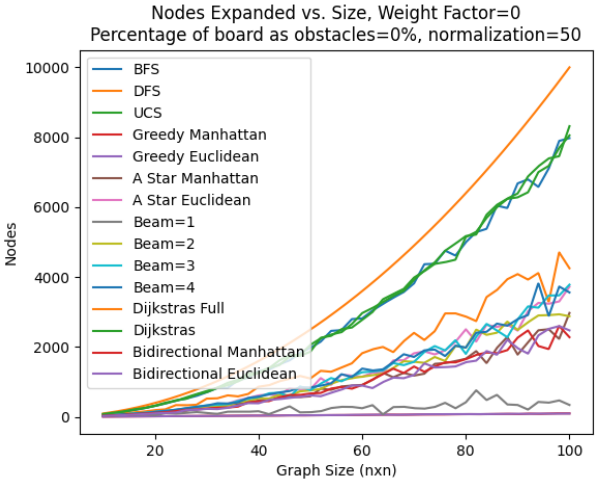
The first analysis is the most basic, but also the most important. Each algorithm was tested on boards starting from a size of 10 all the way up to 200, and the average time and nodes explored were plotted. The two figures below are the result.



This visualization shows that no algorithm is inherently worse than the others, but it does reveal some insight into the time and space used by each. Beam search with a beam length of 1 has a time complexity that rises much quicker than the others, and Bidirectional manhattan search does a good job of staying low in it's time complexity. In the space realm, BFS, UCS, and Dijkstra's all explore many more nodes than the others, and both Bidirectional searches explore the least amount of nodes.

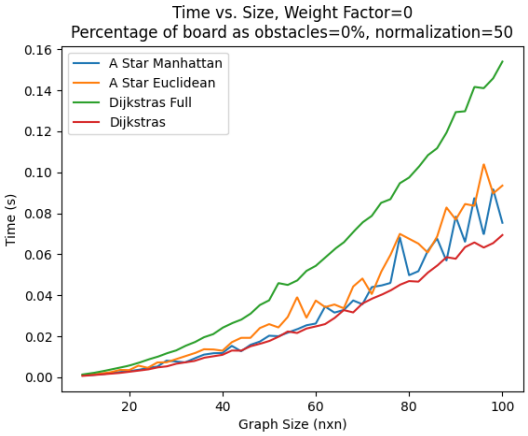
The next interesting result came from the analysis of how algorithms perform when there isn't a maze to search through. The following graphs show the performance of each algorithm on a graph that doesn't have any weight associated with it. In a board like this, the least number of iterations and nodes expanded will win because having cost only relate to each step significantly changes the performance of these algorithms. The following figures show the significant changes on a 10x10 board up to a 100x100 for each algorithm.

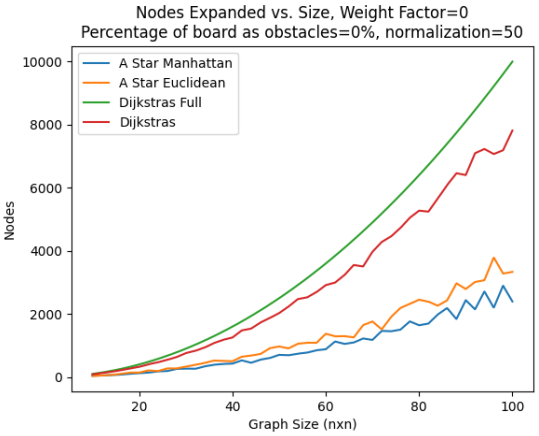


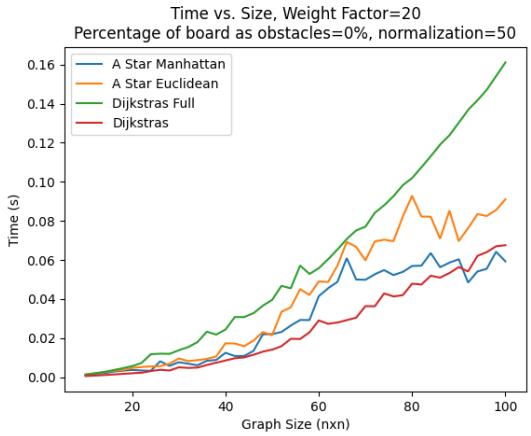


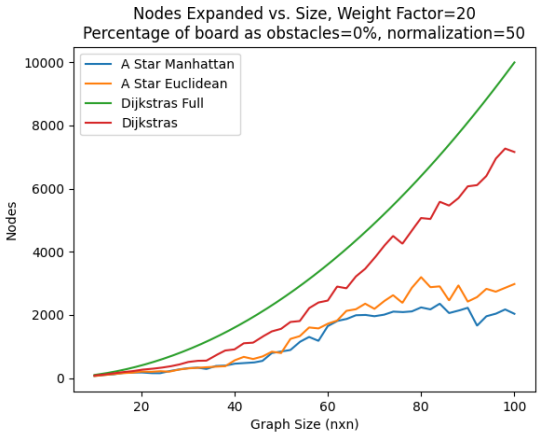
By far the most startling difference is that Beam search with a beam size of 1 is now very fast, and also expands very few nodes. This is because when there is no weight, the optimal path is the shortest path. Beam with a length of 1 works by expanding only one node each iteration, and this is the best node because it will have the lowest heuristic score, the path is nearly perfectly direct to the goal.

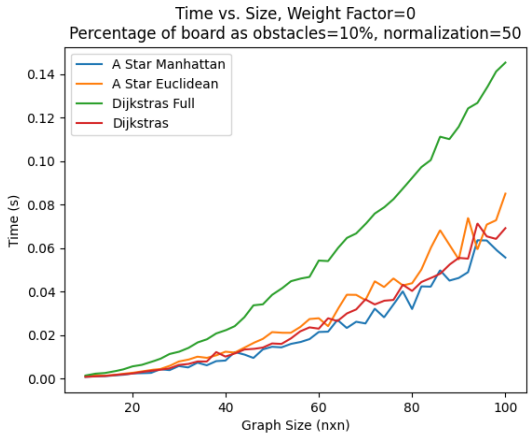
Another interesting result came from looking at the comparisons between just the two Dijkstra variants and the manhattan and euclidean versions of A*. These two algorithms always find the best cost path, so now it comes down to how fast do they do it, and likewise, how many iterations does this take. The figures below show the performance on 10x10 to 100x100 sized boards with all permutations of no obstacles, 10% obstacles, uniform weights and maximum weight of 20.

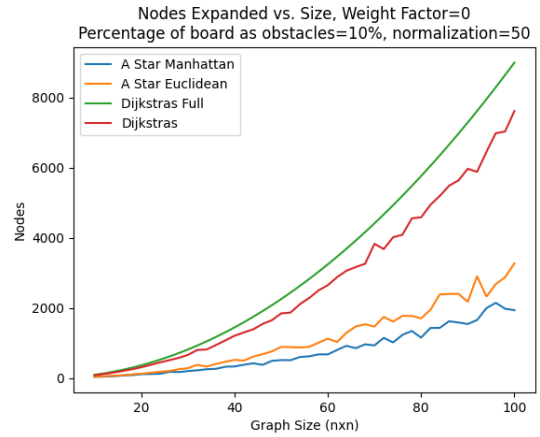


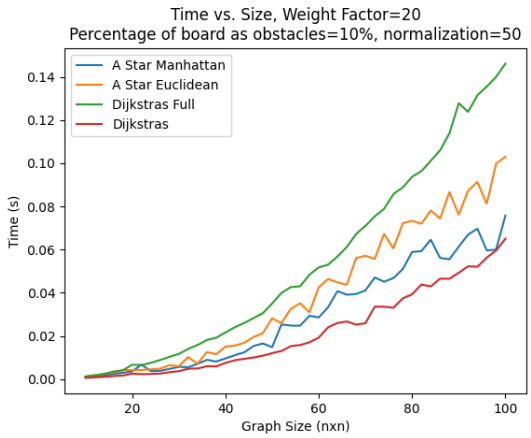


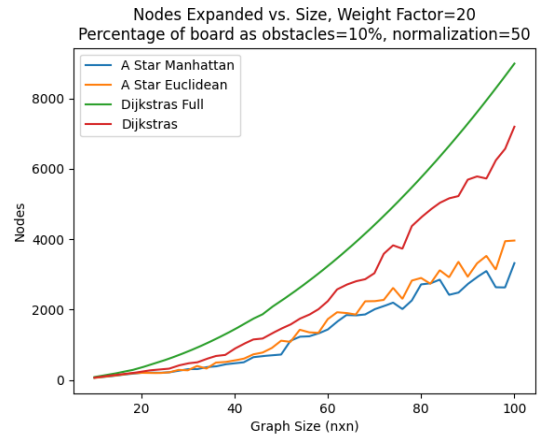












These results show an interesting trend, the maximum weight of the board is much more important in showing a difference in performance than the number of obstacles. In both tests where the board was a uniform weight, each algorithm except for Dijkstra's full exploration variant had a similar time increase. Once the weight of each node is taken into account, Dijkstra's end when goal encountered variant starts to beat out both A* versions in time.

Although the time complexity bounces around, the number of nodes expanded has a much more predictable behaviour. Dijkstra's full exploration variant always had to expand the most nodes, because it by definition explores every single node in the graph. This is what accounts for the significantly longer time over the other three algorithms. Next in the exploration list is the other Dijkstra version. This algorithm also explores many nodes as it has to explore until it reaches a goal, but it doesn't have to explore the entire board. Both A* algorithms explore roughly the same number of nodes each time, but significantly less than both Dijkstra algorithms. Why then is A* slower than Dijkstra in the time taken if it has to explore fewer nodes? The A* algorithm is more complicated than Dijkstra's, and therefore each iteration takes longer. The clear advantages of Dijkstra's algorithm come through in this analysis. It is fast. But space and memory capacity is sacrificed for this speed. A* may take marginally longer, but it uses nearly half the space. Both of these algorithms have their own specialties and unique advantages, but the most importantly, they will both always come to the same path cost, and this will be the optimal path.

Pathfinding and Human Thought

Pathfinding is an action that humans use every day without much difficulty. For example, walking through a crowded hallway or over an uneven sidewalk. These examples each have a certain cost to each step, such as avoiding collisions with other people and preventing tripping on rugged terrain. Our analysis simulates this kind of human thought process and reasoning by having our algorithms make decisions due to cost and heuristic value, and by adding in obstacles for our algorithms to find a path around. Algorithms that are uninformed such as BFS simulate the thought process of a human who is not informed about the changing distance of them to their goal while they move, while informed algorithms take this into account at every step.

Conclusion

After extensive analysis and testing of each algorithm individually, and then the algorithms against each other, there are some key takeaways. First, the uses for these algorithms are not fixed. If all one cares about is finding a solution, then there is no reason an uninformed search could not provide this solution. But if one needs the best path, A* or Dijkstra's may be the ones to choose to ensure the optimal solution is found. Second, all of these algorithms could probably benefit from the implementation of dynamic programming. This would especially add adaptability into the algorithms and they would be able to handle a board that periodically changes by reverting back to an older path. This would be interesting to explore, but there is the issue that a solution could potentially never be found, especially if the goal changes so frequently.

Next, one of the main lessons learned was during the implementation of these algorithms. Each algorithm may require different information from the graph inorder to correctly traverse the graph. This required a graph that would have sufficient data for each algorithm and lead to not implementing a few algorithms we had identified early in the project.

Lastly, there is so much more to explore in the field of path finding, and this was only able to touch the mere surface. The implementation and analysis of the algorithms in this project centered around something a human could visualize. Many problem spaces in the real world operate in much more complex realms that a human couldn't so easily visualize. There are so many possible applications of these algorithms, finding the cheapest path in a board only begins to realize the full potential.

Work Cited

Coppin, Ben. Artificial Intelligence Illuminated. United States, Jones and Bartlett Publishers, 2004.

Erickson, J. (2019). Shortest Paths. In Algorithms (pp. 278-282). Independently Published. doi:https://jeffe.cs.illinois.edu/teaching/algorithms/book/08-sssp.pdf

Even, Shimon. Graph Algorithms. United States, Cambridge University Press, 2011.

Frana, Phil (August 2010). "An Interview with Edsger W. Dijkstra". Communications of the ACM. 53 (8): 41–47. doi:10.1145/1787234.1787249

Pushpak Bhattacharyya. "Beam Search". Indian Institute of Technology Bombay, Department of Computer Science and Engineering (CSE). pp 39-40.