Final Project: Finding a Path Through Minneapolis

Abstract

In this paper, a modified Iterative-Deepening A* algorithm (IDA*) will be used to find the shortest path through Minneapolis. The IDA* algorithm written for this problem has two possible heuristics, a simple straight line heuristic and a ALTBestp heuristic. While ALTBestp is theoretically faster whith less memory usage, the results of the experiment show little difference in memory usage or runtimes. Therefore, the heuristics can be seen as having equal use.

1 Introduction

Computer science is a field that has been filled with many challenges as it has formed over the decades but one of its more interesting problems has been path-finding. Path-finding is a skill that comes naturally to humans and can be elusive to computers. This is due to a variety of reasons from the number of variables to consider to the computation power required for comparisons. Variables such as distance, time, the cost of travel, and places to visit can all factor into the process of path-finding. For humans, this is a simple matter of prioritizing different variables before attempting to find a path and then choosing a path which minimizes the variables best. The human brain can process all of this data relatively quickly and come up with a solution to satisfy those priorities easily. However, for computers and software, each variable needs to be chosen and added into the program. The data associated with the variable also needs to be stored and tracked as the program goes through the process of finding and comparing all the possible paths it comes across. For each additional variable accounted for, the matter of finding a path becomes greatly more complicated as each variable has data to be tracked and compared against. For example, when humans want to get from point A to point B, they dont start walking in a straight line between the points because there is most likely an obstacle in the way. Humans get directions and travel by way of street names and landmarks such as traveling down Simpson Drive or taking the first right past the grocery store. Computers need a great deal more detail because they need to be told about all of the relevant information. If the computer wasn't given the information about an obstacle it would attempt to move through it anyway

while a human could see it and work around the obstacle. A computer also needs to be constantly told what its current position is in relation to itself, the starting point, and the ending point depending on the variables the computer is judging the path by. Overall, the situation can become very complicated very quickly depending on both the methods used and the variables by which the path is judged. To simplify the process, we will be judging the path based solely on its distance traveled in this problem.

For this problem, the computer needs to find a path through Minneapolis, Minnesota using only the coordinates for the road network of Minneapolis. The data of the road network of Minneapolis comes from the file map.lisp, provided by James Parker (Parker. 2016). The road network is represented in the file where each line is a different road. Each line holds five numbers representing whether the street is a one-way or two-way street and the xy-coordinates of the start and end of the road. For this problem, the data will be used to find the shortest route through the Minneapolis road network starting and ending with specific coordinates.

2 Literature Review

Introduction

Finding the shortest route to a goal has been a significant problem in network analysis. A particular problem to focus on would be finding the shortest path through a network of roads in a city. It is possible to solve this kind of problem with a simple search algorithm, such as a variation of A*. A modified Iterative-Deepening A* algorithm would an efficient way to find the shortest path through a city. To understand why, it is useful to know some of the basics of path-finding, such as uninformed and informed search.

Uninformed Search vs Informed Search

Path finding has two basic types of searches, uninformed searches and informed searches. Uninformed searches, such as depth-first search, breadth-first search, and random walk, look for a goal knowing only the initial state and the goal state. Informed searches, such as hill-climbing, best-first search, and A*, have additional information the algorithms can use to find the goal. Depth-first search works well for graph traversal and travels deeper into the graph until it hits an endpoint (Chiong et.al. 2008). At that point, it backtracks to the most recent node visited, before searching the next node possible (Chiong et.al. 2008). Breadth-first search explores a graph or tree level by level, moving horizontally to search all the neighboring nodes before moving deeper to find the goal (Chiong et.al. 2008). Random-walk selects its successor at random so it can move vertically or horizontally at each step. The algorithm stops when goal node is first found (Chiong et.al. 2008). All of these uninformed algorithms stop searching when the goal is found for the first time. This can often leads to sub-optimal solutions. The informed searches are better at finding the optimal solution due to the additional information they can access.

Examples of informed searches are hill-climbing, best-first search, and A*. Hill-climbing is a search that continuously moves forward based on an evaluation equation (Chiong et.al. 2008). The search travels to a node with the least evaluation function cost and prunes the rest (Chiong et.al. 2008). The evaluation function used by Chiong was f(Node)=d(Node) where d(Node) is the straight line distance between the current node and the goal node (Chiong et.al. 2008). Best-first search is similar to hill-climbing but allows backtracking, which means when the search hits an endpoint it can backtrack to an earlier node which can be expanded further (Chiong et.al. 2008). It does this by maintaining a list of nodes to be further explored and expands on those nodes until the best solution is found (Chiong et.al. 2008). A* is an extension of best-first search which uses a different evaluation function. The evaluation function consists of the cost from the start node to the current node and the estimated cost from the current node to the goal node (Chiong et.al. 2008). Due to the evaluation functions conditions, A* will choose the shortest path which is likely to be optimal if the heuristic created by the evaluation function is both admissible and consistent (Chiong et.al. 2008). While the search portion is identical to best-first search, the heuristic, or evaluation function, of A* takes into account distance already traveled as well as how close it is to the goal node (Chiong et.al. 2008). Due to this, the A* search can find the optimal solution faster and more accurately than best-first search as well.

Chiong tested all of these algorithms on Borneo Island, Malaysia with the road network connecting 100 major cities and towns. The algorithms were tested with networks of 8, 15, 75 and 100 cities (Chiong et.al. 2008). The uniformed searches were always able to find the goal towns but usually found sub-optimal solutions, leaving shorter paths unfound. Hill-climbing worked well for a small number of cities where connections between cities were high but often got stuck in the local optima when the number of cities and towns became large because it could not backtrack. This was seen by a low average accuracy below 45

A* and its Variations

The search algorithm A* can be used to find the shortest path to a goal. The algorithm can also be expanded upon to create variations such as Hierarchical Path-Finding A* (HPA*) and Iterative-Deepening A* (IDA*). However, these variations work to varying degrees of success depending on what the graph looks like. A* is an informed search that includes an estimate of path-completion which allows it to often find the optimal path (Zeng et. al. 2009). The estimate is formed with additive cost measures of f(n) = g(n) + h(n) where f is the total cost, g is the distance from the start node to the current node, and h is the heuristic distance of the current node to the goal node (Dechter et. al. 1985). The heuristic built by this path estimate allows A* to work well with graphs with a geographical basis such as road networks. This is because the algorithm can take advantage of spatial coordinates and trim the search for the shortest path (Zeng et. al. 2009). As long as the heuristic is both admissible and consistent, the solution found will be optimal (Dechter et. al. 1985). However, the algorithm also has a large space complexity and can be forced to search arbitrary large regions of the search space (Dechter et. al. 1985). With the large overhead, other variations of A*, like HPA*, are more efficient than A*.

Hierarchical Path-Finding A* is one variation of A* that works well with extremely large networks such as the road network across the United States. HPA* works by breaking a graph into sub-problems of different clusters to reach a goal. The heuristics used are different depending on the level of clusters used (Botea et al. 2004). Only the current data is stored in memory which reduces the amount of storage used (Botea et al. 2004). For example, say you are traveling from home in Boston to your brothers house in Los Angeles. HPA* would find the shortest path by finding the best path from home to the interstate outside Boston, the best path from the interstate to Los Angeles, and lastly the best path from the interstate to your brothers house (Botea et al. 2004). This type of algorithm works best when there are clear boundaries the data can be grouped by which doesnt necessarily translate to a single city road network. The algorithm also has many more computations than A* in smaller searches in which case the overhead of HPA* is larger than the potential savings (Botea et. al. 2004). However, with extremely large networks, HPA* will have less memory usage than A* due to its clustering (Botea et. al. 2004). While IDA* may have more repeated computations, IDA* also has an improvement of memory usage when compared to A*.

Iterative-Deepening A* is a combination of depth-first search and A*. IDA* works by searching one connection level at a time before moving onto the next (Cazenave. 2006). The algorithm travels deeper into the graph until the given threshold is exceeded at which point it cuts off a branch and backtracks to the closest previous node to try again (Korf. 1985). If the goal is not found, a percentage of the threshold is iterated to become the new threshold and the algorithm searches again (Korf. 1985). IDA* returns the first solution it finds or failure if no solution is found (Korf. 1985). Since IDA* expands all nodes at a given cost before expanding nodes at a greater cost, the first solution it finds will be the solution with the lowest cost making it the optimal solution (Korf. 1985). IDA* can have significant overhead when compared to A* but there are modifications for IDA* which reduce the space requirements and decrease running time (Chakrabarti et. al. 1991, Cazenave. 2006).

Alterations to the way nodes are stored and the heuristic can have a large effect on the space requirements and runtime of IDA*. One modification Chakrabarti et. al. propose to reduce the overhead for finding successive nodes is to use a set of buckets to group together f-cost values which exceed the threshold cutoff for the iteration (Chakrabarti et. al. 1991). If no solution is found, this value is incremented to use for the next iteration (Chakrabarti et. al. 1991). This adaption is more difficult to implement and, while it reduces the required memory usage, does not create a significant advantage over A* (Chakrabarti et. al. 1991). Another improvement is changing the open node storage from linked lists to an array of stacks (Cazenave. 2006). The running time of IDA* is usually proportional to the number of nodes expanded and switching to an array of stacks reduces the number of expanded nodes (Edelkamo et. al. 2001). This change allows faster access time and smaller storage (Cazenave. 2006). Another improvement to implement is the ALTBestp admissible heuristic which Cazenave found to improve the runtime of IDA* since it took less time at each node (Cazenave. 2006). With the improvements, the space complexity and the running time of IDA* are reduced to make it more efficient than A* (Cazenave. 2006).

Conclusion

When a problem requires path-finding to find the shortest path, informed searches are more efficient at finding the optimal solution because they have access to additional information that uninformed searches do not. Of those informed searches, A* variations are more efficient to use because they estimate the complete distance from the start node to the goal node (Zeng et. al. 2009). HPA*, using clusters of data to finding possible paths, is more efficient than A* but less well suited for smaller search spaces where the overhead is greater than the possible advantage (Botea et. al. 2004). With the modifications from Cazenave, IDA* is also more efficient than A* (Cazenave. 2006). Between HPA* and IDA*, the more efficient algorithm will change depending on the search space and size. In the case of a network of roads within one city, IDA* would be easier to implement than HPA* and still be complete and optimal due to the lack of possible boundaries. Therefore, in a moderate road network like you would find in a city, a modified IDA* would be a better option then A* and HPA*.

3 Approaching the Problem

From previous research, it was decided to use Iterative-Deepening A* for a variety of reasons. IDA* would provide a complete and optimal search that would work well with the road network data provided. Early attempts were made to use lisp to adjust an already available IDA* algorithm code to be able to use the road network data since that was the language the data was presented with. The next attempt was to write an IDA* program based off of the basic algorithm for Iterative-Deepening A*.

The first attempts to solve the road network problem came from the language lisp. Publically available algorithms from *aima.cs.berkeley.edu*, a website attached to the textbook *Artificial Intelligence: A Modern Approach*, provided several already made functions that could be adjusted to work with the road network data (Russell et. al. 2010). The first attempt was to use the function tree-ida*-search from the file ida.lisp to find the shortest path through Minneapolis (Russell et. al. 2010). This could not work for several reasons. The first was that the data could not be put into a compatible form easily. To use the tree-ida*-search function, the data, which was presented as type map problem, needed to be a type iterative problem (Russell et. al. 2010). To convert the data into this type, it would have needed to have points created for each intersection, be reread into a function to make it into an iterative problem, and then used. The second reason was that the tree formed for the function would have had to have roads relisted repeatedly. This means each path would have created it's own branch which would have made the tree unnecessarily large and complicated. Alternatively, the problem could have been solved by altering a route-finding function from the file "route-finding.lisp" (Russell et. al. 2010). The difficulty with these functions was that the data needed to be specialized to work with the functions. The map problem type of the data needed to be converted to another type known was a route traveling problem. This couldnt be done with the data because the raw data was in the wrong form and would have been difficult to convert as extensively as was needed to work with the

functions. Unfortunately, while these methods were understood theoretically, it was more difficult to do in practice so another method was used.

When the problem could not be solved in lisp, a second option of writing a IDA* function in C was chosen. The language C was chosen due to its flexibility and easy usability. The algorithm was based off of a basic IDA* algorithm given in pseudocode by Wikipedia ("Iterative Deepening A*"). The algorithm written originally worked by starting at the coordinates given and calling a search function that searched for connecting road networks up to a specific boundary distance. This bound was set by the minimum distance found between the possible roads. The search function returned either a negative number, in which case the goal exit was found, or a positive number which represented the new boundary. The function itself was correct, however the map representation within the program was not.

The first representation of the road network was with a grid made up of an array. The road network was stored in a double array of structs called Map. Within the structs, known as Ngraphs, was an int h to represent the heuristic distance of the road end and an int array called neighbors to represent the roads the road was connected to. The x and y coordinates of the beginning of the road were the keys to the array Map. The value of each of these points was the struct containing that roads heuristic and the roads neighbors array indicated zeros for xy-coordinates which werent connecting roads and ones for xy-coordinates did contain connecting roads. This representation was possible for the small sample network of 6 roads but was incredibly inefficient. Only small portions of the grid were used so a great majority of each array was empty. The double array containing an array of equal size within each index of it also took too much memory and resulted in segmentation fault when used with a larger data set. This problem was solved by creating a single array of structs called Road to contain all of the road network information. The structs were made up of an int h to hold the heuristic, an int to represent whether the street is a oneway or twoway, and two Location structs which held the beginning and end coordinates of the road. This change reduced the amount of memory needed for storage and made it simpler to access the information as you did not need to iterate through both rows and columns of a double array. This was the representation used for gathering data for the problem.

4 Experiment Design and Results

Experiment Design

To solve the problem of finding the shortest path through the Minneapolis road network, a IDA* algorithm was written and tested using two different heuristics. The first was a simple heuristic which found the straight line distance between the end of the current road and the goal exit. The second was a heuristic proposed by Tristan Cazenave called the ALTBestp admissible heuristic (Cazenave. 2006). The experiment called to test these heuristics against each other and see if there was an improvement in runtime and memory usage as Cazenave proposed.

The first stage of this experiment was creating the IDA* algorithm to test the heuristics.

The IDA function was created so that the boundary started at zero and expanded every time the search function returned with a positive number. This became the new boundary. When the function returned negative, the exit goal had been reached and the path length could be shown. The search function works primarily through recursion. At the beginning of the function, the f-cost, the distance traveled plus the heuristic of the current road, is compared to the boundary. If the f-cost is larger than the boundary, the f-cost is returned. The current road index is then checked with the is-goal function where, if the function returns true, a negative one is returned to indicate that the goal has been reached. If neither of these if statements are true, then the function moves on to search the neighboring roads. This was done by taking a road and searching for connecting roads by looking through the road array for the roads whos starting coordinates match the ending coordinates of the current road. When it finds a match in the array, the distance between the end of the current road and the end of the new neighboring road is added together which gives the new distance traveled cost. The neighboring road index, the new distance traveled cost, and the distance boundary are used to make the recursive call to continue searching the road network. By using recursion, the search function travels through all possible paths within the boundary. Both the IDA function and the search function remain constant through the experiment. The difference is which heuristic is used.

The problem to solve in this experiment is to find the shortest path through Minneapolis using an Iterative-Deepening A* algorithm. The differences to test in the experiment are the effects of using a simple straight line heuristic or Cazenaves ALTBestp heuristic. The simple heuristic is a standard straight line goal distance heuristic. The heuristic for each road is found by using the make-h function. This function takes in the coordinates of the current road. It calculates the straight line distance between the ending coordinates of the current road and the coordinates of the exit goal. This heuristic result is then stored in the roads array and can be called upon in the search. The second heuristic comes from Cazenaves article *Optimizations of data structures, heuristics and algorithms for path-finding on maps* (Cazenave. 2006). The heursitc is called ALTBestp heuristic. It is admissible and therefore useable in for the IDA* algorithm. The heuristic works by computing the distance to all points from a given point and then computing the heuristic. Below is a proof by Cazenave for the admissibility of the ALT heuristic which is the basis for the ALTBestp heuristic.

If d(X,Y) is the shortest path between X and Y, and if the distances are precomputed between pPos, the current node is cPos, and the goal node is at gPos, then the following inequalities are true:
(1) d(cPos,pPos) $\leq d(cPos, gPos) + d(gPos, pPos)$
(2)$d(cPos, gPos) \leq d(cPos, pPos) + d(pPos, gPos)$
(3)$d(gPos, pPos) \leq d(gPos, cPos) + d(cPos, pPos)$

From 1 and 3 the following can be shown, respectively:
(4) d(cPos,gPos) $\geq d(cPos, pPos)d(gPos, pPos)$
(5)$d(gPos, cPos) \geq d(gPos, pPos)d(cPos, pPos)$

Given that d(gPos,cPos) = d(cPos,gPos) and abs is the absolute value:
(6) d(gPos,cPos) $\geq abs(d(gPos, pPos)d(cPos, pPos))$

Therefore, an admissible heuristic would be
(7) h = abs(d(gPos,pPos) d(cPos,pPos))

For the ALTBestp heuristic, the heuristic chooses for h the maximum value out of the ALT values given by each pPos. For the experiment, the values were calculated with all roads outside of the current road.

Results

Both the simple heuristic and the ALTBestp heuristic were tested with a sample road network of 6 roads and the Minneapolis network of 1357 roads. The sample network started at the road at index 4 with the coordinates (1,1). The goal exit was at the coordinates (8,8). The Minneapolis network started at index 533 with the coordinates (390,6319). The goal exit was at the coordinates (1046,10500). Below are the tables for both the sample network and the Minneapolis network. For both networks, the simple and ALTBestp heuristics were tested by 100 trials each.

Table 1: Sample Network Data for Simple and ALTBestp Heuristics

Heuristic Type	Percentage Optimal	Average Memory Usage (bytes)	Average Runtime (seconds)
Simple	100	5436	0.000000
ALTBestp	100	5436	0.000000

Table 2: Minneapolis Network for Simple and ALTBestp Heuristics

Heuristic Type	Percentage Optimal	Average Memory Usage (bytes)	Average Runtime (seconds)
Simple	100	5476	0.000000
ALTBestp	100	5476	0.000000

5 Result Analysis

The results of this experiment can be seen in tables 1 and 2 above. From a numerical point of view, there is a large difference between the size of the sample road network and the Minneapolis road network. The sample road network only contains 6 roads while the Minneapolis network contains 1357 roads. While both heuristics resulted in a 100 percent optimality for both networks, the difference in network size resulted in remarkably small differences in both memory usage and runtime.

Memory Usage Analysis

Memory usage can vary depending on storage methods, the size of memory needed for storage, and the amount of computations needed. In this experiment, the changes that could cause a change in memory usage would be the memory for storage and the memory for computations. There was a change in memory usage due to size differences. As seen in table 1, the memory usage of the small sample network averaged 5436 bytes between the simple and ALTBestp heuristics. The Minneapolis network, seen in table 2, had an average memory usage of 5476 bytes between the simple and ALTBestp heuristics. This 40 byte difference may be due to the larger size of the roads array. Separately, the two tables show similar results for the different networks.

The memory usage results for the small sample network and the larger Minneapolis network are very similar. For the sample network, the average memory usage for both the simple heuristic and the ALTBestp heuristic was the same. Both heuristics resulted in an average memory usage of 5436 bytes. For the Minneapolis network, the average memory usage was 5476 bytes for both the simple heuristic and the ALTBestp heuristic. The lack of difference between the two heuristic results could be due to the structure of how the heuristic is computed during a network search. Both the simple and ALTBestp heuristic is computed during the search process and have similar methods of computation. While the ALTBestp is required to compute more values and make comparisons between those values, the difference of memory requirement between the two functions is likely minimal. The difference, or lack thereof, between the heuristic runtimes may be due to a similar reason.

Runtime Analysis

The runtimes of the sample network and larger Minneapolis network were very low. For both networks, the average runtime was 0.000000 seconds. Measurements could only be taken with precision up to the 6th decimal place so it can be assumed that the run time was at least less than 0.0000005 seconds, or 0.5 microseconds. The average runtimes for the simple and ALTBestp heuristics for the sample network and the Mineapolis network could only be measured as less than 0.5 microseconds due to this measurement precision. It is likely there is a difference between at least the runtimes between the sample network and the Minneapolis network but it could not be measured due to lack of precision.

6 Summary

The problem of this article was to find the shortest route through a Minneapolis road network given the beginning and end points of each road. After comparing variations of A*, such as General A*, Hierarchical Path-finding A*, and Iterative-Deepening A*, it was decided that Iterative-Deepening A* would be used to solve the problem. During the experiment, two separate heuristics were used for the IDA*; a simple straight line heuristic and Cazenava's ALTBestp heuristic. Both the sample sample road network and the larger Minneapolis network resulted in similar results for optimality, memory usage, and runtime.

Both the sample road network and the Minneapolis network had similar results from this experiment. For both networks, out of 100 trials per heuristic, the simple and ALTBestp heuristic found the optimal path 100 percent of time. The difference in memory usage between the simple and ALTBestp heuristics was zero. The 40 byte difference between the sample network and the Minneapolis network is likely due to the roads array size. There is also no noticeable difference between runtimes for the two heuristics. Both were measured at less than 0.5 microseconds so the difference is likely negligible. All of this adds up to the simple heuristic and the ALTBestp heuristic being equally good heuristics up to the size of a 1357 element array. In the future, this experiment could be redone with the two-way street data or a larger data set. While the data for whether or not a road is a two-way was provided and stored, it was not used to find a path through Minneapolis. In this case, the shortest path found did not travel on the same street twice.

References

[1] Botea, Adi and Müller, Martin and Schaeffer, Jonathan. "Near optimal hierarchical path-finding." *Journal of game development* 1, no. 1 (2004): 7-28.

[2] Cazenave, Tristan. "Optimizations of data structures, heuristics and algorithms for path-finding on maps." In *Computational Intelligence and Games, 2006 IEEE Symposium on*, pp. 27-33. IEEE, 2006

[3] Chakrabarti, P.P. and Ghose, S. and Sarker, S.C. and Sarker U.K. Reducing Reexpansions in Iterative-Deepening Search by Controlling Cutoff Bounds. *Artificial Intelligence* 50 (1991): 207-221

[4] Chiong, Raymond, Jofry Hadi Sutanto, and Wendy Japutra Jap. "A comparative study on informed and uninformed search for intelligent travel planning in Borneo Island." *Information Technology, 2008. ITSim 2008. International Symposium on* vol. 3, pp. 1-5. IEEE, 2008.

[5] Dechter, Rina and Pearl, Judea. "Generalize Best-First Search Strategies and the Optimality of A*." *Journal of ACM* vol. 32, no. 3 (1985): 505-536.

[6] Edelkamo, Stefan and Korf, Richard E. and Reid, Michael. Time Complexity of Iterative-Deepening-A* *Artificial Intelligence* 129 (2001): 199-218.

[7] -. "Iterative deepening A*." *Wikipedia.com*, last modified 20 March 2016, https://en.wikipedia.org/wiki/Iterative_deepening_A*

[8] Korf, Richard E. "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search." *Artificial Intelligence* vol. 27, no. 1 (1985): 97-109.

[9] Parker, James. "map.lisp." *University of Minnesota, College of Science and Engineering.* 12 April 2016, http://www-users.cselabs.umn.edu/classes/Spring-2016/csci4511/assignments/project/map.lisp.

[10] Russell, Stuart and Peter Norvig. *Artificial Intelligence: A Modern Approach, 3rd Addition.* New Jersey: Pearson Education Inc, 2010

[11] Zeng, W and R. L. Church Finding shortest paths on real road networks: the case for A*. *International Journal of Geographical Information Science* vol. 23, no. 4 (2009): 531-543.