Artificial Intelligence Coursework I

Felix Vorländer[1] [40407962]

[1] Napier University, Edinburgh EH10 5DT, UK
40407962@live.napier.ac.uk

Abstract. The following work aims to describe and compare various path finding algorithms. Initially, I will describe the general procedure of the algorithms in an intelligible way. Afterwards, I will go into more detail about the properties the algorithms inhibit and compare advantages and downsides that come with them in different use cases.

1 Research of route finding algorithms

1.1 Breadth-first search

There are various approaches to solve the problem of finding routes within given maps. Algorithms that are capable of doing so are implemented frequently and reliably in software we use every day. If algorithms are used to solve such problems, they usually aim to present a result with certain properties. For example, it could aim to find an *optimal route* (the shortest possible route to a certain point) or to be *complete* (always be able find a route to a certain point on a map and be able to determine when there is no existing route to that point) [1].

The Breadth-first search algorithm is complete and can be optimal under certain conditions. It is designed to traverse a tree architecture until it finds a predefined goal node. This allows certainty about the shortest route being found as soon as the goal node is reached as longs as "the path cost is a nondecreasing function of the depth of the node"[2]. In other words: As long as the graph's edges do not have discrete weights or only the number of moves actually matter, the algorithm will always find the best solution.

We presume that a map like environment is presented to the algorithm in form of a graph. To allow a breadth-first search to function the graph needs to be converted to a tree-like structure. This can be achieved by defining the start node as the root and adding branches for each connected node. This can then be repeated for each node, ignoring branches that would lead back to nodes that were already created (see **Fig. 1.** A graph with its corresponding breadth-first tree.**Fig. 1**). Note how the connection between node two and four is ignored for example and how it would cause an infinite loop if it was not.

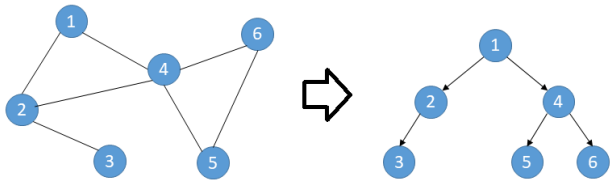


Fig. 1. A graph with its corresponding breadth-first tree.

After the tree is created the algorithm will now visit each node row by row. Let's say the given goal node is five. After checking if the start node (one) is already the goal node, it would continue to check node two and four in the second row. In the third row it would check node three before stopping at the goal node five. It is then capable of creating the corresponding path to that node by repetitively going back to the current nodes parent until it reaches the start node again, remembering the route it took (see **Fig. 2**). This route can then be presented as the solution [2]. Note how node six has never been checked and the optimal route could still be found. If the algorithm reaches the last node of the tree without finding the goal node it can certainly determine that no possible route exists for the given task. Thus, the algorithm is complete.

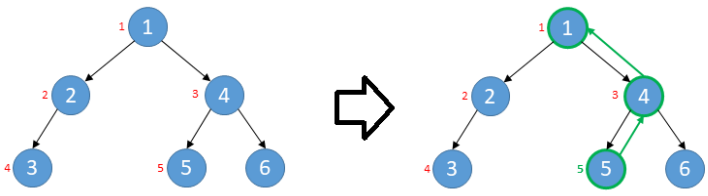


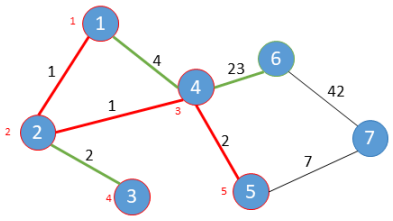
Fig. 2. The procedure of the breadth-first search.

However, if the graph's edges were weighted, the algorithm would not be capable of always finding an optimal solution. This is because the first route to the goal node is not necessarily the cheapest. Note how the tree's architecture ignores possible detours that could potentially have lower traveling costs.

1.2 A* Algorithm

Since A* can be interpreted as an extension to Dijkstra's problem definition of "[finding] the path of minimum total length between two given nodes [3]", I will initially focus on Dijkstra's algorithm from his original paper released in 1959. In the field of Artificial Intelligence the addressed version of Dijkstra is typically known as the uniform cost search, although "[t]he two algorithms […] are logically equivalent [4]."
After initializing unknown cost values as infinite [4] Dijkstra will check nodes in a graph and remember the total costs and routes it takes to get there based on the weights of the checked edges. The algorithm starts by checking the costs to every neighbor of the start node. From that point forward, it will proceed in a *greedy* [5] way thus always choosing the next node to visit (i.e. *close*) based on it having the lowest total cost to be reached out of the list of all checked nodes. The chosen node's edges will then be checked as before, possibly updating the total cost values of the nodes neighbors as well as the corresponding routes. As soon as this procedure causes the goal node to be visited (not just checked), the algorithm can stop and present an optimal solution [3].

Fig. 3 shows a suitable example for this. The red numbers describe the order in which the nodes are visited. The green line describes checked nodes and edges while the red line describes the resulting solution after termination. Note how the cost value to reach node four was updated. Initially, it was checked as a neighbor of node one with the cost value of four. Next, node two (the cheapest relative to node one) was visited and the costs to reach node four were checked again. Since it is cheaper to reach node four from node one the costs were updated and the resulting route has been remembered.



Node	Cost
1	0
2	1
3	3
4	~~2~~
5	4
6	25
7	INF.

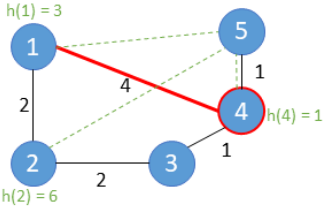
Fig. 3. Terminated Dijkstra Algorithm with node visit order and cost history.

In addition to this procedure the A* algorithm uses a heuristic function to estimate the costs of future routes. This value is added to the costs of a node before choosing the next visit. A typical realization would be to check how close a node is to the goal node hence presuming that the nodes linear distance is able to indicate the quality of leading to a suitable path. Affirmatively, this is often the case in typical map-like environments.

However, the optimality of A* depends on the heuristic function to be **admissible** and thus "never overestimating the cost to reach the goal [2]." Since a heuristic function is always admissible if it is **consistent** [2] the admissibility of a linear distance heuristic on a graph that uses displacement as its weights can be easily proven. For every node p_n, n describing the position in the shortest possible route and $d(p_1, p_2)$ describing the distance between two nodes, the heuristic function will always hold the following and is therefore consistent:

$$h(p_1) \geq d(p_1, p_2) + h(p_2). \text{ [6]}$$

Fig. 4 allows a better understanding of how the A* algorithm uses the heuristic value to make a more deliberate decision. The green line presents the heuristic values of checked nodes while the red line presents the route that was chosen based on that value added to its edge's cost. Note how A* chooses node four while Dijkstra would have chosen node two first. Also note, how the heuristic value from the start node one can never be surpassed, thus making the heuristic function consistent and A* optimal for the given graph.



Node	Cost + h(p)
1	0
2	2+6 **= 8**
3	INF
4	4+1 **= 5**
5	INF

Fig. 4. A decision of the A* algorithm based on edge weights and heuristic values.

1.3 Beam Search

Algorithms that are remembering metrics for a large amount of nodes can struggle with bigger search problems because of their memory consumption [7]. Especially for more complex search problems the previously described algorithms can be memory heavy and time inefficient. Beam search allows to "[reduce] memory consumption […] at the cost of finding longer paths [8]". This constitutes Beam search as a suitable candidate for specific environments with possibly insufficient memory space for complete search trees, such as machine translation systems for example [9].

The standard version of the Beam search is associated with the Breadth-first search strategy [10] as described in 1.1. It begins by generating a search tree just like the named algorithm does but it only allows a predefined branching factor, called **beam width**. This causes non-promising nodes to be **pruned** (reducing size removing sections of the tree) and only a specified number of nodes to be expanded. The decision about what nodes are pruned is based on a problem-specific heuristic. This causes only the most promising branches to be expanded [10] until the goal node is reached.

In conclusion, if the beam width was set to one, the algorithm would act like a greedy search. However, if no limit was set, the algorithm would act like a complete breadth-first search. Note that the memory usage is always determined by the beam width multiplied with the tree depth that is needed to find a solution.

By possibly pruning routes that lead to an optimal solution the algorithm cannot ensure optimality. This behavior also results in "no guarantee that beam search will find […] any solution at all, even when a solution exists [10] ". The algorithm is therefore incomplete as well.

There are many variations of the Beam search algorithm. For example, some versions exchange the breadth-first with a depth-first strategy and ensure that a specific memory size is not exceeded by simply deleting least-promising nodes as soon as the **beam** (list of expandable nodes) is filled [11].

Fig. 5 shows an example of a tree that is pruned by the standard Beam search algorithm. Note how possible routes are pruned even though there is no certainty about them not leading to a goal state.

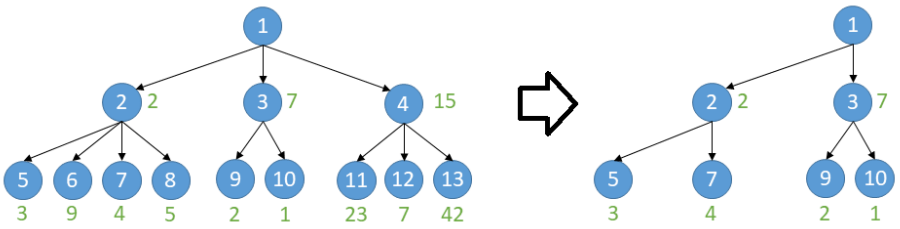


Fig. 5. Example of a pruned Beam Search tree based on heuristic values with a beam width of two.

2 Method Evaluation

The given problem of finding a route in a distance-weighted graph with 1000 nodes introduces an interesting example of application for the presented algorithms. Especially if the algorithm should also be able to determine when there is no possible route within the graph at all. The completeness of an algorithm can only be assured if it can handle that situation and not cause infinite loops or similar behavior that does not lead to a valid solution output. Additionally, if a graph exceeds a certain amount of node connections a brute-force approach to solve it becomes impossible. The amount of possible solutions grows exponentially and each solution must be stored together with its resulting edge weights and then be compared to find the shortest possible route. An example for how fast this can reach memory limits can be seen in **Fig. 6**. Every third node can either choose to progress to the top or the bottom row. Similar to a binary number this results in 2^ndifferent possible routes where n is the number of decisions to go up or down. For a graph with 1000 nodes that results in about 2^{333} possible solutions that must be compared and evaluated even though the graph structure is fairly simple. Note that there are just $\sim 2^{260}$ atoms in the observable universe [12] which makes the task of storing every route physically impossible and that a typical graph can have a lot more nodes connected to each other in a more complicated manner which is why the problem must be approached in a more deliberate way.

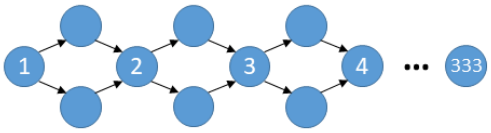


Fig. 6. A simple connection graph with about 1000 nodes

The Breadth-first search is suited didactically to present an example of how algorithms are able to solve route finding problems. It approaches the task in a reasonable way that is easy to understand and it is able to secure optimality as well as completeness for certain graph structures. However, its need to check every single node until it finds the solution can make it time consuming and memory heavy. Even though the algorithm can ensure completeness its inability to work with weighted graphs make it unfit in many cases. When the algorithm is used to find a route to a distance-weighted graph with 1000 nodes it can find a solution and even a usual home computer is able to handle the memory that is needed to check and store the amount of nodes. However, the only assurance the algorithm gives is the usage of a minimum amount of steps to reach the goal which does not have to correlate with the length of the resulting route. Since the weights of the graph are ignored, optimality cannot be assured. Since optimality and speed have a very high priority in the predefined task of this coursework the algorithm is unsuited.

Similarly, the Beam search algorithm cannot assure optimality either. Since a 1000 node graph can usually be stored in the internal memory of a typical home computer the memory saving approach of the Beam search is unappealing for the given task. In addition, it may not even find any solution at all if the tree is pruned in an unlucky way. The strength of Beam search lay in its memory saving approach which makes it suitable for more specific use cases on small memory devices or specifically predefined graph environments. If the problem is defined by finding a shortest route as fast as possible on a graph that needs a reasonable memory effort another approach should be preferred.

In contrast, the more memory heavy approach of storing every node of the graph in different lists and update their costs on a regular basis seems more applicable. This is what the Dijkstra (Uniform Cost Search) algorithm does and as a tradeoff it delivers a complete and optimal solution in a reasonable amount of time. The advancement of the algorithm with a heuristic value allows to accelerate the path finding time even more, which is why A* is a very suitable approach to solve the 1000 node graph problem. Since the weights of the graph are all defined as linear distances, the heuristic function can be constructed in an admissible way by using the direct distance to the goal node as *h(n).* Even if the heuristic function was not admissible and the solution therefore not always optimal, A* is still complete and delivers a fast solution that is “good enough” in most cases. This is why it is widely used in real life use cases to find routes quickly and reliably in videogames or actual navigation systems [13].

References

1. Coppin, B. 2004. Artificial intelligence illuminated. Jones & Bartlett Learning: 79–80.
2. Russell, Stuart; Norvig, Peter (2010) [1995]. Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall 81-82, 94-96
3. Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. Numerische Mathematik, 1(1): 269–271.
4. Felner, A. 2011. Position paper: Dijkstra's algorithm versus uniform cost search or a case against Dijkstra's algorithm, SOCS: 47–51.
5. Black, Paul E. 2005. "greedy algorithm". Dictionary of Algorithms and Data Structures. U.S. National Institute of Standards and Technology (NIST). https://xlinux.nist.gov/dads/HTML/greedyalgo.html Retrieved 15 November 2018.
6. Dechter, R., Pearl, J. 1985. Generalized best-first search strategies and the optimality of A*, J. ACM 32: 505–536.
7. Korf, R. 1993 Linear-space best-first search. Artificial Intelligence, 62(1): 41–78.
8. Furcy, D., Koenig, S. 2005. Limited discrepancy beam search. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI): 125–131.
9. Tillmann, Christoph and Hermann Ney. 2003. Word reordering and a dynamic programming beam search algorithm for statistical machine translation. Computational Linguistics, 29(1): 97–133.
10. Zhou, R., & Hansen, E. A. 2005. Beam-stack search: Integrating backtracking with beam search. In Biundo, S., Myers, K., & Rajan, K. (Eds.), Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling. ICAPS, AAAI Press: 90–98.
11. Rich, E. Knight, K. 1995. Artificial Intelligence, McGraw-Hill, New York.
12. Schutz, B. 2003. Gravity from the Ground Up, Cambridge University Press, pp. 361–362.
13. Santoso, A., Setiawan, A. 2010. Performance analysis of Dijkstra, A* and Ant algorithm for finding optimal path case study: Surabaya citymap. International Conference, Medical Image Computing.