A* Algorithm

Yue Zhou

August 4, 2018

1 Abstract

In this paper, we are going to discuss how A* works as a best-first search method. Best-first search is a search algorithm which returns the nodes satisfying the searching goal. A* algorithm gives a glance of optimal solution to find the shortest path in a weighted graph. Heuristic function is the core of optimization method in A* algorithm. In real problems, A* is efficient on both time and space complexity, meanwhile it grants the solution.

2 Introduction

In a weighted graph, we are often asked to search a path from start node to end node with the minimum distance. A* algorithm is an algorithm to find the path with least weight by using heuristic function. Comparing to searching every possible path in a graph, such as Dijkstra's algorithm, A* improves performance in such following ways. From time complexity perspective, A* algorithm indicates the search direction.Hence, some nodes and paths are not needed to be searched. From space complexity perspective, A* algorithm does not store every node from the graph, it only stores the "currently working" nodes into a set, so called open list.

3 Estimation function

In the A* algorithm, the main idea is to apply the function f(n) = g(n) + h(n), while g(n) represents the cost of path from start node to current node, h(n) represents heuristic value, and f(n) represents the estimated total cost of path from start to reach final destination, via the current node n.
Particularly, "the heuristic function h(n) informs the search direction from current vertex to the final destination. It generates an informed way of interpreting the direction from current node to the final destination."[1] It is interpreting because the value of h(n) is not necessarily equal to the actual value from current node to destination, such as f(n). The value of h(n) has to be less or equal to the actual cost from current node n to the final destination. Thus, we always have h(n)<=d(n-1,n)+h(n-1).(d is the distance from node n-1 to n.) In an actual problem, there are two piece of considerations that needs to be balanced for a heuristic function. One side is the cost of deriving the value of heuristic function at the current node. The other side is how accurate the value of heuristic function at current node is.
In a short word, the estimation function f(n) is equal to the sum of the actual cost function g(n) from start node to the current node, and the estimated cost function h(n) from current node n to the final destination.
(For the purpose of convenience, we use term "goal" to substitute term "final destination" in the rest of paper.)

4 Usage of heuristic function

From Amit's thoughts on pathfinding, we have a glimpse of heuristic function in real examples.[5] Suppose that we are playing a game based on a two-dimensional space. The space is consisted of blocks. We are trying to travel from a start block to an end block with the minimum cost.(There

are many blocks in between start and end node.) We have three types of blocks: ground, ocean, and air. The ground block has travel cost of 1, the ocean block has travel cost of 2, and the air block has travel cost of 3. In order to simplify it, we build x and y axis on this two-dimension board.
When we are looking for the shortest path, there are some strategies that can be applied. Firstly, heuristic function does not need to be global; it can be local. For example, there are only ground type in a bunch of connected blocks (maybe a few blocks of other types).Then we can consider all the estimated distance between two blocks in this local area as 1. Secondly, we may have many heuristic functions h1(n), h2(n)...,hn(n) for different local areas. When we calculate the global heuristic function, we can choose the minimum heuristic function among all the heuristic functions as our global heuristic function. Thirdly, there are different types calculation methods for heuristic function depending on how many directions we can travel from current node.
Manhattan Distance:
Manhattan distance defines 4 directions of movement from the current block. If we define D as the global minimum cost to move from a block to an adjacent block. Then heuristic function is defined as: h(n) = D * (dx+dy). dx=abs(node.x-goal.x).dy=abs(node.y-goal.y).
Diagonal Distance:
In diagnoal distance, we can travel toward 8 directions from a block. Define D2 as the minimum cost to move diagonally from one block to another. Then the heuristic function is defined as: h(n)= D * (dx+dy) + (D2- 2*D)*min(dx,dy).
dx=abs(node.x-goal.x).dy=abs(node.y-goal.y).
Euclidean distance:
In euclidean distance, we can travel toward any direction from a block. The heuristic function is defined as: h(n)= D * sqrt(dx*dx + dy * dy).
dx=abs(node.x-goal.x).dy=abs(node.y-goal.y).
In this case, we have figured out there are different ways to define a heuristic function in A* algorithm. Based on the definition of space, we choose the corresponding method to calculate heuristic function.

5 Features of A* Algorithm

A* algorithm solves the pathfinding problem. The reason why A* has a better performance than Dijkstra's algorithm is because we have a indicated searching direction in A* algorithm by heuristic function. First of all, heuristic function is admissible. By saying "Admissible",it means for each node n, the estimated distance f(n) never exceeds the actual distance from start to current node n, g(n). According to Russell,"a heuristic function is said to be admissible if it never overestimates the cost of reaching the goal, i.e. the cost it estimates to reach the goal is not higher than the lowest possible cost from the current point in the path.[2]" In another word, an admissible heuristic function never overestimates the distance from current node to the goal.
A* algorithm's feature heavily depends on the property of heuristic function. Heuristic function improves the performance in the following ways: Optimality, completeness, accuracy, and execution.
Completeness means A* algorithm always returns a solution if there is, and it returns all the possible solutions(selected by heuristic function). It is complete because the algorithm does not end until all the nodes are driven from the open list. A* algorithm grants the ability of finding all possible solutions chosen by heuristic function from the start vertex to the goal.
Optimality means A* algorithm always returns the best result from all the possible results as solution. In order to have the best result, we have to ensure the heuristic function is admissible.
Accuracy means the the node we have stored in open list contains the nodes that are on the least cost path. In other words, the A* algorithm always returns the best possible solution based on how heuristic function is defined.
Execution time means the running time is different when we use different heuristic function. Some heuristic functions can filter more nodes than other heuristic functions, so the it shrinks execution time. Some heuristic functions only converge faster when the node is approaching the goal.
In order to operate A* algorithm, heuristic function demands a given goal(where to go) and a method to calculate heuristic function(estimated distance from current node to the goal). Unlike Dijkstra's algorithm, a given goal must be known before we can operate A* algorithm. Furthermore, we need to construct an heuristic function to estimate the distance from current vertex to

goal.
In A* algorithm, goal has to be given. Since heuristic function is the function to determine the distance from the current node to goal. Without given goal, there is no value of heuristic function. Thus, there is no application of heuristic function. In addition, A* turns to be dijktra's algorithm in this case.
In A* algorithm, a node may be processed by multi times. It happens when we take a node out from closed list to open list. (This particular case will be analyzed in pseudocode section.) That is, we still process each node less times in A* algorithms compared to Dijkstra's algorithm.
The required condition for heuristic function is to calculate the heuristic value from the current node to the goal. By given the goal and the current node, there is an simplified way to get an estimation on distance rather than calculating the real distance. If we still have to find the real distance for each node, then it is no difference from dijkstra's algorithm.
In another word, A* algorithm is an informed version of Dijkstra algorithm. The major part that differs A* from Dijkstra is heuristic function. In A*, heuristic function supplies information when we make a choice of which node to go in the next step. Meanwhile, Dijkstra algorithm treats every node equally and tracks every possible path.
According to Korf, "some authors use the definition of " best-first search" to categorize a search with heuristic function that estimates the distance from current node to the goal. The type of search, exploring the shorter estimated distance first, is called greedy best-first search, or pure heuristic search."[3] It shares the similar methodology with greedy algorithm because they both select the "best candidate from the current condition.

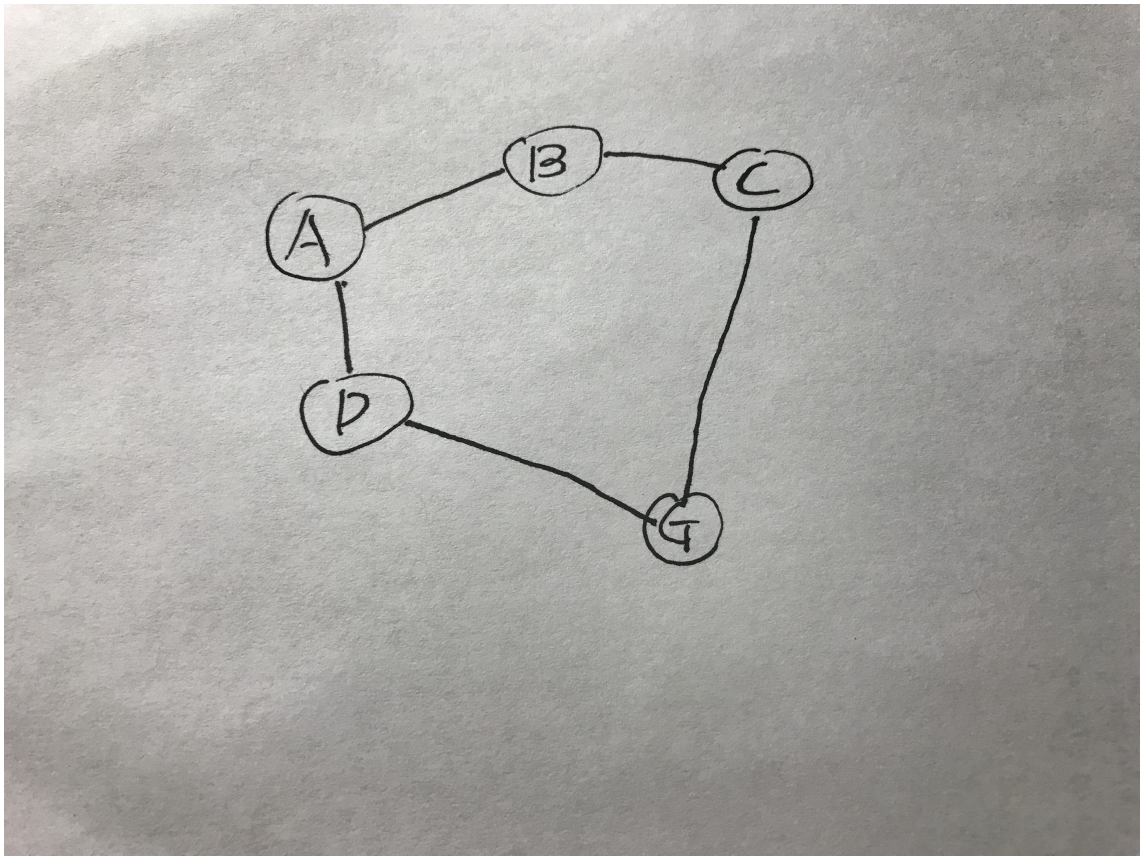
6 Pseudocode

```
1 Put node_start in the OPEN list with f(node_start) = h(node_start) (initialization)
2 while the OPEN list is not empty
3     Take from the open list the node node_current with the lowest
4         f(node_current) = g(node_current) + h(node_current)
5    if node_current is node_goal we have found the solution; break
6     Generate each state node_successor that come after node_current
7     for each node_successor of node_current
8        Set successor_current_cost = g(node_current) + d(node_current, node_successor)
9        if node_successor is in the OPEN list
10           if g(node_successor) ≤ successor_current_cost continue (to line 20)
11       else if node_successor is in the CLOSED list
12           if g(node_successor) ≤ successor_current_cost continue (to line 20)
13           Move node_successor from the CLOSED list to the OPEN list
14       else
15       node_successor to the OPEN list
16           Set h(node_successor) to be the heuristic distance to node_goal
17
18       Set g(node_successor) = successor_current_cost
19       Set the parent of node_successor to node_current
20
21     Add node_current to the CLOSED list
22
23 if(node_current != node_goal ) exit with error (the OPEN list is empty)
```

Here it comes analysis of the pseudocode of A* algorithm. Firstly, we create an open list to store all the current nodes that are being processed. (On a two-dimensional graph, it is the "outer edges" of all the processed nodes.)
Secondly, we take the least cost node out from the open list. The cost of each node is determined by the estimation function f(n) = g(n) + h(n).
Thirdly, we compare the current node with the goal, to see whether they are the same node. If they are, then we are done. If they are not the same node, we will start processing the rest steps: In the code line 6, we process all the successors of the current nodes.(All the successors are equally the outer edge of the processed nodes in two-dimension graph.) In the code line 8, we calculate all the actual distance for each successor.(Though the original definition of successor is the least

greater node than the current node, here it means the node connecting to the current node and has the least greater estimation cost.)
From line 10 to line 16, that is in the "if" condition, we try to add $node_successor$ to the open list.(We add the node successor either from closed list or not from closed list, depending on value of function g($node_successor$). At the end we adjust the heuristic distance of $node_successor$. For the outer "if" condition, from line 9 to line 21, we adjust the cost from start node to successors as the actual cost, g(n). Then add all the current nodes to the closed list.
Particularly, the content from line 11 to line 13 gives us a condition of moving a node's successor out of closed list to open list. It happens only when the current node is in open list and the node successor is in closed list.
Line 23 grants the pseudocode is not an endless loop. If open list is empty and the current node still has not reached goal, we should execute.

7 Particular example



For example, we have the weighted graph above and we try to find the optimal solution of the shortest path from node A to node G. In the graph, the cost for each path is equally to be 10.
First, we need to build an heuristic function. One way to define the heuristic function is to take real physical distance from each node to node G. So I pull out a ruler out and measure the distance and get h(A) = 5, h(B) = 4.3, h(C) = 4, h(D) = 3.9. Thus, we have the estimated function values as f(A) = h(A) + g(A) = 5 + 0 = 5, f(B) = h(B) + g(B) = 4.3 + 10 = 14.3; f(c) = h(C) + g(C) = 4 + 20 = 24. f(D) = 3.9 + 10 = 13.9. Thus, from step one, we can tell that f(B) has the least value between the value of f(B) and the value of f(C) (B and C are two nodes in the open list), so we choose B as our second node. This is a very simply example of A* algorithm. It shows the effectiveness of heuristic function and the logic in A* algorithm. Even in a more complex case, this methodology still applies.

8 Time complexity and Space complexity

The time complexity of A* depends on the heuristic. In the worst case, heuristic function has neither contribution nor guided effect toward reducing the time complexity. Thus, the algorithm explore the entire graph, just like Dijkstra algorithm. A specific example is the heuristic function is zero everywhere. In this case, running A* is as same as running Dijkstra, and we still go through every possible path and record the shortest path. Thus, the time complexity is as same as the time complexity in Dijkstra, $O(E + VlgV)$.
However, in most applications, the running time is shorter, because heuristic function plays a role in the algorithm. For instance, heuristic function indicates there is a path from start node to goal with zero cost(which is unlikely happening in applications), then we only need to choose this path as our solution. The time complexity is $\theta(1)$.
The average running time of A* algorithm is $O(E)$. In A* algorithm, it always choose the path with the least value estimated function as the path to continue. In addition, A* algorithm only exams the paths that are "relatively less in cost" than the paths that are "relatively more in cost". Thus, A* algorithm only exams certain paths, without examining every path in the graph. The running time of A* is $O(V)$.
We only store part of the nodes in the open list and closed list, and the stored nodes are less than the total number of nodes in graph. The space complexity is $O(V)$.

9 When to use A*

When we are looking for an acceptable path or even the shortest path. There is no need to use brutal force method to list out all the possible paths, so we can simply apply A* algorithm and take advantage of heuristic function. Importantly, an acceptable solution and optimal solution have different requirement for heuristic function. An optimal solution requires a precise heuristic function.
Another case to apply A* algorithm is when the number of paths is enormous and finite. In this case, the number of steps it takes from the start node to the goal is unknown. Thus, we do not want to run brutal force algorithm to search the optimal solution because the running time might be sufficiently large. Thus, we can create an heuristic function and apply A* algorithm to this problem to get an estimation of real running time. If the running time is manageable, then we can approximate the running time before we apply brute force algorithm to this problem.
When heuristic function is not accurate enough, then A* algorithm might not be a good choice. For instance, if the heuristic value is larger than the actual distance from current node to the goal, then the "exceeded distance" will cause bias on making the choice of which path to go. In this case, we urge an improvement on heuristic function.
In conclusion, it is up to the programmer to choose use A* algorithm or some other algorithms instead. If the following conditions are satisfied, then we can apply A* algorithm.
1. Graph G(V,E) has no negative distance edge, d(u,v).
2. we have an admissible heuristic function.
3. A start node and a goal(end node).

10 Conclusion

A* is one of the best-first search algorithms. Compared to the traditional type of searching, A* algorithm has optimization on both time and space complexity.(No other algorithm searches less vertices in a graph than A* algorithm.) The core of A* algorithm, heuristic function, has contribution on completeness, optimality, accuracy, and excution time. Thus, building heuristic should be the concentrated part when A* algorithm is applied.

11 Citation

[1] Poole, David L., and Alan K. Mackworth. Artificial Intelligence: Foundations of Computational Agents. Cambridge University Press, 2010. https://artint.info/html/ArtInt_56.html
Russell, S.J.; Norvig, P. (2002). Artificial Intelligence: A Modern Approach. Prentice Hall. ISBN

0-13-790395-2.

Korf, Richard E. (1999). "Artificial intelligence search algorithms". In Atallah, Mikhail J. Handbook of Algorithms and Theory of Computation. CRC Press. ISBN 0849326494. [4]http://mat.uab.cat/~alseda/MasterOpt/AStar-Algorithm.pdf [5]http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html