4. A* HEURISTIC SEARCH OF A GIVEN GRAPH

08/07/2019

INTRODUCTION

A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

$$f(n) = g(n) + h(n)$$

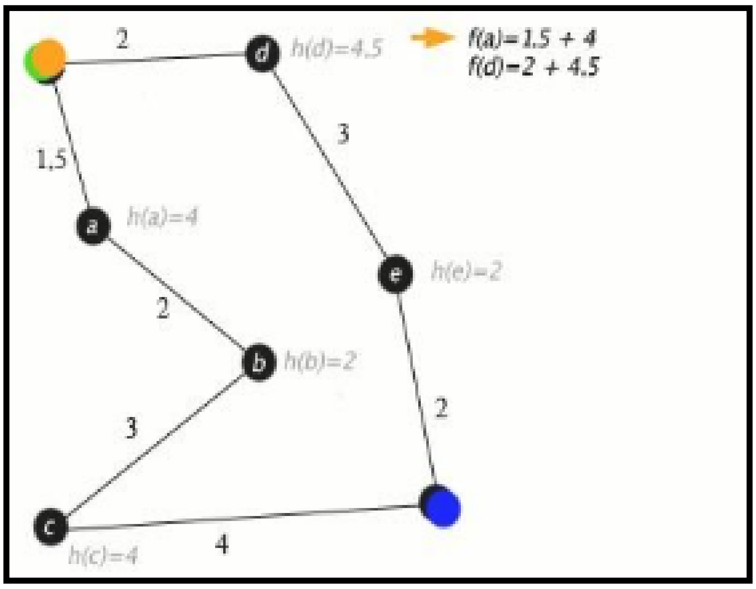
where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal. A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended. The heuristic function is problem-specific. If the heuristic function is admissible, meaning that it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal.

IMPLEMENTATION USING DATA STRUCTURE

Typical implementations of A* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the *open set* or *fringe*. At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty).[a] The f value of the goal is then the cost of the shortest path, since h at the goal is zero in an admissible heuristic.

Example

An example of an A* algorithm in action where nodes are cities connected with roads and h(x) is the straight-line distance to target point:



The A* algorithm also has real-world applications. In this example, edges are railroads and h(x) is the great-circle distance (the shortest possible distance on a sphere) to the target. The algorithm is searching for a path between Washington, D.C. and Los Angeles.

There are a number of simple optimizations or implementation details that can significantly affect the performance of an A* implementation. The first detail to note is that the way the priority queue handles ties can have a significant effect on performance in some situations. If ties are broken so the queue behaves in a LIFO manner, A* will behave like depth-first search among equal cost paths (avoiding exploring more than one equally optimal solution).

When a path is required at the end of the search, it is common to keep with each node a reference to that node's parent. At the end of the search these references can be used to recover the optimal path. If these references are being kept then it can be important that the same node doesn't appear in the priority queue more than once (each entry corresponding to a different path to the node, and each with a different cost). A standard approach here is to check if a node about to be added already appears in the priority queue. If it does, then the priority and parent pointers are changed to correspond to the lower cost path. A standard binary heap based priority queue does not directly support the operation of searching for one of its elements, but it can be augmented with a hash table that maps elements to their position in the heap, allowing this decrease-priority operation to be performed in logarithmic time. Alternatively, a Fibonacci heap can perform the same decrease- priority operations in constant amortized time.

Dijkstra's algorithm, as another example of a uniform-cost search algorithm, can be viewed as a special case of A* where for all *x*.[10][11] General depth-first search can be implemented using A* by considering that there is a global counter *C* initialized with a very large value. Every time we process a node we assign *C* to all of its newly discovered neighbors. After each single assignment, we decrease the counter *C* by one. Thus the earlier a node is discovered, the higher its value. Both Dijkstra's algorithm and depth-first search can be implemented more efficiently without including an value at each node.

APPLICATIONS OF A* HEURISTICS

A* is commonly used for the common pathfinding problem in applications such as video games, but was originally designed as a general graph traversal algorithm. It finds applications to diverse problems, including the problem of parsing using stochastic grammars in NLP. Other cases include an Informational search with online learning

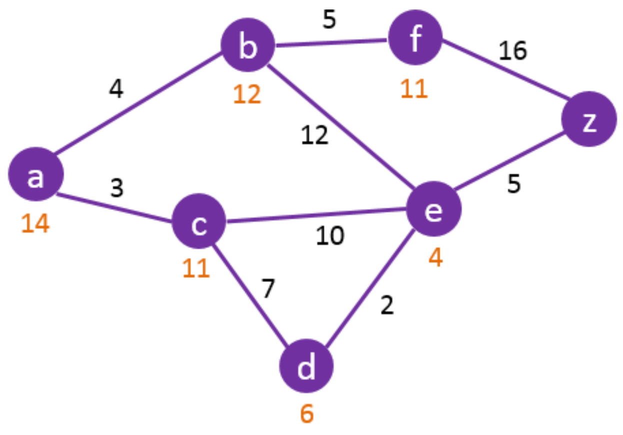
RELATIONS TO OTHER ALGORITHMS

What sets A* apart from a greedy best-first search algorithm is that it takes the cost/distance already traveled, $g(n)$, into account.

Some common variants of Dijkstra's algorithm can be viewed as a special case of A* where the heuristic for all nodes; in turn, both Dijkstra and A* are special cases of dynamic programming. A* itself is a special case of a generalization of branch and bound and can be derived from the primal- dual algorithm for linear programming.

PYTHON PROGRAM TO IMPLEMENT A* HEURISTICS

Write a program in python to implement A* heuristic search in the given graph below.

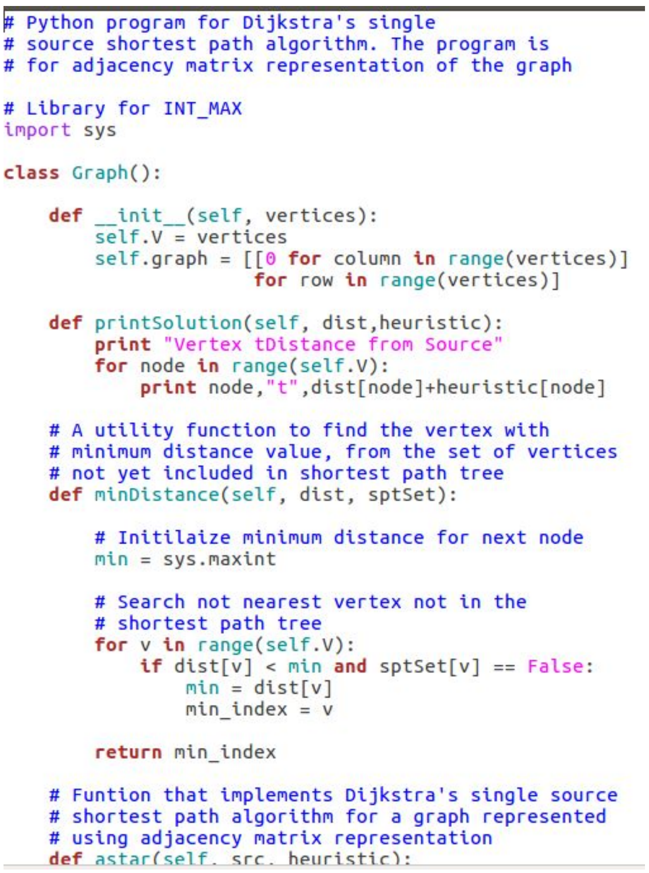


A* Search Algorithm

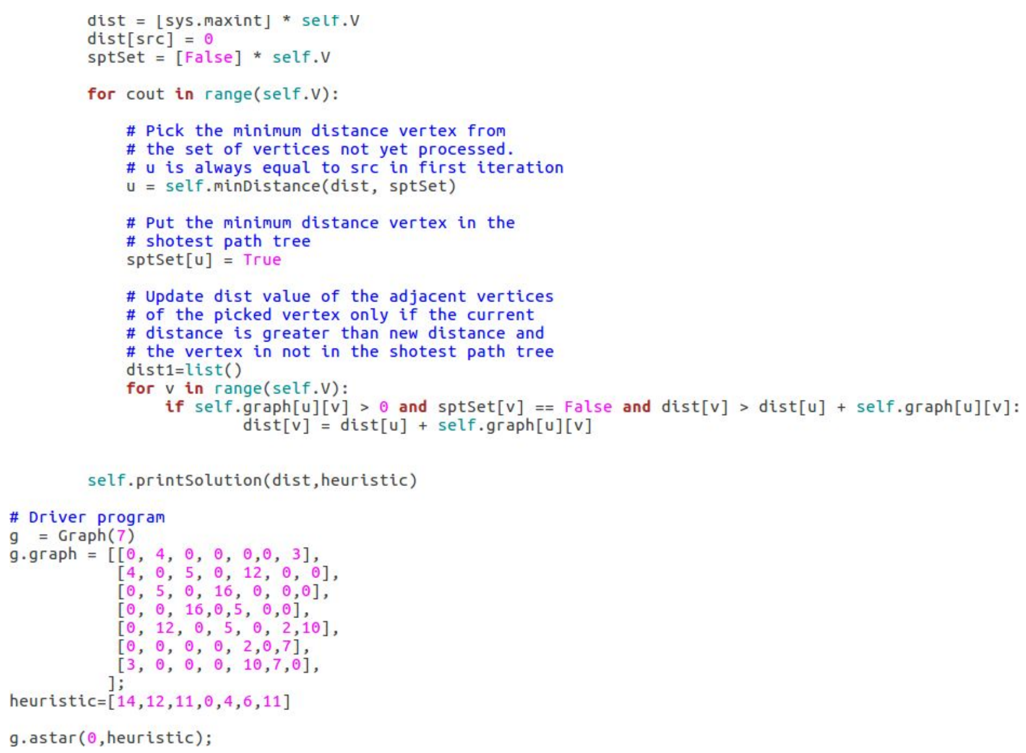
What is the shortest path to travel from A to Z?

Numbers in orange are the heuristic values, distances in a straight line (as the crow flies) from a node to node Z.

CODE-



```
# Python program for Dijkstra's single
# source shortest path algorithm. The program is
# for adjacency matrix representation of the graph

# Library for INT_MAX
import sys

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printSolution(self, dist,heuristic):
        print "Vertex tDistance from Source"
        for node in range(self.V):
            print node,"t",dist[node]+heuristic[node]

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minDistance(self, dist, sptSet):

        # Initilaize minimum distance for next node
        min = sys.maxint

        # Search not nearest vertex not in the
        # shortest path tree
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]
                min_index = v

        return min_index

    # Funtion that implements Dijkstra's single source
    # shortest path algorithm for a graph represented
    # using adjacency matrix representation
    def astar(self, src, heuristic):
```



```
        dist = [sys.maxint] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):

            # Pick the minimum distance vertex from
            # the set of vertices not yet processed.
            # u is always equal to src in first iteration
            u = self.minDistance(dist, sptSet)

            # Put the minimum distance vertex in the
            # shotest path tree
            sptSet[u] = True

            # Update dist value of the adjacent vertices
            # of the picked vertex only if the current
            # distance is greater than new distance and
            # the vertex in not in the shotest path tree
            dist1=list()
            for v in range(self.V):
                if self.graph[u][v] > 0 and sptSet[v] == False and dist[v] > dist[u] + self.graph[u][v]:
                        dist[v] = dist[u] + self.graph[u][v]


        self.printSolution(dist,heuristic)

# Driver program
g  = Graph(7)
g.graph = [[0, 4, 0, 0, 0,0, 3],
           [4, 0, 5, 0, 12, 0, 0],
           [0, 5, 0, 16, 0, 0,0],
           [0, 0, 16,0,5, 0,0],
           [0, 12, 0, 5, 0, 2,10],
           [0, 0, 0, 0, 2,0,7],
           [3, 0, 0, 0, 10,7,0],
          ];
heuristic=[14,12,11,0,4,6,11]

g.astar(0,heuristic);
```

OUTPUT-

```
Vertex tDistance from
  Source 0 t 14
 1 t 16
 2 t 20
3 t 17
4 t 16
5 t 16
6 t 14
```

ADVANTAGES AND DISADVANTAGES OF A * HEURISTICS

A* is just like Dijkstra, the only difference is that A* tries to look for a better path by using a heuristic function which gives priority to nodes that are supposed to be better than others while Dijkstra's just explore all possible paths.

Its optimality depends on the heuristic function used, so yes it can return a non optimal result because of this and at the same time better the heuristic for your specific layout, and better will be the results (and possibly the speed).

It is meant to be faster than Dijkstra even if it requires more memory and more operations per node since it explores a lot less nodes and the gain is good in any case.

Precomputing the paths could be the only way if realtime results are required and the graph is quite large, but usually the required node is found before traversing the entire graph.

COMPLEXITY OF A* HEURISTIC

The time complexity of A* depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) *d*: $O(b^d)$, where *b* is the branching factor (the average number of successors per state).[20] This assumes that a goal state exists at all, and is reachable from the start state; if it is not, and the state space is infinite, the algorithm will not terminate.

The heuristic function has a major effect on the practical performance of A* search, since a good heuristic allows A* to prune away many of the b^d nodes that an uninformed search would expand. Its quality can be expressed in terms of the *effective* branching factor *b**, which can be determined empirically for a problem instance by measuring the number of nodes expanded, *N*, and the depth of the

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d.$$

solution, then solving[21]

Good heuristics are those with low effective branching factor (the optimal being *b** = 1). The time complexity is polynomial when the search space is a tree, there is a single goal state, and the heuristic function *h* meets the following condition:

$$|h(x) - h^*(x)| = O(\log h^*(x))$$

where h^* is the optimal heuristic, the exact cost to get from *x* to the goal. In other words, the error of *h* will not grow faster than the logarithm of the "perfect heuristic" h^* that returns the true distance from *x* to the goal.

CONCLUSION

DIJKSTRA vs A* HEURISTIC

Dijkstra's algorithm doesn't pay any attention to which direction it is going,this may cause to explore vertices are not even close to the target vertex,thus increases the time to find the shortest path. A* is just an optimization of Dijkstra for a specific source and for a specific target. A* uses heuristic which just happens to be an estimation of how far we're from the target vertex,this heuristic is what gives A* its real power because having of a sense of directionality of exploration could dramatically increases the time to find the shortest path.