Shortest Path using **Dijkstra's and $\mathcal{A}^*$ Algorithm**

Arjun Reddy Baddam
Computer Science
Indiana State University
Terre Haute, IN, USA

December 9, 2013

Abstract

Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree.

$\mathcal{A}^*$ algorithm is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between points, called nodes.

1 Dijkstra's Algorithm

1.1 Introduction

In 1959 a three pages long paper entitled A Note on Two Problems in Connexion with Graphs was published in the journal Numerische Mathematik. In this paper Edsger W. Dijkstra - then a twenty-nine-year-old computer scientist - proposed algorithms for the solution of two fundamental graph theoretic problems: the minimum weight spanning tree problem and the shortest path problem. We shall say a bit more about this paper later on in this discussion. For the time being suffice it to say that today Dijkstra's Algorithm for the shortest path problem is one of the most celebrated algorithms in computer science (CS) and a very popular algorithm in operations research (OR).

In the literature this algorithm is often described as a greedy algorithm. For example, the book Algorithmics (Brassard and Bratley

[1988, pp. 87-92]) discusses it in the chapter entitled Greedy Algorithms. The Encyclopedia of Operations Research and Management Science (Gass and Harris [1996, pp. 166-167]) describes it as a "... node labelling greedy algorithm ... " and a greedy algorithm is described as "... a heuristic algorithm that at every step selects the best choice available at that step without regard to future consequences ..." (Gass and Harris [1996, p. 264]).

1.2 Applications

Real time applications of Dijkstra's Algorithm are:

- Traffic Information System
- Mapping(Map Quest, Google Maps)
- Routing Systems

1.3 History

Dr. Dijkstra says "One morning I was shopping in Amsterdam with my young fianceè, and tired, we sat down on the cafe terrace to drink a cup of coffee and I was thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention. In fact, it was published in 1959, three years late. The publication is still readable, it is, in fact, quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame"[4].

Dijkstra's Algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1956 and published in 1959. Dijkstra's original algorithm does not use a min-priority queue and runs in $O(|V|^2)$(where $|V|$ is the number of vertices).The idea of this algorithm is also given in (Leyzorek et al. 1957). The implementation based on a min-priority queue implemented by a Fibonacci heap and running in $O(|E| + |V|log|V|)$(where $|E|$ is the number of edges) is due to (Fredman Tarjan 1984). This is asymptotically the fastest known

single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights.[2]

1.4 Algorithm

procedure DIJKSTRA ALGORITHM(V) ▷ Graph G, with edges E to the form (v1, v2) and vertices V, and source vertex, s
    dist array to distances from the source to each vertex.
    prev: array to pointers to preceding vertices.
    i: loop index
    F: list of finished vertices
    U: list of heap unfinished vertices
    for $i \leftarrow 0, v-1$ **do**
        dist[i] = INFINITY
        prev[i] = NULL
    end for
    dist[s]=0
    while F is a missing vertex **do**
        pick the vertex, v, in U with the shortest path to s
        add v to F
        for each edge of v, (v1, v2) **do**
            if dist[$v1$] + length($v1, v2$) ¡ dist[$v2$] **then**
                dist[$v2$] = dist[$v1$] + length($v1, v2$)
                prev[$v2$] = $v1$
                possibly update U, depending on implementation
            end if
        end for
    end while
end procedure
[2]

1.5 Example of Dijkstra's Algorithm

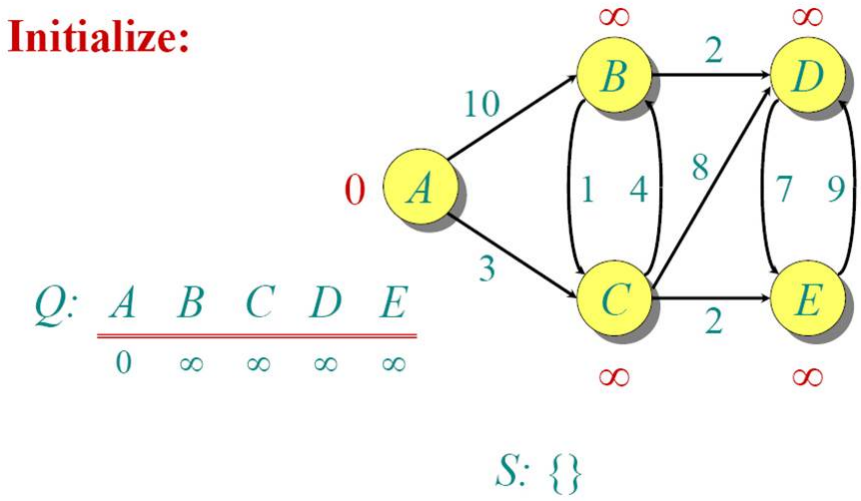


Figure 1: Example

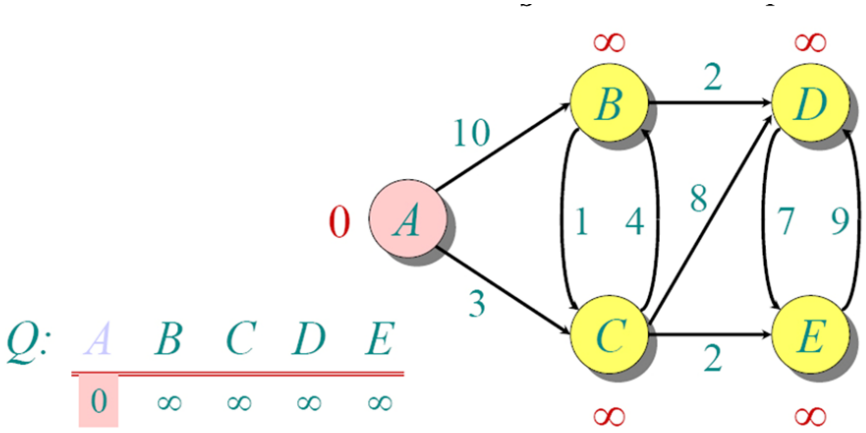


Figure 2: Example

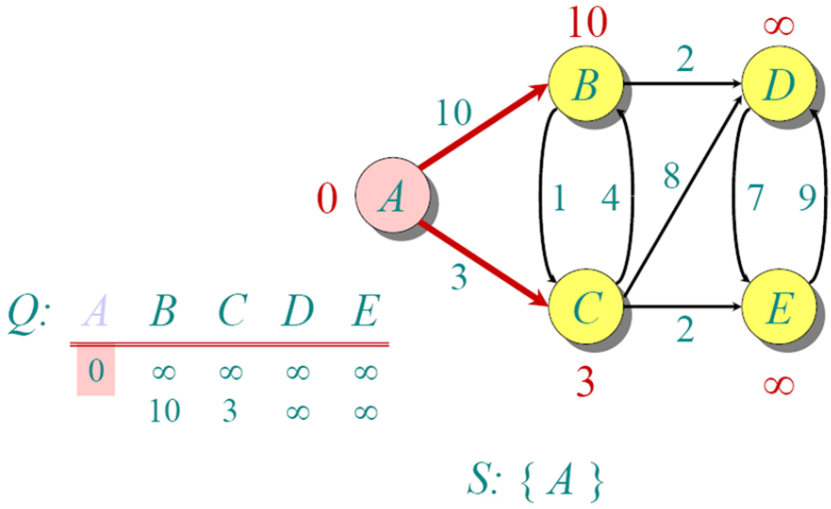


Figure 3: Example

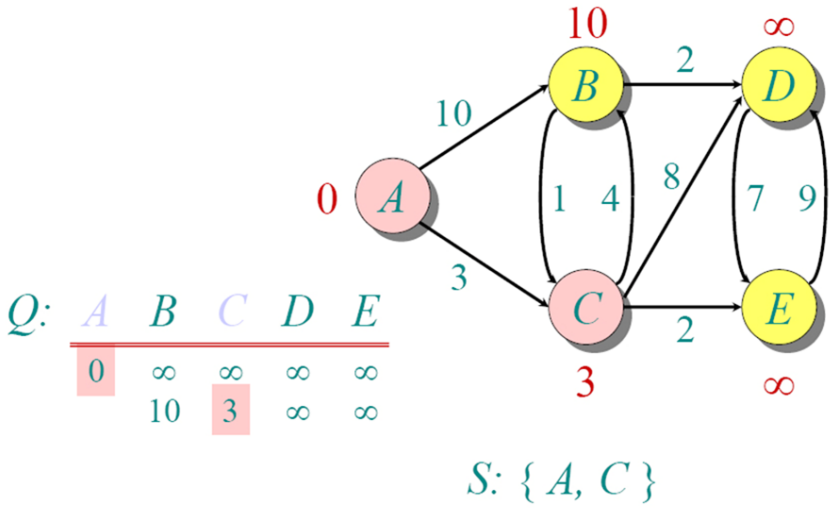


Figure 4: Example

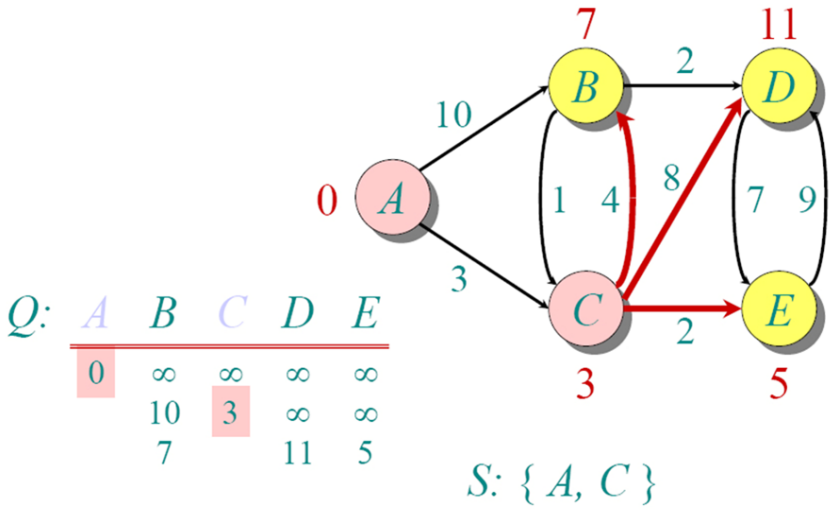


Figure 5: Example

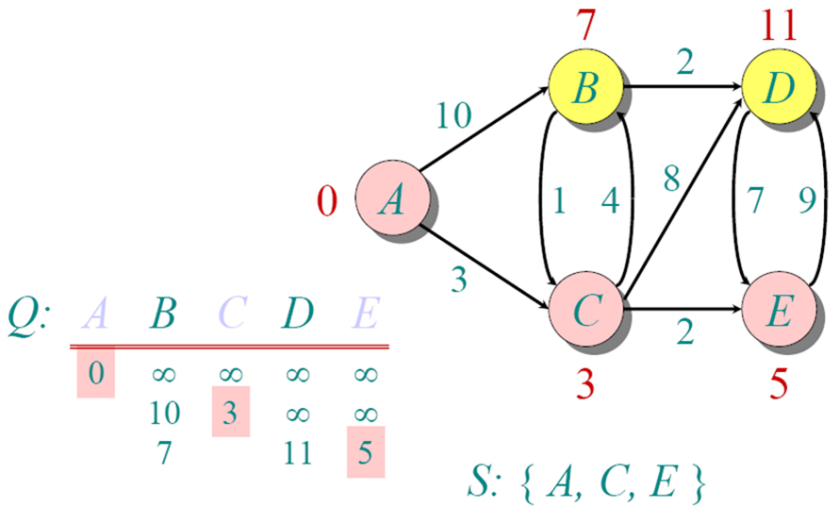


Figure 6: Example

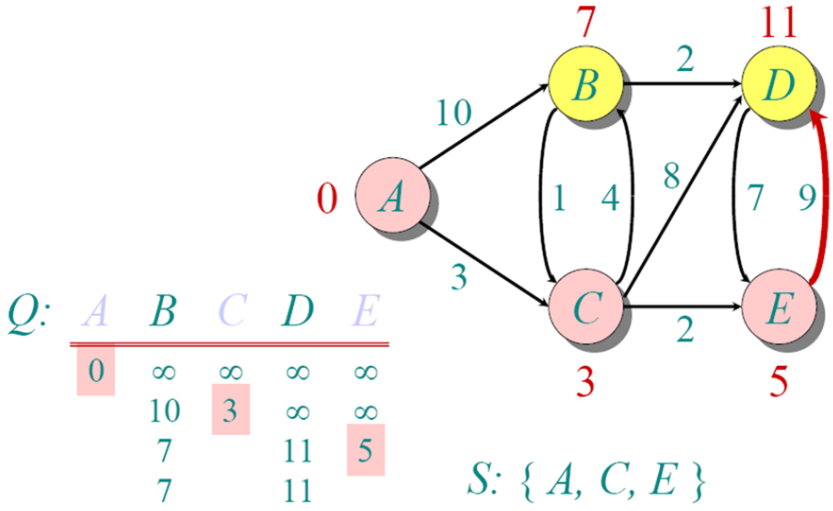


Figure 7: Example

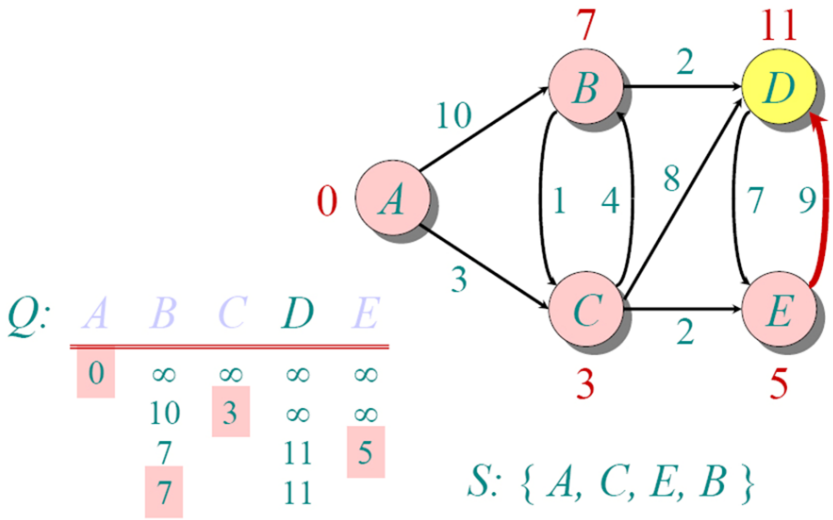


Figure 8: Example

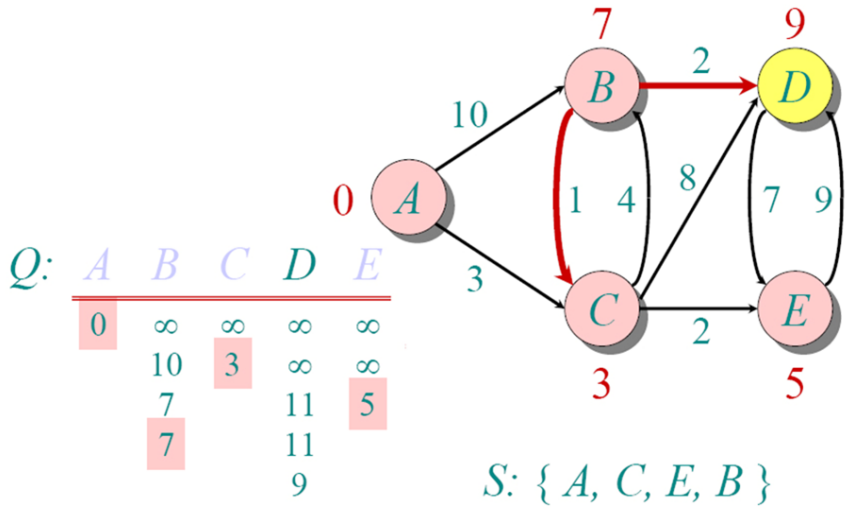


Figure 9: Example

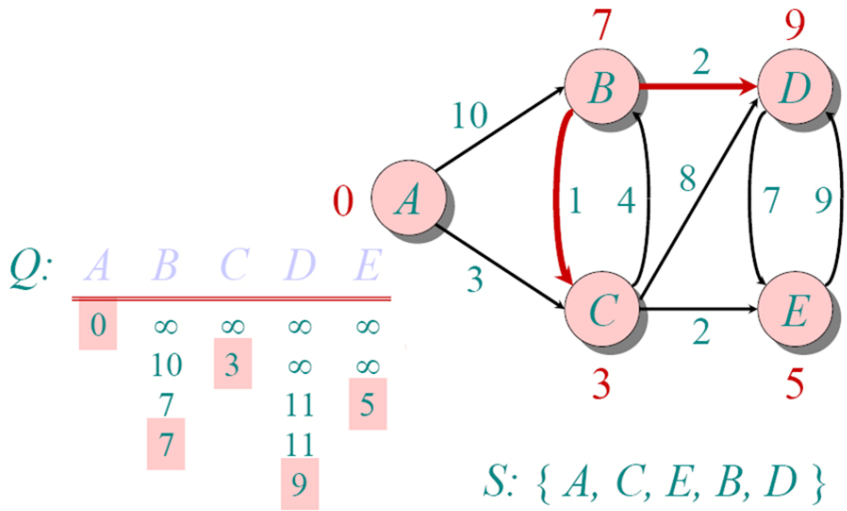


Figure 10: Example

1.6 Time Complexity

- **Best Case:**

  The simplest implementation is store vertices in an arrary or linked list. This will produce a running time of:
  $O(|V|^2 + |E|)$

  For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of:
  $O((|E| + |V|)log|V|)$

- **Worst Case:**

  The worst case running time for Dijkstra's Algorithm is:
  $O(|V|^3 + |E|)$

[1]

2 $\mathcal{A}^*$ Algorithm

2.1 Introduction

$\mathcal{A}^*$ uses a best-first search and finds the least-cost path from a given initial node to one goal node (out of one or more possible goals).

It uses a distance-plus-cost heuristic function (usually denoted $f(x)$) to determine the order in which the search visits nodes in the tree. The distance-plus-cost heuristic is a sum of two functions:

* $f(x) = g(x) + h(x)$
* the path-cost function, which is the cost from the starting node to the current node (usually denoted $g(x)$)
* and an admissible "heuristic estimate" of the distance to the goal (usually denoted $h(x)$).

The $h(x)$ part of the $f(x)$ function must be an admissible heuristic; that is, it must not overestimate the distance to the goal. Thus, for an application like routing, $h(x)$ might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points or nodes.[5]

2.2 Applications

Real time applications of $\mathcal{A}^*$ Algorithm are:

- $\mathcal{A}^*$ mainly used in Computer Gaming, Robotics and Navigation systems/maps.
- For heuristic search is applied to construct a Neural Network Structure(NS).

2.3 History

Peter Hart, Nils Nilsson and Betram Raphael of Stanford Research Institute (now SRI Interna tional) first described the algorithm in 1968. It is an extension of Edsger Dijkstra's 1959 algorithm. $\mathcal{A}^*$

achieves better time performance by using heuristics. Is it a computer algorithm that is widely used in pathfinding and graph traversal, the proc ess of plotting an efficiently traversable path between points, called nodes. Noted for its performance and accuracy, it enjoys widespread use. Bertram Raphael suggested some significant improvements upon this algorithm, calling the revised version A2. Then Peter E. Hart introduced an argument that established A2, with only minor changes, to be the best possible algorithm for finding shortest paths. Hart, Nilsson and Raphael then jointly developed a proof that the revised A2 algorithm was optimal for finding shortest paths under certain well-defined conditions.[3]

2.4 Algorithm

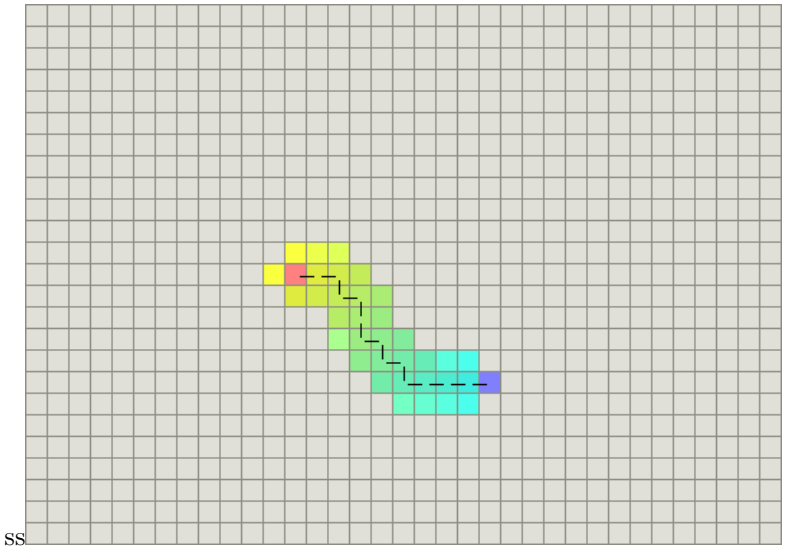
```
procedure FUNCTION(findPath(Node start, Node end))
    List open, closed()
    open.empty()
    closed.empty()
    start.f = start.g = start.h = 0
    open.push(start)
    while !open.isempty() do
        bestNode = open.popLowestFScoreNode()
        if bestNode == end then ▷ Iterate from end to start via
parents                                          ▷ Construct the path
        end if
        closed.push(bestNode)

        parent = bestNode
        for each adjacentNode in bestNode.adjacentNodes() do
            if adjacentNode.isTraversable() then
                if !closed.exists(adjacentNode) then
                    if !open.exists(adjacentNode) then
                        adjacentNode.h = heuristicCostTo(end)
                        adjacentNode.g = parent.g + costTo(adjacentNode)
                        adjacentNode.f=adjacentNode.g+adjacentNode.h
                        adjacentNode.parent = parent
                        open.push(adjacentNode)
                    else                                ▷ Edge relaxation
                        openNode = open.find(adjacentNode)
```

```
                    adjacentNode.g = parent.g + costTo(adjacentNode)
                    if adjacentNode.g ¡ openNode.g then
                        openNode.g = adjacentNode.g
                        openNode.parent = parent
                    end if
                end if
            end if
        end if
    end for
end while
end procedure
```

[5]

2.5 Example

$\mathcal{A}^*$ is like other graph-searching algorithms in that it can potentially search a huge area of the map. It.s like Dijkstra.s algorithm in that it can be used to find a shortest path. It.s like Greedy Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as Best-First-Search:



ss

Figure 11: Example with concave obstacle

In the example with a concave obstacle, A* finds a path as good

as what Dijkstra.s algorithm found:

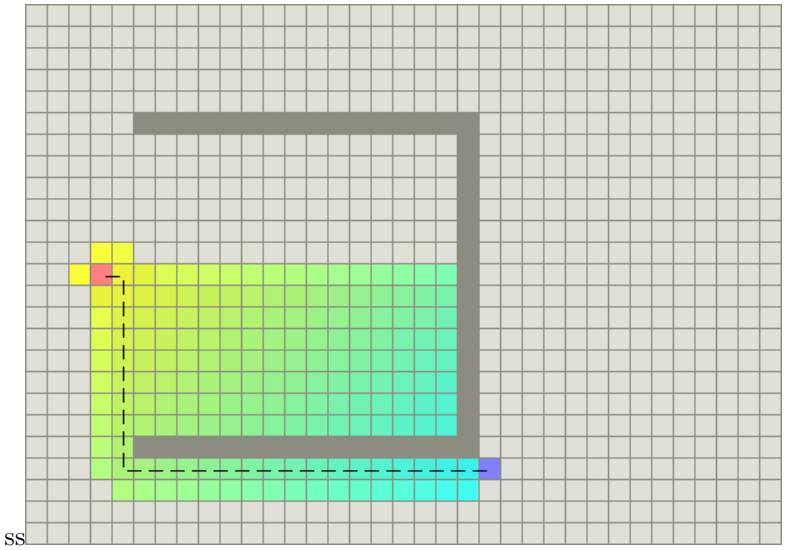


Figure 12: Example

The secret to its success is that it combines the pieces of information that Dijkstra's algorithm uses (favoring vertices that are close to the starting point) and information that Best-First-Search uses (favoring vertices that are close to the goal). In the standard terminology used when talking about $\mathcal{A}^*$, $g(n)$ represents the exact cost of the path from the starting point to any vertex n, and $h(n)$ represents the heuristic estimated cost from vertex n to the goal. In the above diagrams, the yellow (h) represents vertices far from the goal and teal (g) represents vertices far from the starting point.$\mathcal{A}^*$ balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest $f(n) = g(n) + h(n)$

2.6 Time Complexity

The time complexity of $\mathcal{A}^*$ depends on the heuristic. In the worst case, the number of nodes expanded is exponential in the length of the solution.

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

where h^* is the optimal heuristic and it is also defined as true cost of getting from n to the goal.For almost all heuristics in practical use, the error is at least proportional to the path cost, and the resulting exponential growth eventually.[6]

3 Bibilography

References

[1] Thomas H. Cormen., Charles E. Leiserson., Ronald L. Rivest., Clifford Stein *"Introduction to Algorithms" Third Edition*, 1999.

[2] Wikipedia Dijkstra's Algorithm
http://en.wikipedia.org/wiki/Dijkstra's algorithm

[3] Wikipedia $\mathcal{A}^*$ Search Algorithm
http://en.wikipedia.org/wiki/A* search algorithm

[4] Dr. Dijkstra Personal Interview
http://tinyurl.com/lbfocu3

[5] A* Pathfinding for Beginners
http://www.policyalmanac.org/games/aStarTutorial.htm

[6] GameProgramming - $\mathcal{A}^*$ Comparision
http://theory.stanford.edu/ amitp/GameProgramming/AStarComparison.html