Research Paper on Hierarchical Path-Finding

Topic: Hierarchical Path-Finding

Addie Audette, Bug Lee, Annorah Lewis, Luke Marks

November 28, 2022

1 Introduction

1.1 The Motivation

The problem of finding an optimal path arises in many application domains including navigation, robotics, networking, and video games. There are different flavors of algorithms that correctly find the shortest path between two nodes in a graph.

In an unweighted graph, Breath-first search (BFS) is the goto algorithm. For a more general graph where an edge weight can contain either a positive or negative value, the Bellman-Ford algorithm is capable of finding the shortest path between two nodes. However, when the problem domain is restricted to only containing non-negative edge weights, the Dijkstra can find the shortest path with less computation demand than the Bellman-Ford. Nowadays, the A* algorithm, the successor of Dijkstra, is more commonly used as it demands even less computation with the help of an admissible heuristic function.

However, the naive A* algorithm is no longer sufficient for modern real-time applications. Given the sheer size of the nodes for the graph in modern applications, computation demands are too high to run a naive A* algorithm simultaneously for hundreds, if not thousands, of agents. Thus, like many problems in computer science, adding a hierarchy is a solution to reduce the complexity of the problem.

2 Dijkstra algorithm

2.1 Shortest-path tree

Dijkstra algorithm finds the shortest path to all the vertices from source vertex s, given that the graph $G = (V, E)$ contains only non-negative weight for all edges. In other words, it forms the tree that represents the shortest paths to all of the vertices in the graph. Figure 1

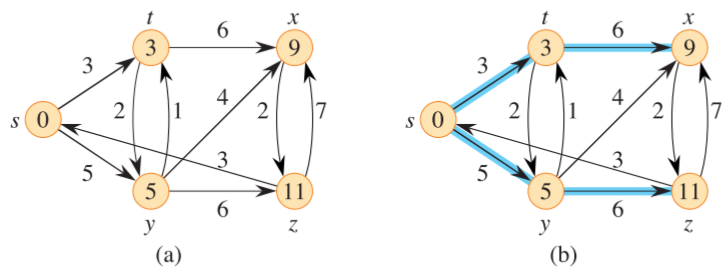

Figure 1: (a) A weighted, directed graph with source s. (b) The blue edges represent the shortest-path tree rooted at the source s. The figure was taken from *Introduction to Algorithms* by CLRS[2].

Dijkstra algorithm is a type of greedy algorithm where it makes a locally optimal decision in each step.

At first, the source vertex only knows the distance to its neighbors and treats other vertices as if they are infinitely far away.

In each step, the vertex that is closest to the source vertex, among the unvisited vertices, is added to the shortest path tree and gets marked visited. Then, the algorithm updates the distance to the vertices that are now reachable (but still unvisited) by the newly visited vertex. It then selects the that is closest to the source vertex among the unvisited vertices again and repeats the process until all the vertex get marked visited.

2.2 Implementation

In practice, the algorithm keeps track of the vertex that is closest to the source vertex using a min-priority queue. Also, the algorithm does not explicitly construct a shortest-path tree using some form of tree data structure. Rather, it records the parent pointer for each vertex. Then, the shortest path from the destination vertex to the source vertex can be found by following the parent pointers. Figure 2

2.3 Pseudocode

The following pseudocode was adapted from *Introduction to Algorithms* by CLRS[2].

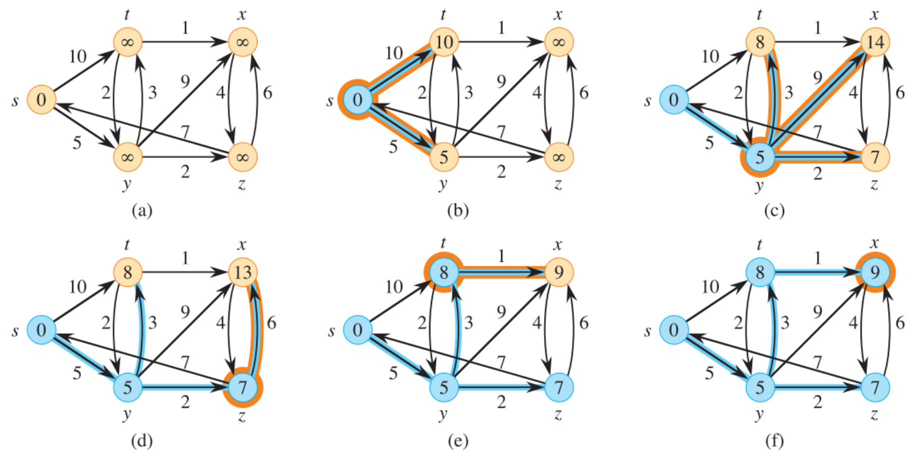

Figure 2: The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and blue edges indicate predecessor values. Blue vertices belong to the vertices that are marked visited, and tan vertices are the not yet visited vertices. The figure was taken from *Introduction to Algorithms* by CLRS[2].

INITIALIZE-SINGLE-SOURCE(G, s)

// $v.d$ = distance from s to v
// $v.\pi$ = parent vertex of v

for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi =$ NIL
$s.d = 0$

DIJKSTRA(G, w, s)

```
// G = (V, E); graph with positive weight edges
// s ∈ V, source vertex
// w(u, v) = weight of edge (u, v)

INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅     // Set of vertices that shortest path from s is known
Q = ∅     // Min-priority Queue
          // Q also holds unvisited vertices.
for each vertex u ∈ G.V
    INSERT(Q, u)
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v in G.Adj[u]
        // found shorter way to reach v
        if v.d > u.d + w(u, v)
            v.d = u.d + w(u, v)     // Update distance to reach from s to v
            v.π = u                 // Update parent pointer
            DECREASE-KEY(Q, v, v.d)
```

2.4 Correctness proof

Let S be the set of vertices that are visited at some point during the algorithm. We want to prove that path found by Dijkstra results in the shortest path for all $u \in S$.

First, consider the base case where $|S| = 1$. That is the case when S contains only the source s. Since the distance from s to s is 0, it is clear that the base case holds for $|S| = 1$.

For the inductive step, assume that Dijkstra finds the shortest path for all $u \in S$, where $|S| \leq k$ for some $k \leq |V|$. Let v be the $k+1$ vertex, the next closest vertex among unvisited vertex. Let u be the vertex in visited set S that have closest to v. We want to prove that S maintains the shortest path tree after visiting v.

We prove this by contradiction. Suppose there is another shortest path to reach v through some unvisited vertex x to v. By construction, v is the closest among the unvisited vertices to some visited vertex u. By the inductive hypothesis, we know that path from s to u is the shortest. Since both path(s, u) and path(u, v) are shortest, the path through x cannot be shorter. This contradicts the assumption that there exists a shorter path to s to v through some unvisited vertex x. In the other words, any other path to v must be longer than or equal to the path(s, u) and then path(u, v). This concludes that the inductive step holds for any $2 \leq k \leq |V|$.

Therefore, Dijkstra output the shortest path for all $u \in V$ by the end of the execution when $|S| = |V|$.

2.5 Runtime Analysis

3 A* algorithm

3.1 "Dijkstra with a twist" [1]

Most often, finding a path from s to some target vertex t using Dijkstra is overkill. Dijkstra algorithm finds the shortest path to all vertices from s when the application only needs one shortest path from s to t. So, the following modifications on the Dijkstra algorithm can give out better performance for finding the shortest path from s to t.

- Dijkstra algorithm blindly selects vertex with minimum distance from s each step. Instead, make a clever guess in each step where the algorithm selects a vertex that is likely part of the shortest path from s to t.[1, 3]
- Terminate once the shortest path from s to t is found.

The above modification on Dijkstra is known as the A* algorithm. Given that the A* algorithm makes an appropriate guess for each step, the A* algorithm guaranteed to return shortest path between source to goal vertex. Also, the A* algorithm computationally more efficient than Dijkstra since it avoids adding unrelated shortest paths to vertices other than t. The guessing stretegy that A* must implement is known as admissible heuristic function.

3.2 Admissible Heuristic

For A* to work correctly and efficiently, the A* algorithm must guess each step that

- Heuristic:
  Minimize wasting computation on finding sub-paths that are obviously not part of the optimal path[3], but also
- Admissiblity:
  Should not ignore the sub-path that can be part of the optimal path[3].

In many situations, there is extra information available for the given problem domain other than the distance between two vertices. For example, imagine finding the shortest route between two cities in the road network. Not only do we know the distance between two pairs of cities, but also know how all cities are positioned on the map. From this, we would unlikely consider traveling to a city in the opposite direction of a destination even if it is located close to the source. Rather, we would first consider a nearby city on the way to the destination.

In the above example, we considered the euclidian distance between a city and the destination in addition to the distance to reach the city. Knowing the euclidian distance was a heuristic to make an appropriate guess. That is, the goal was to select a city with minimum cost $f = g(v) + h(v)$, where $g(v)$ is the exact distance to reach some city v and $h(v)$ is the euclidian distance between city v to destination. The guessing was also admissible since it is certain that the road route cannot be shorter than the euclidian distance between them[3]

3.3 Implementation

Implementation is similar to Dijkstra, but the difference is that the Min-priority queue is sorted based on the cost f instead of distance. Note that if no heuristic is used, $h(v) = 0$ for all $v \in V$, A* implementation is equivalent to Dijkstra.

3.4 Pseudocode

INITIALIZE-SINGLE-SOURCE(G, s)

// $v.f$ = cost of v: $f = g + h$
// $v.g$ = exact distance from s to v
// $v.\pi$ = parent vertex of v
// $h(v)$ = estimate distance from v to target

for each vertex $v \in G.V$
    $v.g = \infty$
    $v.f = \infty$
    $v.\pi =$ NIL
$s.g = 0$
$s.f = h(s)$

A* SEARCH(G, s, t, w, h)

```
// G = (V, E); graph with positive weight edges
// s ∈ V, source vertex
// t ∈ V, target vertex
// w(u, v) = weight of edge (u, v)
// h(v) = estimate distance from v to t

INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅      // Set of vertices that shortest path from s is known
Q = ∅      // Min-priority Queue sorted by cost f.
           // Q also holds unvisited vertices.
for each vertex u ∈ G.V
    INSERT(Q, u)
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    if u = t
        break

    for each vertex v in G.Adj[u]
        // found shorter way to reach v
        if v.g > u.g + w(u, v)
            v.g = u.g + w(u, v)     // Update exact distance to reach from s to v
            v.f = v.g + h(v)        // Update cost to of v
            v.π = u                 // Update parent pointer
            DECREASE-KEY(Q, v, v.f)
```

References

[1] M. Buckland. *Programming Game AI by Example*, pages 241–247. Wordware Publishing, 2005.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*, pages 620–624. The MIT Press, 4th ed edition, 2022.

[3] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Transctions on systems science and cybernetics*, pages 100–107, 1968.