Research Paper on Hierarchical Path-Finding

Topic: Hierarchical Path-Finding

Addie Audette, Bug Lee, Annorah Lewis, Luke Marks

December 14, 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 (or near shortest) path between two or more nodes in a graph.

Two algorithms, Dijkstra and A*, are widely known and guarantee to find the shortest path in general cases. When the problem domain is restricted to only containing non-negative edge weights, the Dijkstra can find the shortest path to all vertices that are connected to the graph[4]. On the other hand, the A* algorithm, the successor of Dijkstra, is generally more suitable for application as it demands 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 number 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 [1]. 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[3]. In other words, it forms the tree that represents the shortest paths to all of the vertices in the graph. Figure 1

Dijkstra algorithm is a type of greedy algorithm where it makes a locally optimal decision in each step. The following describe the high-level idea:

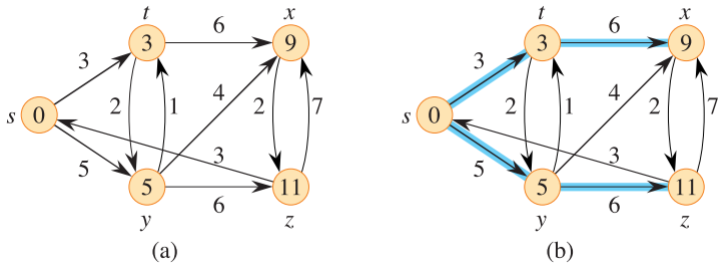

Figure 1: (a) A weighted, directed graph with source s. (b) The blue edges represent the shortest-path tree rooted at the source s. The figure was taken from *Introduction to Algorithms* by CLRS[3].

1. At first, the source vertex only knows the distance to its neighbors and treats other vertices as if they are infinitely far away.
2. Among the univisited vertices, visit the vertex that is closest to the source vertex. That is, add the selected vertex to the shortest path tree and mark visited.
3. Then, updates the distance to the vertices that are now reachable (but still unvisited) by the newly visited vertex.
4. Repeats the step 2-3 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[3].

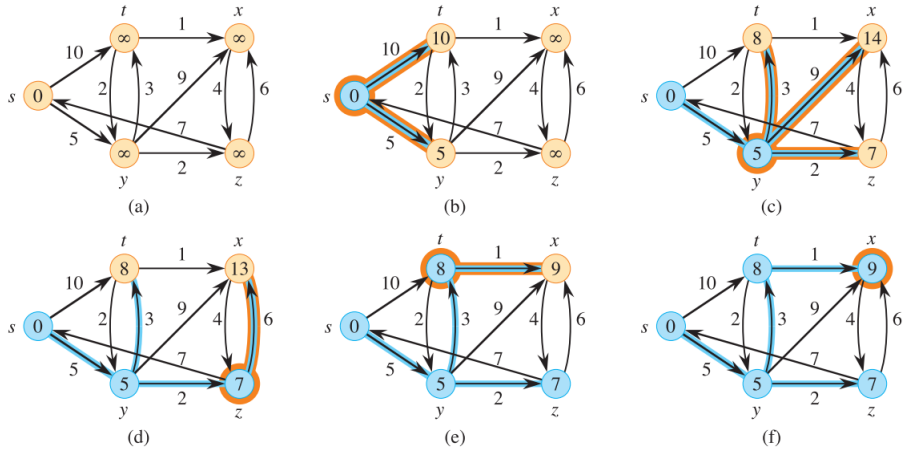

Figure 2: The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and blue edges indicate predecessor values. Blue vertices belong to the vertices that are marked visited, and tan vertices are the not yet visited vertices. The figure was taken from *Introduction to Algorithms* by CLRS[3].

INITIALIZE-SINGLE-SOURCE(G, s)

1 // $v.d$ = distance from s to v
2 // $v.\pi$ = parent vertex of v
3
4 **for** each vertex $v \in G.V$
5 $\quad v.d = \infty$
6 $\quad v.\pi = \text{NIL}$
7 $s.d = 0$

```
DIJKSTRA(G, w, s)
 1  // G = (V, E); graph with positive weight edges
 2  // s ∈ V, source vertex
 3  // w(u, v) = weight of edge (u, v)
 4
 5  INITIALIZE-SINGLE-SOURCE(G, s)
 6  S = ∅      // Set of vertices that shortest path from s is known
 7  Q = ∅      // Min-priority Queue
 8             // Q holds unvisited vertices.
 9  for each vertex u ∈ G.V
10      INSERT(Q, u)
11  while Q ≠ ∅
12      u = EXTRACT-MIN(Q)
13      S = S ∪ {u}
14      for each vertex v in G.Adj[u]
15          // found shorter way to reach v
16          if v.d > u.d + w(u, v)
17              v.d = u.d + w(u, v)      // Update distance to reach from s to v
18              v.π = u                  // Update parent pointer
19              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 $|S| = |V|$.

3 A* algorithm

3.1 "Dijkstra with a twist" [2]

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.[2, 5]
- 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 target 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 unnecessary exploration on finding sub-paths that are obviously not part of the optimal path[5], but also
- Admissibility:
  Should not ignore the sub-path that can be part of the optimal path[5].

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 consider a nearby city closer to the destination unless it is deadend.

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 $f(v) = g(v) + h(v)$, where $g(v)$ is the exact distance to reach some city v and $h(v)$ is the euclidian distance

between city v to destination. On the other hand, we must be sure that no overestimations were made. If we accidentally overestimate one of the sub-path that is part of the optimal path, then the optimal sub-path can be ignored due to its high cost and other non-optimal paths might be selected instead. Fortunately, the guess was safe from overestimation since it is certain that the road route cannot be shorter than the euclidian distance between them[5]. From this, we find that euclidian distance also satisfies admissibility.

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)

```
1  // v.f = cost of v:  f = g + h
2  // v.g = exact distance from s to v
3  // v.π = parent vertex of v
4  // h(v) = estimate distance from v to target
5
6  for each vertex v ∈ G.V
7        v.g = ∞
8        v.f = ∞
9        v.π = NIL
10 s.g = 0
11 s.f = h(s)
```

```
A*-SEARCH(G, s, t, w, h)
 1  // G = (V, E); graph with positive weight edges
 2  // s ∈ V, source vertex
 3  // t ∈ V, target vertex
 4  // w(u, v) = weight of edge (u, v)
 5  // h(v) = estimate distance from v to t
 6
 7  INITIALIZE-SINGLE-SOURCE(G, s)
 8  S = ∅      // Set of vertices that shortest path from s is known
 9  Q = ∅      // Min-priority Queue sorted by cost f.
10             // Q holds unvisited vertices.
11  for each vertex u ∈ G.V
12      INSERT(Q, u)
13  while Q ≠ ∅
14      u = EXTRACT-MIN(Q)
15      S = S ∪ {u}
16      if u = t
17          break
18
19      for each vertex v in G.Adj[u]
20          // found shorter way to reach v
21          if v.g > u.g + w(u, v)
22              v.g = u.g + w(u, v)      // Update exact distance to reach from s to v
23              v.f = v.g + h(v)         // Update cost to of v
24              v.π = u                  // Update parent pointer
25              DECREASE-KEY(Q, v, v.f)
26
27  return PATH(G, s, t)
```

4 Hierarchical Path Finding

4.1 Path finding algorithm for modern applications

In practice, a naive A* algorithm is still not sufficient for many modern applications. First, many modern applications require computation to happen in real-time for hundreds, if not thousands, users/agents simultaneously[1]. Secondly, the shift to mobile applications has put more limitations on memory and CPU usage[1]. For this reason, many variants of the A* algorithm such as IDA*, D*, etc have been proposed to combat the complexity of modern applications. Like many problems in computer science, one approach is to divide and conquer. That is, instead of running the A* algorithm in the whole graph, restructure the problem in a way to use the A* algorithm in a smaller sub-graph and then combine the result to find the shortest path from source to destination in an original graph. This is known as the Hierarchical Path Finding algorithm.

4.2 Hierarchy

Section 3.2 give an illustration of how A* mimics the decision-making process of human by use of admissible heuristic. However, A* lack the arguably most important skill that human use: abstraction. Consider the following example of how a human might plan his/her trip from Virginia Tech to the University of Virginia. The example is adapted from *Near Optimal Hierarchical Path-Finding* [1].

1. Plan a route from Virginia Tech to a major highway entrance in Blacksburg.
2. Plan a route from Blacksburg to Charlottesville.
3. Plan a route from the highway exit in Charlottesville to the University of Virginia.

In the above example, the complexity of finding a path from Virginia Tech to the University of Virginia was reduced in each step. The traveler only needs detailed maps for two cites [1], Blacksburg and Charlottesville. Once the traveler enters the highway, a simplified highway map can be used and ignore all other road details of other cities that pass by [1].

Likewise, the idea of Hierarchical Path Finding is to form a region by clustering the vertices and introducing high-level regional routes Figure 3. Then, the steps for finding a path from the source vertex to the destination vertex are (1) compute a local (low-level) route to the source vertex to the source region entrance, (2) compute a regional (high-level) route from the source region to destination region, and (3) compute a local (low-level) route to destination region exit to destination vertex.

4.3 Time optimization: All-pair shortest path

For applications where computational time is more critical than memory usage, caching technique can be used[2]. That is, the shortest path between regions can be pre-computed for all regions and stored inside a table/matrix. The simplest form is to run Dijkstra on each region, but more sophisticated algorithms should be used in practice. This is also known as finding an all-pair shortest path. Once stored inside a table/matrix, finding a regional route from the source region to the destination region can be performed in constant time during runtime.

4.4 Near optimal path

One caveat of the Hierarchical Path Finding algorithm is that it does not guarantee the shortest path between the source and destination vertex. Using the human traveler analogy from section 4.2, a highway route may make travelers travel around the city, adding more distance needed to travel compared to traveling through city roads[1]. The *Near Optimal Hierarchical Path-Finding* apply a path-smoothing procedure to make a path found by Hierarchical Path Finding within 1% optimal compared to shortest path[1].

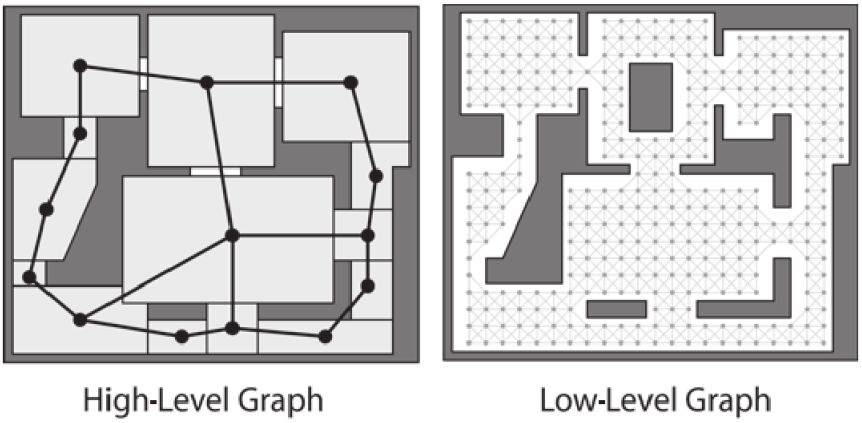

Figure 3: Forming regions (high-level graph) by clustering neighboring vertices. The figure was taken from *Programming Game AI by Example* by Buckland[2].

4.5 Implementation

Compare to Dijkstra and naive A*, there are more tunable variables that implementers need to consider; the number of hierarchy levels, cluster/region size, and placement of regional entrances/exits. In practice, optimizations like preprocessing/caching regional routes and path smoothing may be necessary. To avoid overwhelming the algorithm with optimization details, our implementation assumes the following simplifications: (1) an input graph is static and known in advance, (2) 2 levels of the hierarchy, (3) preselecting regional entrances/exits in reachable locations, (4) no preprocessing/caching, and (5) no path refinement or smoothing.

4.6 Pseudocode

HIERARCHICAL-PATH-FINDING(G, s, t, w, h)

```
1  // G = (V, E); graph with positive weight edges
2  // s ∈ V, source vertex
3  // t ∈ V, target vertex
4  // w(u, v) = weight of edge (u, v)
5  // h(v) = estimate distance from v to t
6
7  Path = {}                          // Hold global path from s to t
8  L_2 = {}                           // Hold regional level path from s to t
9  G_2 = CreateLevel2(G)              // Create regional vertices from G
10 G_2.V = G_2.V ∪ {s, t}             // Add s and t to regional vertices
11
12 L_2 = A*-search(G_2, s, t, w, h)   // Compute regional shortest path
13
14 // Compute local shortest path between two regions inside regional shortest path
15 // then add to global path
16 for i = 1 to |L_2| − 1
17       L_1 = A*-search(G, L_2[i], L_2[i + 1], w, h)   // shortest path between L_2[i] and L_2[i + 1]
18       Path = Path ∪ L_1
19
20 return Path
```

5 Conclusion

The Dijkstra and A* have their strong points and cannot argue that one is better than the other in all situations. However, knowing that most applicants do not need additional computation for finding a path to all vertices from the source, A* became a preferred method of the path-finding algorithm in most cases. Hierarchical Path Finding follows the similar footstep. Many applications nowadays are more constrained by the resource than optimality, allowing Hierarchical Path Finding to rise despite its additional complexity and non-optimality. For our next step, we plan to implement and showcase the three algorithms, Dijkstra, A*, and Hierarchical Path Finding discussed in this paper. Throughout the process, we will explore whether the performance of Hierarchical Path Finding is worth the additional complexity.

References

[1] A. Botea, M. Müller, and J. Schaeffer. Near optimal hierarchical path-finding. *J. Game Dev.*, 1:1–30, 2004.

[2] M. Buckland. *Programming Game AI by Example*, pages 241–247, 359–360, 373. Wordware Publishing, 2005.

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

[4] J. Erickson. *Algorithms*, pages 278, 284, 289. Jeff Erickson, 2019.

[5] 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.