CS774 - Optimization Techniques
Mid-Term Project Report

by
Abhinav Prakash(14014)
Pratik Mishra(14493)

In the first part of the report we are covering the algorithms/techniques for designing the shortest path algorithm for static graphs

Static Graphs : $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ & $\boldsymbol{w_{ij}} : \boldsymbol{E} \rightarrow \mathbb{R}$ do not change

In Part 2, we have a brief look at the Travelling salesman Problem.

Shortest Path Problem

In graph theory, the **shortest path problem**is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

We can categorise our problem based on the nature of source and destination given as input.

- The **single-source shortest path problem**,in which we have to find shortest paths from a source vertex $\boldsymbol{v}$ to all other vertices in the graph.
- The **single-destination shortest path problem**,in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex $\boldsymbol{v}$. This can be reduced to the single-source shortest path problem by reversing the arcs in the directed graph.
- The **all-pairs shortest path problem**,in which we have to find shortest paths between every pair of vertices $\boldsymbol{v}$, $\boldsymbol{v'}$ in the graph.

We will be mainly concerned with single-source shortest path problem & all-pairs shortest path problem.

We will be studying them for static as well as dynamic graphs.

In our dynamic graph, there can be many variations, like, cases where the weight of edges vary or cases where nodes are dynamically inserted and removed.

Definition :
(Linear programming formulation)

Given a directed graph (V, E) with source node s,target node t, and cost w_{ij} for each edge (i, j) in E, consider the program with variables x_{ij}

$$\min \sum_{i,j \in A} w_{ij} x_{ij} \text{ provided } x_{ij} \geq 0$$

$$\forall i \quad \sum_j x_{ij} - \sum_j x_{ji} = \begin{cases} 1 & \text{if } i = s \\ -1 & \text{if } i = t \\ 0 & \text{otherwise} \end{cases}$$

The intuition behind this is that x_{ij} is an indicator variable for whether edge (i, j) is part of the shortest path: 1 when it is, and 0 if it is not. We wish to select the set of edges with minimal weight, subject to the constraint that this set forms a path from s to t (represented by the equality constraint: for all vertices except s and t the number of incoming and outcoming edges that are part of the path must be the same (i.e., that it should be a path from s to t).

This LP has the special property that it is integral; more specifically, every basic optimal solution (when one exists) has all variables equal to 0 or 1, and the set of edges whose variables equal 1 form an s-t directed path.

The **dual**for this linear program is

$$\max y_t - y_s$$

Given $\forall i,j \quad y_j - y_i \leq w_{ij}$

and feasible duals correspond to the concept of a consistent heuristic for the A* algorithm for shortest paths. For any feasible dual y the reduced costs $w'_{ij} = w_{ij} - y_j + y_i$ are nonnegative and A* essentially runs Dijkstra's algorithm on these reduced costs.

Single-source shortest paths

• **For Undirected Graphs** $w_{ij} : E \rightarrow \mathbb{R}^{+}$

Dijkstra's Algorithm $O(|E| + |V| \log |V|)$

• **Unweighted graphs**

Breadth-first Search $O(|E| + |V|)$

• **Directed acyclic graphs**

Topological Sorting $O(|E| + |V|)$

• **Directed graphs with nonnegative weights**

Bellman - Ford Algorithm $O(|V||E|)$

Dijkstra's algorithm with Fibonacci heap $O(|E| + |V| \log |V|)$

• **Directed graphs with arbitrary weights without negative cycles**

Bellman - Ford Algorithm $O(|V||E|)$

All-pairs shortest paths

• **Undirected graph** ($w_{ij} : E \rightarrow \mathbb{R}^{+}$)

Floyd-Warshall Algorithm $O(|V|^3)$

• **Directed graph**

$w_{ij} : E \rightarrow \mathbb{R}$ (***no negative cycles***)

Floyd-Warshall Algorithm $O(|V|^3)$

Johnson - Dijkstra Algorithm $O(|E||V| + |V|^2 \log |V|)$

There has been a recent development where the complexity has been improved to $O(|E||V| + |V|^2 \log \log |V|)$ using Fibonacci Heaps.

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph. Dijkstra's original algorithm does not use a min-priority queue and runs in $O(|V|^2)$ time (where $|V|$ is the number of nodes). 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.

```
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph:           // Initialization
6          dist[v] ← INFINITY                // Unknown distance from source to v
7          prev[v] ← UNDEFINED               // Previous node in optimal path from source
8          add v to Q                        // All nodes initially in Q (unvisited nodes)
9
10     dist[source] ← 0                      // Distance from source to source
11
12     while Q is not empty:
13         u ← vertex in Q with min dist[u]  // Source node will be selected First
14         remove u from Q
15
16         for each neighbor v of u:         // where v is still in Q.
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]:             // A shorter path to v has been found
19                 dist[v] ← alt
20                 prev[v] ← u
21
22     return dist[], prev[]
```

Practical optimizations

In common presentations of Dijkstra's algorithm, initially all nodes are entered into the priority queue. This is, however, not necessary: the algorithm can start with a priority queue that contains only one item, and insert new items as they are discovered (instead of doing a decrease-key, check whether the key is in the queue; if it is, decrease its key, otherwise insert it). This variant has the same worst-case bounds as the common variant, but maintains a smaller priority queue in practice, speeding up the queue operations.

The **Bellman–Ford algorithm** is an algorithm that compute the shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.

Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm. If a graph contains a "negative cycle" (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no ***cheapest*** path: any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect negative cycles and report their existence.

Algorithm:

Like Dijkstra's Algorithm, Bellman–Ford is based on the principle of relaxation, in which an approximation to the correct distance is gradually replaced by more accurate values until eventually reaching the optimum solution. In both algorithms, the approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value with the length of a newly found path. However, Dijkstra's algorithm uses a priority queue to greedily select the closest vertex that has not yet been processed, and performs this relaxation process on all of its outgoing edges; by contrast, the Bellman–Ford algorithm simply relaxes ***all*** the edges, and does this $|V| - 1$ times.
In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances. This method allows the Bellman–Ford algorithm to be applied to a wider class of inputs than Dijkstra.

Bellman–Ford runs in $O(|V| |E|)$ time.

```
function Bellman_Ford(list vertices, list edges,
   vertex source) ::distance[],predecessor[]


   // This implementation takes in a graph, represented as
   // lists of vertices and edges, and fills two arrays //
   (distance and predecessor) with shortest-path
   // (less cost/distance/metric) information

   // Step 1: initialize graph
   for each vertex v in vertices:
```

```
      distance[v] := inf          // At the beginning , all vertices have a weight of
Infinity

      predecessor[v] := null      // And a null predecessor
   distance[source] := 0          // Except for the Source, where the Weight is zero


   // Step 2: relax edges repeatedly
   fori from1 tosize(vertices)-1:
       for eachedge (u, v) withweight w inedges:
           ifdistance[u] + w < distance[v]:
               distance[v] := distance[u]
               + w predecessor[v] := u


   // Step 3: check for negative-weight cycles
   for eachedge (u, v) withweight w inedges:
       ifdistance[u] + w < distance[v]:
           error"Graph contains a negative-weight
   cycle" returndistance[], predecessor[]
```

Floyd–Warshall algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles).A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between ***all*** pairs of vertices, though it does not return details of the paths themselves.

```
1 let dist be a |V| × |V| array of minimum distances initialized
to ∞ (infinity) 2 for each vertex v
3 dist[v][v] ← 0 4
for each edge (u,v)
5    dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
6 for k from 1 to |V|
7    for i from 1 to |V|
8       for j from 1 to |V|
9          if dist[i][j] > dist[i][k] + dist[k][j]
 10            dist[i][j] ← dist[i][k] + dist[k][j]
 11         end if
```

Algorithm

The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with $\Theta(|V|^3)$ comparisons in a graph. This is remarkable considering that there may be up to $\Omega(|V|^2)$ edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal.

Comparison with other shortest path algorithms

The Floyd–Warshall algorithm is a good choice for computing paths between all pairs of vertices in dense graphs, in which most or all pairs of vertices are connected by edges. For sparse graphs with nonnegative edge weights, a better choice is to use Dijkstra's algorithm from each possible starting vertex, since the running time of repeated Dijkstra ($O(|V|\,|E|\log|V|)$ using binary heaps) is better than the $\Theta(|V|^3)$ running time of the Floyd–Warshall algorithm when $|E|$ is significantly smaller than $|V|^2$. For sparse graphs with negative edges but no negative cycles, Johnson's algorithm can be used, with the same asymptotic running time as the repeated Dijkstra approach. There are also known algorithms using fast matrix multiplication to speed up all-pairs shortest path computation in dense graphs, but these typically make extra assumptions on the edge weights (such as requiring them to be small integers). In addition, because of the high constant factors in their running time, they would only provide a speedup over the Floyd–Warshall algorithm for very large graphs.

Introduction to the Travelling Salesman Problem

The travelling salesman problem (TSP) asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

The traveling salesman problem can be described as follows:

TSP = {(*G, f, t*): *G* = (*V, E*) a complete graph,
f is a function $V \times V \rightarrow Z$, *t* Z,
G is a graph that contains a traveling salesman tour with cost that does not exceed *t*}.

Example:

Consider the following set of cities:

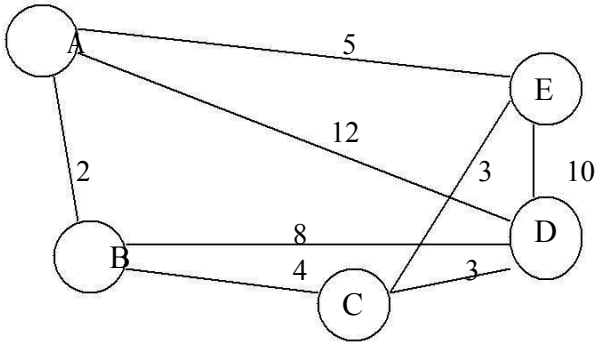

Figure 10.1 A graph with weights on its edges.

The problem lies in finding a minimal path passing from all vertices once. For example the path Path1 {A, B, C, D, E, A} and the path Path2 {A, B, C, E, D, A} pass all the vertices but Path1 has a total length of 24 and Path2 has a total length of 31.

Definition:

A Hamiltonian cycle is a cycle in a graph passing through all the vertices once.

Theorem :
The traveling salesman problem is NP-complete.

Proof:
First, we have to prove that TSP belongs to NP. If we want to check a tour for credibility, we check that the tour contains each vertex once. Then we sum the total cost of the edges and finally we check if the cost is minimum. This can be completed in polynomial time thus TSP belongs to NP.
Secondly we prove that TSP is NP-hard. One way to prove this is to show that Hamiltonian cycle $\leq_P$ TSP

(given that the Hamiltonian cycle problem is NP-complete). Assume $G = (V, E)$ to be an instance of Hamiltonian cycle. An instance of TSP is then constructed. We create the complete graph $G' = (V, E')$, where $E' = \{(i, j): i, j \in V \text{ and } i \neq j\}$. Thus, the cost function is defined as:

$$t(i, j) = \begin{cases} 0 & \text{if } (i, j) \in E, \\ 1 & \text{if } (i, j) \notin E. \end{cases} \qquad 10.1$$

Now suppose that a Hamiltonian cycle h exists in G. It is clear that the cost of each edge in h is 0 in G as each edge belongs to E. Therefore, h has a cost of 0 in G' . Thus, if graph G has a Hamiltonian cycle then graph G'has a tour of 0 cost.
Conversely, we assume that ***G'*** has a tour ***h'*** of cost at most 0. The cost of edges in ***E'*** are 0 and 1 by definition. So each edge must have a cost of 0 as the cost of ***h'*** is 0. We conclude that ***h'*** contains only
edges in E.
So we have proven that G has a Hamiltonian cycle if and only if G**' has a tour of cost at most 0.**
Thus TSP is NP-Complete.

Methods to solve the traveling salesman problem

Since TSP is a known NP-Hard problem, it does not have any polynomial time optimum solutions. We do not consider the exponential Dynamic Programming solution here because of its infeasibility. We look at approximate solutions for TSP that achieve polynomial time.

Using the triangle inequality to solve the traveling salesman problem

Definition:
If for any set of vertices $a, b, c \in V$, it is true that t $(a, c) \leq t(a, b) + t(b, c)$ where t is the cost function, we say that t satisfies the triangle inequality.

First, we create a minimum spanning tree the weight of which is a lower bound on the cost of an optimal traveling salesman tour. Using this minimum spanning tree we will create a tour the cost of which is at most 2 times the weight of the spanning tree. We present the algorithm that performs these computations using the MST-Prim algorithm.

Approximation-TSP
Input: A complete graph $G (V, E)$
Output: A Hamiltonian cycle

1. **Select a "root" vertex** $r \in V[G]$.
2. Use MST-Prim (G, c, r) to compute a minimum spanning tree from r.
3. Assume L to be the sequence of vertices visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H that visits the vertices in the order L.

Theorem :

Approximation-TSP is a 2-approximation algorithm with polynomial cost for the traveling salesman problem given the triangle inequality.

Proof:

Approximation-TSP costs polynomial time as was shown before.

Assume H^* to be an optimal tour for a set of vertices. A spanning tree is constructed by deleting edges from a tour. Thus, an optimal tour has more weight than the minimum- spanning tree, which means that the weight of the minimum spanning tree forms a lower bound on the weight of an optimal tour.

$$c(t) \leq c(H^*). \qquad 10.2$$

Let a full walk of T be the complete list of vertices when they are visited regardless if they are visited for the first time or not. The full walk is *W*. In our example:
W = A, B, C, B, D, B, E, B, A,.
The full walk crosses each edge exactly twice. Thus, we can write:

$$c(W) = 2c(T). \qquad 10.3$$

From equations 10.2 and 10.3 we can write that

$$c(W) \leq 2c(H^*), \qquad 10.4$$

Which means that the cost of the full path is at most 2 time worse than the cost of an optimal tour. The full path visits some of the vertices twice which means it is not a tour. We can now use the triangle inequality to erase some visits without increasing the cost. The fact we are going to use is that if a vertex a is deleted from the full path if it lies between two visits to b and c the result suggests going from b to c directly.

In our example we are left with the tour: A, B, C, D, E, A. This tour is the same as the one we get by a preorder walk. Considering this preorder walk let H be a cycle deriving from this walk. Each vertex is visited once so it is a Hamiltonian cycle. We have derived H deleting edges from the full walk so we can write:

$$c(H) \leq c(W) \qquad 10.5$$

From 10.4 and 10.5 we can imply:

$$c(H) \leq 2\, c(H^*). \qquad 10.6$$

This last inequality completes the proof.

Further Work to be done:

We continue to look for better solutions than 2-approximate Algorithm. For example, Christofides Algorithm is 1.5-approximate. We also look at a Heuristic solution by Karp and the Branch and bound algorithm.

References

1. https://en.wikipedia.org/wiki/Shortest_path_problem

2. https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

3. https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

4. https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

5. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.(1990). ***Introduction to Algorithms*** (1st ed.). MIT Press and McGraw-Hill. ISBN 0 -262-03141-8. See in particular Section 26.2, "The Floyd–Warshall algorithm", pp. 558–565 and Section 26.4, "A general framework for solving path problems in directed graphs", pp. 570–576.

6. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford(2001) [1990]. "Single-Source Shortest Paths and All-Pairs Shortest Paths". ***Introduction to Algorithms*** (2nd ed.). MIT Press and McGraw-Hill. pp. 580–642. ISBN 0 -262-03293-7.

7.http://users.cecs.anu.edu.au/~Alistair.Rendell/Teaching/apac_comp3600/module4/all_pairs_shortest_paths.xhtml

8.Pettie, Seth (26 January 2004). "A new approach to all-pairs shortest paths on real-weighted graphs". ***Theoretical Computer Science***. **312**(1): 47–74.