Shortest Path

CS 5633 Analysis of Algorithms

Computer Science
University of Texas at San Antonio

November 13, 2024

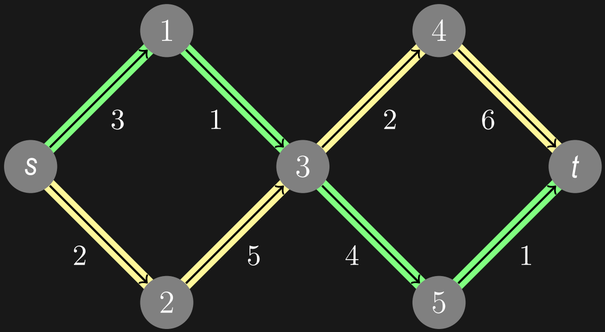
Shortest Path

Shortest Path

- A common application of graphs is to model some sort of transportation network (airports, roads, etc.).
- If we are currently at a location s and wish to travel to a location t, then we may want to find a path which starts at s and ends at t. There may be many such paths, and in the application, some paths may be much more expensive to follow than others.
- We can assign a weight to each edge $\{u, v\}$ of the graph which represents the cost of moving from location u to location v. Now consider some path p from s to t. The weight of the path $w(p)$ is the sum of the edge weights along the path.

Example of Shortest Path

- Two potential paths
  - Path 1: $s \to 1 \to 3 \to 4 \to t$; weight: $3 + 1 + 4 + 1 = 9$.
  - Path 2: $s \to 2 \to 3 \to 5 \to t$; weight: $2 + 5 + 2 + 6 = 15$.



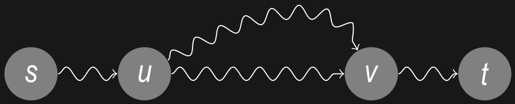
Determining the Shortest Path

- In this setting, we would be interested in computing a shortest path from s to t. A shortest path is a path of minimum weight from s to t. Let $\delta(u, v)$ denote the weight of a shortest path between any two vertices u and v in the graph ($\delta(u, v) = \infty$ if there are no paths from u to v).
- In some applications we may want to have negative weights on an edge. Note that if there is a negative-weight cycle, then some shortest paths may not exist (usually due to negative loops).

$$
\begin{tikzcd}
 & & 2 \arrow[dll, "-2"'] & \\
s \arrow[r, "1"'] & 1 \arrow[rr, "1"'] & & 3 \arrow[ul, "-2"'] \arrow[r, "1"'] & t
\end{tikzcd}
$$

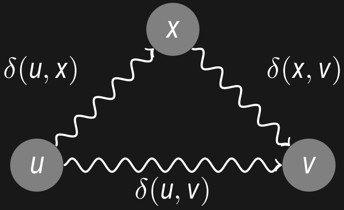
Building Shortest Path on Shortest Paths

- Similar to other optimization problems, we can find the shortest path from s to t based on other shortest paths.
- Theorem: The sub-path of a shortest path is also a shortest path.
- Proof: Consider the shortest path P from s to t. Two vertices u and v are on this path. The sub-path of P from u to v must be the shortest between u and v. Otherwise, we can construct a new path P' using the shortest path between u and v, and P' is shorter than P, thus P cannot be the shortest path between s and t.



Traingle Inequality

- The following theorem is known as the *triangle inequality* and is also important in the computation of shortest paths.
- Theorem: For all $u, v, x \in V$, we have $\delta(u, v) \leqslant \delta(u, x) + \delta(x, v)$.



Dijkstra’s Algorithm

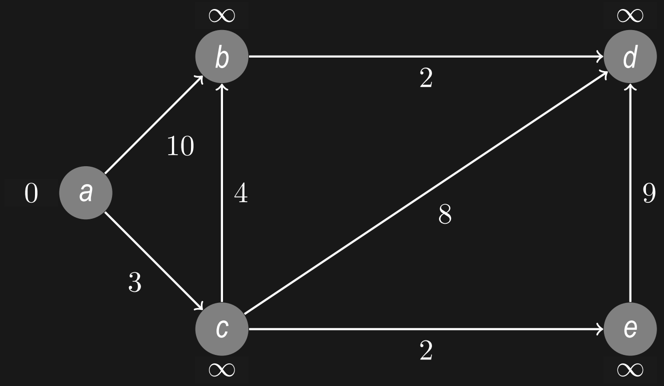
Single-source Shortest Path

- We will now consider the *single-source shortest path* problem in which we are given a graph with a designated source vertex s, and we wish to compute the shortest path weights $\delta(s, v)$ for each $v \in V$.
- We will assume all edge weights are nonnegative so that we will not have any negative weight cycles.

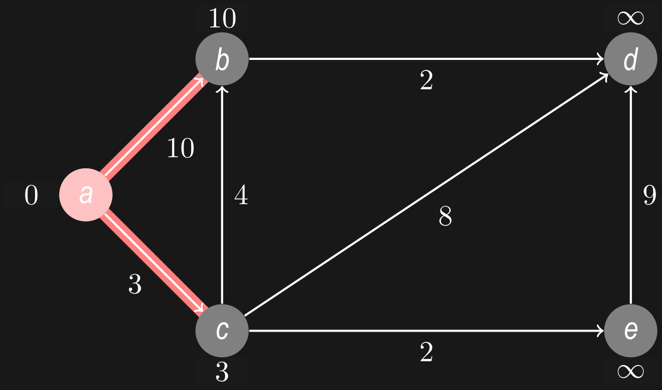
Dijkstra's Algorithm

1. Maintain a set S of vertices whose shortest path weights from s are known, that is $dist[v] = \delta(s, v)$.
2. At each step, add the vertex $v \in \{V - S\}$ whose $d[v]$ is minimal to S.
3. Update $dist[u]$ for any vertex u adjacent to v.

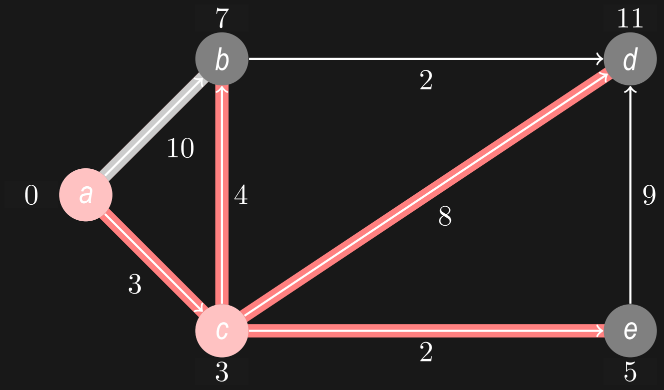
An Example of Dijkstra's Algorithm



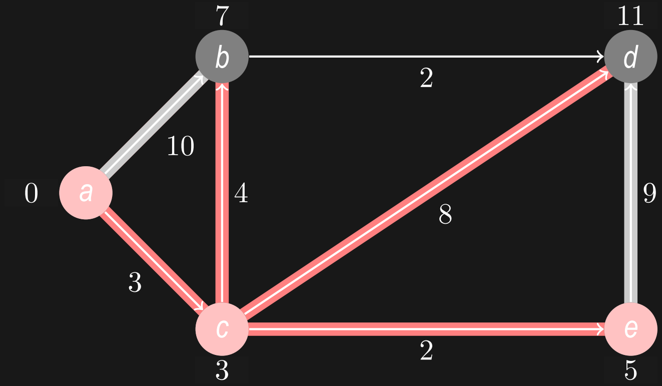
An Example of Dijkstra's Algorithm



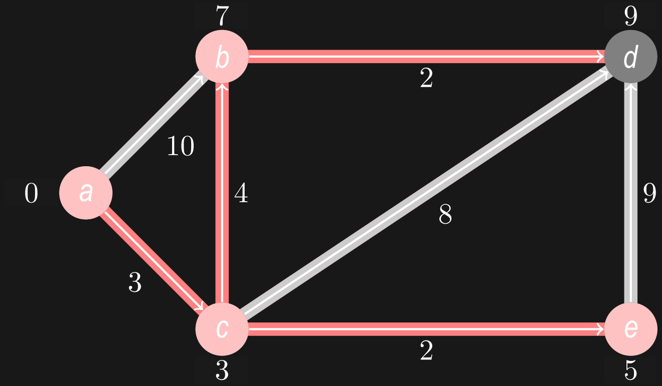
An Example of Dijkstra's Algorithm



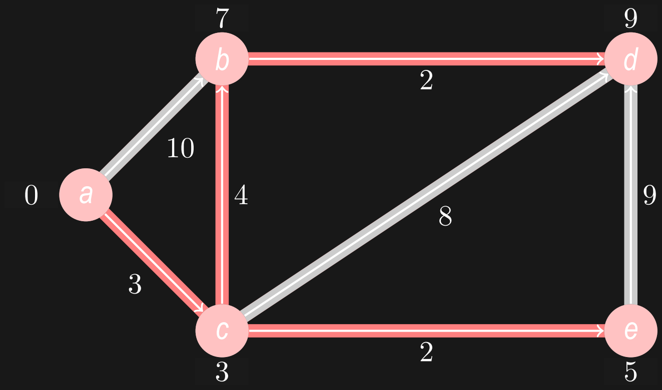
An Example of Dijkstra's Algorithm



An Example of Dijkstra’s Algorithm



An Example of Dijkstra's Algorithm



Implementing Dijkstra’s Algorithm

- Dijsktra’s algorithm requires maintaining a sorted list vertices based on their current path weights.
- The algorithm also requires dynamic updates and extracting min from the list.
- A priority queue is clearly a good option for these tasks.

Pseudo-code of Dijkstra's Algorithm

Algorithm 1: Dijkstra's algorithm with a priority queue.

```
Function Dijkstra_Shortest_Path(graph G(V, E), source s)
    Q = empty priority queue;
    S = empty set; // set of vertices with known shortest path;
    for each u ∈ V do
        u.dist = ∞; // Initially, all vertices have a ∞ long path from s;
        u.prev = NIL; // previous vertex in the shortest path is unknown;
        Q.add(u); // add u to priority queue;
    s.dist=0; // s → s has 0 weight; Q is implicitly updated;
    while Q is no empty do
        u = Q.Extract_Min();
        S.add(u);
        for each neighbor v of u do
            dist = u.dist + weight(u, v);
            if dist < v.dist then
                v.dist = dist;
                v.prev = u;
```

Run-time Dijkstra's Algorithm

- Assuming binary tree is used as the priority queue. Update on the priority queue and extracting min both cost $O(\lg |V|)$.
- The loop at line 10 performs $|V|$ Extract_Min in total. Therefore, the total cost of Extract_min is $O(|V| \lg |V|)$.
- The loop at line 15 performs $|E|$ update at most. Therefore, the total cost of updates is $O(|E| \lg |V|)$.
- The overall cost of Dijkstra's Algorithm is then $O((|E| + |V|) \lg |V|) = O(|E| \lg |V|)$.

Correctness of Dijkstra's Algorithm

- Theorem:
  (i) For all $v \in S : v.dist = \delta(s, v)$.
  (ii) For all $v \notin S : v.dist$ is the weight of a shortest path from s to v that uses only vertices in S (besides v itself).
- The implication of this theorem is that Dijkstra's algorithm terminates with $v.dist = \delta(s, v)$ for each $v \in V$ (because each $S = V$ at the end of the algorithm).
- The proof is by induction. It is clearly true in the base case ($s.dist = 0$ and $v.dist = \infty$ for all $v \neq s$). Assume (i) and (ii) are true before an iteration, and we will show it remains true after another iteration.
- Let u be the vertex added to S in this iteration. So $d[u] \leq d[v]$ for all $v \in S$.

Correctness of Dijkstra's Algorithm cont.

- Induction proof of (i):
  1. Assume the contradiction is true, i.e, let u be the first vertex for which $u.dist \neq \delta(s, u)$ when it is added to set S.
  2. If $u.dist \neq \delta(s, u)$, when there must be a shortest path P, and the $weight(P) < u.dist$. That is, there must be some vertex y in $\{V - S\}$ on the path P (since u is the first vertex in S violates (i)).
  3. As y is on u's shortest path, we should have $y.dist < u.dist$.
  4. However, because u is chosen before y, we must have $u.dist < y.dist$, contradicting that $y.dist < u.dist$.
  5. Therefore, the assumption that "(i) is false" must be false.

Correctness of Dijkstra's Algorithm cont.

- Induction proof of (ii):
  1. Consider a $v \in S$. Let P be the shortest path from s to v using only vertices in S (except v itself).
  2. Case 1: If P does not contain u, then (ii) is true by induction hypothesis.
  3. Case 2: If P contains u,
     - 3.1 P consists of vertices of $S - \{u\}$, then through u to v.
     - 3.2 Based on (i), $u.dist$ is the weight of u's shortest path.
     - 3.3 Therefore, $u.dist + weight(u, v)$ is the weight of the shortest path from s to v using only vertices in S.
     - 3.4 Because of we set $v.dist$ to be $u.dist + weight(u, v)$, (ii) is true at the end of current iteration (or the beginning of the next iteration).

Shortest Path of Unweighted Graph

- Now consider the *unweighted case* in which we want to find a path with the smallest number of edges.
- Certainly Dijkstra's Algorithm can still work (we can just set the weight of each edge to be 1). Can we do better?
- Idea: do a modified BFS search starting from s. Recall BFS traverses the graph in "layers" where each layer will be the same distance from s.

Output Shortest Paths

- So far we have only been computing the *length* of a shortest path. What if we want to know what the shortest path actually is?
- We can build a shortest path tree similarly to how we built MST, BFS, and DFS trees (we remember the "predecessor" for each vertex during the computation).
- The shortest paths can be easily reconstructed backwardly using *v.prev* that we have set in the pseudo-code.

All-pairs Shortest Paths

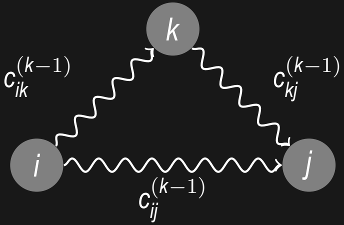
Floyd-Warshall Algorithm

- The **Floyd-Warshall algorithm** is a dynamic programming algorithm for the all-pairs shortest path problem.
- Suppose the graph is given as an *adjacency matrix* $A = (a_{ij})$ where a_{ij} is the weight of the edge from i to j.
- Let $c_{ij}^{(k)}$ denote the weight of a shortest path from i to j with intermediate vertices on the path belonging to the set $\{1, 2, \ldots, k\}$.
  - Note that $\delta(i, j) = c_{ij}^{(n)}$.

Floyd-Warshall Algorithm cont.

- The algorithm is to show that $c_{ij}^{(k)}$ for each $1 \leq i, j, k \leq n$ can be computed using dynamic programming.

$$c_{ij}^{(k)} = \min(c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)})$$



Pseudo-code of Floyd-Warshall Shortest Path Algorithm

Algorithm 2: Floyd-Warshall algorithm.

Function FW_All_Shortest_Paths(graph $G(V, E)$)
    c = a matrix of $|V| \times |V|$ with values of ∞;
    // set initial values;
    for each vertex v **do**
        $c[v, v]$ = 0;
    for each edge (u, v) **do**
        $c[u, v]$ = $weight(u, v)$;
    // update the values based on c's recursive definition;
    for $k = 1$ **to** $|V|$ **do**
        for $i = 1$ **to** $|V|$ **do**
            for $j = 1$ **to** $|V|$ **do**
                if $c[i, j] > (c[i, k] + c[k, j])$ **then**
                    $c[i, j] = c[i, k] + c[k, j]$;

Run Time of Floyd-Washall Algorithm

- The first two loops have a run time of $\Theta(|V| + |E|)$.
- The loop starts at line 8 has a run time of $\Theta(|V|^3)$.
- Considering $|E| < |V|^2$, the total run time of this algorithm is $\Theta(|V|^3)$.