CSE 3500 Algorithms and Complexity – Fall 2016
Lecture 17: October 25, 2016

Dijkstra's Algorithm

- Dijkstra's algorithm for the SSSP problem generates the shortest paths in nondecreasing order of the shortest path weights. I.e., the shortest path whose path weight is the least (from across all the shortest paths) is generated first; the shortest path whose path weight is the next smallest is generated next; and so on.
- The algorithm at any given time keeps a set $S \subseteq V$ of nodes such that for the nodes in S we have already computed the weights of the shortest paths from the source node. To begin with $S = \{s\}$.
- The algorithm has $n-1$ phases ($n = |V|$) where in each phase a new node enters S. When $S = V$, the algorithm terminates.
- The nodes that are not in S will be stored in a 2-3 tree Q.
- For any node $u \notin S$ we define a function $dist[u]$. We define $dist[u]$ to be the weight of the shortest path from among all the s to u paths all of whose intermediate nodes come from S.
- When a node $u \notin S$ is inserted into Q, its key value will be $dist[u]$.
- We claim that the next node u that should enter S is the node from Q whose $dist$ value is the least (from among all the nodes in Q). This can be proven by contradiction. Assume to the contrary that the next node u that enters S has an intermediate node from $V - S$. Let this node be w. We can express the shortest path weight from s to u as $pathweight(s, w) + pathweight(w, u)$ (where $pathweight(a, b)$ stands for the weight of the shortest path from the node a to the node b). Since there are no negative edges, it follows that $pathweight(w, u)$ is nonnegative. This in turn means that $pathweight(s, w) \leq pathweight(s, u)$. This is a contradiction since in this case we should generate the shortest path to w before generating the shortest path to u.

A pseudocode

- Input is a directed weighted graph $G(V, E)$ and a source node $s \in V$. The output are the shortest paths from s to every other node. Let $|V| = n$ and $|E| = m$. A pseudocode for the Dijkstra's algorithm follows.

1) $S = \{s\}$;
2) **for** each $u \in V - \{s\}$ **do**
3) $\quad dist[u] = cost(s, u)$;
4) $\quad$ Insert u into a 2-3 tree Q with $dist[u]$ as the key;
5) **for** $i = 1$ **to** $n - 1$ **do**
6) $\quad$ Identify the node u from Q with the least key;
7) $\quad S = S \cup \{u\}$; Delete u from Q;
8) $\quad$ **for** each $w \in Adj(u)$ **do**
9) $\quad\quad$ **if** $dist[u] + cost(u, w) < dist[w]$ **then** $dist[w] = dist[u] + cost(u, w)$;

- **Run time analysis:** Steps 1 through 4 take a total of $O(n \log n)$ time. Step 9 takes $O(\log n)$ time since it involves changing the key value of a node in Q. This can be done by deleting the node first and then reinserting the node with a new key value.

  The **for** loop of step 8 takes a total of $O(d_u \log n)$ time (for each iteration of the **for** loop of step 5) where d_u is the out-degree of the node u. When summed over all the iterations of the **for** loop of step 5, step 8 takes a total of $O(\sum_{u \in V} d_u \log n) = O(m \log n)$ time.

  Steps 6 and 7 take a total of $O(n \log n)$ time summing over all the iterations of the **for** loop of step 5.

  In summary, the total run time of Dijkstra's algorithm is $O((m + n) \log n)$.

An Example

- Consider the following graph: $G(V, E)$ with $V = \{s, 1, 2, 3, 4, 5\}$ and $cost(s, 1) = 15, cost(s, 3) = 2, cost(s, 4) = 10, cost(1, 2) = 5, cost(2, 4) = 1, cost(3, 4) = 1, cost(3, 2) = 3, cost(3, 1) = 3, cost(5, 2) = 5$, and $cost(5, 4) = 2$.

- At the beginning we have: $S = \{s\}$; $dist[1] = 15, dist[2] = \infty, dist[3] = 2, dist[4] = 10$, and $dist[5] = \infty$.

- When $i = 1$: Node 3 has the least *dist* value and hence it enters S next. The current value of $dist[1]$ is 15 which is larger than $dist[3] + cost(3, 1) = 5$ and hence $dist[1]$ changes to 5. Likewise, $dist[2]$ changes to 5 and $dist[4]$ changes to 3. $dist[5]$ continues to be ∞.

- When $i = 2$: Node 4 enters S. The *dist* values of the nodes $1, 2$, and 5 do not change.

- When $i = 3$: The nodes 1 and 2 have the same least *dist* value of 5. We can break the tie arbitrarily. Let the node 1 enter S next. The *dist* values of the nodes 2 and 5 do not change.

- When $i = 4$: The node 2 enters S next. The *dist* value of the node 5 does not change.

- When $i = 5$: The node 5 enters S next.
- In summary, the shortest path weights for the nodes $1, 2, 3, 4$, and 5 are $5, 5, 2, 3$, and ∞, respectively.

Note: Dijsktra's algorithm might give an incorrect answer when the input graph has negative edges.

- Consider the following graph: $G(V, E)$ where $V = \{s, 1, 2\}$ and $cost(s, 1) = 10, cost(s, 2) = 5$, and $cost(1, 2) = -8$.
- Let's employ Dijkstra's algorithm on this graph.
- At the beginning we have: $S = \{s\}$; $dist[1] = 10$, and $dist[2] = 5$.
- When $i = 1$: The node 2 has the least *dist* value and hence it will enter S next. This means that the algorithm will output 5 as the weight of the shortest path from s to 2, which is incorrect! There is a shorter path from the node s to 2 with a weight of 2.

Dynamic Programming

- There are problems for which greedy algorithms may not yield acceptable solutions. For example, there are problems for which multiple decision sequences might have to be investigated to arrive at an optimal solution. For such problems we can attempt to use the dynamic programming approach.
- There are three steps involved in any dynamic programming algorithm:
  1. Identify a function such that the solution we are looking for is the value of this function at a specific point;
  2. Write a recurrence relation for this function; and
  3. Start from the base case values of the function. Use the recurrence relation to evaluate the function at as many points as needed before the point of interest is reached.
- We will illustrate the dynamic programming approach using several examples. The first example to be investigated is the 0/1 knapsack problem.
- 0/1 knapsack: Input are n objects. Object i has a profit of p_i and a weight of w_i, for $1 \leq i \leq n$. We are also given a knapsack of capacity m. The problem is to compute $x_1, x_2, \ldots, x_n$ such that each x_i is either zero or one (for $1 \leq i \leq n$), $\sum_{i=1}^{n} w_i x_i \leq m$, and $\sum_{i=1}^{n} p_i x_i$ is maximum.