Algorithms in Systems Engineering
ISE 172

Lecture 20

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Chapter 7
- References
  - CLRS Chapter 24
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

Finding Shortest Paths in Acyclic Networks

- In acyclic networks, finding shortest paths is easier than in networks that have cycles.
- If we topologically order the nodes first, then we can compute shortest paths.
- Let $G = (V, E)$ be a given graph and let `order` be a topological ordering of the nodes. Then we can solve the SPP as follows.

```
def acyclic_shortest_paths(G):
    dist = {}
    for j in range(len(G.get_node_list())):
        m = order[j]
        if len(G.get_in_neighbors(m)) == 0: dist[m] = 0
        for n in G.get_neighbors(m):
            estimate = dist[m] + G.get_edge_weight(m ,n)
            if dist[n] > estimate:
                dist[n] = estimate
    return dist
```

Proof of Correctness for Algorithm

Claim: When the algorithm examines a node, its distance label is optimal.

Proof

Base Case: When we examine node 1, $d(1) = 0$ is correct. When we examine node 2, $d(2) = d(1) + c_{12}$ is correct because only node 1 can be inbound to node 2 (topological ordering).

Induction: Suppose that the algorithm has examined nodes $1, 2, \cdots, k$ and the distance labels are correct. Show that the distance label for node $k+1$ is correct.

Let the shortest path from s to $k+1$ be $s - i_1 - i_2 - \cdots - i_h - i_k + 1$.

Solving SPP with Non-Negative Arc Lengths

- When there are cycles, the situation is a bit more complex.
- Dijkstra's Algorithm generalizes the algorithm for the acyclic case.
- The difference is the order in which the nodes are examined.
- Nodes are divided into two groups
  - temporarily labeled
  - permanently labeled
- In order to produce the shortest paths tree, we keep track of the *predecessor node* each time a label is updated.
- Basic Idea: Fan out from source and permanently label nodes in order of distance from the source.

Dijkstra's Algorithm

- We will assume for now that the edge lengths are all positive.
- The idea of Dijkstra's Algorithm is to perform a graph search, updating the shortest known path to each encountered vertex as the search evolves.
- Throughout the algorithm, we maintain a quantity $d(v)$ for each node v, which represents the length of the shortest path found so far.
- We will call $d(v)$ the current *estimate* for node v.
- We start by assigning $d(r) = 0$ for the source node and $d(v) = \infty$ for all other nodes v.
- The node v to be processed next is the unprocessed node for which $d(v)$ is minimized.
- The processing step consists of updating the estimates for all the neighbors of v.

Algorithm Summary

- We are given a graph $G = (V, E)$ and a source node node r from which we want to find shortest paths to all other nodes.
- Algorithm
  - Initialize by assigning $d(r) = 0$ for the source node and $d(v) = \infty$ for all other nodes $v \in V \setminus \{r\}$.
  - Place r on the list Q of unprocessed nodes and set the list S of processed nodes to be empty.
  - While Q is not empty
    - Choose $v \in Q$ such that $d(v) = \min_{u \in Q} d(u)$.
    - Remove v from Q and add it to S
    - For each neighbor x of v, set $d(x) = \min\{d(x), d(v) + w_{\{v,x\}}\}$.
- When the algorithm is completed, we will have $d(v) = \delta(v)$ for all $v \in V$ and the search tree will be a shortest paths tree.
- Why is this algorithm correct?

Proof of Correctness

Claim: At the end of any iteration the following inductive hypotheses hold:

1. The distance label $d(i)$ is optimal for any node i in the set S.
2. The distance label $d(j)$ for any node $j \in \bar{S}$ is the length of the shortest path from the source to j such that all internal path nodes are in S.

Proof Strategy

- Show that statements 1 and 2 are true after the first iteration.
- Assume that they are true after iteration $i-1$ and prove that they hold after iteration i.
- (Assume iteration i moves node i from $\bar{S}$ to S.)

Dial's Implementation

- Node selection is bottleneck operation.
- Maintain distances in sorted fashion using following property

  Proposition 1. *The distance labels that Dijkstra's Algorithm designates as permanent are non-decreasing.*

- Create $nC + 1$ sets numbered $0, 1, \cdots, nC + 1$ and store all nodes with temporary distance label k in bucket k
- Reduce number of buckets to $C + 1$ using following property

  Proposition 2. *If $d(i)$ is the distance label designated as permanent at the beginning of an iteration, then at the end of an iteration $d(j) \leq d(i) + C$ for each finitely labeled node $j \in \bar{S}$.*

- Algorithm runs in $O(m + nC)$ time

Implementing the Algorithm with Priority Queues

- To implement the algorithm, we need to maintain the node list L as a *priority queue*.
- Recall from Lecture 15 that a priority queue is a data structure for maintaining a list of items that have associated *priorities*.
- The usual operations are
  - construct a queue from a list of items.
  - find the item with the highest priority.
  - insert an item.
  - delete an item.
  - change the priority of an item.
- A "naive" implementation is to simply maintain a vector of the estimates for each node and then scan through it each time to find the minimum.
- We may be able to do better using a heap.

Review: Heaps

- A *heap* is a binary tree with additional structure that allows it to function efficiently as a priority queue.
- The additional structure needed to support these operations is that the record stored at each node has a higher priority than either of its children.
- Any node with this property is said to satisfy the *heap property*.
- Consider a tree in which all nodes except for the root have the heap property.
- We can easily transform this into a tree in which every node has the heap property (how?).
- This operation is called `heapify()`.
- By calling `heapify()` on each node, starting at the lowest level and working upward, we can transform an unordered binary tree into a heap.

Review: Operations on a Heap

- The node with the highest priority is always the root.
- To delete a record
  - Exchange its record with that of a leaf.
  - Delete the leaf.
  - Call `heapify()`.
- To add a record
  - Create a new leaf.
  - Exchange the new record with that of the parent node if it has a higher priority.
  - Continue to do this until all nodes have the heap property.
- We can change the priority of a record in a similar fashion.

Running Time of Dijkstra's Algorithm

- This algorithm fits our definition of a graph search algorithm (roughly speaking), but cannot be analyzed in exactly the same way.
- We will consider the processing step to include both
  - deletion of the next node to be processed, and
  - adjustment of the estimates of all its neighbors.
- The "naive" implementation
  - The processing step has a running time in $O(n)$, for a total running time of $O(m + n^2) = O(n^2)$.
  - This is linear if the graph is dense.
- Using a heap
  - The deletion operation takes $O(\lg n)$ time and adjusting one estimate also takes $O(\lg n)$ time.
  - The time spent deleting elements is thus in $O(n \lg n)$
  - The total time spent adjusting the estimates is in $O(m \lg n)$.
  - This gives a total running time of $O((m + n) \lg n)$, which is just $O(m \lg n)$ if the graph is connected.
- Which algorithm is better?

Running Times of Other Implementations

d-Heap: $O(m \log_d n + nd \log_d n)$ ($d = \max\{2, \lceil m/n \rceil\}$)

Fibonacci Heap: $O(m + n \log n)$ (best strongly polynomial time algorithm)

Johnson's: $O(m \log \log C)$

Radix Heap: $O(m + n \log(nC))$

Fibonacci Radix: $O(m + n\sqrt{logC})$