References for Today’s Lecture

• Required reading
  – Section 21.2

• References
  – AMO Sections 4.5–4.7
  – CLRS Section 24.3
Solving SPP with Non-Negative Arc Lengths

- When there are cycles, the situation is a bit more complex.
- **Dijkstra’s Algorithm** generalizes the algorithm from Lecture 7 for the acyclic case.
- The difference is the order in which the nodes are examined.
- As before, 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

Input: An network $G = (N, A)$ and a vector of arc lengths $c \in \mathbb{Z}_+^A$

Output: $d(i)$ is the length of a shortest path from node $s$ to node $i$ and $\text{pred}(i)$ is the immediate predecessor of $i$ in an associated shortest paths tree.

$S := \emptyset$
$\bar{S} := N$
$d(i) \leftarrow \infty \forall i \in N$
$d(s) \leftarrow 0$ and $\text{pred}(s) \leftarrow 0$

while $|S| < n$ do
    let $i \in \bar{S}$ be the node for which $d(i) = \min\{d(j) : j \in \bar{S}\}$
    $S \leftarrow S \cup \{i\}$
    $\bar{S} \leftarrow \bar{S} \setminus \{i\}$
    for $(i, j) \in A(i)$ do
        if $d(j) > d(i) + c_{ij}$ then
            $d(j) \leftarrow d(i) + c_{ij}$ and $\text{pred}(j) \leftarrow i$
        end if
    end for
end while
Example of Dijkstra’s Algorithm
Proof of Correctness

Claim 1. 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 $\overline{S}$ to $S$.)
Running Time of Dijkstra’s Algorithm

• Note that Dijkstra’s Algorithm is a graph search procedure.

• It is very similar to Prim’s Algorithm.

• At each step, we need to update some node labels and then be able to determine the node with the minimum label.

• What is the running time for a naive implementation?
Dial’s Implementation

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

**Property 1. [4.5]** The distance labels that Dijkstra’s Algorithm designates as permanent are non-decreasing.

- Create $nC + 1$ buckets 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

**Property 2. [4.6]** 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 $j \in \overline{S}$.

- Algorithm runs in $O(m + nC)$ time
Implementation with Priority Queues

• To get a strongly polynomial time algorithm, we must use a more general data structure for maintaining a priority queue.

• For a given order set $H$, this data structure should support the operations

  – `push(item, value)` (to add and change value of an item)
  – `peek()`
  – `pop()`
Binary Heaps

• A binary heap is a balanced 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 each node has a higher priority than either of its children.

• Balanced binary trees can be stored very efficiently in a single array.
  – The root is stored in position 0.
  – The children of the node in position $i$ are stored in positions $2i + 1$ and $2i + 2$.
  – This determines a unique storage location for every node in the tree and makes it easy to find a node’s parent and children.
  – Using an array, basic operations can be performed very efficiently.
Creating the Heap

• Any node whose priority is higher than either of its children 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.

• This is how we create the initial heap.

• Note that this step is unnecessary for implementing Dijkstra’s. Why?
Operations on a Heap

- The node with the highest priority is always the root.
- To change the priority of a node
- To insert a node
- To delete a node

What are the running times of these operations?
Analyzing Dijkstra’s with a Binary Heap
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**:
- **Radix Heap**: $O(m + n \log(nC))$
- **Fibonacci Radix**: $O(m + n \sqrt{\log C})$