CSE 101, Winter 2016

Design and Analysis of Algorithms

Lecture 7: Bellman-Ford, SPs in DAGs, PQs

Class URL: http://vlsicad.ucsd.edu/courses/cse101-w16/
Negative Edges

- Dijkstra’s algorithm assumes that the shortest path from s to v must pass through vertices that are closer to s than v.
- This fails when there are negative edges in G (Figure 4.12)
Bellman-Ford Algorithm

• Idea: Successive Approximation / Relaxation
  – Find SP using $\leq 1$ edges
  – Find SP using $\leq 2$ edges
  – ...
  – Find SP using $\leq n-1$ edges $\rightarrow$ have true shortest paths
• Let $l_j^{(k)}$ denote shortest $v_0 - v_j$ pathlength using $\leq k$ edges
Bellman-Ford Algorithm

• **Idea: Successive Approximation / Relaxation**
  – Find SP using \( \leq 1 \) edges
  – Find SP using \( \leq 2 \) edges
  – …
  – Find SP using \( \leq n-1 \) edges \( \rightarrow \) have true shortest paths

• Let \( l_{j}^{(k)} \) denote shortest \( v_0 \rightarrow v_j \) path length using \( \leq k \) edges

• Then, \( l_{i}^{(1)} = d_{0j} \forall j = 1, \ldots, n-1 \) \( \// d_{ij} = \infty \) if no \( i-j \) edge

• In general, \( l_{j}^{(k+1)} = \min \{ l_{j}^{(k)} , \min_{i} (l_{i}^{(k)} + d_{ij}) \} \)
  – \( l_{j}^{(k)} \): don’t need \( k+1 \) arcs
  – \( \min_{i} (l_{i}^{(k)} + d_{ij}) \): view as length-k SP plus a single edge
Bellman-Ford vs. Dijkstra

EXAMPLE: \( SP(S, A) = S \rightarrow D \rightarrow C \rightarrow A \)

\( S \rightarrow D \rightarrow C \) is found at Pass 2, allowing \( S \rightarrow D \rightarrow C \rightarrow A \) to be found at Pass 3
Bellman-Ford (avoiding unneeded work)

Example: SP(S,A) = S → D → C → A

S → D → C is found at Pass 2, allowing S → D → C → A to be found at Pass 3
Bellman-Ford vs. Dijkstra

PASS: 1                2                        3                         4
Label A  8   min([8], 2+∞) = 8   min([8], 3+4) = 7   min([7], 4+1) = 5*
B  3   min([3], 2+∞) = 3*
C  ∞   min([∞], 2+2) = 4   min(4, 3+∞) = 4*
D  2*
Special Case: Longest/Shortest Paths in DAGs

- **(Single-Source) Longest-Path Problem**: well-defined only when there are no cycles
- **DAG**: can topologically sort the vertices
  \[ \rightarrow \text{labels } v_1, \ldots, v_n \text{ s.t. all edges directed from } v_i \text{ to } v_j, \ i < j \]

- Let \( l_j \) denote **longest** \( v_0 \rightarrow v_j \) path length
  - \( l_0 = 0 \)
  - \( l_1 = d_{01} \quad // \quad d_{ij} = -\infty \) if no i-j edge
  - \( l_2 = \max(d_{01} + d_{12}, d_{02}) \)
  - In general, \( l_k = \max_{j < k} (l_j + d_{jk}) \)

\[
l(z) = \max (l(x) + d_{xz}, l(y) + d_{yz})
\]
**Special Case: Longest/Shortest Paths in DAGs**

- **(Single-Source) Longest-Path Problem**: well-defined only when there are no cycles
- **DAG**: can topologically sort the vertices
  - labels $v_1, \ldots, v_n$ s.t. all edges directed from $v_i$ to $v_j$, $i < j$

- Let $l_j$ denote **longest** $v_0 - v_j$ path length
  - $l_0 = 0$
  - $l_1 = d_{01}$ // $d_{ij} = -\infty$ if no $i$-$j$ edge
  - $l_2 = \max(d_{01} + d_{12}, d_{02})$
  - In general, $l_k = \max_{j < k} (l_j + d_{jk})$

- **Shortest path length** in DAG
  - replace max by min
    - $l(z) = \max (l(x) + d_{xz}, l(y) + d_{yz})$
    - // use $d_{ij} = +\infty$ if no $i$-$j$ edge
DAG Longest/Shortest Paths Complexity

- Generic Bellman-Ford = $O(VE)$
- In a DAG: Topological sort = $O(V+E)$ (DFS)

- Edges **out of** vertex $v$ aren’t “processed” (traversed) until after all edges **in to** $v$ have been processed
  - Runtime $O(V+E)$
  - **Exercise:** Understand why runtime is $O(V+E)$ for both longest-path and shortest-path in a DAG

- Application: PERT (program evaluation and review technique) — **critical path** is the longest path in the DAG
Dynamic Sets

• Dynamic sets (data structures):
  – change a dictionary, e.g., add/remove words
  – reuse of **structured** information
  – fast updating for on-line algorithms

• Elements:
  – **key** is element ID
    • dynamic set of key values
  – **satellite** information associated with key

• Operations
  – **query**: return information about the set
  – **modify**: change the set
Dynamic Set Operations

- **Search**(S,k)
  - Given set S and key value k, return pointer x to an element of S such that key[x] = k, or NIL if no such element

- **Insert**(S,x)
  - Augment set S with element pointed to by x

- **Delete**(S,x)
  - Given pointer x to an element in set S, remove x from S

- **Minimum**(S) / **Maximum**(S)
  - Given totally ordered set S, return pointer to element of S with smallest / largest key
Dynamic Set Operations

• **Predecessor / Successor**\((S, x)\)
  – Given element \(x\) whose key is from a totally ordered set \(S\), return a pointer to next smaller / larger element in \(S\), or \(NIL\) if \(x\) is minimum / maximum element

• **Union**\((S, S')\)
  – Given two sets \(S, S'\), return a new set \(S = S \cup S'\)
Elementary Data Structures

- Different data structures support/optimize different operations
- **Stack** has *top*, LIFO policy
  - insert = push x: top(S) = top(S)+1; S[top(S)] = x  $O(1)$
  - delete = pop  $O(1)$

  \[
  \begin{array}{cccccccc}
  1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  \hline
  15 & 6 & 2 & 9 & & & & \\
  \end{array}
  \]

  top[S] = 4

  \[
  \begin{array}{cccccccc}
  1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  \hline
  15 & 6 & 2 & 9 & 17 & & & \\
  \end{array}
  \]

  After push(S,17)  top[S] = 5

- **Queue** has *head*, *tail*, FIFO policy
  - insert = enqueue: add element to the tail  $O(1)$
  - delete = dequeue: remove element from the head  $O(1)$

  \[
  \begin{array}{cccccccc}
  1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  \hline
  & & 15 & 6 & 2 & 9 & & \\
  \end{array}
  \]

  head = 2  tail = 6

  \[
  \begin{array}{cccccccc}
  1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  \hline
  & & 15 & 6 & 2 & 9 & 8 & \\
  \end{array}
  \]

  head = 2  After enqueue(Q,8)  tail = 7
Priority Queue (PQ) Abstract Data Structure

• Operations:
  – Insert(S,x) : add element x
  – Minimum(S) / Maximum(S): return element with min/max key
  – DeleteMin(S) / DeleteMax(S): return min/max key, remove element

• Applications
  – Simulation systems: key ≡ event time
  – OS scheduler: key ≡ job priority
  – Numerical methods key ≡ inherent error in operation
  – Dijkstra’s shortest-path algorithm
  – Prim’s minimum spanning tree algorithm

• Question: How do we use a PQ to sort?
What Are Naïve PQ Implementations?

• **Answer**: Insert elements one by one; perform DeleteMin n times]

• Unordered list
  – Insert O(1)
  – DeleteMin O(n)

• Ordered list
  – Insert O(n)
  – DeleteMin O(1)

• **Observation**: If Insert, DeleteMin could each be accomplished in O(log n) time, then we would have an O(n log n) sorting algorithm (= heapsort)
Heaps

• A heap is a binary tree of depth $d$ such that
  – (1) all nodes not at depth $d-1$ or $d$ are internal nodes
    $\rightarrow$ each level is filled before the next level is started
  – (2) at depth $d-1$ the internal nodes are to the left of the leaves and have degree 2, except perhaps for the rightmost, which has a left child
    $\rightarrow$ each level is filled left to right

• A max heap (min heap) is a heap with node labels from an ordered set, such that the label at any internal node is $\geq (\leq)$ the label of any of its children
  $\rightarrow$ All root-leaf paths are monotone
Heaps, and Sorting With Heaps

• **Fact:** Every node in a heap is the root of a heap (!)

• **How do we store a heap?**
  – Implicit data structure:  // maxheap example
    Array index: 1 2 3 4 5 6 7 8 9 10
    Value: 20 11 5 5 3 2 3 4 1 2

• **How do we sort using a heap?**
  – **Insert:** Put new value at A[n]; fix violation of heap condition ("re-heapify")
  – **DeleteM**: Remove root; replace by A[n]; re-heapify
    • If maxheap, DeleteMax (return largest element first)
    • If minheap, DeleteMin (return smallest element first)
Heaps

• Pointers:
  – Parent
  – Left, Right (children)

• Parent \( \leq \) Child \( \rightarrow \) this is a minheap example
Heap Operations

**Insert**($S,x$): $O$(height) $\rightarrow$ $O$(log $n$)

**Extract-min**($S$): return head, replace head key with the last key, float down $\rightarrow$ $O$(log $n$)

“Float down”: If heap condition violated, swap with smaller child

Keep swapping with parent until heap condition satisfied
O(n log n) Heapsort

- Build heap easy time bound: \( n \times O(\log n) \) time
  - for \( i = 1..n \) do insert \((A[1..i], A[i])\)
- Extract elements in sorted order: \( n \times O(\log n) \) time
  - for \( i = n..2 \) do
    - Swap \((A[1] \leftrightarrow A[i])\)
    - Heapsize = Heapsize-1
    - Float down \( A[1] \)
Actual Time To Build Heap: $O(n)$

- Heapify $(i,j)$ makes range $[i,j]$ satisfy heap property:
  
  ```
  Heapify (i,j)  // minheap
  if i not a leaf and child of i is < i
  let k = smaller child of i
  interchange a[i], a[k]
  Heapify (k,j)
  ```

BuildHeap: for $i = n$ to 1 do Heapify $(i,n)$
Actual Time To Build Heap: $O(n)$

- Heapify $(i,j)$ makes range $[i,j]$ satisfy heap property:

  Heapify $(i,j)$  // minheap
  if $i$ not a leaf and child of $i$ is $< i$
  let $k =$ smaller child of $i$
  interchange $a[i], a[k]$
  Heapify $(k,j)$

BuildHeap: for $i = n$ to $1$ do Heapify $(i,n)$

- We will show that BuildHeap actually takes $O(n)$ time (!)

- Observation: If vertices $i+1, \ldots, n$ are roots of heaps, then after Heapify$(i,n)$ vertices $i, \ldots, n$ will be roots of heaps
Actual Time To Build Heap: $O(n)$

- **BuildHeap**: for $i=n$ to 1 do Heapify $(i,n)$
- Observation: If vertices $i+1$, ..., $n$ are roots of heaps, then after Heapify$(i,n)$ vertices $i,...,n$ will be roots of heaps
- Let $T(h) \equiv$ time for Heapify on $v$ at height $h \rightarrow T(h) = O(h)$
Actual Time To Build Heap: $O(n)$

- **BuildHeap**: for $i=n$ to 1 do Heapify $(i,n)$
- Observation: If vertices $i+1, \ldots, n$ are roots of heaps, then after Heapify $(i,n)$ vertices $i, \ldots, n$ will be roots of heaps

- Let $T(h) \equiv$ time for Heapify on $v$ at height $h \Rightarrow T(h) = O(h)$

- Heapify called once for each $v$
  \[ \Rightarrow \text{total BuildHeap time is } O(\Sigma_v h(v)) \]
- Vertex at height $i$ is root of heap with $2^{i+1}$ nodes
  \[ \Rightarrow \left\lceil \frac{n}{2^{i+1}} \right\rceil \text{ vertices at height } i \]
  \[ \Rightarrow \Sigma i \cdot \frac{n}{2^i} \text{ is upper bound on BuildHeap time} \]

**Fact:** $\Sigma i/2^i = 2 \Rightarrow O(n)$ bound

\[
X = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \ldots = [\frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots] + [\frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \ldots] = 1 + \frac{X}{2} = 2
\]