CSE 101, Winter 2018
Design and Analysis of Algorithms
Lecture 7: Bellman-Ford, SPs in DAGs, PQs
Class URL: http://vlsicad.ucsd.edu/courses/cse101-w18/
Figure 4.10: Single-Edge Extensions of SPs

- Find shortest path to $v_{k+1}$ by extending by a single edge the shortest path to one of $v_0, v_1, \ldots, v_k$

All $y \in R$: know true SP costs from $s$

If Dijkstra picks $v$ to have next permanent label, then there is no “closer” vertex to $s$ (not already in $R$)

* green path cannot have shorter path cost

Added after class

Lec. 6
Proof of Dijkstra Correctness

**Note:** P is the entire purported path that Dijkstra "failed to find" part of P.

Suppose v gets next permanent label (by definition of temp label of y).

Can path \( P \) be shorter than \( s \rightarrow u \rightarrow v \) path??

Suppose \( P \) is shorter...

\[ \text{length}(P) \geq \text{length}(P') + l(x,y) \geq d^*(x) + l(x,y) \geq l(y) \geq l(v) \]

**Suppose** \( P \) is shorter...

All edge weights non-negative.
Dijkstra key points ...

- Greedy
- Can be incorrect if \( \exists \) a neg-weight edge
- \( \text{min label} \leftrightarrow PQ \) (log \( |V| \) insertion if binary heap)
  \[ V \cdot \log V \]
  \[ E \cdot \log V \]
  \[ \Rightarrow O((V+E) \log V) \]
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 ≤ 1 edges
  – Find SP using ≤ 2 edges
  – ...
  – Find SP using ≤ n-1 edges \(\rightarrow\) have true shortest paths

• Let \(l_j^{(k)}\) denote shortest \(v_0 - v_j\) pathlength using ≤ \(k\) 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 \( v_0 \rightarrow v_j \) pathlength using \( \leq k \) edges
- Then, \( l_i^{(1)} = d_{0j} \) \( \forall j = 1, \ldots, n-1 \) \( \quad \text{// } d_{ij} = \infty \text{ if no } i-j \text{ edge} \)
- 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

**PASS:** 1 2 3 4

**LABEL**

A \(8\) min \((8, 3+4, \infty +1, 2+ \infty) = 7\) min \((7, 3+4, 4+1, 2+\infty) = 5\) min \((5, 3+4, 4+1, 2+\infty) = 5\)

B \(3\) min \((3, 8+4, \infty+\infty, 2+\infty) = 3\) min \((3, 7+4, 4+\infty, 2+\infty) = 3\) min \((3, 5+4, 4+\infty, 2+\infty) = 3\)

C \(\infty\) min \((\infty, 8+1, 3+\infty, 2+2) = 4\) min \((4, 7+1, 3+\infty, 2+2) = 4\) min \((4, 5+1, 3+\infty, 2+2) = 4\)

D \(2\) min \((2, 8+\infty, 3+\infty, \infty+2) = 2\) min \((2, 7+\infty, 3+\infty, 4+2) = 2\) min \((2, 5+\infty, 3+\infty, 4+2) = 2\)

**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)

PASS:  1  2  3  4

LABEL  A  min(8, 3+4, ∞ +1) = 7  etc.

B  min(3, 8+4) = 3  etc.

C  min(∞, 8+1, 2+2) = 4  etc.

D  min(2, ∞+2) = 2  etc.

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
PASS: 1 2 3 4

Label A 8 \( \min([8], 2+\infty) = 8 \) \( \min([8], 3+4) = 7 \) \( \min([7], 4+1) = 5^* \)

B 3 \( \min([3], 2+\infty) = 3^* \)

C \( \infty \) \( \min([\infty], 2+2) = 4 \) \( \min(4, 3+\infty) = 4^* \)

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

\[
l(z) = \max (l(x) + d_{xz}, l(y) + d_{yz})
\]
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
  // use $d_{ij} = +\infty$ if no i-j edge

\[ l(z) = \max (l(x) + d_{xz}, l(y) + d_{yz}) \]
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 \(\text{key}[x] = k\), 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: $\text{top}(S) = \text{top}(S)+1$; $S[\text{top}(S)] = x$ $O(1)$
- delete = pop $O(1)$

![Stack example](image)

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

![Queue example](image)
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
    → 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
    → 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
  → 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**  
    parent of $A[i]$ has index $= i \div 2$
  - **Left, Right (children)**  
    children of $A[i]$ have indices $2i, 2i+1$

- **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, ..., 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 $\Rightarrow T(h) = O(h)$

• Heapify called once for each v $\Rightarrow$ total BuildHeap time is $O(\sum_v h(v))$
• Vertex at height i is root of heap with $2^{i+1}$ nodes $\Rightarrow \lceil n/2^{i+1} \rceil$ vertices at height i $\Rightarrow \sum i \cdot n/2^i$ is upper bound on BuildHeap time

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

\[
X = 1/2 + 2/4 + 3/8 + 4/16 + \ldots = [1/2 + 1/4 + 1/8 \ldots] + [1/4 + 2/8 + 3/16 + \ldots] \\
= 1 + X/2 = 2
\]