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 \( v_0, v_1, \ldots, v_k \)

\[ R \]

Known region

\[ s \]

\[ u \]

\[ v \]

\( \forall x \in R: \) know true SP costs from \( s \)

\( \heartsuit \) if Dijkstra picks \( v \) to have next permanent label, then there is no "closer" vertex to \( s \) (not already in \( R \))

\( \times \) green path cannot have shorter path cost
Proof of Dijkstra Correctness

Suppose $v$ gets next permanent label

(by def'n of temp label of $y$)

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

because Dijkstra chose $v$!

Suppose $P$ is shorter...

$\text{length}(P) \leq \text{length}(P^{'}) + l(x,y) + \text{length}(\text{green subpath})$ from $y$ to $v$.

$P$ is the shortest path from $s$ to $v$.

Note: $P$ is the entire purported path Dijkstra found but Dijkstra failed to part of $P$.

$\star \text{length}(P) = \text{length}(P^{'}) + l(x,y) + \text{length} (\text{green subpath})$

$SP(s,x) \equiv \text{not necessarily } P'$

all edge weights non-neg

$\text{SP cost from } s \text{ to } x$

$\text{SP}(s,x)$

$\star \text{length}(P) = \text{length}(P^{'}) + l(x,y) \geq d^*(x) + l(x,y) \geq l(y) \geq l(v)$
Dijkstra key points ...

- **Greedy**
- Can be incorrect if \( \exists \) a neg-weight edge
- \( \min \) label \( \leftrightarrow \) \( PQ \) \( \quad (\log V) \) \( \) 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)
Familiar Recurrence: Pascal’s Triangle

- \( \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \)
  
  #combinations of \( n \) objects taken \( k \) at a time

2 cases:

1. “my choice includes the \( n \)th object”

2. “my choice does not include the \( n \)th object”
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

Each iteration "grows the edge budget by one (edge)"
Bellman-Ford Algorithm

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

- Let \( l^{(k)}_j \) denote shortest \( v_0 - v_j \) path length using \( \leq k \) edges

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

- In general, \( l^{(k+1)}_j = \min \{ l^{(k)}_j, \ \min_i (l^{(k)}_i + d_{ij}) \} \)
  - \( l^{(k)}_j \): don’t need \( k+1 \) arcs
  - \( \min_i (l^{(k)}_i + d_{ij}) \): view as length-\( k \) SP plus a single edge
Want SP costs from $v_0$ to each $v_j$

Vertices

\[ j = 1, 2, 3, \ldots, n-1 \]

\[ k = 1, 2, 3, \ldots, n-1 \]

\[ \leq \text{#edges} \]

\[ O(E) \]

\[ l^{(2)} \]

\[ l^{(1)} \]

\[ = \min \left( l^{(1)}, l^{(2)} \right) \]

\[ \min_{i \neq 1} l^{(1)} + d^{(1)} \]

\[ 6 \]

\[ 10 \]

\[ 20 \]

\[ 3 + 2 \]

\[ \infty + 15 \]

\[ = 5 \]
Bellman-Ford Takeaways

- Relaxing or successively approximating the shortest paths from source \((v_0)\) to all other vertices \((v_j)'s\) in passes that “grow” the “edge budget” (number of edges allowed in SPs)

- There are \(O(V)\) passes

- There is a “recurrence” that has two cases (looking back from the \((k+1)^{st}\) pass to the \(k^{th}\)-pass shortest-path costs): either the \((k+1)^{st}\) edge didn’t help (red case), or else it helped (blue case)

- Delivers correct shortest-path costs even if negative-weight edges are present (but, would not if there are negative-weight cycles)

- Can detect presence of negative-weight cycles by seeing if path costs when \(k = |V|\) have decreased from path costs when \(k = |V| - 1\)

- Each pass requires \(O(E)\) work because we look at all incident (incoming, in directed case) edges of each vertex (blue case in previous slide) when we take a min over “how to get to \(v_i\) in \(\leq k\) edges, plus the edge cost from \(v_i\) to \(v_j\)”
Bellman-Ford Execution (w/unneeded work)

PASS: 1

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

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

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

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

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

\( S \rightarrow D \rightarrow C \) 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  8  \(\min(8, 3+4, \infty +1) = 7\)  etc.

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

C  \(\infty\)  \(\min(\infty, 8+1, 2+2) = 4\)  etc.

D  2  \(\min(2, \infty+2) = 2\)  etc.

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
Dijkstra Execution

PASS: 1                2                        3                         4
Label  A  min([8], 2+\infty) = 8  min([8], 3+4) = 7  min([7], 4+1) = 5*
B        min([3], 2+\infty) = 3*
C        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
  - labels $v_1, \ldots, v_n$ s.t. all edges directed from $v_i$ to $v_j$, $i < j$

- Let $l_j$ denote *longest* $v_0 - v_j$ 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** $v_0 - v_j$ 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, $l_k = \max_{j<k} (l_j + d_{jk})$

- **Shortest** pathlength in DAG
  - replace max by **min**
    - // use $d_{ij} = +\infty$ if no $i$-$j$ edge

\[
\begin{align*}
l(z) &= \min(l(x) + d_{xz}, l(y) + d_{yz})
\end{align*}
\]
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 into $v$ 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
Lecture 1 Fun Problems

- Triangle finding in a graph
  \[ (i, j, k) \text{ s.t. } (v_i, v_j), (v_j, v_k), (v_i, v_k) \in E \]

- Celebrity Problem

- Max-Min Problem
  \( O(n) \) possible
  LB's
  DFS/sink?
Any Other Questions About MT?

• Bring a dark writing implement and photo ID

• Know where you are seated (seating chart)

• General structure is known; “advice” Doc is posted;
**Figure 4.8** Dijkstra’s shortest-path algorithm.

procedure \textit{dijkstra}\((G, l, s)\)

Input: Graph \(G = (V, E)\), directed or undirected;
positive edge lengths \(\{l_e : e \in E\}\); vertex \(s \in V\)

Output: For all vertices \(u\) reachable from \(s\), \(\text{dist}(u)\) is set to the distance from \(s\) to \(u\).

\[
\begin{align*}
\text{for all } u & \in V: \\
\text{dist}(u) & = \infty \\
\text{prev}(u) & = \text{nil} \\
\text{dist}(s) & = 0
\end{align*}
\]

\(H = \text{makequeue}(V)\) (using \text{dist}-values as keys)

while \(H\) is not empty:
\(u = \text{deletemin}(H)\)

\[
\text{for all edges } (u, v) \in E: \\
\text{if } \text{dist}(v) > \text{dist}(u) + l(u, v): \\
\text{dist}(v) = \text{dist}(u) + l(u, v) \\
\text{prev}(v) = u \\
\text{decreasekey}(H, v)
\]
PQ Implementations

Which heap is best?
The running time of Dijkstra’s algorithm depends heavily on the priority queue implementation used. Here are the typical choices.

| Implementation  | delete/\min  | insert/decrease\key | \(|V| \times \text{delete/\min} + (|V| + |E|) \times \text{insert}\) |
|----------------|--------------|---------------------|-------------------------------------------------|
| Array          | \(O(|V|)\)  | \(O(1)\)            | \(O(|V|^2)\)                                    |
| Binary heap    | \(O(\log |V|)\) | \(O(\log |V|)\)      | \(O((|V| + |E|) \log |V|)\)                      |
| d-ary heap     | \(O\left(\frac{d \log |V|}{\log d}\right)\) | \(O\left(\frac{\log |V|}{\log d}\right)\) | \(O\left((|V| \cdot d + |E|) \frac{\log |V|}{\log d}\right)\) |
| Fibonacci heap | \(O(\log |V|)\) | \(O(1)\) (amortized) | \(O(|V| \log |V| + |E|)\)                         |

So for instance, even a naive array implementation gives a respectable time complexity of \(O(|V|^2)\), whereas with a binary heap we get \(O((|V| + |E|) \log |V|)\). Which is preferable?

This depends on whether the graph is sparse (has few edges) or dense (has lots of them). For all graphs, \(|E|\) is less than \(|V|^2\). If it is \(\Omega(|V|^2)\), then clearly the array implementation is the faster. On the other hand, the binary heap becomes preferable as soon as \(|E|\) dips below \(|V|^2/\log |V|\).
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 \(O(1)\)

```
1 2 3 4 5 6 7
15 6 2 9 [ ] [ ] [ ]
```

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

```
1 2 3 4 5 6 7
[ ] [ ] [ ] [ ] [15 6 2 9]
```

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

```
1 2 3 4 5 6 7
15 6 2 9 17 [ ] [ ]
```

- 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 \equiv \text{event time}
  – OS scheduler: key \equiv \text{job priority}
  – Numerical methods key \equiv \text{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")
  - **DeleteMax:** 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** \( \text{parent of } A[i] \text{ has index } = i \div 2 \)
  - **Left, Right (children)** \( \text{children of } A[i] \text{ have indices } 2i, 2i+1 \)
- **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
O(n log n) Heapsort

- Build heap easy time bound: \( n \times O(\log n) \) time
  - for \( i = 1..n \) do \text{insert} \((A[1..i], A[i])\)

- Extract elements in sorted order: \( n \times O(\log n) \) time
  - for \( i = n..2 \) do
    - \text{Swap} \((A[1] \leftrightarrow A[i])\)
    - \text{Heapsize} = \text{Heapsize-1}
    - \text{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, ..., n are roots of heaps, then
  after Heapify(i,n) vertices i,...,n will be roots of heaps
Actual Time To Build Heap: $O(n)$

- **BuildHeap**: for $i=1$ to $n$ 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 $h$ $\Rightarrow T(h) = O(h)$
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 $O(\Sigma v \cdot h(v))$

- Vertex at height $i$ is root of heap with $2^{i+1}$ nodes
  $\Rightarrow \lceil n/2^{i+1} \rceil$ vertices at height $i$
  $\Rightarrow \Sigma i \cdot n/2^i$ is upper bound on BuildHeap time

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

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