Lecture 21: Shortest Path
(CLRS 24.3)

April 11, 2002

1 Shortest path

• Last time we started discussing shortest path problems.

  Example: Shortest path from $a$ to $e$ (of length 21)

  ![Graph Diagram]

  • We discussed some interesting properties of shortest path, e.g. that if $P = u = v_0, v_1, v_2, \ldots, v_k = v$ is shortest path from $u$ to $v$ then for all $i < k$, $P' = u = v_0, v_1, v_2, \ldots, v_i$ is shortest path from $u$ to $v_i$.

  • Today we will first solve the single source shortest path problem (SSSP):

    - Given a graph $G = (V, E)$ with non-negative weights and a source vertex $s$, find the length of the shortest path from $s$ to all other vertices $v \in V$. The length/weight $w(P)$ of a path $P = v_0, v_1, v_2, \ldots, v_k$ is defined as $\sum_{i=1}^{k} w(v_{i-1}, v_i)$.

  • Note:

    - A SSSP solution is a spanning tree for $G$.
    - Solving the SSSP problem with source $u$ is the best algorithm we know for computing the shortest path between $u$ and $v$.

1.1 SSSP for graphs with non-negative weights—Dijkstra’s algorithm

• Recall Prim’s greedy minimum spanning tree algorithm:

  - Grows tree out from source $s$; repeatedly add minimum edge out of tree
  - Correct by “cut theorem”
  - Implemented using priority queue on vertices not yet in the tree
- Dijkstra’s greedy algorithm for SSSP works almost the same way:
  - Grow set (tree) $S$ of vertices we know the shortest path to; repeatedly add new vertex $v$ that can be reached from $S$ using one edge. $v$ is chosen as the vertex with the minimal path weight among paths $<s=v_0,v_1,\ldots,v_i,v>$ with $v_j \in S$ for all $j \leq i$
  - Implemented using priority queue on vertices in $V \setminus S$.

Dijkstra(s)

FOR each $v \in V$ DO
  $d[v] = \infty$
  INSERT $(Q, v, \infty)$
OD

$S = \emptyset$

$d[s] = 0$

CHANGE$(Q, s, 0)$

WHILE $Q$ not empty DO
  $u = \text{DELETEMIN}(Q)$
  $S = S \cup \{u\}$
  FOR each $e = (u, v) \in E$ with $v \in V \setminus S$ DO
    IF $d[v] > d[u] + w(u, v)$ THEN
      $d[v] = d[u] + w(u, v)$
      CHANGE$(Q, v, d[v])$
      visit$[v] = u$
    FI
  OD
OD

- Example:
• Analysis:
  - While loop runs $|V|$ times $\Rightarrow$ we perform $|V|$ DELETEMIN operations
  - We perform at most one CHANGE operation for each of the $|E|$ edges
    $\Downarrow$
    $O(|E| + |V| \log |E|) = O(|E| \log |V|)$ running time

• Note:
  - Running time like Prim’s minimal spanning tree algorithm
  - Algorithm computes shortest path tree (stored using visit[v]) which can be used to find actual shortest paths
  - Algorithm works for directed graphs as well
  - Like Prim’s algorithm, Dijkstra’s algorithm can be improved to $O(|V| \log |V| + |E|)$ using another heap (Fibonacci heap)
• Correctness:
  
  – We prove correctness by induction on size of $S$
  
  – We will prove that after each iteration of the while-loop the following invariant holds:
    
    a) $v \notin S \Rightarrow d[v]$ is length of shortest path from $s$ to $v$ among path of the form $<s,v_0,v_1,\ldots,v_k,v>$ where $v_1,v_2,\ldots,v_k \in S$
    
    b) $v \in S \Rightarrow d[v] = \delta(s,v)$ ($\delta(s,v)$ is length of shortest path from $s$ to $v$)
    
    \[\downarrow\]
    
    When algorithm terminates ($S = V$) we have solved SSSP
  
  – Proof:
    
    Invariant trivially holds initially ($S = \emptyset$). To prove that invariant holds after one iteration of while-loop, given that it holds before the iteration, we need to prove that after adding $u$ to $S$:
    
    a) $d[v]$ correct for all $(u,v) \in E$ where $v \notin S$
      
      • Easily seen to be true since $d[v]$ explicitly updated by algorithm (all the new paths to $v$ of the special type go through $u$)
    
    b) $d[u] = \delta(s,u)$
      
      • Assume $d[u] > \delta(s,u)$, that is, the found path is not the shortest
      
      • Consider shortest path to $u$ and edge $(x,y)$ on this path where $x \in S$ and $y \notin S$ (such an edge must exist since $s \in S$ and $u \notin S$)

    \[
    \begin{array}{c}
    \text{Shortest path from } s \text{ to } u \\
    \text{Path from } y \text{ to } u \text{ has weight } w
    \end{array}
    \]

    • We chose $u$ such that $d[u]$ was minimized $\Rightarrow d[y] > d[u] \Rightarrow w$ must me $< 0 \Rightarrow$ contradiction since all weights are non-negative (note that we use that $d[y]$ is shortest path to $y$)

2 All pairs shortest path (APSP)—non-negative weights

• In the APSP problem, we want to compute the shortest path between any two vertices $u,v \in V$
  
  – Note that the output is of size $O(|V|^2)$ so we cannot hope to design a better than $O(|V|^2)$ time algorithm

• We can solve the problem simply by running Dijkstra’s algorithm $|V|$ times $\Rightarrow O(|V| \cdot |E| \log |V|)$ algorithm
  
  – In the worst case (dense graph) this is $O(|V|^3 \log |V|)$
• We can obtain a much simpler $O(|V|^3)$ algorithm by working on adjacency matrix $A$:

```plaintext
FOR k = 1 to |V| do
  FOR i = 1 to |V| DO
    FOR j = 1 to |V| DO
      FI
    OD
  OD
OD
```

• Correctness:

  - We prove correctness by induction
  - We will prove that after each iteration of the $k$-loop the following invariant holds:
    After the $k$'th (out of $|V|$) iterations, $A[i, j]$ contains the length of shortest path from $v_i$ to $v_j$ that (apart from $v_i$ and $v_j$) only contains vertices of index at most $k$
    $$\downarrow$$
    When algorithm terminates we have solved APSP
  - Proof:
    * Invariant holds initially (we start with adjacency matrix $A$).
    * When “adding” vertex with index $k$ we explicitly check all new paths between $v_i$ and $v_j$ through $v_k$ for all $|V|^2$ pairs.

• Note:

  - We can easily produce adjacency-matrix from adjacency list in $O(|V|^2)$ time
  - Algorithm runs in $O(|V|^3)$ time, even if the graph is sparse. Using algorithm based on Dijkstra’s algorithm we will get much better performance for sparse graphs.
  - Using more efficient heap, algorithm based on Dijkstra’s algorithm can be improved to $O(|V|^2 \log |V| + |V| \cdot |E|) = O(|V|^3)$