CS 151

Graph Algorithms

Monday, December 3, 12

Announcements

1. prelab 10 is due now. Except for the three parts you couldn't do, which you should hand in on Friday.

Monday, December 3, 12

(Possible Negative) Weighted Shortest Paths

1. input: a graph G=(V,E) with arbitrary edge costs and a start vertex s
2. goal: determine the cost of the least-cost path from s to every vertex in V (and while you're at it, compute the path itself.)
3. Q: will Dijkstra's still work?
   A: sadly, no
   • Dijkstra's doesn't work because once we explore a vertex, we never change its s.p. cost, but with negative costs we may want to.
   • also, shortest paths may not even exist if there are negative-cost cycles!
4. we'll need a completely new algorithm this time.
   idea: Bellman-Ford(s), a dynamic programming algorithm
   • solve the shortest paths problem using recursion...
   • but then we'll need a completely new algorithm this time.
   • instead of shortest paths, we'll need a completely new algorithm this time.
   • the shortest paths problem has no dynamic programming solutions.
   • we'll need a completely new algorithm this time.
   • dynamic programming solutions don't work because once we explore a vertex, we never revisit it.
5. Ex. a recursive solution for Fib is:
   - fib(k) = fib(k-1) + fib(k-2)
   - set F[0] = 0, F[1] = 1
   - for k=2,3,...n
   - set F[k] = F[k-1] + F[k-2]
   - return F[k]
   C: will this work?
   Q: will Dijkstra's still work?
   (and while you're at it, compute the path itself.)
   • goal: determine the cost of the least-cost path from s to every vertex in V
   • input: a graph G=(V,E) with arbitrary edge costs c and a start vertex s
   • output: the cost of the shortest path from s to every vertex in V.

   First up: figure out a recursive solution.
   Let BF(v,k) be the cost of the shortest path from s to v on at most k edges.
   Note that the s.p. from s to v on at most k edges either
   • uses up to k edges (these cases are not mutually exclusive),
   • or uses at most k-1 edges (these cases are not mutually exclusive).
   • BFS(v,k) be the cost of the shortest path from s to v on at most k edges.

   Note that the s.p. from s to v on at most k edges either
   • uses up to k edges (these cases are not mutually exclusive),
   • or uses at most k-1 edges (these cases are not mutually exclusive).
   • BFS(v,k) be the cost of the shortest path from s to v on at most k edges.

   Let BF(v,k) be the cost of the shortest path from s to v on at most k edges.
   The shortest s-v path on at most k edges is the minimum of the 2 cases:
   BF(v,k) = min{ BF(v,k-1), min{ BF(u,k-1) + c(u,v)} }

   u: (u,v) ∈ E
   (these cases are not mutually exclusive, that's okay.)

   As base cases, we need BF(v,0) = ∞, BF(s,0) = 0.

   BF(v,k) = min{ BF(v,k-1), min{ BF(u,k-1) + c(u,v)} }
   (these cases are not mutually exclusive, that's okay.)

   BF(v,k) = min{ BF(v,k-1), min{ BF(u,k-1) + c(u,v)} }

   u: (u,v) ∈ E
   (these cases are not mutually exclusive, that's okay.)
The Bellman-Ford Algorithm

Recursive solution would be

\[
BF(v, k) =
\begin{cases}
0 & \text{if } k = 0 \text{ and } v = s,
\infty & \text{if } k = 0 \text{ (and } v \neq s),
\min \{ BF(v, k-1), min \{ BF(u, k-1) + c(u, v) \} \} & \text{for all other } v \text{ and } k > 0.
\end{cases}
\]

But this solution could make an exponential number of recursive calls, blech!

Unwind it to its iterative solution and solve all vertices and all k at once:

\[
BF() \text{ -- iteratively compute } BF(v, k) \text{ for all } v \text{ and all } k = 0, \ldots, n
\]

\[
\begin{align*}
\text{create } n \times n \text{ array } BF \\
\text{initialize } BF[s][0] = 0, \text{ for all other } v, BF[v][0] = \infty \\
\text{for } k = 1, 2, \ldots, n \\
\text{for each vertex } v \in V \\
\text{set } BF[v][k] = \min \{ BF[v][k-1], \min \{ BF[u][k-1] + c(u, v) \} \}
\end{align*}
\]

\[
\text{return } BF[*][n-1]
\]

Example

\[
\begin{array}{ccccccc}
& 0 & 1 & 2 & 3 & 4 & 5 \\
0 & & & & & & \\
1 & & & & & & \\
2 & & & & & & \\
3 & & & & & & \\
4 & & & & & & \\
5 & & & & & & \\
6 & & & & & & \\
\end{array}
\]

RT: inner for loop runs in time \( \sum_v \deg(v) \in \mathcal{O}(m) \), so whole alg is \( \mathcal{O}(mn) \).

Example

\[
\begin{array}{ccccccc}
& 0 & 1 & 2 & 3 & 4 & 5 \\
0 & & & & & & \\
1 & & & & & & \\
2 & & & & & & \\
3 & & & & & & \\
4 & & & & & & \\
5 & & & & & & \\
6 & & & & & & \\
\end{array}
\]

Example

\[
\begin{array}{ccccccc}
& 0 & 1 & 2 & 3 & 4 & 5 \\
0 & & & & & & \\
1 & & & & & & \\
2 & & & & & & \\
3 & & & & & & \\
4 & & & & & & \\
5 & & & & & & \\
6 & & & & & & \\
\end{array}
\]

Example
BF() -- iteratively compute BF(v,k) for all v and all k=0,...,n

• create n x n array BF
• initialize BF[s][0] = 0, for all other v, BF[v][0] =
• for k=1,2,...,n
• for each vertex v in V
• set BF[v][k] = min { BF[v][k-1], min{ BF[u][k-1] + c(u,v) } }
• return BF[*][n-1]
BF( ) -- iteratively compute BF(v,k) for all v and all k=0,...,n

• create n x n array BF
• initialize BF[s][0] = 0, for all other v, BF[v][0] = \infty
• for k=1,2,...,n
• for each vertex v in V
• set BF[v][k] = \min \{ BF[v][k-1], \min{ BF[u][k-1] + c(u,v) } \}
• return BF[*][n-1]

Note: neg cycle exists if and only if BF[v][n] = BF[v][n-1] = \infty for some v.