Greedy algorithms

Shortest paths in weighted graphs

Tyler Moore

CSE 3353, SMU, Dallas, TX

Lecture 13

Some slides created by or adapted from Dr. Kevin Wayne. For more information see http://www.cs.princeton.edu/~wayne/kleinberg-tardos. Some code reused from Python Algorithms by Magnus Lie Hetland.

Greedy algorithms: greed is good?

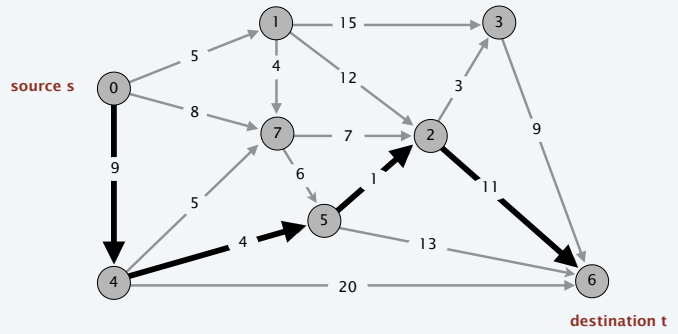
Greed, for lack of a better word, is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures, the essence of the evolutionary spirit. Greed, in all of its forms; greed for life, for money, for love, knowledge, has marked the upward surge of mankind and greed, you mark my words, will not only save Teldar Paper, but that other malfunctioning corporation called the U.S.A.

Greedy algorithms

- A **greedy algorithm** builds a solution incrementally, making the best local decision to construct a global solution
- The clever thing about greedy algorithms is that they find ways to consider only a portion of the solution space at each step
- We've already seen two greedy algorithms
  1. Gale-Shapley algorithm to solve stable-matching problem: men propose to their best choice, women accept/decline without considering other prospective offers
  2. Earliest-finish algorithm to solve interval-scheduling problem: choose the job that finishes first and doesn't conflict with jobs already accepted

Shortest-paths problem

Problem. Given a digraph $G = (V, E)$, edge lengths $\ell_e \geq 0$, source $s \in V$, and destination $t \in V$, find the shortest directed path from s to t.



length of path = 9 + 4 + 1 + 11 = 25

3

Car navigation

4

Shortest path applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in LaTeX.
- Urban traffic planning.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Optimal truck routing through given traffic congestion pattern.

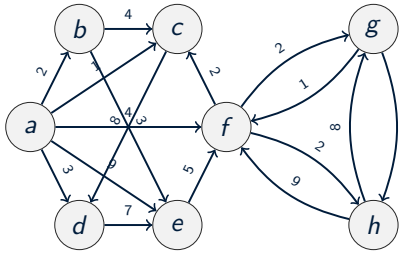
Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

5

The many cases of finding shortest paths

- We've already seen how to calculate the shortest path in an unweighted graph (BFS traversal)
- We'll now study how to compute the shortest path in different circumstances for *weighted* graphs
  1. Single-source shortest path on a weighted DAG
  2. Single-source shortest path on a weighted graph with nonnegative weights (Dijkstra's algorithm)
  3. Single-source shortest path on a weighted graph including negative weights (Bellman-Ford algorithm)

Weighted Graph Data Structures



Nested Adjacency Dictionaries w/ Edge Weights

```
N = {
'a':{'b':2,'c':1,'d':3,'e':9,'f':4},
'b':{'c':4,'e':3},
'c':{'d':8},
'd':{'e':7},
'e':{'f':5},
'f':{'c':2,'g':2,'h':2},
'g':{'f':1,'h':6},
'h':{'f':9,'g':8}
}
>>> 'b' in N['a'] # Neighborhood membership
True
>>> len(N['f'])   # Degree
3
>>> N['a']['b']
# Edge weight for (a, b)
2
```

Shortest paths in DAGs

- Recursive approach to finding the shortest path from a to z
  1. Assume we already know the distance $d(v)$ to z for each of a's neighbors $v \in G[a]$
  2. Select the neighbor v that minimizes $d(v) + W(a, v)$

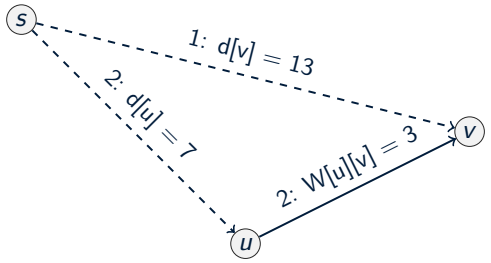
Recursive solution to finding shortest path in DAGs

```
def rec_dag_sp(W, s, t):    #Shortest path from s to t
    @memo                   #Memoize f
    def d(u):               #Distance from u to t
        if u == t: return 0#We're there!
        # Return the best of every first step
        return min(W[u][v]+d(v) for v in W[u])
    return d(s)             #Apply f to actual start node
```

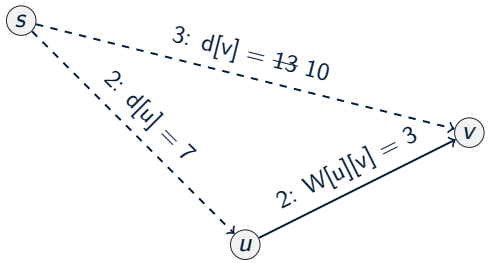

Shortest paths in DAGs: Iterative approach

- The iterative solution is a bit more complicated
  1. We must start with a topological sort
  2. Keep track of an upper bound on the distance from a to each node, initialized to ∞
  3. Go through each vertex and *relax* the distance estimate by inspecting the path from the vertex to its neighbor
- In general, *relaxing* an edge (u, v) consists of testing whether we can shorten the path to v found so far by going through u; if we can, we update $d[v]$ with the new value
- Running time: $\Theta(m + n)$

Relaxing edges



Relaxing edges

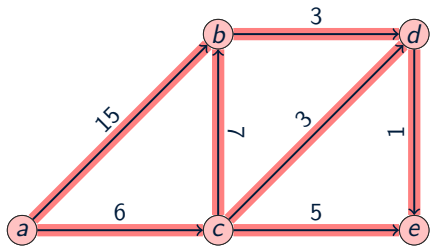



```
inf = float('inf')
def relax(W, u, v, D, P):
    d = D.get(u,inf) + W[u][v]# Possible shortcut estimate
    if d < D.get(v,inf):      # Is it really a shortcut?
        D[v], P[v] = d, u     # Update estimate and parent
        return True           # There was a change!
```


Iterative solution to finding shortest path in DAGs

```
def dag_sp(W, s, t):        #Shortest path from s to t
    d = {u:float('inf') for u in W}             # Distance estimates
    d[s] = 0                #Start node: Zero distance
    for u in topsort(W):    #In top-sorted order...
        if u == t: break    #Have we arrived?
        for v in W[u]:      #For each out-edge ...
            d[v] = min(d[v], d[u] + W[u][v])    # Relax the edge
    return d[t]             #Distance to t (from s)
```


Shortest-paths on weighted DAG example



Topological sort: a, c, b, d, e

	d[Node]: upper bd. dist. from *a*					
Node	init.	1 (u=a)	2 (u=c)	3 (u=b)	4 (u=d)	5 (u=e)
a	0	0	0	0	0	0
b	∞	15	13	13	13	13
c	∞	6	6	6	6	6
d	∞	∞	9	9	9	9
e	∞	∞	11	11	10	10

Shortest-paths on weighted DAG: exercise

But what if there are cycles?

- With a DAG, we can select the order in which to visit nodes based on the topological sort
- With cycles we can't easily determine the best order
- If there are no negative edges, we can traverse from the starting vertex, visiting nodes in order of their estimated distance from the starting vertex
- In Dijkstra's algorithm, we use a priority queue based on minimum estimated distance from the source to select which vertices to visit
- Running time: $\Theta((m+n)\lg n)$
- Dijkstra's algorithm combines approaches seen in other algorithms
  1. Node discovery: bit like breadth-first traversal
  2. Node visitation: selected using priority queue
  3. Shortest path calculation: uses relaxation as in algorithm for shortest paths in DAGs

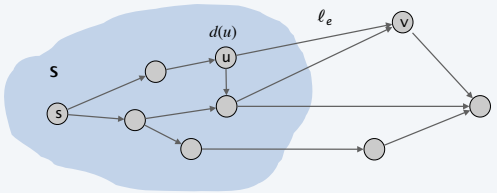
Dijkstra's algorithm

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined the shortest path distance $d(u)$ from s to u.

- Initialize $S = \{ s \}$, $d(s) = 0$.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v) : u \in S} d(u) + \ell_e,$$

shortest path to some node u in explored part, followed by a single edge (u, v)



6

Dijkstra's algorithm

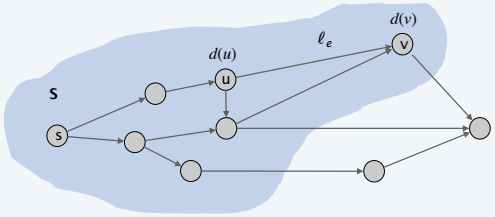
Greedy approach. Maintain a set of explored nodes S for which algorithm has determined the shortest path distance $d(u)$ from s to u.

- Initialize $S = \{ s \}$, $d(s) = 0$.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v) : u \in S} d(u) + \ell_e,$$

add v to S, and set $d(v) = \pi(v)$.

shortest path to some node u in explored part, followed by a single edge (u, v)



7

Dijkstra's algorithm: proof of correctness

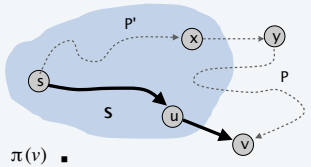
Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest $s \to u$ path.

Pf. [by induction on $|S|$]

Base case: $|S| = 1$ is easy since $S = \{ s \}$ and $d(s) = 0$.

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S, and let (u, v) be the final edge.
- The shortest $s \to u$ path plus (u, v) is an $s \to v$ path of length $\pi(v)$.
- Consider any $s \to v$ path P. We show that it is no shorter than $\pi(v)$.
- Let (x, y) be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it reaches y.



$$\ell(P) \geq \ell(P') + \ell(x,y) \geq d(x) + \ell(x,y) \geq \pi(y) \geq \pi(v) \quad \blacksquare$$

nonnegative lengths; inductive hypothesis; definition of π(y); Dijkstra chose v instead of y

8

Dijkstra's algorithm: efficient implementation

Critical optimization 1. For each unexplored node v, explicitly maintain $\pi(v)$ instead of computing directly from formula:

$$\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e \,.$$

- For each $v \notin S$, $\pi(v)$ can only decrease (because S only increases).
- More specifically, suppose u is added to S and there is an edge (u, v) leaving u. Then, it suffices to update:

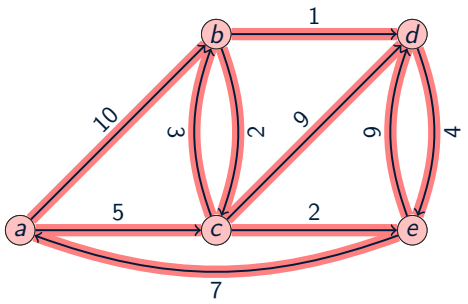
$$\pi(v) = \min\{\, \pi(v),\ d(u) + \ell(u, v) \,\}$$

Critical optimization 2. Use a priority queue to choose the unexplored node that minimizes $\pi(v)$.

Dijkstra's algorithm

```
from heapq import heappush, heappop
def dijkstra(G, s):
    D, P, Q, S = {s:0}, {}, [(0,s)], set()      # Est., tree, queue, vis
    while Q:                                    # Still unprocessed nodes?
        _, u = heappop(Q)                       # Node with lowest estimate
        if u in S: continue                     # Already visited? Skip it
        S.add(u)                                # We've visited it now
        for v in G[u]:                          # Go through all its neighbors
            relax(G, u, v, D, P)  # Relax the out-edge
            heappush(Q, (D[v], v))# Add to queue, w/est. as pri
    return D, P                                 # Final D and P returned
```


Dijkstra's algorithm example



Node	d[Node]: upper bd. dist. from *a*					
	init.	1 (u=a)	2 (u=c)	3 (u=e)	4 (u=b)	5 (u=d)
a	0	0	0	0	0	0
b	∞	10	8	8	8	8
c	∞	5	5	5	5	5
d	∞	∞	14	13	9	9
e	∞	∞	7	7	7	7

Dijkstra's algorithm: exercise