6.006 Final Review Session II

Graph Algorithms
Dynamic Programming

Dynamic Programming

Summary:

* DP ≈ "careful brute force"
* DP ≈ guessing + recursion + memoization
* DP ≈ dividing into reasonable # subproblems whose solutions relate — acyclicly — usually via guessing parts of solution

* time = # subproblems · $\underbrace{\text{time/subproblem}}_{\text{treating recursive calls as } O(1)}$

  = # subproblems · # guess choices · time/guess

  — essentially an amortization
  — count each subproblem only once; after first time, costs O(1) via memoization

* DP often ≈ shortest paths in some DAG

Dynamic Programming

* 5 easy steps to dynamic programming:

1. define subproblems — count # subprobs.
2. guess (part of solution) — count # choices
3. relate subprob. solutions — compute time/subprob.
4. recurse + memoize — time = time/subprob. · # subprobs.

   OR build DP table bottom-up
   - check subprobs. acyclic/topological order
5. solve original problem: = a subproblem

   OR by combining subprob. solutions (⇒ extra time)

Dynamic Programming

Good practice problems:

Fall 2012 final: Problem 12 -- Optimizing Santa
Fall 2014 final: Problem 5 -- Hanging posters
> Fall 2014 final: Problem 6 -- Palindromes
> Spring 2014 final: Problem 7 -- 3D blocks
Spring 2014 final: Problem 10
Spring 2015 final: Problem 9-b

Link to 6.006 from previous semesters:
http://courses.csail.mit.edu/6.006/

Dynamic Programming (Fall 2014 final)

Problem 6. Palindromes [20 points]

Given a string $x = x_1, \ldots, x_n$, design an efficient algorithm to find the minimum number of characters that need to be inserted to make it a palindrome (recall that a palindrome is a string such as "racecar" the reads the same backwards). Analyze the running time of your algorithm and justify its correctness.

For example, when $x =$ "ab3bd", we need to insert two characters (one "d" and one "a"), to get either the of the palindromes "dab3bad" or "adb3bda".

1. Define subproblems. How many are there?
2. What are the possible guesses we have to consider?
3. Define recurrence.
4. Compute time per subproblem.
5. Obtain running time.
6. Solve original problem

Dynamic Programming (Spring 2014 final: Problem 7 -- 3D blocks)

Problem 7. [15 points] Suppose you are given a set of n rectangular three-dimensional blocks, where block B_i has length l_i, width w_i, and height h_i, all real numbers. You are supposed to determine the maximum height of a tower of blocks that is as tall as possible, using any subset of the blocks.

There are two constraints:

1. You are not allowed to rotate the blocks: the length always refers to the east-west direction, the width is always north-south, and the height is always up-down.
2. You are only allowed to stack block B_i on top of block B_j if $l_i \leq l_j$ and $w_i \leq w_j$, that is, the two dimensions of the base of block B_i are no greater than those of block B_j.

1. Assume that l[i] <= l[j] when i < j (this assumption can be weakened if we sort all blocks according to l and breaking ties according to w)
2. Define subproblems. How many are there?
3. What are the possible guesses we have to consider?
4. Define recurrence.
5. Compute time per subproblem. Obtain running time.
6. Solve original problem

All pairs shortest paths

Floyd-Warshall algorithm - $O(|V|^3)$

Subproblems: **P(i,j,k):**

Find shortest path from i to j using only nodes <=k as intermediate.

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each vertex v
   dist[v][v] ← 0
for each edge (u,v)
   dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
for k from 1 to |V|
   for i from 1 to |V|
      for j from 1 to |V|
         if dist[i][j] > dist[i][k] + dist[k][j]
             dist[i][j] ← dist[i][k] + dist[k][j]
         end if
```

All pairs shortest paths

Johnson's algorithm - $O(|V||E|+|V|^2*\log|V|)$

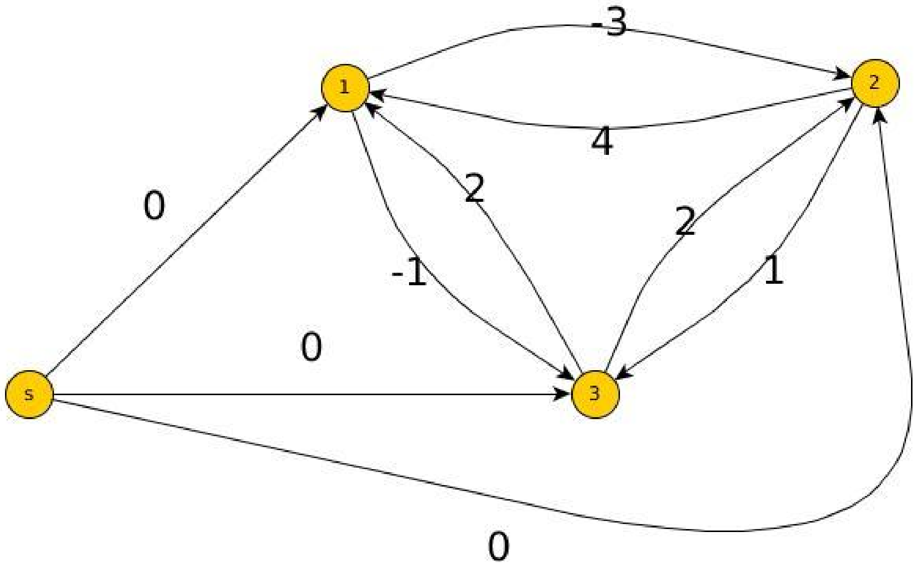
Idea:

- Reduce to a graph that has non-negative weights: $w_h(u,v)= w(u,v)+h(u)-h(v)$ (Find appropriate function h using Bellman-Ford) - $\mathbf{O(|V||E|)}$
- Run Dijkstra's algorithm $|V|$ times to compute $\delta_h(u,v)$ - $\mathbf{O(|V||E|+|V|^2*\log|V|)}$
- Recover the original shortest path lengths $\delta(u,v)=\delta_h(u,v)+h(v)-h(u)$ - $\mathbf{O(|V|^2)}$

Example of solving difference constraints

Adapted from Problem 9 in Spring 14 Final: Solve the following difference constraints:

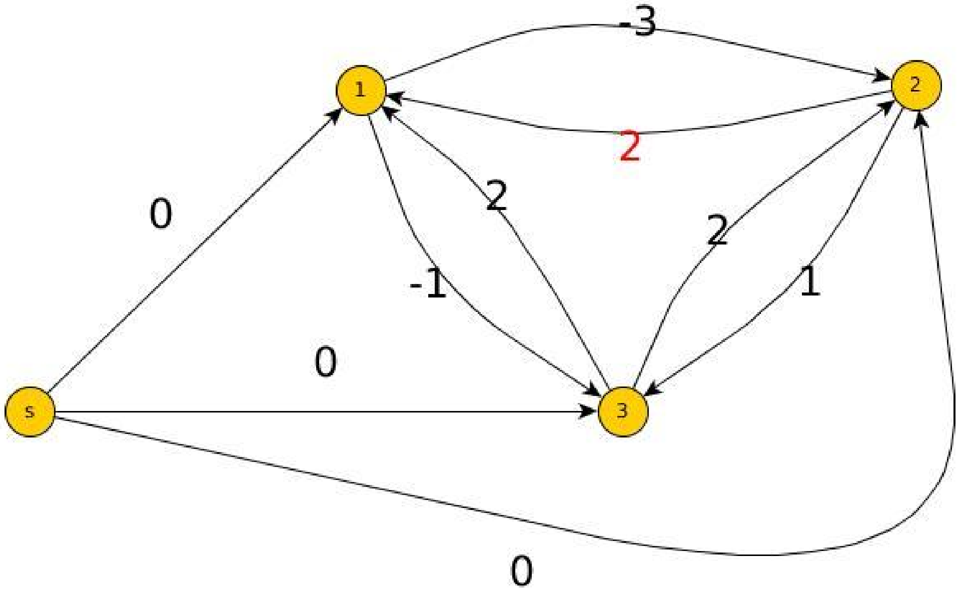
$$x_1 \leq x_2 + 4$$
$$x_2 \leq x_1 - 3$$
$$x_1 \leq x_3 + 2$$
$$x_3 \leq x_1 - 1$$
$$x_2 \leq x_3 + 2$$
$$x_3 \leq x_2 + 1$$



Example of solving difference constraints

Adapted from **Problem 9 in Spring 14 Final**: Solve the following difference constraints:

$$x_1 \leq x_2 + 4$$
$$x_2 \leq x_1 - 3$$
$$x_1 \leq x_3 + 2$$
$$x_3 \leq x_1 - 1$$
$$x_2 \leq x_3 + 2$$
$$x_3 \leq x_2 + 1$$



Shortest paths problems

Adapted from **Spring 11 final Problem 5** : Compute all pairs shortest paths in a graph G=(V,E,w) where the weight of paths that have **at least 11 edges** is **doubled**.

- Create a graph G'=(V',E',w') that has 12 copies of the graph G
- For every edge {u,v} in E, add $\{u_i,v_{i+1}\}$ (0<= i <=10). Also add $\{u_{11},v_{11}\}$
- Run Floyd-Warshall on G'
- For each {u,v} in E pick the smallest among $\{\delta(u_0,v_i) \mid 0 \le i \le 10\} \cup \{2 * \delta(u_0,v_{11})\}$

Alternatively: Modify Floyd-Warshall to keep track of shortest paths between i and j, having exactly k edges (k<=10) or >10 edges.

Graph Representations

Adjacency Matrices:

Adjacency matrix A is a 2D matrix, $[1, \ldots, n, 1, \ldots, n]$.

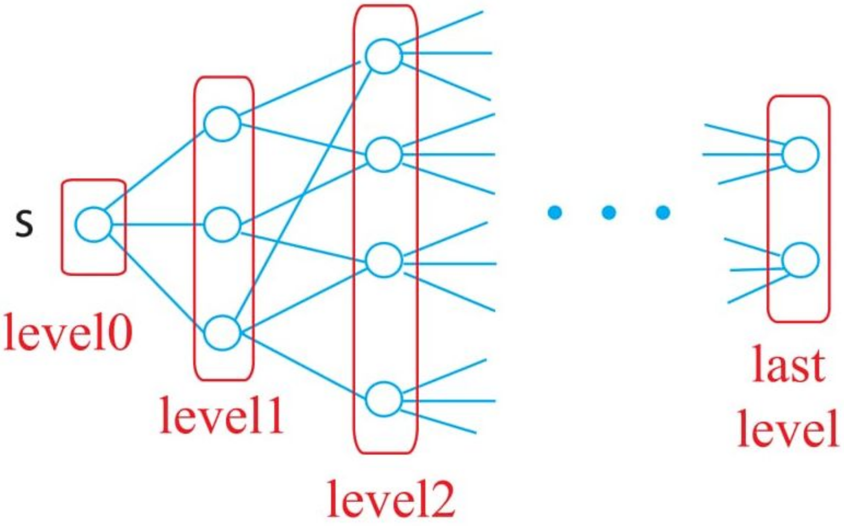
$$A[i,j] = \begin{cases} 1 \text{ if } (i,j) \in E \\ 0 \text{ if } (i,j) \notin E \end{cases}$$

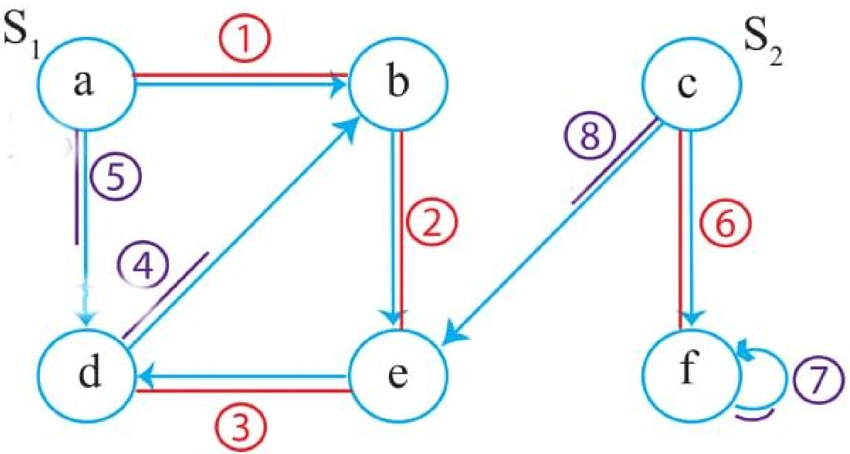
Adjacency Lists

Array Adj of $|V|$ linked lists

- for each vertex $u \in V$, $Adj[u]$ stores u's neighbors, i.e., $\{v \in V \mid (u,v) \in E\}$. (u,v) are just outgoing edges if directed.

BFS / DFS





BFS / DFS

```
BFS (V,Adj,s):
    level = { s: 0 }
    parent = {s : None }
    i = 1
    frontier = [s]
    while frontier:
        next = [ ]
        for u in frontier:
            for v in Adj [u]:
                if v not in level:
                    level[v] = i
                    parent[v] = u
                    next.append(v)
        frontier = next
        i + =1
```

```
DFS (V, Adj)
    parent = { }
    for s in V:
        if s not in parent:
            parent [s] = None
            DFS-visit (Adj, s)
```

```
DFS-visit (Adj, s):
    for v in Adj [s]:
        if v not in parent:
            parent [v] = s
            DFS-visit (Adj, v)
```

BFS / DFS Applications

BFS

- Shortest paths in unweighted graphs

DFS

- Edge Classification
- Topological Sort
- Cycle Detection

Runtimes: O(V+E) [using adjacency lists]

Single Source Shortest Paths

Path: A sequence of vertices (v_0, v_1, … , v_k), such that,

for all 0 <= i < k, (v_i, v_{i+1}) is in E.

Simple Path: A sequence of vertices (v_0, v_1, … , v_k), such that,

For all 0 <= i < k, (v_i, v_{i+1}) is in E and

for all 0 <= i, j <= k, if i =/= j then v_i =/= v_j.

Single Source Shortest Paths

Problem: Given a graph G and a vertex s, find the shortest paths from s to each other vertex in G.

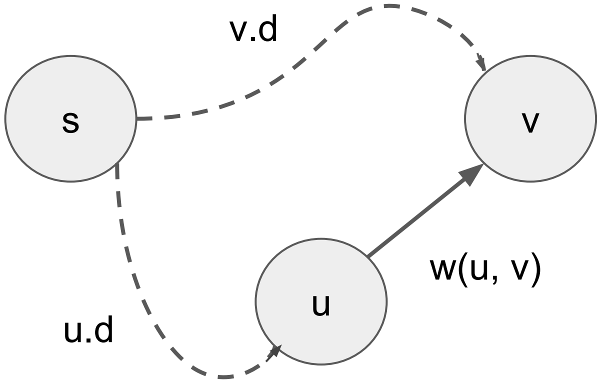
First Try

Brute Force Algorithm: Try every possible path. Running time = $2^{\Omega(n)}$

Relaxation

Would it be shorter if I use (u, v)?

```
relax(u, v):
    if  u.d + w(u, v) < v.d
        v.d = u.d + w(u, v)
        v.parent = u
```



Shortest Paths in DAGs

SP_DAG(G):

```
initialization(V)
topo_sort(G)
for each u in V in topological order:
    for each v in Adj[u]:
        relax(u, v)
```

Running Time: O(V+E)

Bellman - Ford Algorithm

Fix an ordering of edges <e_1, e_2, … , $e_{|E|}$>.

Bellman-Ford(G):

```
initialization(V)
for i = 1 to |V| - 1:
    for j = 1 to |E|:
        relax(e_j)
```

Running Time: O(VE)

Can handle negative weights.

Can find negative weight cycles.

Dijkstra’s Algorithm

- Works out one node at a time
- Maintains a priority queue of best path weights to nodes
- Runtime is O(V*(EXTRACT-MIN)+E*(DECREASE+KEY))
- With a Fibonacci heap, this is O(V log V + E)

Dijkstra's: example problem

A large metropolitan area's police departments would like to compute the shortest path from any police station to any intersection in the area, where each portion of a street between intersections has a specified time-to-travel. How can one find these shortest paths to each intersection in $O(S+N \log N)$, for S street segments and N intersections?