EE360C: Algorithms

Dynamic Programming

Pedro Santacruz

Department of Electrical and Computer Engineering
University of Texas at Austin

Fall 2014

A Review of Algorithmic Techniques

Greedy

Build up a solution incrementally, myopically optimizing some local criterion

Divide and Conquer

Break up a problem into two (or more) subproblems, solve each subproblem independently, and combine the solutions to the subproblems to form a solution to the original problem

Dynamic Programming

Break up a problem into a series of overlapping subproblems and build up solutions to larger and larger subproblems.

A History

Bellman pioneered the systematic study of dynamic programming in the 1950s.

Etymology

- Dynamic programming = planning over time
- Secretary of Defense at the time was hostile to mathematical research
- Bellman sought an impressive name to avoid confrontation
  - "something not even a Congressman could object to"

Dynamic Programming

Dynamic Programming is an algorithm design technique that, like divide and conquer, relies on solutions to sub problems to solve a problem

- in dynamic programming, however, the subproblems are not independent
- a dynamic programming algorithm solves each of these subproblems just once, saving time in comparison to a traditional divide and conquer approach

Dynamic programming is commonly used for **optimization problems** in which many possible solutions exist; the algorithm helps us find one of the possibilities

- every solution to the problem has a value
- the algorithm helps us find a solution with the optimal value

Dynamic Programming Structure

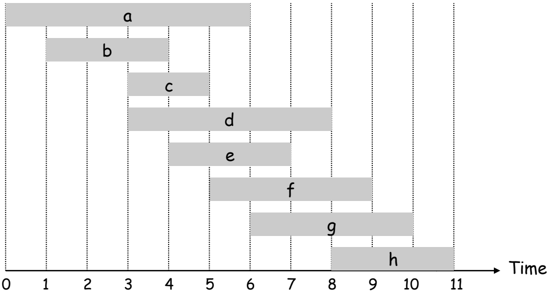
Dynamic programming algorithms typically involve the following steps:

- characterize the structure of an optimal solution
- recursively define the value of an optimal solution
- compute the value of an optimal solution in a bottom-up fashion
- construct an optimal solution from the computed information

Weighted Interval Scheduling

The weighted interval scheduling problem:

- Job j starts at s_j, finishes at f_j, and has weight or value v_j
- Two jobs are *compatible* if they do not overlap
- Goal: find maximum weight subset of mutually compatible jobs

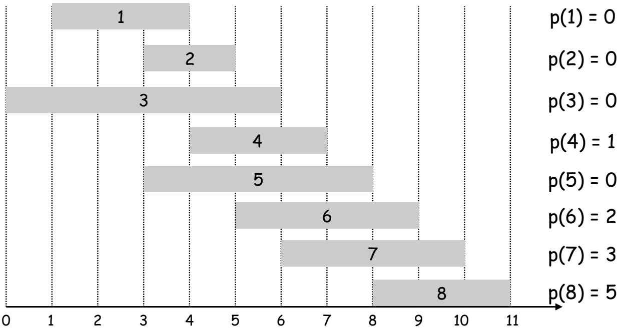


Unweighted Interval Scheduling

Recall that greedy algorithm that works perfectly (optimally) if all the weights are 1. The greedy algorithm can fail spectacularly if arbitrary weights are allowed.

Back to Weighted Interval Scheduling

Label (and sort) jobs by finishing time ($f_1 \leq f_2 \leq \ldots \leq f_n$). Define $p(j)$ to be the largest index $i < j$ such that job i is compatible with job j.



Weighted Interval Scheduling: Binary Choice

OPT(j) is the value of the optimal solution to the problem consisting of job request $1, 2, \ldots, j$.

- **Case 1:** OPT selects job j.
  - then OPT can't include any of the jobs in the range $p(j) + 1, p(j) + 2, \ldots j - 1$
  - OPT must include the optimal solution to the problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$
- **Case 2:** OPT does not select job j.
  - OPT must include the optimal solution to the problem consisting of remaining compatible jobs $1, 2, \ldots j - 1$

$$\mathrm{OPT}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + \mathrm{OPT}(p(j)), \mathrm{OPT}(j-1) & \text{otherwise} \end{cases}$$

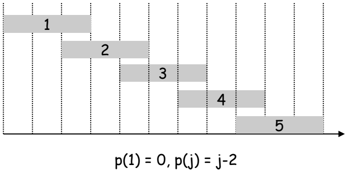
Weighted Interval Scheduling: Brute Force

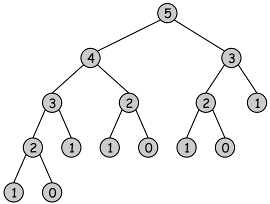
```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

Weighted Interval Scheduling: Brute Force is Bad

The recursive algorithm fails spectacularly because of redundant sub-problems (it's an exponential algorithm)

- the number of recursive calls for a family of "layered" instances grows like the Fibonacci sequence





WEIGHTED INTERVAL SCHEDULING: MEMOIZATION

Memoization: store the results of each sub-problem in a cache; look them up as needed.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty    ← global array
M[j] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(w_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

Weighted Interval Scheduling: Memoization is Good

What is the running time of the memoized algorithm? $O(n \lg n)$

- Sort by finish time: $O(n \lg n)$
- Computing $p(\cdot)$: $O(n)$ after sorting by start time
- `M-Compute-Opt(j)`: each invocation takes $O(1)$ time and either:
  - returns an existing value `M[j]`
  - fills in one new entry `M[j]` and makes two recursive calls
- Define a progress measure Φ as the number of nonempty entries in `M[]`.
  - initially $\Phi = 0$; throughout $\Phi \leq n$
  - a call to `M-Compute-Opt(j)` increases Φ by at most 1; Φ cannot be increased more than n times; there are at most $2n$ recursive calls
- so the overall running time of `M-Compute-Opt(n)` is $O(n)$

Weighted Interval Scheduling: Finding a Solution

This algorithm has only computed the optimal value (i.e., the weight of the optimal schedule). What if we want the solution itself?

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (v_j + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

Bottom-Up Weighted Interval Scheduling

To do this via dynamic programming, basically, we unwind the recursion...

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(v_j + M[p(j)], M[j-1])
}
```

When Dynamic Programming Applies

For an optimization problem to be a good candidate for dynamic programming, it should exhibit:

- optimal substructure
- overlapping problems

Optimal Substructure

A problem exhibits the optimal substructure property if the optimal solution contains within it optimal solutions to subproblems

- be careful, though, optimal substructure may also make a problem a good candidate for a greedy solution

We use the optimal substructure by then solving the subproblems bottom-up, combining the subproblem solutions as we move up.

Overlapping Subproblems

If the recursive solution is going to solve the same subproblem multiple times, then we're wasting computation

- this is in contrast to problems that generate unique subproblems at every split, for which divide and conquer is a good fit
- one alternative for overlapping subproblems is *memoization*
  - it's like dynamic programming, but it fills the table in using a top-down approach that's more like traditional recursion
  - you create a table to use in a recursive algorithm, and the recursive calls fill in the table as they return
  - results can then be just computed once and looked up thereafter
- more traditional dynamic programming is "bottom up"

Segmented Least Squares

Least Squares

- Foundational problem in statistic and numerical analysis
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, find a line $y = ax + b$ that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

Solution

The solution from calculus tells us that the minimum error is achieved when:

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}$$

and

$$b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

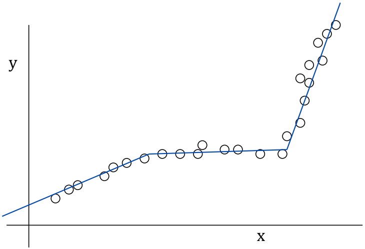
Segmented Least Squares

Segmented Least Squares

- Points lie roughly on a sequence of several line segments
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \cdots < x_n$, find a sequence of lines that minimizes $f(x)$

Question

What is a reasonable choice for $f(x)$ to balance accuracy (goodness of fit) and parsimony (number of lines)



Segmented Least Squares

The Tradeoff

We have to tradeoff the number of lines for the summed error

Find a sequence of lines that minimizes

- The sum of the sums of the squared errors E in each segment
- The number of lines L

This results in a tradeoff function $E + cL$ for some constant c

Dynamic Programming: Multiway Choice

Notation

- $OPT(j)$ = minimum cost for points $p_1, p_2, \ldots, p_j$
- $e(i, j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$

To Compute $OPT(j)$

- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some i
- Cost = $e(i, j) + c + OPT(i-1)$

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j}(e(i,j) + c + OPT(i-1)) & \textit{otherwise} \end{cases}$$

Segmented Least Squares: Algorithm


```
INPUT: n, p_1,…,p_N , c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error e_ij for
         the segment p_i,…, p_j

   for j = 1 to n
      M[j] = min_{1 ≤ i ≤ j} (e_ij + c + M[i-1])

   return M[n]
}
```

Running Time

- Bottleneck: computing $e(i, j)$ for $O(n^2)$ pairs, $O(n)$ per pair using the previous formula

The Knapsack Problem

Knapsack Problem

- Given n objects and a "knapsack"
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$
- Knapsack has a capacity of W kilograms
- Goal: fill the knapsack so as to maximize the total value

Greedy?

Remember the greedy algorithm we explored was not optimal...

Knapsack: A False Start

Definition: $OPT(i)$ is the max profit for the subset of items $1, \ldots, i$

- Case 1: OPT does not select item i
  - OPT selects the best of $1, 2, \ldots, i-1$
- Case 2: OPT selects item i
  - accepting item i does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion

We need more subproblems!

Dynamic Programming: Adding a Variable

Definition: $OPT(i, w)$ is the max profit of items $1, \ldots, i$ with weight limit w

- Case 1: OPT does not select item i
  - OPT selects best of $\{1, 2, \ldots, i-1\}$ using weight limit w
- Case 2: OPT selects item i
  - new weight limit $w - w_i$
  - OPT selects best of $\{1, 2, \ldots, i-1\}$ using this new weight limit

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max(OPT(i-1, w), v_i + OPT(i-1, w - w_i)) & \text{otherwise} \end{cases}$$

Knapsack: Bottom Up

```
Input: n, w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (w_i > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}

return M[n, W]
```

Knapsack: Running Time

Running Time

- $\theta(nW)$
- But I swear I've heard that the knapsack algorithm is NP-Complete!
- What does that mean?
- How is that possible? Didn't I just write down a polynomial time algorithm?
- Nope. It's "pseudo-polynomial"

An Approximation Algorithm

There is a polynomial time algorithm that produces a feasible solution whose value is within 0.01% of the optimal.

The Coin Changing Problem

The Problem

You are given k denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. We want to find an algorithm that makes change for an amount of money n using as few coins as possible.

First Question

The coin changing problem exhibits optimal substructure. Consider any optimal solution to make change for n cents using our k denominations of coins. Consider breaking that solution into two different pieces along any coin boundary. Suppose that one half of the solution amounts to b cents and the other half to $n - b$ cents. Then the solution to the each half must be an optimal way to make change for b cents (or $n - b$ cents) using the same k denominations of coins. **Prove this.**

The Coin Changing Problem

The Problem

You are given k denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. We want to find an algorithm that makes change for an amount of money n using as few coins as possible.

Second Question

Let $C[p]$ be the minimum number of coins of the k denominations that sum to p cents. **Recursively define the value of the optimal solution.** To get you started, there must exist some "first coin" d_i, where $d_i \leq p$.

The Coin Changing Problem

The Problem

You are given k denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. We want to find an algorithm that makes change for an amount of money n using as few coins as possible.

Third Question

Provide an algorithm (e.g., pseudocode) to compute the value of the optimal solution bottom up. Provide another algorithm to construct the optimal solution from the computed information.

The Coin Changing Problem

The Problem

You are given k denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. We want to find an algorithm that makes change for an amount of money n using as few coins as possible.

The Algorithm to Reconstruct the Solution

MakeChange(S, d, n)

```
while n > 0
        Print S[n]
        n ← n − d[S[n]]
```

THE COIN CHANGING PROBLEM

The Problem

You are given k denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. We want to find an algorithm that makes change for an amount of money n using as few coins as possible.

Fourth Question

What is the running time of your algorithm?

Running Time

CHANGE is $\Theta(nk)$ (which is pseudo-polynomial in k, the input size and dependent on n, just like knapsack was dependent on W). MAKECHANGE is $O(n)$ since n is reduced by at least 1 in every iteration of the **while** loop. The total space requirement is $\Theta(n)$

RNA Secondary Structure

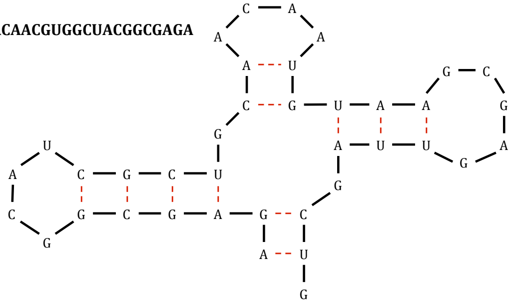
RNA

String $B = b_1 b_2 \ldots b_n$ over the alphabet $\{$ A, C, G, U$\}$

Secondary Structure

RNA is single stranded, so it tends to loop back and form *base pairs* with itself. This structure is often essential to the function of the molecule. Legal base pairs are (A, U) or (C, G).

Ex: **GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA**



RNA Secondary Structure

Secondary Structure

A set of pairs $S = \{(b_i, b_j)\}$ that satisfy:

- **Watson and Crick:** S is a matching and each pair in S is a Watson-Crick component (matching A to U or C to G)
- **No Sharp Turns:** The ends of each pair are separated by at least 4 intervening bases. That is, if $(b_i, b_j) \in S$, then $i < j - 4$.
- **Non-Crossing:** If (b_i, b_j) and (b_k, b_l) are two pairs in S, the we cannot have $i < k < j < l$

Free Energy

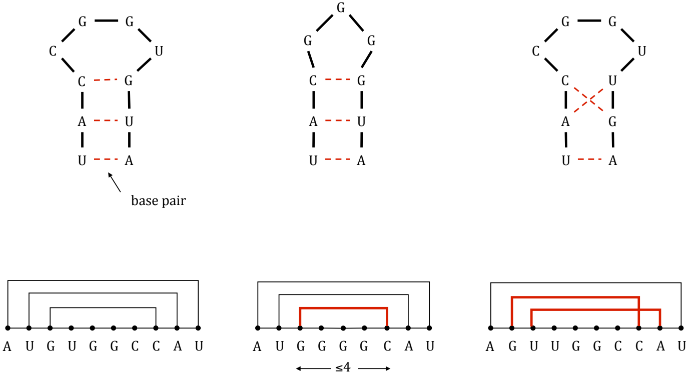
The usual hypothesis is that an RNA molecule will form the secondary structure that has the optimum total *free energy*, which we approximate by the number of base pairs.

Goal

Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

RNA Secondary Structure: Examples

Examples.



ok

sharp turn

crossing

Dynamic Programming Over Intervals

Notation

$OPT(i, j)$ is the maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$

- Case 1: if $i \geq j - 4$
  - $OPT(i, j) = 0$ by the no sharp turns condition
- Case 2: Base b_j is not involved in a pair.
  - $OPT(i, j) = OPT(i, j - 1)$
- Case 3: Base b_j pairs with b_t for some $i \leq t < j - 4$
  - The non crossing constraint decouples the resulting subproblems
  - $OPT(i.j) = 1 + \max_t(OPT(i, t - 1) + OPT(t + 1, j - 1))$ such that $i \leq t < j - 4$ and (b_j, b_t) is a Watson and Crick pair

Bottom Up Dynamic Programming Over Intervals

Question

What order to solve the subproblems?

Answer

Do the shortest intervals first.

```
RNA(b_1,…,b_n) {
   for k = 5, 6, …, n-1
      for i = 1, 2, …, n-k
         j = i + k
         Compute M[i, j]

   return M[1, n]
}
```

using recurrence

Dynamic Programming Summary

Recipe

- Characterize the structure of the problem
- Recursively define the value of an optimal solution
- Compute the value of the optimal solution
- Construct the optimal solution from computed information

Dynamic Programming Techniques

- Binary choice: weighted interval scheduling.
- Multi-way choice segmented least squares.
- Adding a new variable: knapsack
- Dynamic programming over intervals: RNA secondary structure

Top-down vs. Bottom-up?

Different people have different intuitions

Longest Common Subsequence

This is a common problem in biological applications: find the longest common sequence of ACTG in two different pieces of DNA

- useful in finding similar proteins, etc.

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, a sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a **subsequence** of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, $x_{i_j} = z_j$.

(For example $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$.)

Our goal, is, given two sequences X and Y, find the longest common subsequence of the two sequences.

Characterizing the Longest Common Subsequence

Theorem: Optimal Substructure of an LCS

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. if $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

Proof (Part I)

If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to get a common subsequence with length $k+1$, which would contradict the premise. So $z_k = x_m = y_n$. And therefore Z_{k-1}, the prefix of Z with z_k removed is a longest common subsequence of X_{m-1} and Y_{n-1}.

Proof (Parts II and III)

If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a subsequence of X and Y, which contradicts the premise that Z was an LCS.

A Recursive Solution to LCS

If $x_m = y_n$, there is one subproblem to solve: find the LCS of X_{m-1} and of Y_{n-1}. If $x_m \neq y_n$, then there are two subproblems to solve: find the LCS of X_{m-1} and Y and the LCS of X and Y_{n-1}.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Computing the Length of an LCS

We use dynamic programming to compute solutions to the $\Theta(mn)$ subproblems in a bottom-up fashion

- we create a table $c[0\,..\,m, 0\,..\,n]$ and compute its entries in row major order (filling in the first row, left to right, then the second row, etc.)
- we keep a second table $b[0\,..\,m, 0\,..\,n]$ whose entries point to the table entry that corresponds to the best subproblem for the problem $b[i, j]$ (to help in reconstructing the optimal solution)

Computing the Length of an LCS (cont.)

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
      do c[i, 0] ← 0
for j ← 1 to n
      do c[0, j] ← 0
for i ← 1 to m
      do for j ← 1 to n
            do if x_i = y_j
                  then c[i, j] ← c[i − 1, j − 1] + 1
                        b[i, j] ← ↖
                  else  if c[i − 1, j] ≥ c[i, j − 1]
                        then c[i, j] ← c[i − 1], j]
                              b[i, j] ← ↑
                        else  c[i, j] ← c[i, j − 1]
                              b[i, j] ← ←
return c and b
```

AN EXAMPLE

i		j → 0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2	B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3	C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4	B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5	D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6	A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7	B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

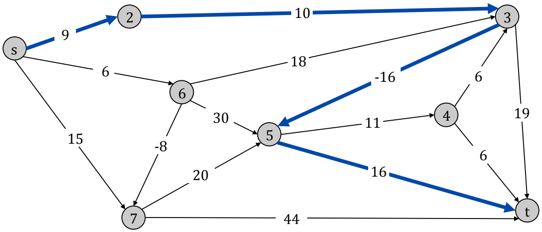
Shortest Paths

Shortest Path Problem

Given a direct graph $G = (V, E)$ with the edge weights c_{vw} (which can include negative edge weights), find the shortest path from node s to node t.

Example

Nodes represent agents in a financial setting and c_{vw} is the cost of a transaction in which we buy from agent v and sell immediately to w



Negative Cost Cycles

Observation

If some path from s to t contains a negative cost cycle, there does not exist a shortest s-t path; otherwise there exists a path that is a simple path.

Shortest Paths and Dynamic Programming

Definition: $OPT(i, v)$ is the length of the shortest v-t path P using at most i edges.

- Case 1: P uses at most $i - 1$ edges
  - $OPT(i, v) = OPT(i - 1, v)$
- Case 2: P uses exactly i edges
  - if (v, w) is the first edge, the OPT uses (v, w), and then selects the best w-t path using at most $i - 1$ edges

$$OPT(i, v) = \begin{cases} 0 & \text{if } i = 0 \\ \min(OPT(i-1, v), \min_{(v,w)\in E}(OPT(i-1, w) + c_{vw})) & \text{otherwise} \end{cases}$$

IMPLEMENTATION

See the book for the Bellman-Ford algorithm implementation and a discussion of the running time.

Distance Vector Routing

Communication Network

- nodes = routers
- edges = direct communication link
- cost of edge = delay on link (which is naturally non-negative, but we use Bellman-Ford algorithm anyway!)

Dijkstra's Algorithm

Requires global information of network

Bellman-Ford

Uses only local knowledge of neighboring nodes

Synchronization

We don't expect the routers to run in lock-step; Bellman-Ford can tolerate asynchronous routing updates and can be shown to still converge on the correct shortest path.

Distance Vector Protocol

Distance Vector Protocol

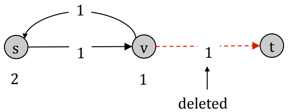
- Each router maintains a vector of shortest path lengths to every other node (distances) and the first hop on each path (directions)
- Algorithm: each router performs n separate computations, one for each potential destination node
- “Routing by rumor”

Examples

RIP, Xerox XNS RIP, Novell’s IPX RIP, Cisco’s IGRP, DEC’s DNA Phase IV, AppleTalk’s RTMP

Caveat

Edge costs may change during the algorithm; inconsistent router state can lead to the counting to infinity problem



Path Vector Protocols

Link State Routing

- Each router stores the entire path (not just the distance and next hop)
- Based on Dijkstra's algorithm
- Avoids the counting to infinity problem
- Requires significantly more storage

Examples

Border Gateway Protocol (BGP), Open Shortest Path First (OSPF)

Questions

?