CSC 373: Algorithm Design and Analysis
Lecture 8

Allan Borodin

January 23, 2013

Lecture 8: Announcements and Outline

Announcements

- No lecture (or tutorial) this Friday.
- Lecture and tutorials as usual on Monday.
- First Problem Set is now complete and due Friday, February 1.
- First Term Test in tutorials on Monday, February 4.

Today's outline

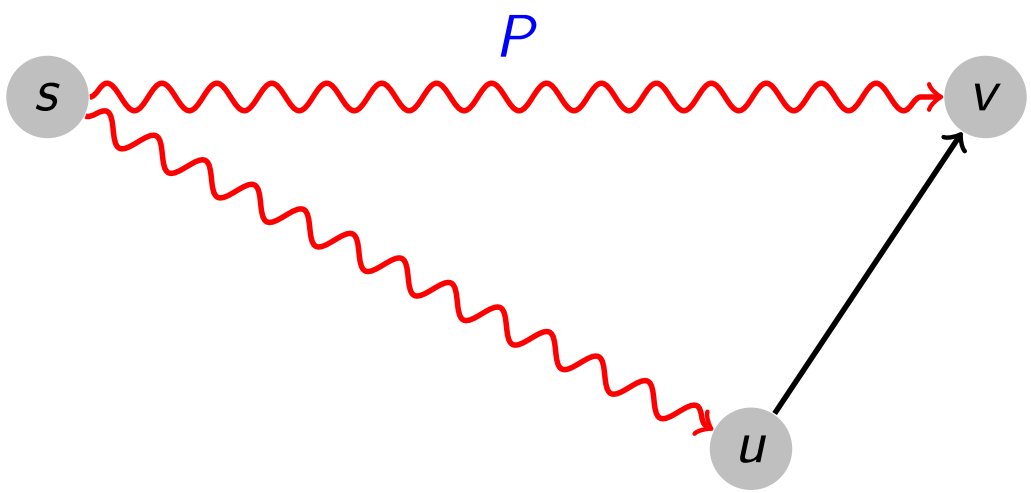
- Reflections on the DP algorithm for the least cost paths problem
- The all pairs least cost problem and an alternative DP
- The chain matrix product problem
- The edit distance problem

Review: A DP with a somewhat different style

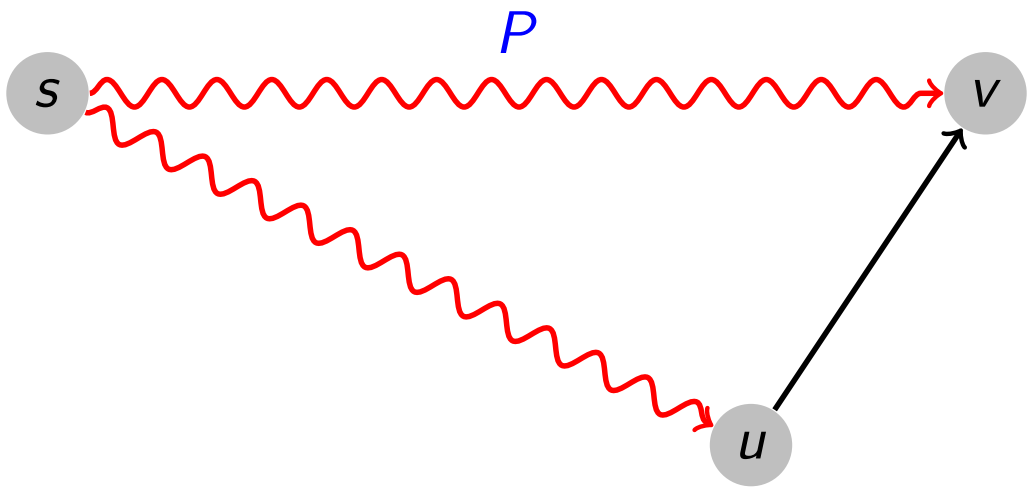
- We considered the single source least cost paths problem which is efficiently solved by Dijkstra's greedy algorithm for graphs in which all edge costs are non-negative.
- The least cost paths problem is still well defined as long as there are no negative cycles; that is, the least cost path is a simple path.
  - There can also be non simple least cost paths when there are zero cost cycles but there will always be a simple path having the least cost and hence we can restrict attention to such simple paths.
- Basically the same algorithm can be used to compute the least cost paths from all nodes to a single terminal node t (again assuming no negative cycles).
- Given the Bellman Ford algorithm for shortest paths, it is not difficult to detect when a directed graph has a negative cost cycle (and find such a cycle if it exists). See, for example, Section 6.10 of KT.

Single source least cost paths for graphs with no negative cycles

- Following the DP paradigm, we consider the nature of an optimal solution and how it is composed of optimal solutions to "subproblems".
- Consider an optimal simple path P from source s to some node v.
  - This path could be just an edge.
  - But if the path P has length greater than 1, then there is some node u which immediately proceeds v in P. If P is an optimal path to v, then the path leading to u must also be an optimal path.



Single source least cost paths for graphs with no negative cycles



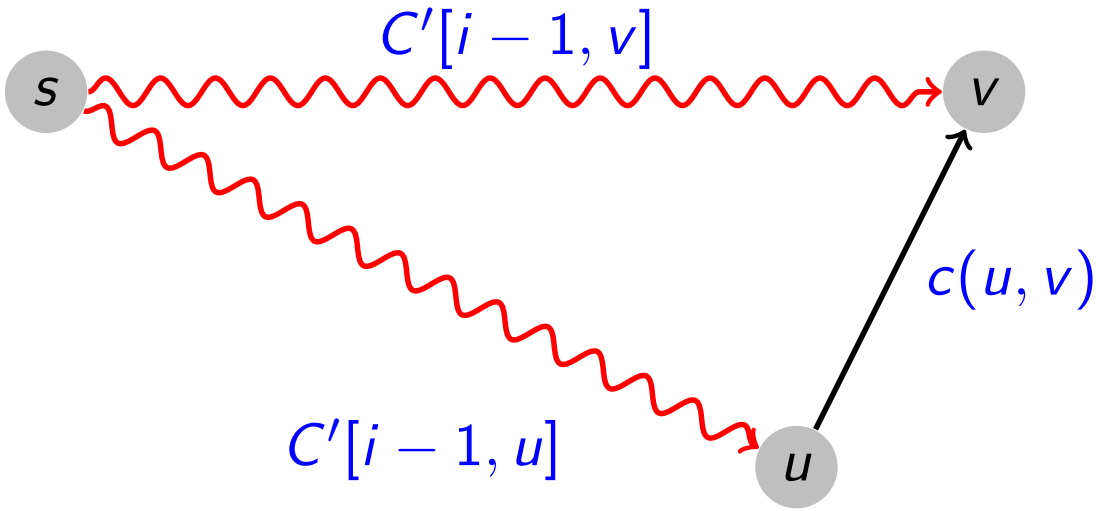
- This leads to the following semantic array:

$C[i, v]$ = the minimum cost of a simple path with path length at most i from source s to v. (If there is no such path then this cost is ∞.)

- The desired answer is then the single dimensional array derived by setting $i = n - 1$. (Any simple path has path length at most $n - 1$.)

How to construct the computational array?

- We can construct $C'[i, v]$ from $C'[i-1, \ldots]$ as follows:



- Let $C'[i, v]$ be the minimum value among
  - $C'[i-1, v]$
  - $C'[i-1, u] + c(u, v)$ for all $(u, v) \in E$.

Corresponding computational array

- The computational array is defined as:

$$C'[i,v] = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ \min\{A, B\} & \text{otherwise} \end{cases}$$

$$A = C'[i-1, v]$$

$$B = \min\left\{ C'[i-1, u] + c(u,v) : (u,v) \in E \right\}$$

- Why is this slightly different from before?
  - Namely, showing the equivalence between the semantic and computationally defined arrays is not an induction on the indices of the input items in the solution.
  - But it is based on some other parameter (i.e. the path length) of the solution.
- Time complexity: n^2 entries $\times$ $O(n)$ per entry $= O(n^3)$ in total.

Computing maximum cost path using the same DP?

- To define this problem properly we want to say "maximum cost simple path" since cycles will add to the cost of a path.
- (For least cost we did not have to specify that the path is simple once we assumed no negative cycles.)
- Suppose we just replace min by max in the least cost DP. Namely,

$M[i, v]$ = the maximum cost of a simple path with path length at most i from source s to v. (If there is no such path then this cost is $-\infty$.)

- The corresponding computational array would be

$$M'[i,v] = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ -\infty & \text{if } i = 0 \text{ and } v \neq s \\ \max\{A, B\} & \text{otherwise} \end{cases}$$

$$A = M'[i-1, v]$$

$$B = \max\Big\{ M'[i-1, u] + c(u,v) : (u,v) \in E \Big\}$$

- Is this correct?

What goes wrong?

- The problem calls for a maximum simple path but the recursion

$$B = \max\Big\{ M'[i-1, u] + c(u, v) : (u, v) \in E \Big\}$$

  does not guarantee that the path through u will be a simple path as v might occur in the path to u. Algorithm would work for a DAG.

- In fact, determining the maximum cost of a simple path is NP-hard.
  - A special case of this problem is the Hamiltonian path problem: does a graph $G = (V, E)$ have a simple path of length $|V| - 1$?
  - The Hamiltonian path problem is a variant of the "notorious" (NP-hard) traveling salesman problem (TSP).

- See Section 6.6 of DPV for how to use DP to reduce the complexity from the naive $O(n!)$ to $O(n^2 2^n)$.

Stirling's approximation

$$n! \approx \sqrt{2\pi n}\ \left(\frac{n}{e}\right)^n$$

The all pairs least cost problem

- We now wish to compute the least cost path for all pairs (u, v) in an edge weighted directed graph (with no negative cycles).
- We can repeat the single source DP for each possible source node: complexity $O(n^4)$
- We can reduce the complexity to $O(n^3 logn)$ using the DP based on the semantic array

$E[j, u, v]$ = cost of shortest path of path length at most 2^j from u to v.

- What is corresponding computational array?

Another DP for all pairs (DPV section 6.6)

- Let's assume (without loss of generality) that $V = \{1, 2, \ldots, n\}$.
- We now define the semantic array

$G[k, u, v]$ = the least cost of a (simple) path π from u to v such that the internal nodes in the path π are in the subset $\{1, 2, \ldots, k\}$.

- The computational array is

$$G'[0, u, v] = \begin{cases} 0 & \text{if } u = v \\ c(u, v) & \text{if } (u, v) \text{ is an edge} \\ \infty & \text{otherwise.} \end{cases}$$

$$G'[k+1, u, v] = \min\{A, B\}$$

where $A = G'[k, u, v]$ and $B = G'[k, u, k+1] + G'[k, k+1, v]$.

- Like the recursion for the previous array $E'[j, u, v]$, the recursion here uses two recursive calls for each entry.
- Time complexity: n^3 entries $\times$ $O(1)$ per entry $= O(n^3)$ in total.

A similar DP (using 2 recursive calls)

The chain matrix product problem (DPV section 6.5

- We are given n matrices (say over some field) $M_1, \ldots, M_n$ with M_i having dimension $d_{i-1} \times d_i$.
- **Goal:** compute the matrix product

$$M_1 \cdot M_2 \cdot \ldots \cdot M_n$$

using a given subroutine for computing a single matrix product $A \cdot B$.

- We recall that matrix multiplication is associative; that is,

$$(A \cdot B) \cdot C = A \cdot (B \cdot C).$$

- But the number of operations for computing $A \cdot B \cdot C$ generally depends on the order in which the pairwise multiplications are carried out.

The matrix chain product problem continued

- Let us assume that we are using classical matrix multiplication and say that the scalar complexity for a $(p \times q)$ times $(q \times r)$ matrix multilication is pqr.
- For example say the dimensions of A, B and C are (respectively) 5×10, 10×100 and 100×50.
- Then using $(A \cdot B) \cdot C$ costs $5000 + 25000 = 30000$ scalar operations whereas $A \cdot (B \cdot C)$ costs $50000 + 2500 = 52500$ scalar ops.
- **Note:** For this problem the input is these dimensions and not the actual matrix entries.

Parse tree for the product chain

- The matrix product problem then is to determine the parse tree that describes the order of pairwise products.
- At the leaves of this parse tree are the individual matrices and each internal node represents a pairwise matrix multiplication.
- Once we think of this parse tree, the DP is reasonably suggestive:

The root of the optimal tree is the last pairwise multiplication and the subtrees are subproblems that must must be computed optimally.

The DP array for the matrix chain product problem

- The semantic array:

$C[i,j]$ = the cost of an optimal parse of $M_i \cdot \ldots \cdot M_j$ for $1 \leq i \leq j \leq n$.

- The recursive computationally array:

$$C'[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\{C'[i,k] + C'[k+1,j] + d_{i-1}d_k d_j : i \leq k < j\} & \text{if } i < j \end{cases}$$

- This same style DP algorithm (called **DP over intervals**) is also used in the RNA folding problem (in Section 6.5 of KT) as well as in computing optimal binary search trees (see section 15.5 in CLRS).

- Essentially in all these cases we are computing an optimal parse tree.

The sequence alignment (edit distance) problem

The edit distance problem

- Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ over some finite alphabet S.
- **Goal:** find the best way to "match" these two strings.

- Variants of this problem occur often in bio-informatics as well as in spell checking.
- Sometimes this is cast as a maximization problem.
- We will view it as a minimization problem by defining different distance measures and matching symbols so as to minimize this distance.

A simple distance measure

- Suppose we can delete symbols and match symbols.
- We can have a cost $d(a)$ to delete a symbol a in S, and a cost $m(a, b)$ to match symbol a with symbol b (where we would normally assume $m(a, a) = 0$).
- As in any DP we consider an optimal solution and let's consider whether or not we will match the rightmost symbols of X and Y or delete a symbol.

The DP arrays

- The semantic array:

$$E[i,j] = \text{the cost of an optimal match of } x_1 \dots x_i \text{ and } y_1 \dots y_j.$$

- The computational array:

$$E'[i,j] = \begin{cases} 0 & \text{if } i = j = 0 \\ d(y_j) + E'[i, j-1] & \text{if } i = 0 \text{ and } j > 0 \\ d(x_i) + E'[i-1, j] & \text{if } i > 0 \text{ and } j = 0 \\ \min\{A, B, C\} & \text{otherwise} \end{cases}$$

where $A = m(x_i, y_j) + E'[i-1, j-1]$, $B = d(x_i) + E'[i-1, j]$, and $C = d(y_j) + E'[i, j-1]$.

- As a simple variation of edit distance we consider the maximization problem where each "match" of "compatible" a and b has profit 1 (resp. $v(a,b)$) and all deletions and mismatches have 0 profit.
- This is a special case of unweighted (resp. weighted) bipartite graph matching where edges cannot cross.

DP concluding remarks

- In DP algorithms one usually has to first generalize the problem (as we did more or less to some extent for all problems considered). Sometimes this generalization is not at all obvious.

- What is the difference between divide and conquer and DP?

  - In divide and conquer the recursion tree never encounters a subproblem more than once.
  - In DP, we need memoization (or an iterative implementation) as a given subproblem can be encountered many times leading to exponential time complexity if done without memoization.
  - See also the comment on page 169 of DPV as to why in some cases memoization pays off since we do not necessaily have to compute every possible subproblem. (Recall also the comment by Dai Tri Man Le in Lecture 6.)