CSC373: Algorithm Design, Analysis and Complexity
Winter/Spring 2020

Allan Borodin and Sara Rahmati

Week of January 20-24, 2020

Week 3 : Annoucements and agenda

Assignment 1 was posted on January 12 and is due Thursday, February 6 at 4:59 PM. There will be a small bonus question to be posted tomorrow around 10AM.
The quiz (at end of tutorial) covers divide and conquer and greedy algorithms.

Some questions regarding Assignment 1 on Piazza. We do **not** monitor everyday. We try to answer questions posed on piazza but questions can also be answered in the lecture or the tutorials or by fellow students.

Some smalll changes/clarifications with respect to Assignment 1.

- Rewording of question 4, part b. In your example for a 3-colorable graph G such that the greedy algorithm does not optimally colour G, you can order the vertices in any way which then fixes the way the bgraph is coloured.
- In question 3, we changed "$1 \leq i < j \leq n$" to "$1 \leq i \leq j \leq n$".
- In question 2, polynomial inputs and outputs are given in terms of their coefficients. **Questions?**

This weeks agenda

1. Finishing up greedy algorithms (for now)
   - Dijkstra's algorithm: greedy or dynamic programming?
   - Summarizing the greedy paradigm
   - Some quick concluding remarks on greedy algorithms
2. Begin *dynamic programming (DP)*. Getting to DP quickly so assignment questions will be (more) understandable.
   - The dynamic programming (DP) paradigm
   - The weighted interval selection problem
   - The knapsack problem
   - The edit distance and longest common subsequence problems.
   - Shortest and least cost problems in directed graphs
   - The matrix chain problem

Dijkstra's algorithm

We assume that you may have already seen Dijskstra's algorithm so we will not spend too much time on this algiorithm. But we do assume that you understand this algorithm and the problem it is solving.

We are given an edge weighted directed graph $G = (V, E, d)$ with a starting node s and a non negative cost or distance function $d : E \rightarrow \mathbb{R}^{\geq 0}$.

Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes.
    For each $u \in S$, we store a distance $d(u)$.
Initially $S = \{s\}$ and $d(s) = 0$.
While $S \neq V$
    Select a node $v \notin S$ with at least one edge from S for which
        $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible.
    Add v to S and define $d(v) = d'(v)$.
EndWhile

Figure: Dijkstra's algorithm from DPV text

Comments on Dijkstra's algorithm

Dijkstra's algorithm is a seminal algorithm and the basis for navigation systems.

As long as there are no negative cycles, the meaning of a "least cost path" is well defined even if there are negative cost edges.

The assumption that edges have non-negative costs is necessary for Dijkstra's algorithm. FInd an example for which Dijkstra will not result in shortest paths.

Is Dijkstra's algorithm a greedy algorithm?
This rather "academic question" brings us back to the question as to what is a greedy algorithm. What is the objective of the algorithm?

Comments on Dijkstra's algorithm

Dijkstra's algorithm is a seminal algorithm and the basis for navigation systems.

As long as there are no negative cycles, the meaning of a "least cost path" is well defined even if there are negative cost edges.

The assumption that edges have non-negative costs is necessary for Dijkstra's algorithm. FInd an example for which Dijkstra will not result in shortest paths.

Is Dijkstra's algorithm a greedy algorithm?
This rather "academic question" brings us back to the question as to what is a greedy algorithm. What is the objective of the algorithm?
When the objective if to find a least cost path from s to all nodes $v \in V$, then Dijkstra's algorithm is a greedy algorithm (within the informal paradigm which we first described and within a more precise definition which we will mention next). It is an *adaptive order* greedy algorithm in the sense that the next edge that will be considered and greedily added to the set S depends on what edges are already in S.

Summarizing the greedy paradigm

- Informally, (most) greedy algorithms consider one input item at a time and make an irrevocable ("greedy") decision about that item before seeing more items.
- To make this precise for any given problem we have to say
  1. how input items are represented ; for example, what other information could we put into the representation of an interval?
  2. how an algorithm determines the order in which input items are considered.
- Key to formalizing our intuitive idea of a greedy algorithm: we need to define the class of orderings of the input items that will be allowed. We cannot allow any ordering of the input set or else one can usually take (say) exponential time to compute an "optimal ordering".
- If we try to make this precise by stating that the ordering must be done in say time $O(n \log n)$ (or even $poly(n)$) then we are in the situation of trying to prove that something cannot be done in a given time bound. The Turing tarpit

One way to formalize how to order

- For a given problem, assume that input items belong to some set $\mathcal{J}$.
- For any execution of the algorithm, the input is a finite subset $\mathcal{I} \subset \mathcal{J}$.
- Let $f : \mathcal{J} \to \Re$ be any function; that is, we do not place any restriction on the complexity or even the computability of the function.
- Then for any actual input set $\mathcal{I} = \{I_1, \ldots, I_n\}$, the function f induces a total order on the input set (where we can break ties using the index of the input items as given).
- In a fixed order the function f is set initially. In an adaptive order, there can be a different function f_i in each iteration i with f_i depending on the items considered in iterations $j < i$.

Jeff Erickson's comment on greedy algorithms

5.4 Warning: Greed is Stupid

If we're very very very very lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a *greedy* algorithm. The general greedy strategy is look for the best first step, take it, and then continue. While this approach seems very natural, it almost never works; optimization problems that can be solved correctly by a greedy algorithm are *very* rare. Nevertheless, for many problems that should be solved by dynamic programming, many students' first intuition is to apply a greedy strategy.

For example, a greedy algorithm for the edit distance problem might look for the longest common substring of the two strings, match up those substrings (since those substitutions don't cost anything), and then recursively look for the edit distances between the left halves and right halves of the strings. If there is no common substring—that is, if the two strings have no characters in common—the edit distance is clearly the length of the larger string. If this sounds like a stupid hack to you, pat yourself on the back. It isn't even *close* to the correct solution.

Everyone should tattoo the following sentence on the back of their hands, right under all the rules about logarithms and big-Oh notation:

Greedy algorithms never work!
Use dynamic programming instead!

What, never?
No, never!
What, *never*?
Well. . . hardly ever.[6]

DP and Linear programming: two sledgehammers of the *algorithmic craft*

The following is the first paragraph of chpater 6 of DPV.

> In the preceding chapters we have seen some elegant design principles—such as divide-and-conquer, graph exploration, and greedy choice—that yield definitive algorithms for a variety of important computational tasks. The drawback of these tools is that they can only be used on very specific types of problems. We now turn to the two *sledgehammers* of the algorithms craft, *dynamic programming* and *linear programming*, techniques of very broad applicability that can be invoked when more specialized methods fail. Predictably, this generality often comes with a cost in efficiency.

My view of greedy algorithms

- First, the previous comments are in the context of emphasizing DP and LP algorithms for optimization problems and were a deliberate overstating of the point.
- My view of greedy algorithms is that while they may not often be optimal or as good as more sophisticated algorithms, there are many cases where they work well either in terms of provable approximations or "in practice".
- Moreover, in some cases we imediately need something that works and knowing some basic approaches to a problem becomes a starting point. If nothing esle, greedy algorithms can be a benchmark for comparison against more sophisticated algorithms.
- DP algorithms, once they are formulated, often seem quite apparant. But coming up with a correct DP formulation is often not so obvious. In contrast, coming up with a correct (albeit possibly one having poor performance) greedy algorithm is usually easy to do.
- Finally, there are applications (e.g. auctions) where conceptual simplicity is a virtue (and maybe even a necessity) in itself.

Dynamic programming - what's in a name

The followings Bellman quote in Erickson's notes on dynamic programming explain the origins of the term. Richard Bellman introduced this algorithmic technique during the Cold War.

> *The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word* 'research'*. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term* 'research' *in his presence. You can imagine how he felt, then, about the term* 'mathematical'*. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?*
>
> — Richard Bellman, on the origin of his term 'dynamic programming' (1984)

Figure: Richard Bellman quote on the naming of dynamic programming

What is dynamic programming?

Here is the wikipedia "definition":

In computer science, mathematics, management science, economics and bioinformatics, dynamic programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space. (Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup.) The technique of storing solutions to subproblems instead of recomputing them is called "memoization".

What is dynamic programming?

Here is the wikipedia "definition":

In computer science, mathematics, management science, economics and bioinformatics, dynamic programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space. (Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup.) The technique of storing solutions to subproblems instead of recomputing them is called "memoization".

What is the difference between divide and conquer and dynamic programming?

Dynamic programming

- Dynamic programming (DP) began as and remains a very general algorithmic approach for solving optimization problems.
- Its usage now goes beyond that but still optimization is the main use.
- To begin to understand dynamic programming, we will start by developing a DP algorithm for the *weighted* interval selection. Our second example of the use of DP will be for the knapsack problem.

The weighted interval selection problem (WISP)

Goal: Find a non-intersecting set of intervals so as to maximize the sum of interval weights (i.e., values) in the chosen set.

- Can we use a greedy algorithm? (Recall Erickson's warning.)

Why not use greedy for WISP?

- All the possible ways of ordering the input items that we can think of will not only fail to be optimal but can produce arbitrarily bad solutions for some instances.
- Some possible orderings: by non increasing weight, by non increasing weight/interval length.
- Moreover, for a general greedy formalization it can be proven that no greedy algorithm can provide a good solution (in the worst case).
- There are some extensions to a greedy approach which do allow constant approximations (i.e. by allowing revocable acceptances) and even optimality (i.e. by a local ratio/primal dual algorithm that uses a reverse delete phase). This is another reason why one should not be so negative about greeedy algorithms.

The DP approach

- Let's consider an optimal solution and once again assume that the intervals have been sorted by non-decreasing finishing time.
- Then in an optimal solution OPT, either the last interval I_n was selected or it was not.
  - If not, then we must be using an optimal solution for the first $n-1$ intervals.
  - If I_n is in OPT then no interval in OPT can end after time s_n.
  - Furthermore (and this is the essential aspect of DP), the intervals ending by s_n must be chosen optimally.

Note

- Once again we will define the problem so that an interval can start when another one ends.
- We can easily modify things if we do not want to allow an interval to start at precisely the time another ends.

The value/profit of an optimal solution

- The previous observation leads us to compute the entries (for $i = 1, \ldots, n$) in the following "semantic array"

$V[i] =$ max profit obtainable by a set of intervals which are a subset of the first i intervals $\{I_1, \ldots, I_i\}$

- The optimal value then is $V[n]$.
- We can also define $V[0] = 0$.
- To compute the entries of this array, it is helpful to define

$pred(i) =$ the largest index j such that $f_j \leq s_i$

(If we don't allow a job to start where another ends we would then have $f_j < s_i$.)

Recursively computing the $V[i]$

- $V'[0] = 0$
- $V'[i] = \max\{A, B\}$ for $i > 0$, where

$$A = V'[i-1] \text{ and } B = V'[pred(i)] + w_i.$$

- Here B (resp. A) corresponds to the case that the ith interval is used (resp. not used) in the optimum solution for the first i intervals.
- We can arbitrarily assume that we take the solution corresponding to case A when $A = B$.

Claim

$$V[i] = V'[i] \quad \text{for all } i = 1, 2, \ldots, n.$$

Iterative vs recursive implementation

- We can clearly compute the entries of $V'[i]$ iteratively for $i = 0, 1, \ldots, n$. Time bound is $O(n \log n)$ for sorting and for computing pred[i] values.

- What if we use a recursive program directly following the definition of V'?
  - Suppose for all $i = 1, 2, \ldots, n-1$, interval I_i overlaps I_{i+1} and no other I_j for $j > i+1$.
  - This leads to the complexity recurrence

$$T[n] = T[n-1] + T[n-2]$$

    whose solution (recall Fibonacci sequences) is exponential in n.

- Memoization avoids this problem. In some sense, memoization is one of the defining characteristics (say verses divide and conquer) of DP algorithms.

Why two arrays V and V'?

- The semantic array is defined to say what we are trying to compute.
- The recursively defined computational array is essentially a high level code for how to compute the entries of the semantic array.
- The creative aspect of DP is coming up with an appropriate semantic array that has to provide us with enough information to obtain the desired result as well as being easy to compute.
- And although it often seems tedious, we need a proof that $V = V'$.
- In fact, we should have been doing the same testing of equality for divide and conquer algorithms.

Computing an optimal solution and not just the optimal value

- So far we only computed the value of an optimal solution (for WISP) but we can easily adapt the DP solution to compute the solution as well.

- While there are somewhat more efficient ways to do this, the conceptually simplest thing to do is to maintain an array, say S, where $S[i]$ contains the partial solution corresponding to the value $V[i]$.

- It should be clear from the recursion defining V' how to do this.

$$S'[i] = \begin{cases} \varnothing & \text{if } i = 0 \\ S'[i-1] & \text{if } V'[i] = V'[i-1] \\ S'[pred(i)] \cup \{i\} & \text{otherwise.} \end{cases}$$

Characteristics of dynamic programming

DP algorithms exploit the *optimal substructure* property of the problem being solved. That is, an optimal solution contains within it, optimal solutions to subproblems.

As you will have seen, the wikipedia page (and other explanations of dynamic programming) emphasize memoization as a defining characteristic. And it is this aspect of reusing subproblems that distingusihes DP from divide and conquer. In fact, as one now thinks about our divide and conquer examples, the recursive subproblems are all disjoint and hence memoization is not needed.

Bellman argues against trying to formalize the meaning of dynamic programming stating that although solme solutions "are forced upon us ....experience alone, combined with often laborious trial and error, will yield suitable formulations of involved processes".

So lets consider more examples. But first a comment about efficient implementation.

A comment on efficient implementations of DP

Dai Tri Man Le (a fiormer student) makes the following observation on implementing a DP algorithm:

> *One problem with using DP in practice is the memory issue. When the program uses too much memory, it's no longer fast. That's why sometimes one uses recursion instead of sequentially implemented DP, although the worst cases can be terrible. Recently I was able to improve some worst case of an algorithm used in industry from 24 hours to 5 mins using memoization. I didn't even need to memorize everything, just the most recently computed results, and it's already sufficient to see the improvement. It's also interesting that when I didn't restrict the size of the look up (hash) table as much so that it can memoize more things, the algorithm became slower. So a lot of tuning was needed for the code to perform well.*

As stated in the first week, this course is not concerned with implementation issues, as important as they are.

The Knapsack problem

- In the knapsack problem we are given a set of n items $I_1, \ldots, I_n$ and a size bound B where each item $I_j = (s_j, v_j)$ with s_j being the size of the item and v_j the value.
- A feasible set is now a subset of items S such that the sum of the sizes of items in S is at most the bound B.
- Goal: Find a feasible set S that maximizes the sum of the values of items in S.

- Often (e.g., KT and CLRS texts) one uses w_j for the "weight" (meaning size in my terminology) of the item rather than s_j but I am avoiding that due to our earlier use of w_j to denote the weight or profit of an interval in the WISP.
- In general we can allow real valued parameters but in some algorithms need to restrict attention to integral parameters. But by scaling inputs this is not a significant restriction.
- This is known to be an NP hard problem but as we shall see it is only "weakly NP hard". It remains an NP hard problem even when $v_j = s_j$ for all j.

A first attempt

- Here is a plausible DP approach. Lets assume all sizes are integral. Suppose we consider an optimal solution and consider the last item placed in the knapsack.
- Then after placing that item in the knapsack (say having sisze s), we have reduced the available space to $B - s$.
- So it seems that we need to have a semantic array

$V[b] =$ max profit/value obtainable within size bound b for $0 \leq b \leq B$.

- The recursive array

$$V'[b] = \begin{cases} 0 & \text{for } b \leq 0 \\ \max_j \{ V'[b - s(j)] + v(j) : j = 1, 2, \ldots, n \} & \text{for } b > 0 \end{cases}$$

- Does this work and if not why not?

A correct approach

- The previous approach did not work because it allows using an item more than once.
- Instead we can use

$V[i, b]$ = the maximum profit possible using only the first i items and not exceeding the bound b.

- The corresponding computational array is :

$$V'[i, b] = \begin{cases} 0 & \text{if } i = 0 \text{ or } b = 0 \\ \max\{C, D\} & \text{if } s_i \leq b \end{cases}$$

where

$$C = V'[i-1, b] \text{ and } D = V'[i-1, b-s_i] + v_i.$$

- This algorithm has running time $O(nB)$ and is pseudo polynomial time. Why is it not polynomial time?

A second DP algorithm for the knapsack problem

- In the first algorithm, if the sizes (or the bound B) are small (i.e. $B = poly(n)$) then the algorithm runs in polynomial time.
- What if the values $\{v_i\}$ are integral and small?
- Consider the following semantic array

$$W[i, v] = \begin{cases} \text{minimum size required to obtain at least profit } v \text{ using} \\ \qquad\qquad \text{a subset of the items } \{I_1, \ldots, I_i\} \text{ if possible} \\ \infty \text{ otherwise} \end{cases}$$

- The desired optimum value is $\max\{v : W[n, v] \text{ is at most } B\}$.

Corresponding computational array

- The corresponding computational array is :

$$W'[i,v] = \begin{cases} \infty & \text{if } i = 0 \text{ and } v > 0 \\ 0 & \text{if } i \leq 0 \text{ and } v \leq 0 \\ \min\{C, D\} & \text{otherwise.} \end{cases}$$

  where

$$C = W'[i-1, v] \text{ and } D = W'[i-1, v-v_i] + s_i.$$

- This DP remains pseudo polynomial time but now the complexity is $O(nV)$ where $V = v_1 + v_2 + \ldots + v_n$.

An FPTAS for the knapsack problem

- This algorithm can be used as the basis for an efficient approximation algorithm for all input instances.

- The basic idea is relatively simple:
  - The high order bits/digits of the values can determine an approximate solution (disregarding low order bits after rounding up).
  - The fewer high order bits we use, the faster the algorithm but the worse the approximation.
  - The goal is to scale the values in terms of a parameter ϵ so that a $(1+\epsilon)$ approximation is obtained with time complexity polynomial in n and $(1/\epsilon)$.
  - The details are given in the DPV text (section 9.2.4) or the KT text (section 11.8).
  - Namely, KT set $\hat{v}_i = \lceil \frac{v_i n}{\epsilon v_{max}} \rceil$ where $v_{\max} = \max_j\{v_j\}$. DPV use the floor $\lfloor\ \rfloor$.
  - The running time is $O(n^3/\epsilon)$. Somewhat better bounds are know.

Looking ahead toward discussion of NP complete problems

- In term of computing optimal solutions, all "NP complete optimization problems" (i.e. optimization problems corresponding to NP complete decision problems) can be viewed (up to polynomial time) as a single class of problems.
- But in the world of approximation algorithms, this single class splits into many classes of approximation guarantees. Up to our believed complexity assumptions, we next discuss these possibilities.

Definition

1. An FPTAS (Fully Polynomial Time Approximation Scheme) algorithm is one that is polynomial time in the encoding of the input and $\frac{1}{\epsilon}$.
2. A PTAS (Polynomial Time Approximation Scheme) algorithm is one that that is polynomial in the encoding of the algorithm but can have any complexity in terms of $\frac{1}{\epsilon}$.

Different approximation possibilities for NP complete optimization

Given widely believed complexity claims

1. An FPTAS
   - e.g. the knapsack problem
2. A PTAS but no FPTAS
   - e.g. makespan (when the number of machines m is not fixed but rather is a parameter of the problem.
3. Having a constant $c > 1$ approximation but no PTAS
   - e.g. JISP
4. An $\Theta(\log n)$ approximation and no constant approximation
   - e.g. set cover H_n essentially tight.
5. No $n^{1-\epsilon}$ approximation for any $\epsilon > 0$
   - e.g. graph colouring and MIS for arbitrary graphs

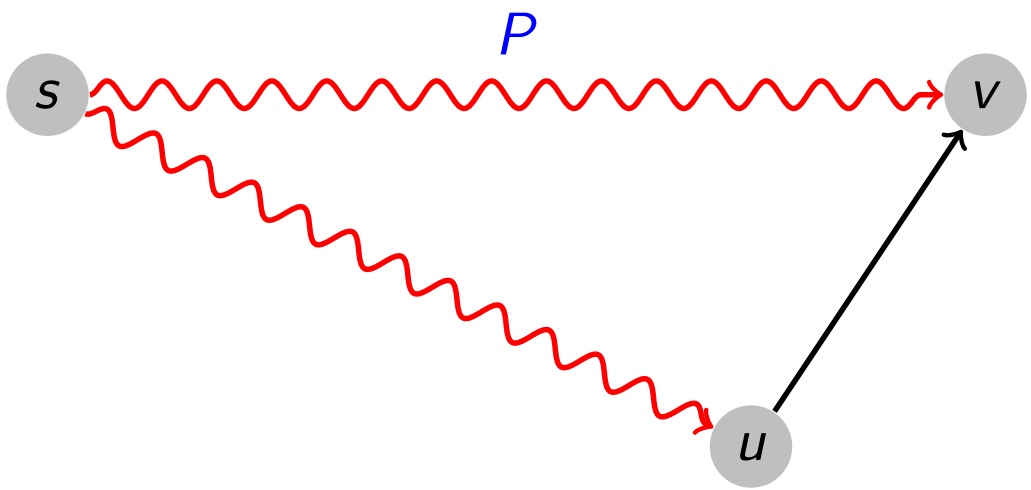
Here n stands for some input size parameter (e.g. size of the universe for set cover and number of nodes in the graph for colouring and MIS).

A DP with a sightly different style

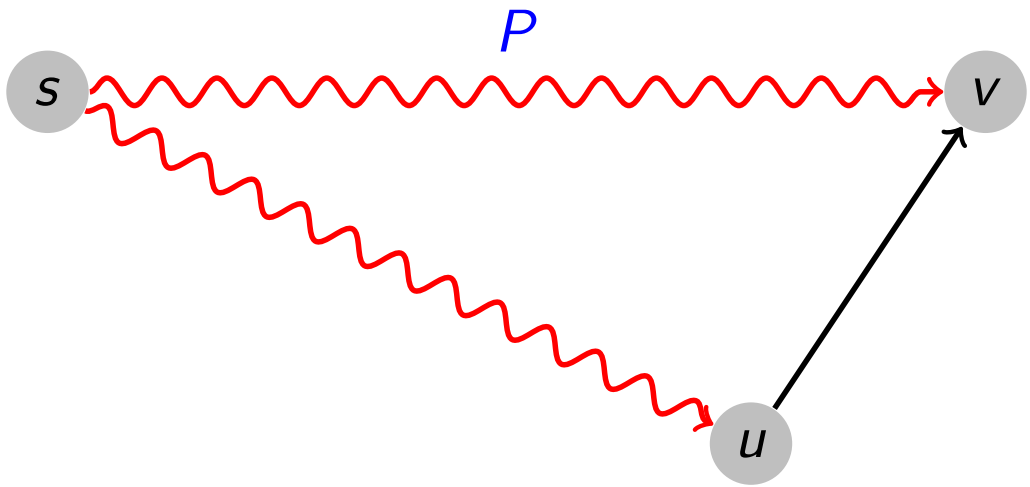
- Let's consider the single source least cost paths problem which is efficiently solved by Dijkstra's greedy algorithm for graphs in which all edge costs are non-negative.
- The least cost paths problem is still well defined as long as there are no negative cycles; that is, the least cost path is a simple path.
- The KT text presents the Bellman-Ford algorithm in Chapter 6.

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



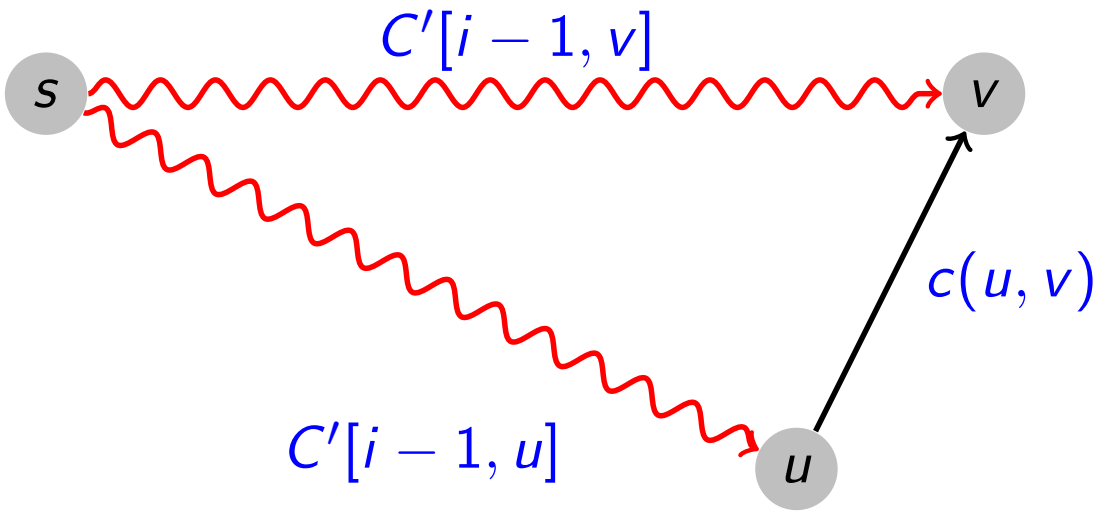
- This leads to the following semantic array:

$C[i, v]$ = 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 (computing the minimum cost path for each vertex) 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

- 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. The 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.