First, a Simple, But Not So Easy Problem of Shortest Path

A Mathematical Conundrum

- Who here has used MatLab?

- It's a program for doing math with matrices!

- (Recall that an $n \times m$ matrix $A$ is a 2D array of numbers $a_{ij}$, $1 \leq i \leq n$, $1 \leq j \leq m$)

- One of the things you can do with matrices is multiply them.

- Given matrices $A_{m \times p}$ and $B_{p \times n}$, product $C = A \cdot B$ is defined by

$$C_{ij} = \sum_{k=1}^{p} a_{ik} \times b_{kj}.$$
• (Recall that matrices must be conformable to multiply: # cols in first equals # rows in second)

• Total # scalar multiplies to compute $C$ is $mpn$.

• We can similarly define multiplication for an arbitrary product of matrices $A_1 \cdot A_2 \cdot \ldots \cdot A_N$ of various sizes, provided each successive pair is conformable.

That’s nice, but where’s the problem?

• Suppose you ask Matlab to compute a matrix product.

• Matrix multiplication is associative, so the program could compute a product $ABC$ as $A \cdot (B \cdot C)$ or $(A \cdot B) \cdot C$.

• However, note that the total work may not be equal for different associative orderings!

• **Example**: $A_{2 \times 12}, B_{12 \times 3}, C_{3 \times 4}$

• For $(AB)C$, we pay $2 \cdot 12 \cdot 3 + 2 \cdot 3 \cdot 4 = 96$

• For $A(BC)$, we pay $12 \cdot 3 \cdot 4 + 2 \cdot 12 \cdot 4 = 240$

• **Problem**: given a chain of matrices $A_1 \ldots A_N$ to multiply, where $A_i$ has size $p_i \times q_i$, find an associative ordering that minimizes the total number of scalar multiplies.

2.1 **Approach**

How should we address this problem?

• A greedy approach might choose least costly pair, most costly pair, leftmost pair, etc to do first.

• Unfortunately, none of these simple choices are optimal for all problem instances.

• We need a different approach!

**New idea**: consider more than one first choice!

• We will derive an ordering from outside in (pick last multiply first).
• Last multiply looks like

\[(A_1 \cdot \ldots \cdot A_k)(A_{k+1} \cdot \ldots \cdot A_N)\]

for some \(1 \leq k \leq N\).

• Each possible split point \(k\) is a choice; call it \(c_k\).

• Some opt soln must make choice \(c_k\) for at least one \(k\) between 1 and \(N\).

• Hence, our algorithm idea is: “compute best soln given each first choice \(c_k\), then take best overall”!

How does this help us?

• We’re going to follow a strategy similar to last time.

• **Inductive Structure**: for each initial choice, we are left with one or more smaller subproblems equivalent to first problem, with no external constraints.

• **Pf**: for each choice \(c_k\), we are left with problem of multiplying \(A_1 \ldots A_k\) and \(A_{k+1} \ldots A_N\) with least cost.

• These problems are the same as the top-level problem, and any solution is compatible with the initial division into \((A_1 \ldots A_k) \times (A_{k+1} \ldots A_N)\).

• **Optimal Substructure**: There must be optimal solns to subproblems induced by the first choice \(c_k\). An optimal solution to the problem must contain optimal solutions to its subproblems. Thus, combining these optimal sub-solutions with \(c_k\) yields a soln \(\Pi_k\) that is optimal among all solns that make choice \(c_k\).

• **Pf**: Let \(\Pi'_{k,\ell}\) and \(\Pi'_{k,r}\) be optimal associative orderings for the left and right subproblems.

• Cost of final multiply is \(p_1q_kq_N\) (computed from sizes of two subparts)

• Total cost of solution \(\Pi_k\) is therefore

\[\text{cost}(\Pi_k) = \text{cost}(\Pi'_{k,\ell}) + \text{cost}(\Pi'_{k,r}) + p_1q_kq_N.\]

• Suppose \(\Pi_k\) is optimal among solns that split first at \(k\), but the solution to one of the subproblems is not optimal. Suppose \(\Pi'_{k,r}\) is not optimal.
• Let $\Pi_{k,r}^*$ be a better solution than $\Pi'_{k,r}$. Then we can replace $\Pi'_{k,r}$ with $\Pi_{k,r}^*$ to derive a better solution to the original problem. This contradicts with the claim that $\Pi_k$ is optimal. Therefore, $\Pi_k$ must contain optimal solutions to its subproblems. QED

2.2 Review of Strategy So Far

So far, we’ve used a generalization of our three-part proof strategy to prove that (1) one of our first choices is compatible with optimality, and (2) we can recursively use the same choice to solve each subproblem induced by each choice.

1. **Complete choice property**: there exists an optimal solution that makes one of the first choices explored by our algorithm.

2. **Inductive structure**: each initial choice leaves a smaller version of the same problem with no external constraints.

3. **Optimal substructure**: for each initial choice, combining it with optimal solutions to the subproblem(s) induced by it yields a solution that is optimal given that choice.

As before, we can use these properties to prove inductively that our algorithm yields a global optimum. Detailed proof is left as an exercise (see handout).

2.3 Formalizing the Solution

So far, we’ve shown that optimizing over a set of initial choices yields a global optimum, but we’ve ignored any discussion of efficiency.

• Rather than trying to write pseudocode right away, let’s first develop a standard way to express our problem decomposition above.

• To compute an optimal solution recursively, we need to compute a large set of subproblems.

• Let the vector $[i,j]$ denote the subproblem of finding the optimal parenthesization of $A_i \ldots A_j$, $1 \leq i \leq j \leq N$.

• Let $T[i,j]$ be the cost of an optimal solution to subproblem $[i,j]$. 
• As we established through our optimal substructure property, if we make first choice \( c_k \) (i.e. divide product after \( A_k \)), we have that

\[
T[i, j \mid k] = T[i, k] + T[k, j] + p_i q_k q_j.
\]

• Moreover as we established through our complete choice property, 

\[
T[i, j] = \min_{i \leq k \leq j} T[i, j \mid k].
\]

• These two statements constitute a recurrence for computing \( T[i, j] \) from optimal solution costs for smaller subproblems.

• The chosen value of \( k \) (i.e. the “argmin” of the minimization) indicates which choice leads to an optimum.

• As with any recurrence, we need to specify one or more base cases whose values are known a priori, so that we have a place to start.

• In this case, \( T[i, i] = 0 \) for any \( i \), since no work is needed to compute a product of one matrix.

• Finally, what’s the goal (the final subproblem)? If our full problem is to multiple \( A_1 \ldots A_N \), we want to compute \( T[1, N] \).

We can describe any algorithm based on recursively optimizing over a collection of choices by a recurrence. The key steps are:

1. Define the domain of subproblems for a problem instance and index them by coordinate vectors (e.g. \([i, j]\)).

2. Define a recurrence that expresses the objective value of an optimal solution to an arbitrary subproblem in terms of the values of optimal solutions to smaller subproblems.

   (Strategy: recurrence should “fall out” of the optimality proof.)

3. Define any base cases necessary to start the recurrence.

4. Define the goal point of the recurrence.

Note that, while the recurrence formally computes only the objective value of an optimal solution, simply keeping track of the choices made in computing this value gives the solution itself.
2.4 Solving the Recurrence Efficiently

As our last step, we need to determine a strategy for efficiently solving our problem. This will fall naturally out of the recurrence!

- Problem size is $N$, number of matrices.
- Naively, we could compute the recurrence “top-down.”
- Each step of the recurrence needs $\Theta(N)$ local work (for min) and spawns $2N$ subproblems of varying sizes.
- If $C(N)$ is cost of computing the recurrence top-down for an input of size $N$, then we have

$$C(N) \geq aN + \sum_{i=1}^{N-1} [C(i) + C(N - i)]$$

$$= aN + 2 \sum_{i=1}^{N-1} C(i).$$

- I’m not sure exactly how fast $C(N)$ grows with $N$, but a simple analysis shows that $C(N) = \Omega(2^N)$.
- You really don’t want an exponential-time algorithm!

Is there a better way to solve this problem?

- Consider computation of $T[1, 4]$ as tree.

Note that there are many redundant subproblem computations. $T[1, 2]$, $T[2, 3]$ and $T[3, 4]$ are all computed twice.
• **Idea:** solve every subproblem only once!

Let’s compute this recurrence *bottom-up* rather than top-down.

• Observe that \( T[i, j] \) depends only on computation of \( T[s] \) for strictly shorter subproblems.

• More precisely, we can compute \( T[i, j] \) given \( T[k, \ell] \) for \( i \leq k < \ell \leq j \).

• These smaller subproblems \([k, \ell]\) are called the *dependencies* of \([i, j]\).

• Suppose we order the subproblems by increasing length follows:

  
  for \( d = 1 \ldots N - 1 \)
  for \( i = 1 \ldots N - d \)
  compute \( T[i, i + d] \) from its dependencies and store it

• When we compute \( T[i, j] \), we have already computed all its dependencies, so this is a feasible subproblem ordering.

OK, let’s do our final complexity analysis.

• Assume that we can look up stored \( T[i, j] \) value in \( O(1) \) time.

• Computing one subproblem minimizes over up to \( N \) constant-sized expressions on stored terms, so requires worst-case time \( \Theta(N) \).

• Number of subproblems in the domain is \( \Theta(N^2) \), since for each of the \( N \) values of \( i \), there are up to \( N \) values of \( j > i \).

• Overall cost of solving each subproblem once to obtain \( T[1, N] \) is therefore \( O(N^3)! \)

The strategy of building a recurrence, then solving it *bottom-up* is called *dynamic programming*. It leads to polynomial-time algorithms for problems that naively seem to require exponential time.

### 2.5 What We Know So Far - Recap

Here are the steps we’ve gone through in developing a DP algorithm.

1. Pick a set of initial choices.

2. Prove optimality of the choice set by showing:
(a) (Complete Choice): for every problem instance, at least one choice is consistent with an optimal solution.

(b) (Inductive Structure): for each initial choice $c_k$, show that making choice $c$ leaves one or more strictly smaller subproblems with no external constraints relative to $c_k$.

(c) (Optimal Substructure): for each initial choice $c_k$, if its induced subproblems are solved optimally, then combining their solutions with choice $c_k$ yields a solution that is optimal among all solutions that make choice $c_k$.

3. Define and index the domain of subproblems.

4. Using the optimality proof, derive a recurrence for the objective value of an optimal solution to an arbitrary subproblem.

5. Define base case(s) and goal point for the recurrence.

6. Find an ordering of subproblems that respects the dependencies of the recurrence.

7. Finally, solve the subproblems in the chosen order, tracking the choice made in each subproblem to obtain the actual solution as well as its objective value.

The above is a good framework for deriving and validating DP algorithms for lots of problems!

**Revisit Our Grid Path Problem**

This will be an informal discussion that provides a (visual) insight to DP. It will include formulation of a sequence alignment problem (a really big deal in computational biology), a dynamic programming solution we briefly discussed at the very beginning of this topic, and its complexity analysis. Here you will see the power of DP - asymptotically we pay nothing in computation time but save space in the order of $O(N)$. This memory saving makes a lot of real problems feasible.
3 Next Problem: Longest Common Subsequence

Here's one of a bunch of stringology problems for which dynamic programming works well.

- Consider the problem of diffing two text files $X$ and $Y$, each consisting of zero or more lines.

- To diff $X$ and $Y$, we first find an ordered list of common lines between them.

- Then, we insert and delete lines between the common lines to turn $X$ into $Y$.

- **Example:** compare $X$

  ```
  foo
  bar
  baz
  quux
  ```

  **to**

  ```
  bar
  xyzzy
  plugh
  baz
  foo
  quux
  ```

- The 3 lines “bar, baz, quux” occur in the same order in each file.

- Resulting diff from $X$ to $Y$ is:

  ```
  -foo
  bar
  +xyzzy
  +plugh
  baz
  +foo
  quux
  ```
To minimize the number of insertions and deletions needed to transform one file to the other, we want to find the longest possible ordered list of common lines between files.

More formally...

- Let $X[1..n]$ and $Y[1..m]$ be two strings of symbols (think of each line as a symbol over a ginormous alphabet)

- Find a sequence of index pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_q, y_q)$ such that
  1. $x_1 < x_2 \ldots < x_q$, and $y_1 < y_2 \ldots < y_q$
  2. $X[x_j] = Y[y_j]$ for $1 \leq j \leq q$
  3. the length $q$ of the sequence is maximal.

- This is the longest common subsequence (LCS) problem.

### 3.1 A DP Algorithm

How to proceed?

- We want to look at ways to match up symbols between $X$ and $Y$.

- A long common subsequence can be divided into its last common pair $(x_q, y_q)$ and the “rest,” which is confined to $X[1..x_q−1]$ and $Y[1..y_q−1]$.

- Intuitively, we want to find that last pair, then recursively find a long common subsequence on those prefixes of $X$ and $Y$.

- Is there an algorithm here?

On with the algorithm!

- Suppose we want an LCS for $X[1..n]$ and $Y[1..m]$.

- If $X[n] \neq Y[m]$, an LCS of $X$ and $Y$ can contain at most one of these two symbols.
• Hence, we have two possible first choices: delete \(X[n]\), or delete \(Y[m]\) (not mutually exclusive).

• If \(X[n] = Y[m]\), we also have a third choice: match up \(X[n]\) to \(Y[m]\).

• **Complete Choice**: there exists an LCS that either matches \(X[n]\) to \(Y[m]\) or fails to use at least one of \(X[n]\) or \(Y[m]\).

• **Pf**: it is not possible to match both \(X[n]\) and \(Y[m]\) to distinct characters in the other sequence, as they are the last in their sequences, and matches must respect order of both seqs. QED

• **Inductive Structure**: making any of the three choices leaves us with a subproblem of the original without external constraints.

• **Pf**: the 3 choices leave LCS subproblems for: \(X[1..n - 1]\) and \(Y[1..m - 1]\); and \(X[1..n - 1]\) and \(Y[1..m - 1]\). QED

• **Optimal Substructure**: for each choice, if we can solve the corresponding subproblem optimally, we get a longest common subseq for the full strings given the choice.

• **Pf**: Let LCS\((A,B)\) be longest common subseq of \(A\) and \(B\).

Then for the 3 cases, we have that \(|\text{LCS}(X,Y)|\) is respectively given by

- \(|\text{LCS}(X[1..n - 1], Y)|\),
- \(|\text{LCS}(X, Y[1..m - 1])|\), and
- \(|\text{LCS}(X[1..n - 1], Y[1..m - 1])| + 1\).

• In each case, apply standard contradiction argument. QED

We’ve practically written the recurrence already! Let’s finish the job.

• **General subproblem computes LCS for prefixes \(X[1..i]\) and \(Y[1..j]\).**  
  Call this subproblem \([i, j]\).

• Let \(L(i, j) = |\text{LCS}(X[1..i], Y[1..j])|\).

• We have

\[
L(i, j) = \max \left\{ \begin{array}{ll}
L(i-1, j) \\
L(i, j-1) \\
L(i-1, j-1) + 1 & \text{(if } X[i] = Y[j])
\end{array} \right\}
\]
• Base cases: \( L(i, 0) = L(0, j) = 0 \) (no symbols for LCS)

• Goal point: compute \( L(n, m) \)

• One of many feasible orderings of subproblems: compute \( L(i, *) \) for each \( i \) from 1 to \( n \), filling in each row in order of increasing \( j \).

• Each subproblem’s opt cost can be computed in \( O(1) \) time from previous subproblems by taking max over 3 cases.

• Conclude that whole algorithm takes \( O(mn) \) time.

Standard traceback gives actual LCS.

### 3.2 Improvements

For a practical implementation, one can often improve the naive recurrence to reduce constant factors, if not asymptotic cost.

• **Claim**: if \( X[i] = Y[j] \), then there exists an LCS of \( X[1..i] \) and \( Y[1..j] \) that includes the pair \((i, j)\).

• **Pf**: Let \( X[1..i] \) and \( Y[1..j] \) be as given, and let \( S \) be any LCS of them.

  • If \( S \) includes pair \((i, j)\), we are done.
  
  • Otherwise, \( S \) must use at least one pair \((i, p)\) or \((q, j)\); if not, we could add \((i, j)\) and get a longer solution.

  • Suppose \( S \) uses \((i, p)\). Delete this pair and add \((i, j)\) to get a solution of the same size. Similar hack fixes \((q, j)\).

  • Hence, can transform \( S \) into new LCS \( S^* \) that uses \((i, j)\). QED
Armed with this fact, we can simplify the recurrence to avoid checking other cases when last 2 chars match:

\[
L(i, j) = \begin{cases} 
L(i - 1, j - 1) + 1 & \text{if } X[i] = Y[j] \\
\max(L(i - 1, j), L(i, j - 1)) & \text{otherwise}
\end{cases}
\]

• What else can we do?
• Suppose \(X = atc\) and \(Y = cag\)
• Trivially, \(L(3, 1) = L(3, 2) = L(3, 3)\), because we cannot match \(X[3]\) anywhere after \(Y[1]\).
• Could use this fact to compute \(L(i, *)\) in time proportional to # of copies of \(X[i]\) in \(Y\), rather than \(|Y|\).
• For diff, where a line of \(X\) may typically occur only once in \(Y\), this is a big win.
• Suppose that, for each \(X[i]\), we have a list of \(j\) such that \(Y[j] = X[i]\), and vice versa.
• (Can build these lists in \(O(m + n)\) time via hashing)
• Exercise: construct sparse version of LCS algorithm that runs in time proportional to total length of these lists.
• There’s actually a (somewhat nontrivial) \(O(n \log n)\) algorithm for LCS, based on the longest increasing subsequence (LIS) computation, that is used by \texttt{diff}.

4 Back to the Sack

Let’s return to our old friend, the 0-1 knapsack problem.

• We are given \(n\) items.
• Item \(x_i\) has weight \(w_i\), value \(v_i\).
• We are also given a capacity \(W\).
• We seek highest-value subset of items with total weight \(\leq W\).
• Must take all or none of an item.
Recall that we failed miserably to apply greedy approach to this problem. Let’s try dynamic programming!

- Put the items in any order you want.
- We will consider each item in this order.
- Idea: do we put last item $x_n$ in knapsack? (Assumes it is feasible; if not, the answer is always no).
- Complete Choice: clearly, opt soln either does or does not contain item $x_n$, so one of two choices is consistent with optimality!
- Inductive Structure: let $(S,W)$ be an instance of the knapsack problem, where $S$ is the item set and $W$ is the knapsack capacity.
  - If we skip $x_n$, subproblem is $(S - \{x_n\},W)$
  - If we add $x_n$, subproblem is $(S - \{x_n\}, W - w_n)$
  - Both subproblems can be solved arbitrarily while producing a solution compatible with our first choice.
- Optimal Substructure: suppose we solve the subproblem $(S',W')$ optimally after making the first choice.
  - If we skip $x_n$, value of solution is that of subproblem.
  - If we add $x_n$, value of solution is that of subproblem plus $v_n$.
  - In each case, apply standard contradiction argument.

OK, on to recurrence...

- What is our domain?
- Given the sorted order on $S$, general subproblem is $(\{x_1 \ldots x_i\}, w)$ for any prefix of $S$ and some $w \leq W$.
- Let’s call the general subproblem $[i, w]$.
- (Note: provided item weights are rational, they can be considered integer, as can $w$, so $0 \leq w \leq W$ is a valid index set.)
- Let $V[i, w]$ be value of an opt solution for this subproblem.
• By substructure, we have that

\[
V[i, w] = \begin{cases} 
\max(V[i - 1, w], V[i - 1, w - w_i] + v_i) & \text{if } w_i \leq w \\
V[i - 1, w] & \text{otherwise}
\end{cases}
\]

• Base cases are \(V[i, 0] = 0\) (since we cannot add to a full knapsack) and \(V[0, w] = 0\) (since we have not yet added any items)

• Goal point is \(V[n, W]\).

• To go bottom-up, consider 2D matrix of 0..n by 0..W

• Initially, fill every point’s storage except bases with value \(-\infty\)

• Compute recurrence in column-major order from top to bottom

• (Note that unreachable points contribute \(-\infty\) to the max and so are not used)

• Domain size is \(\Theta(nW)\), and each subproblem takes \(\Theta(1)\) time.

4.1 A Good Case for Sparse DP

Knapsack is a good example of a problem where subproblems are sparse.

• Defining characteristic: not all points are actually used (note those \(-\infty\)'s).

• When going bottom-up, do not create storage for a point unless it is actually reachable by combining smaller subproblems.

• Since each increment in \(i\) adds a well-defined weight, can determine set of valid subproblems for given \(i\) from those for \(i - 1\).
• **Question:** how few subproblems can we actually compute for a given input of \( n \) items and capacity \( W \)?

• **(Note:** this question is no fun unless all items are feasible, i.e. of weight at most \( W \))

• **My answer:** suppose each \( w_i = W \); then we only compute \( n \) non-base-case points.

• **Question:** can we find an input that forces us to compute \( \Theta(nW) \) points?

• **My answer:** let \( W = \frac{n(n+1)}{2} - 1 \), and let \( w_i = i \).

• We can create solutions from items \( x_1 \ldots x_i \) with any weight between 0 and \( \sum_{k=1}^{i} k = \frac{i(i+1)}{2} \).

• Total number of computed points is therefore \( \Theta(n^3) \).

• (Similarly, can set \( W \approx 2^n \) and force time \( \Theta(n2^n) \).)

### 4.2 Complexity Analysis

Is \( \Theta(nW) \) a good result for worst-case efficiency? How can we tell?

• Consider the trivial knapsack algorithm that considers all subsets of items for feasibility and value.

• This algorithm is \( \Omega(2^n) \), since it considers subsets of \( n \) items.

• An obvious exponential is clearly asymptotically “bad,” but what about dependence on \( W \)?

• **Possible argument:** algorithm scales polynomially with \( W \).

• So, if \( W \) is polynomial in \( n \), cost remains polynomial. (But is this a reasonable restriction?)

• Also, if we double the value of \( W \), the cost doubles. Is this good or bad?

• We need a rational basis for evaluating parameter-dependent runtimes.

Let’s think carefully about how to represent the input.
• Complexity is supposed to be a function of input size.

• **Problem:** we haven’t specified exact representation of input!

• Here’s a sensible proposal . . .

• An input must be specified as a linear string of symbols from a fixed, finite alphabet $\Sigma$.

• Imagine that the symbols are written on a “tape” that is divided into cells, one symbol per cell.

• This model of representing input is intuitively sensible – computers take all input as strings of bits ($\Sigma = \{0, 1\}$)

• Moreover, it matches the abstract representation used by Turing machines, the canonical model of computation.

So what does an input to the knapsack problem look like?

• Must specify $W$, weights $w_i$, and values $v_i$

• Could concatenate these as a string:

  $\#W\#v_1\#w_1\#v_2\#w_2\#\ldots\#v_n\#w_n\#$

• But how do we represent each number?

• If alphabet has $k + 1$ symbols, we can use one symbol for separator and the remaining syms to write numbers in base-$k$ notation.

• Capacity $W$ therefore requires $\log_k W$ symbols to store.

• WLOG, weights $w_i$ may be limited to be $\leq W$.

• Let’s ensure that weights are *dominant* cost for problem specification: each $v_i$ is also at most $W$.

• Then input has total size $\Theta(n \log W)$. 
• (Note that base of log has no place in Θ notation; size of alphabet is immaterial.)

• This is how a real computer would store an input, and it uses asymptotically densest form of storage for numbers.

Let’s now redo the complexity analysis, fixing this representation.

• Let \( b = \log_2 W \), so that \( W = 2^b \).

• Input size is now \( \Theta(nb) \).

• For this size, cost is \( \Theta(n2^b) \).

• Notice that, if we increase the input size (i.e. the space used to store \( W \)) by even one bit, then the cost doubles!

• Suddenly, this idea doesn’t look so attractive anymore!

• **Defn:** Let \( P \) be a combinatorial problem whose input includes a numerical parameter \( W \).

• An algorithm for \( P \) whose cost grows polynomially with the value \( W \) is said to be **pseudopolynomial** in \( W \).

• Such an algorithm is in fact exponential in \( \log W \), the size of \( W \)’s representation in the computer.

• It is *not* a fully polynomial-time solution to \( P \).

• (But it may be useful if you know \( W \) won’t get too big.)

In fact, there is no known algorithm for 0-1 knapsack that is polynomial in both \( n \) and \( W \). More on that later...

## 5 Return to Another Old Problem - Shortest Paths

Let’s look at the ever-popular shortest path problem.

• Given a weighted, directed graph \( G = (V, E) \), a source vertex \( s \), and a target vertex \( t \), find a shortest path in \( G \) from \( s \) to \( t \).

• For fun, let’s allow \( G \) to have negative-weight edges.
• (Clearly, these aren’t distances – could be bonuses/penalties)

• A “shortest path” is now one with weight less than all others (can be negative)

• (Note that Dijkstra’s algorithm cannot deal with this case – it breaks if edge weights are < 0.)

• However, we will forbid negative-weight cycles

• If a cycle with weight < 0 exists, then a path can take arbitrarily many trips around it to get arbitrarily negative weight!

• Without negative-weight cycles, shortest paths are WLOG acyclic.

Let’s consider a DP approach to this problem.

5.1 A Stab at an Algorithm

How might you attack shortest paths via DP?

•

• Complete Choice: Consider a shortest path \( \pi \) from vertex \( s \) to vertex \( t \).

• Either \( \pi \) is a single edge \( s \to t \) (no subproblem), or there is an intermediate vertex \( k \) on the path.

• Inductive Structure: If we say that \( \pi \) from \( s \) to \( t \) passes through \( k \), we are left to determine how to get from \( s \) to \( k \) and from \( k \) to \( t \). These look like inputs to shortest path problem, and they are unconstrained given choices of \( s \), \( t \), and \( k \).

• (We’re allowing repeated vertices – it’s OK, paths with cycles are feasible, though they are not in general shortest.)

• Optimal Substructure: Suppose that for choice \( k \), shortest subpaths have lengths \( \ell(s, k) \) and \( \ell(k, t) \).

• Observe that total length of concatenated path is

\[
\ell(s, t) = \ell(s, k) + \ell(k, t).
\]
• Apply contradiction argument to show that this path is shortest among all those that pass through \( k \). QED

Nifty. Let’s develop a recurrence.

• General subproblem: let \( D(i, j) \) be score of shortest path from vertex \( i \) to vertex \( j \).

• Using our decomposition by choices,

\[
D(i, j) = \min \left\{ w(i, j) \min_k D(i, k) + D(k, j) \right\}
\]

• And the base case is... (ask)

• We can’t actually find one! There is no starting point, as even path from vertex 1 to vertex 2 could go through vertex 37.

• So much for this problem formulation!

• The real issue: our “subproblems” are not actually smaller in any sense – inductive structure argument was incorrect!

Crap! Now what do we do?

### 5.2 Alternate Plan B

What is the problem in Plan A?

• We are not organized.

• Idea to fix it: What if we introduce some structure?

• What about enumerate on path lengths?

• From \( s \) to \( t \), we first consider the path of one step, then all paths of 2 steps, then 3 steps, and so on.

• Will this give us a solution?
5.3 Alternate Plan C

Following algorithm is due to Floyd and Warshall.

- Number the vertices of $G$ $1 \ldots n$.
- A shortest path from $s$ to $t$ has some largest intermediate vertex $k$ (not the endpoints).
- (If $k = 0$, path must be a single edge from $s$ to $t$)
- **Complete Choice**: choose largest intermediate vertex on path to be one of $0 \ldots n$
- (Could skip $s$ and $t$ if desired, since WLOG path need not have cycles.)
- **Inductive Structure**: If shortest path from $s$ to $t$ has largest intermediate vertex $k > 0$, then it decomposes into shortest subpaths from $s$ to $k$ and from $k$ to $t$.

- Again, no constraints given $s$, $t$, and $k$.
- **Check**: do we have a basis for induction now?
- WLOG, shortest path from $s$ to $t$ is acyclic, since no cycle can decrease its total length.
- Hence, $k$ does not occur twice, and so two subpaths have largest intermediate vertex $\leq k - 1$.
- **Optimal Substructure**: Suppose shortest subpaths given max vertex $k$ have lengths $\ell(s, k)$ and $\ell(k, t)$.
  - Total length of resulting path is $\ell(s, k) + \ell(k, t)$.
  - Contradiction argument shows that this path must be a shortest path from $s$ to $t$ given max vertex $k$. QED

OK, recurrence time again.
• Arbitrary subproblem involves getting from node $i$ to node $j$ with max intermediate vertex equal to $k$.

• Three parameters, so index problem as $[i,j,k]$.

• Let $D[i,j,k]$ be length of shortest path from $i$ to $j$ with largest intermediate vertex $k$.

• By substructure, we can write

\[
D[i,j,k] = \min_{p,q<k} (D[i,k,p] + D[k,j,q]) = \min_{p<k} D[i,k,p] + \min_{q<k} D[k,j,q].
\]

• **Base case:** $D[i,j,0]$ is $w(i,j)$ (which may be $\infty$ if there is no edge from $i$ to $j$)

• **Base case:** $D[i,i,0]$ is by defn 0.

• **Goal:** $\min_{k=0}^n D[s,t,k]$

  (Note that goal is not one point but rather $\min$ over $\Theta(n)$ points.)

Ordering and cost?

• Must compute $D[i,j,k]$ for all $i$, $j$, $k \leq n$.

• Hence, must compute $\Theta(n^3)$ domain points.

• Base case lookups in $G$ are $O(1)$, provided we use adjacency matrix.

• Each other points takes $\Theta(n)$ time for the two independent mins.

• Conclude that total cost is $\Theta(n^4)$.

5.4 **We Can Do Better**

Can we get this running time down?

• Note that for each $k$, we are computing

\[
\min_{p<k} D[i,k,p] + \min_{q<k} D[k,j,q]
\]

• This min operation is extremely redundant!
• When moving from $k$ to $k + 1$, each min includes all the terms we’ve already looked at, plus a few more (for $p$ or $q = k + 1$).

• Is there a way to avoid so much repetition?

**Idea:** compute min progressively!

• Let $C[i, j, k] = \min_{p \leq k} D[i, j, p]$

• Then we may write

\[ D[i, j, k] = C[i, k, k - 1] + C[k, j, k - 1]. \]

• Also, we have that

\[ C[i, j, k] = \min\{C[i, j, k - 1], D[i, j, k]\} \]

• We can rewrite the last expression fully in terms of $C$ by subbing in definition of $D$:

\[ C[i, j, k] = \min \begin{cases} C[i, j, k - 1] \\ C[i, k, k - 1] + C[k, j, k - 1] \end{cases} \]

• **Base case:** $C[i, j, 0] = D[i, j, 0] = w(i, j)$, or 0 if $i = j$.

• **Goal point:** $C[s, t, n]$

• **Analysis:** still $\Theta(n^3)$ cells as before, but now each cell takes only $O(1)$ time to compute.

• Total cost is $\Theta(n^3)$!

**Moral:** keep an eye out for opportunities to implement progressive min and max. You could also reformulate choice initially to build in the min: shortest path has largest intermediate vertex *at most* $k$, rather than exactly $k$.

6 Why Do This?

What’s so great about cubic time?

• Observe that one run of DP algorithm actually computes $C[s, t, n]$ for *every* pair of vertices $s$, $t$, not just the pair specified in the input.
• Hence, in $\Theta(n^3)$ time, computes *all-pairs shortest paths*.

• Compare to $\Theta(n^2 \log n)$ to use Dijkstra for single-source shortest paths on a dense graph.