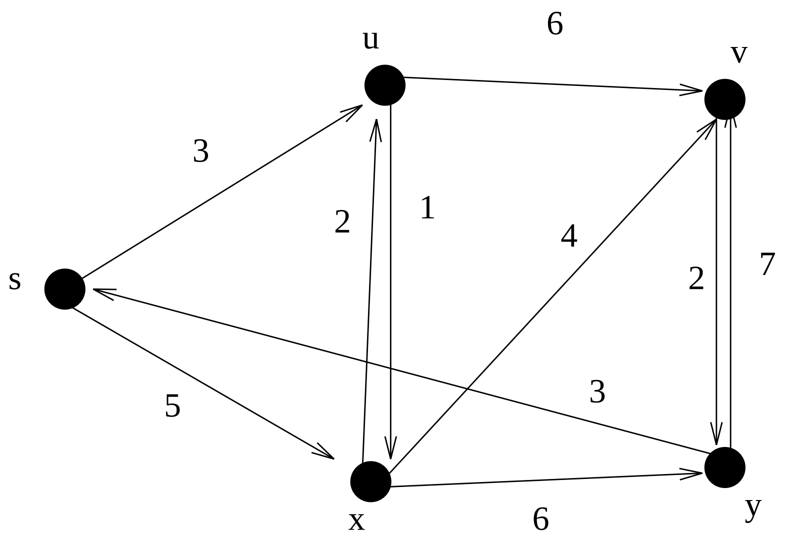
25.1-1 Give two more shortest path trees for the following graph:



Run through Dijkstra's algorithm, and see where there are ties which can be arbitrarily selected.

There are two choices for how to get to the third vertex x, both of which cost 5.

There are two choices for how to get to vertex v, both of which cost 9.

Lessons from the Backtracking contest

- As predicted, the speed difference between the fastest programs and average program dwarfed the difference between a supercomputer and a microcomputer. Algorithms have a bigger impact on performance than hardware!

- Different algorithms perform differently on different data. Thus even hard problems may be tractable on the kind of data you might be interested in.

- None of the programs could efficiently handle all instances for $n \approx 30$. We will find out why after the midterm, when we discuss NP-completeness.

- Many of the fastest programs were very short and simple (KISS). My bet is that many of the enhancements students built into them actually showed them down! This is where profiling can come in handy.

- The fast programs were often recursive.

Winning Optimizations

- Finding a good initial solution via randomization or heuristic improvement helped by establishing a good upper bound, to constrict search.

- Using half the largest vertex degree as a lower bound similarly constricted search.

- Pruning a partial permutation the instant an edge was $\geq$ the target made the difference in going from (say) 8 to 18.

- Positioning the partial permutation vertices separated by b instead of 1 meant significantly earlier cutoffs, since any edge does the job.

- Mirror symmetry can only save a factor of 2, but perhaps more could follow from partitioning the vertices into equivalence classes by the same neighborhood.

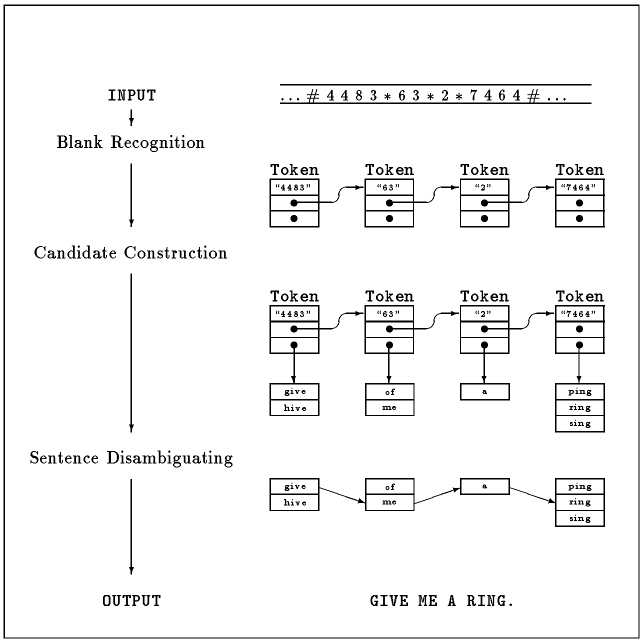
Shortest Paths

Finding the shortest path between two nodes in a graph arises in many different applications:

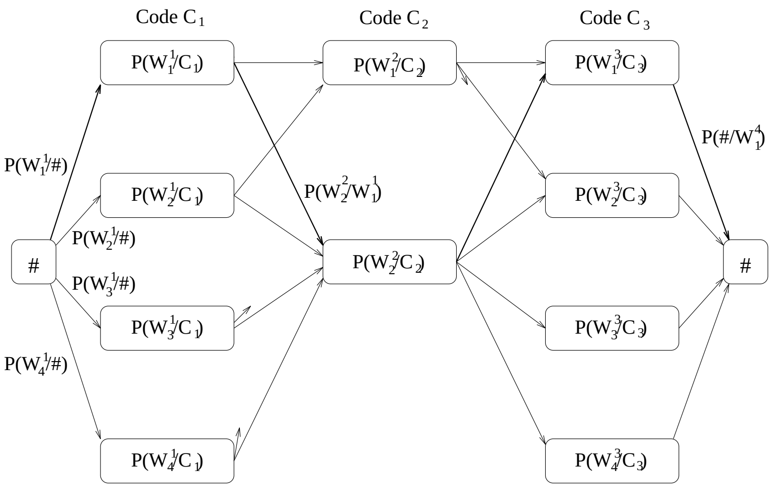
- Transportation problems – finding the cheapest way to travel between two locations.
- Motion planning – what is the most natural way for a cartoon character to move about a simulated environment.
- Communications problems – how look will it take for a message to get between two places? Which two locations are furthest apart, ie. what is the *diameter* of the network.

Shortest Paths and Sentence Disambiguation

In our work on reconstructing text typed on an (overloaded) telephone keypad, we had to select which of many possible interpretations was most likely.



We constructed a graph where the vertices were the possible words/positions in the sentence, with an edge between possible neighboring words.



The weight of each edge is a function of the probability that these two words will be next to each other in a sentence. ‘hive me’ would be less than ‘give me’, for example.

The final system worked extremely well – identifying over 99% of characters correctly based on grammatical and statistical constraints.

Dynamic programming (the Viterbi algorithm) can be used on the sentences to obtain the same results, by finding the shortest paths in the underlying DAG.

Finding Shortest Paths

In an unweighted graph, the cost of a path is just the number of edges on the shortest path, which can be found in $O(n+m)$ time via breadth-first search.

In a weighted graph, the weight of a path between two vertices is the sum of the weights of the edges on a path.

BFS will not work on weighted graphs because sometimes visiting more edges can lead to shorter distance, ie. $1+1+1+1+1+1+1 < 10$.

Note that there can be an exponential number of shortest paths between two nodes – so we cannot report all shortest paths efficiently.

Note that negative cost cycles render the problem of finding the shortest path meaningless, since you can always loop around the negative cost cycle more to reduce the cost of the path.

Thus in our discussions, we will assume that all edge weights are positive. Other algorithms deal correctly with negative cost edges.

Minimum spanning trees are uneffected by negative cost edges.

Dijkstra's Algorithm

We can use *Dijkstra's algorithm* to find the shortest path between any two vertices s and t in G.

The correctness of Dijkstra's algorithm is based on this:

- If $s, \ldots, x, \ldots, t$ is the shortest path from s to t, then $s, \ldots, x$ had better be the shortest path from s to x – or else we could make it shorter.

- The distance from s to x will be less than than of that from s to t, if all edges have positive weight.

This suggests a dynamic programming-like strategy, where we store the distance from s to all nearby nodes, and use them to find the shortest path to more distant nodes.

The shortest path from s to s, $d(s,s) = 0$. If all edge weights are positive, the *smallest* edge incident to s, say (s,x), defines $d(s,x)$.

We can use an array to store the length of the shortest path to each node. Initialize each to ∞ to start.

Soon as we establish the shortest path from s to a new node x, we go through each of its incident edges to see if there is a better way from s to other nodes thru x.

$known = \{s\}$
for $i = 1$ to n do $dist[i] = \infty$
for each edge (s, v), $dist[v] = d(s, v)$
last=s
while $(last \neq t)$
    select v such that $dist(v) = \min_{unknown} dist(i)$
    for each (v, x), $dist[x] = \min(dist[x], dist[v] + w(v, x))$
    last=v
    $known = known \cup \{v\}$

INCLUDE FIGURE OF EXAMPLES OF SHORTEST PATH ALGORITHMS

Complexity $\rightarrow O(n^2)$ if we use adjacency lists and a Boolean array to mark what is known.

This is essentially the same as Prim's algorithm.

An $O(m \lg n)$ implementation of Dijkstra's algorithm would be faster for sparse graphs, and comes from using a heap of the vertices (ordered by distance), and updating the distance to each vertex (if necessary) in $O(\lg n)$ time for each edge out from freshly known vertices.

Even better, $O(n \lg n + m)$ follows from using Fibonacci heaps, since they permit one to do a decrease-key operation in $O(1)$ amortized time.

All-Pairs Shortest Path

Notice that finding the shortest path between a pair of vertices (s, t) in worst case requires first finding the shortest path from s to all other vertices in the graph.

Many applications, such as finding the center or diameter of a graph, require finding the shortest path between all pairs of vertices.

We can run Dijkstra's algorithm n times (once from each possible start vertex) to solve all-pairs shortest path problem in $O(n^3)$. Can we do better?

Improving the complexity is an open question but there is a *super-slick* dynamic programming algorithm which also runs in $O(n^3)$.

Dynamic Programming and Shortest Paths

The four-step approach to dynamic programming is:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute this recurrence in a bottom-up fashion.
4. Extract the optimal solution from computed information.

From the adjacency matrix, we can construct the following matrix:

$D[i,j] = \infty$, if $i \neq j$ and (v_i, v_j) is not in E
$D[i,j] = w(i,j)$, if $(v_i, v_j) \in E$
$D[i,j] = 0$, if $i = j$

This tells us the shortest path going through no intermediate nodes.

There are several ways to characterize the shortest path between two nodes in a graph. Note that the shortest path from i to j, $i \neq j$, using at most M edges consists of the shortest path from i to k using at most $M-1$ edges$+W(k,j)$ for some k.

This suggests that we can compute all-pair shortest path with an induction based on the number of edges in the optimal path.

Let $d[i,j]^m$ be the length of the shortest path from i to j using at most m edges.

What is $d[i,j]^0$?

$$\begin{aligned} d[i,j]^0 &= 0 \text{ if } i = j \\ &= \infty \text{ if } i \neq j \end{aligned}$$

What if we know $d[i,j]^{m-1}$ for all i,j?

$$\begin{aligned} d[i,j]^m &= \min(d[i,j]^{m-1}, \min(d[i,k]^{m-1} + w[k,j])) \\ &= \min(d[i,k]^{m-1} + w[k,j]), 1 \leq k \leq i \end{aligned}$$

since $w[k,k] = 0$

This gives us a recurrence, which we can evaluate in a bottom up fashion:

for $i = 1$ to n
    for $j = 1$ to n
    $d[i,j]^m = \infty$
    for $k = 1$ to n
        $d[i,j]^0$=Min($d[i,k]^m$, $d[i,k]^{m-1} + d[k,j]$)

This is an $O(n^3)$ algorithm just like matrix multiplication, but it only goes from m to $m+1$ edges.

Since the shortest path between any two nodes must use at most n edges (unless we have negative cost cycles), we must repeat that procedure n times ($m = 1$ to n) for an $O(n^4)$ algorithm.

We can improve this to $O(n^3 \log n)$ with the observation that any path using at most $2m$ edges is the function of paths using at most m edges each. This is just like computing $a^n = a^{n/2}$ x $a^{n/2}$. So a logarithmic number of multiplications suffice for exponentiation.

Although this is slick, observe that even $O(n^3 \log n)$ is slower than running Dijkstra's algorithm starting from each vertex!

The Floyd-Warshall Algorithm

An alternate recurrence yields a more efficient dynamic programming formulation. Number the vertices from 1 to n.

Let $d[i,j]^k$ be the shortest path length from i to j using only vertices from $1, 2, ..., k$ as possible intermediate vertices.

What is $d[j,j]^0$? With no intermediate vertices, any path consists of at most one edge, so $d[i,j]^0 = w[i,j]$.

In general, adding a new vertex $k+1$ helps iff a path goes through it, so

$$
\begin{aligned}
d[i,j]^k &= w[i,j] \text{ if } k = 0 \\
&= \min(d[i,j]^{k-1}, d[i,k]^{k-1} + d[k,j]^{k-1}) \text{ if } k \geq 1
\end{aligned}
$$

Although this looks similar to the previous recurrence, it isn't. The following algorithm implements it:

$d^o = w$
for $k = 1$ to n
    for $i = 1$ to n
        for $j = 1$ to n
        $d[i,j]^k = \min(d[i,j]^{k-1}, d[i,k]^{k-1} + d[k,j]^{k-1})$

This obviously runs in $\Theta(n^3)$ time, which asymptotically is no better than a calls to Dijkstra's algorithm. However, the loops are so tight and it is so short and simple that it runs better in practice by a constant factor.