# Lecture 18:
# Shortest Paths (1997)

**Steven Skiena**

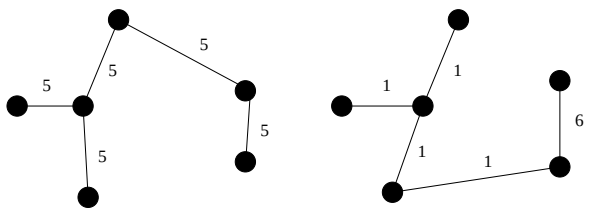
Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/∼skiena

*Describe an efficent algorithm that, given an undirected graph $G$, determines a spanning tree $G$ whose largest edge weight is minimum over all spanning trees of $G$.*

---

First, make sure you understand the question




Lower maximum edge weight  Lower total weight

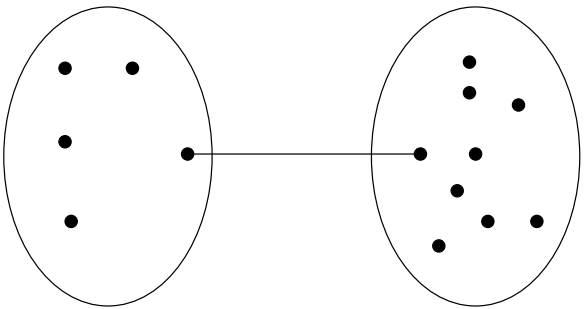
"Hey, doesn't Kruskal's algorithm do something like this." Certainly! Since Krushal's algorithm considers the edges in order of increasing weight, and stops the moment these edges form a connected graph, the tree it gives must minimize the

edge weight.

“Hey, but then why doesn’t Prim’s algorithm also work?”

It gives the same thing as Kruskal’s algorithm, so it must be true that any minimum spanning tree minimizes the maximum edge weight!

Proof: Give me a MST and consider the largest edge weight,



Deleting it disconnects the MST. If there was a lower edge connects the two subtrees, I didn’t have a MST!

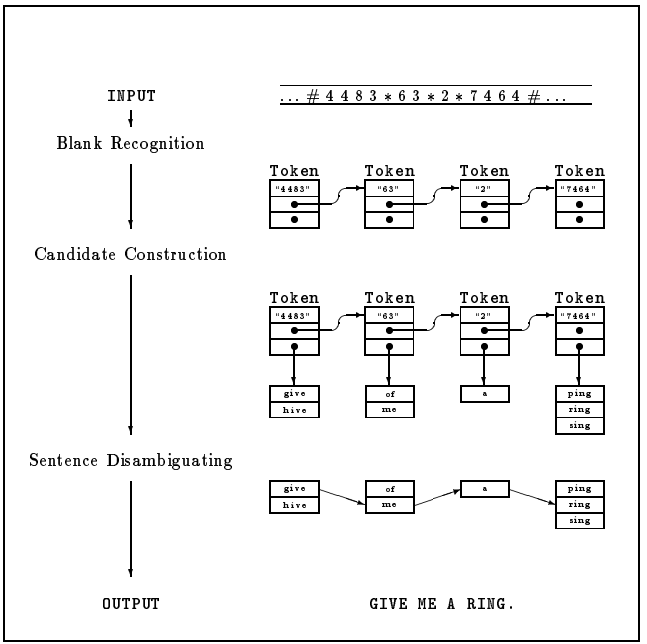
# Shortest Paths

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

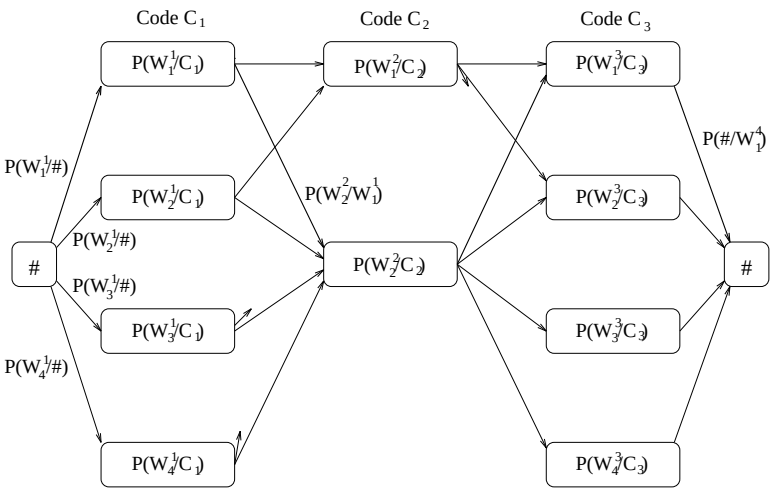
- Transportation problems – finding the cheapest way to travel between two locations.
- Motion planning – what is the most natural way for a cartoon character to move about a simulated environment.
- Communications problems – how look will it take for a message to get between two places? Which two locations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


INPUT
...#4483*63*2*7464#...
Blank Recognition
Token
"4483"
Token
"63"
Token
"2"
Token
"7464"
Candidate Construction
Token
"4483"
Token
"63"
Token
"2"
Token
"7464"
give
hive
of
me
a
ping
ring
sing
Sentence Disambiguating
give
hive
of
me
a
ping
ring
sing
OUTPUT
GIVE ME A RING.

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 principle behind Dijkstra's algorithm is that 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 $x$.

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 $\infty$ 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 $n$, $dist[i] = \infty$
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=$v$
    $known = known \cup \{v\}$

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.