Recall: "Basic Algorithm" for shortest path

\[ \forall u \in V \quad d(u) = \begin{cases} \infty & , u \neq v_0 \\ 0 & , u = v_0 \end{cases} \]

Repeat:

- pick \( u \in V \) \\
- \( d(u) := BF(u) \) \quad 'Bellman-Ford'

Until no change

---

Dijkstra's Algorithm

- works by expanding outward from source, taking into account edge weights, to finding vertex with the least cost

- essentially, running Bellman-Ford on the closest neighbor
- Why Dijkstra Works

Case 1:

$U_0 = \text{destination from } V_0$

$(V_0, \epsilon_0, \epsilon_1)$ is

alternate path to $U_0$

Given the last time the alternate path

goed out of $U$ is $\epsilon_0$.

There was already a path from $V_0$ to $\epsilon_0$.

if $\text{cost (} V_0, \epsilon_0, \epsilon_1, U_0 \text{)} < \text{cost (} V_0, \epsilon_0, U_0 \text{)}$

then $\text{cost (} V_0, \epsilon_0, \epsilon_1 \text{)} < \text{cost (} V_0, \epsilon_0, U_0 \text{)}$

This means that $\epsilon_1$ was closer than $U_0$.

Which is not true since it was already

selected that $U_0$ was the next closest neighbor
to $V_0$. 
Case 2:

$U_0 = \text{destination from } V_0$

$(V_0, E_0, U_0) = \text{alternate path to } U_0$

With $r_0$ and $t_0$ both in $U$, the last step to $U_0$ is from $U$ no matter what.

Also, a direct way from $V_0$ to $t_0$ has to exist because $t_0$ is in $U$.

\[ \therefore \text{if } \text{cost} (V_0, r_0, U_0) > \text{cost} (V_0, t_0, U_0) \]

then $U_0$ would have been mistakenly calculated in the first place.
- The Algorithm

\[ U = \{ v_0 \} \]

\[ \forall u : d(u) = w(v_0, u) \]

Repeat:

\[
\text{pick } u \in V \setminus U \quad \text{"vertices not in } U \text{ yet"}
\text{with least } d(u)
\]

\[ U := U \cup \{ u \} \quad \text{"union, add to } U \text{"} \]

For all \( r \in V \setminus U \):

\[ d(r) = \min \{ d(r), d(u) + w(u, r) \} \]

Until \( U = V \)
Example:

\[ U \]

\[ V_0 \]

\[ d = 5 \]

\[ d = 3 \]

\[ d = \infty \]

since lowest

next iteration:

\[ U \]

\[ V_0 \]

\[ d = 4 \]

\[ d = 3 \]

\[ d = 5 \]

Note:

New Values
- Cost of Dijkstra:

\[ |V| \text{ for outer loop} \]
\[ |V| \text{ for finding lowest } d(u) \]
\[ 1 \text{ for } U := U \cup \{ u \} \]
\[ |V| \text{ for } d(r) = \min \{ d(r), d(u) + w(u, r) \} \]

\[ \Rightarrow O(n \times (n + n + 1)) \Rightarrow O(n^2) \]

with \( n = |V| \)

Priority Queue consideration:
recall (from CS 128?):

```
```

```
```

```
```

\[ \Rightarrow \text{getting the minimum} = O(\log n) \]

\[ \Rightarrow O(m + m \log n + m) \text{ using priority queue} \]

with \( m = |E| \)

- ultimately however, it is usually unknown whether \( m \) is larger than \( n \) substantially
  \[ \Rightarrow \text{dependence on whether graph is sparse} \]
- possible to change algorithm knowing graph details
New Problem

"Imagine ...

"ordered vertices"

$d^k(i,j)$ "the $k$th approximation of distance between $i$ and $j" \Rightarrow "going j from i"

- When $k = n$, the $d^k(i,j)$ is the actual $d(i,j)$

$d^k(i,j) = "cost of shortest path from i to j while visiting only intermediate nodes in V_1, V_2, \ldots, V_k"

Consider :

Goal = compute $d^{k+1}$ from $d^k$
The Algorithm

\[ V = \{ V_0, V_1, \ldots, V_n \} \]

\[ \forall (i, j) : d^0(i, j) = w(i, j) \]

For \( k = 0 \) to \( n-1 \)

\[ \forall i, j : d^{k+1}(i, j) = \min \sum d^k(i, j), d^k(j, k+1) + d^k(k+1, j) \]

end

return \( d^n(i, j) \)
\[ \text{Cost} \Rightarrow O(n^3) \]

- expensive, but finds all shortest paths for all \((i, j)\) pairs in a graph.

Regular Expressions for Graphs

Given:

\[ d(5, 3) = c \ast d \quad \text{"way to 5 to 3"} \]
\[ d(2, 3) = r + bc \ast d \quad \text{"either r or bc \ast d possible from 2 to 3"} \]

note: \(w(i, j)\) = letter of alphabet from \(i\) to \(j\)

also: "+" is union so interchangeable with "∪"
- So...

\[ d^{k+1}(i, j) = \min \{ d^k(i, j), d^k(i, k+1) + d^k(k+1, j) \} \]

can be rewritten as:

\[ d^{k+1}(i, j) = d^k(i, j) \cup d^k(i, k+1) \cdot d^k(k+1, j) \]

- Note: "\( \cup \)" is the dot is concatenation.

- However, newly written form is just "Almost Correct"
- It doesn't take into account loops, which can improve the final result.

Therefore:

\[ d^k(k+1, k+1) \]

needs to be added to the final expression for loop consideration.

- Problems relate back to "Dynamic Programming"