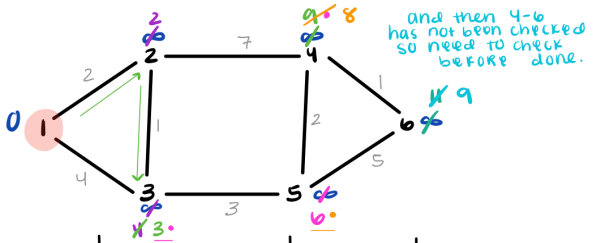
# Dijkstra's

shortest path from one node to all nodes. $O(v^2)$



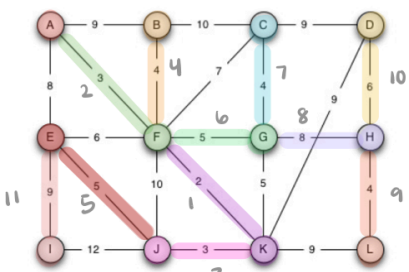

| vertex | shortest d | prev u. | visited order |
|---|---|---|---|
| 1 | 0 | x | 1 |
| 2 | ~~∞~~ 2 | 1 | 2 |
| 3 | ~~∞~~ ~~4~~ 3 | ~~x~~ 2 | 3 |
| 4 | ~~∞~~ ~~9~~ 8 | ~~2~~ 5 | 5 |
| 5 | ~~∞~~ 6 | 3 | 4 |
| 6 | ~~∞~~ ~~9~~ 9 | ~~5~~ 4 | 6 |

1) start u. = 0, others = ∞
2) check every vertex attached to initial vertex
3) pick min value from attached vertices and updated vertex value if shorter path exists
4) Repeat step 3 with final vertices from step 3 and continue until final node.

Note: you need to check all attached vertices from current node before it can be officially "visited" (esp. node 4)

# Prims

start with vertex of order 1 (beg.) then pick shortest length attached to any previous vertex and does not form a cycle.



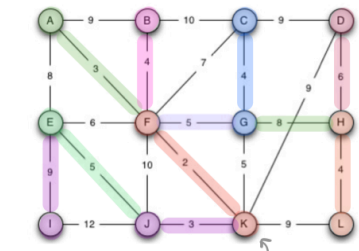
| visited | |
|---|---|
| A | C |
| F | H |
| K | L |
| J | D |
| B | I |
| E | |
| G | |

$O(e \log v)$

order of edges:
(A, F), (F, K), (K, J), (F, B), (E, J)
(F, G), (C, G), (G, H), (L, H), (D, H), (E, I)

# Kruskal's

pick the smallest edge while not creating a cycle



| visited edges | |
|---|---|
| F, K | E, J |
| A, F | F, G |
| J, K | D, H |
| B, F | G, H |
| C, G | E, I |
| H, L | |

$O(e \log v)$

would have formed a cycle!

```
BW-Relax(u, v):
    potentialBW = []
    IF w(u, x) < w(x, v):
        potentialBW.append(w(u,x))
    elseif w(u, x) > w(x, v):
        potentialBW.append(w(x,v)
```

# Floyd-Warshall

shortest path btwn. all pairs of vertices $O(n^3)$

$$\begin{bmatrix} 0 & 3 & \infty & 2 \\ 3 & 0 & 8 & 6 \\ \infty & 8 & 0 & 5 \\ 2 & 6 & 5 & 0 \end{bmatrix}$$

The diagonal never changes, it is ALWAYS 0.
Then $D_x$ has row x and col x stay the same from the previous D.
Then if ∞ is in the $D_x$ row or col. then the #'s in that other row and col. also stay the same as the previous D.

Ex:

$D_1$ =

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 3 | ∞ | 2 |
| 2 | 3 | 0 | 8 | 6 |
| 3 | ∞ | 8 | 0 | 5 |
| 4 | 2 | 6 | 5 | 0 |

then the other #'s need to be updated based on: $d(i,j) > d(i,k) + d(k,j)$

$d(4,2) > d(4,1) + d(1,2)$
$6 > 2 + 3$ TRUE so update!

$d(2,4) > d(2,1) + d(1,4)$
$6 > 3 + 2$ True so update!

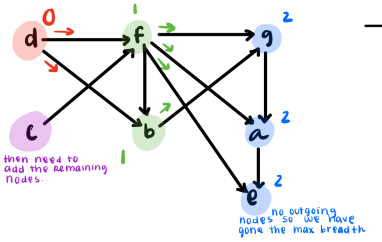
$D_1$ =

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 3 | ∞ | 2 |
| 2 | 3 | 0 | 8 | 5 |
| 3 | ∞ | 8 | 0 | 5 |
| 4 | 2 | 5 | 5 | 0 |

Then Repeat!

# BFS

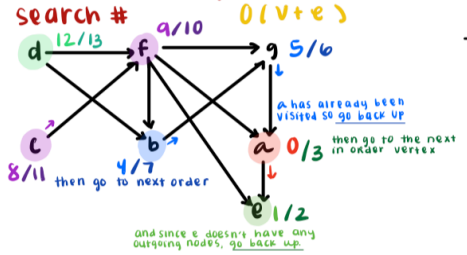
pick any node and then go to the highest order node that is outgoing. Keep track of the levels. $O(n+m)$




| Queue | |
|---|---|
| 0 | d |
| 1 | F |
| 1 | B |
| 2 | g |
| 2 | a |
| 2 | e |
| | C |

# DFS

Start at first vertex and then go to outcoming nodes and update the search # $O(v+e)$




| visited |
|---|
| a |
| e |
| b |
| g |
| c |
| f |
| d |

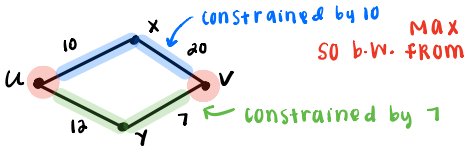
# Topological

Reverse Finish time based on the denominator of DFS $O(v+e)$

d, C, f, b, g, a, e
13, 11, 10, 7, 6, 3, 2

# Bandwidth

measure how much data you can pass through. Pick the min from point u to v, then the max of those mins.




MAX (10, 7) = 10
SO b.w. FROM u to v = 10

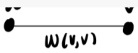
```
function Dijkstra(Graph, source):
    for each vertex v in Graph:                 // Initialization
        dist[v] := infinity                     // initial distance from source to vertex v is set to infinite
        previous[v] := undefined                // Previous node in optimal path from source
    dist[source] := 0                           // Distance from source to source
    Q := the set of all nodes in Graph          // all nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:                       // main loop
        u := node in Q with smallest dist[ ]
        remove u from Q
        for each neighbor v of u:               // where v has not yet been removed from Q.
            alt := dist[u] + dist_between(u, v)
            if alt < dist[v]                    // Relax (u,v)
                dist[v] := alt
                previous[v] := u
    return previous[ ]
```

Floyd-Warshall

```
for i = 1 to N
   for j = 1 to N
      if there is an edge from i to j
         dist[0][i][j] = the length of the edge from i to j
      else
         dist[0][i][j] = INFINITY

for k = 1 to N
   for i = 1 to N
      for j = 1 to N
         dist[k][i][j] = min(dist[k-1][i][j], dist[k-1][i][k] + dist[k-1][k][j])
```

```
KRUSKAL(G):
A = ∅
For each vertex v ∈ G.V:
     MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
     if FIND-SET(u) ≠ FIND-SET(v):
     A = A ∪ {(u, v)}
     UNION(u, v)
return A
```




**SHORTEST:**

Relax (u,v):
  if ( v.d > u.d + w(u,v):
    v.d = u.d + w(u,v)  ← shortest path dist.
    v.numpaths = u.numpaths
  else if (v.d == u.d + w(u,v)
    v.numpaths = v.numpaths + u.numpaths
    ↑ this is the # of paths that are the shortest distance

**LONGEST:**

Relax (u,v):
  if ( v.d < u.d + w(u,v):
    v.d = u.d + w(u,v)
    v.numpaths = u.numpaths
  else if (v.d == u.d + w(u,v)
    v.numpaths = v.numpaths + u.numpaths

```
Prim(G, w, s)
//Input: undirected connected weighted graph G = (V,E) in adj list representation,
        source vertex s in V
//Output: p[1..|V|], representing the set of edges composing an MST of G
01 for each v in V
02    color(v) <- WHITE
03    key(v) <- infinity
04    p(v) <- NIL
05 Q <- empty list   // Q keyed by key[v]
06 color(s) <- GRAY
07 Insert(Q, s)
08 key(s) <- 0
09 while Q != empty
10    u <- Extract-Min(Q)
11    for v in Adj[u]
12       if color(v) = WHITE
13          then color(v) <- GRAY
14               Insert(Q,v)
15               key(v) <- w(u,v)
16               p(v) <- u
17       elseif color(v) = GRAY
18          then if key(v) > w(u,v)
19                  then key(v) <- w(u,v)
20                       p(v) <- u
21    color(v) <- BLACK
22 return(p)
```

```
:cal_sort(N, adj[N][N])
 T = []
 visited = []
 in_degree = []
 for i = 0 to N
         in_degree[i] = visited[i] = 0

 for i = 0 to N
         for j = 0 to N
                 if adj[i][j] is TRUE
                         in_degree[j] = in_degree[j] + 1

 for i = 0 to N
         if in_degree[i] is 0
                 enqueue(Queue, i)
                 visited[i] = TRUE

 while Queue is not Empty
         vertex = get_front(Queue)
         dequeue(Queue)
         T.append(vertex)
         for j = 0 to N
                 if adj[vertex][j] is TRUE and visited[j] is FALSE
                         in_degree[j] = in_degree[j] - 1
                         if in_degree[j] is 0
                                 enqueue(Queue, j)
                                 visited[j] = TRUE
 return T
```

```
DFS-iterative (G, s):                                   //Where G is graph and s is source vertex
  let S be stack
  S.push( s )            //Inserting s in stack
  mark s as visited.
  while ( S is not empty):
      //Pop a vertex from stack to visit next
      v  =  S.top( )
      S.pop( )
      //Push all the neighbours of v in stack that are not visited
      for all neighbours w of v in Graph G:
          if w is not visited :
                  S.push( w )
                  mark w as visited
```

```
BFS (G, s)                      //Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

      mark s as visited.
      while ( Q is not empty)
           //Removing that vertex from queue,whose neighbour will be visited now
           v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                         Q.enqueue( w )              //Stores w in Q to further visit its neighbour
                         mark w as visited.
```

**Class 1)** having low/high stress job. High stress must have week prior = no work.

Subproblem: J(i) = value of best job choices ending in week$_i$ at finishing either H/L stress

Recurrance:

$$J(i) = \begin{cases} 0 & i=0 \\ \max(h_1, l_1) & i=1 \\ \max[l_i + J(i-1), h_i + J(i-2)] & i>1 \end{cases}$$

Now J(i) = best job choice

**Class 2)** max continuous sum

Subprocess: MCS(i) = value of max cont. subseq. sum ending at position i

Recurrance:

$$MCS(i) = \begin{cases} 0 & i=0 \\ a_1 & i=1 \\ \max[a_i, a_i + MCS(i-1)] & i>1 \end{cases}$$

start new seq ↑ attach to tail of prev. best

**HW)** Switch cities (C & D) based on which is cheaper FOR that month.

moving-cost (M) = 10  C = (1, 3, 20, 30)  D = (50, 20, 2, 4)

Subproblem: $[(c_1, d_1) \ldots (c_i, d_i)]$ array for the cost of living in and C(i) = min cost ending in city C and D(i) = min cost ending in city D

Recurrance:

$$C(i) = \begin{cases} c_1 & i=1 \\ \min[C(i-1), D(i-1) + M] + c_i & i>1 \end{cases}$$

$$D(i) = \begin{cases} d_1 & i=1 \\ \min[D(i-1), C(i-1) + M] + d_i & i>1 \end{cases}$$

SO: cost of best plan = min[C(n), D(n)]

**HW)** Place mile post 100m apart if it is not exactly 100m apart → penalty of $(100 - x)^2$ So avg post spots to minimize penalties:

Subproblem: S(i) = min possible total penalty when placing sign at $m_i$ all the way up to $m_i$ answer = S(n)

Recurrance:

$$S(i) = \begin{cases} 0 & i=1 \\ \min_{1 \le j < i} \{S(j) + 100 - (m_i - m_j))^2\} & i>1 \end{cases}$$

Now S(i) = total possible penalty btwn $m_1 \to m_n$

**Class 3)** given 3 strings. Obtain Str. T from reading Str. R and Str. S

Recurrance: "interleaving"

$$IL(i,j) = \begin{cases} \text{true} & i=0 \ \&\ j=0 \\ [t_{i+j} = r_i \text{ and } IL(i-1, j)] \text{ or } [t_{i+j} = s_j \text{ and } IL(i, j-1)] & i>0 \text{ and } j>0 \end{cases}$$

Returns T or F

**Midterm)**

$$C(i) = \begin{cases} 0 & \text{if } i = 0 \\ (m_i - m_{i-1}) \cdot s_{i-1} + C(i-1) & \text{if } i = 1, 2 \text{ (can only use company S)} \\ \min[(m_i - m_{i-1}) \cdot s_{i-1} + C(i-1), 3 \cdot t + C(i-3)] & \text{if } i \ge 3 \text{ (min of choices (a) and (b) above)} \end{cases}$$

Recurrance:

$$C(i) = \begin{cases} 0 & i=0 \\ (m_i - m_{i-1}) \cdot s_{i-1} + C(i-1) & i=1,2 \\ \min[(m_i - m_{i-1}) \cdot s_{i-1} + C(i-1), 3 \cdot t + C(i-3)] & i \ge 3 \end{cases}$$

1. In a weighted graph with start node $s$, there are often multiple shortest paths from $s$ to any other node. We want to use Dijkstra's algorithm to count them (so assume no negative edge weights). To each node $v$, we add a field *v.numPaths* initialized to 0 (with *s.numPaths* initialized to 1). Modify the RELAX routine so that Dijkstra's algorithm will determine both the length of the shortest path to all other nodes *and* the number of such shortest paths.

   *(sol'n:)*

```
procedure relax(u,v)
     if ( v.d > u.d + W(u,v) )
      then  v.d = u.d + W(u,v)
            v.numPaths = u.numPaths
 else if (v.d == u.d + W(u,v) )
            then v.numPaths = v.numPaths + u.numPaths
```