Articulations

- Definition:
  - A node v of a connected graph G is an **articulation** node (also called a cut vertex) if the removal of v and all its incident edges causes G to become disconnected.

- Motivation for articulations:
  - Articulations are important in communication networks.
  - In traffic flows they identify places that will stop traffic between two areas of a city if they become blocked.

Finding Articulations (1)

- Problem:
  - Given any graph $G = (V, E)$, find all the articulation points.

  - Possible strategy:
    - For all vertices v in V:

      Remove v and its incident edges.

      Test connectivity using a DFS.

    - Execution time: $\Theta(n(n + m))$.
      - Can we do better?

Finding Articulations (2)

- A DFS tree can be used to discover articulation points in $\Theta(n + m)$ time.
  - We start with a program that computes a DFS tree labeling the vertices with their discovery times.
  - We also compute a function called $low[v]$ that can be used to characterize each vertex as an articulation or non-articulation point.
    - The root of the DFS tree (the root has a $d[\ ]$ value of 1) will be treated as a special case:

Finding Articulations (3)

- The root of the DFS tree is an articulation point if and only if it has two children.
  - Suppose the root has two or more children.
    - Recall that the back edges never link the vertices in two different subtrees.
    - So, the subtrees are only linked through the root vertex and if it is removed we will get two or more connected components (i.e. the root is an articulation point).
  - Suppose the root is an articulation point.
    - This means that its removal would produce two or more connected components each previously connected to this root vertex.
    - So, the root has two or more children.

Finding Articulations (4)

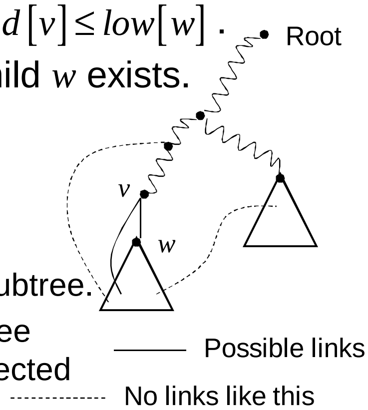
- Computation of $low[v]$.
  - We need another function defined on vertices:
  - This quantity will be used in our articulation finding algorithm:

$low[v]$ = min{$d[v]$, $d[z]$ such that
(u, z) is a back edge for some descendent u of v}

  - So, $low[v]$ is the discovery time of the vertex closest to the root and reachable from v by following zero or more edges downward, and then at most one back edge.

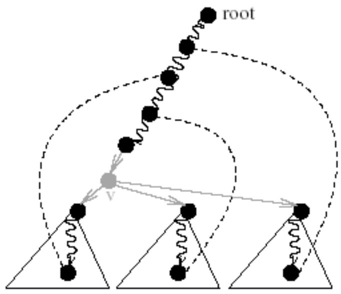
Finding Articulations (5)

- For non-root vertices we have a different test.
  - Suppose v is a non-root vertex of the DFS tree T. Then v is an articulation point of G if and only if there is a child w of v in T with $d[v] \le low[w]$.
  - Sufficiency: Assume such a child w exists.
    - There is no descendent vertex of v that has a back edge going "above" vertex v.
    - Also, there is no cross link from a descendent of v to any other subtree.
    - So, when v is removed the subtree with w as its root will be disconnected from the rest of the graph.



Finding Articulations (6)

- Necessity: Assume no such child w exists.
  - In this case all children of v have a descendent with a back edge going to an ancestor of v.
  - When v is removed each of the children of v will still be connected to some vertex on the path going from the root to the vertex v.
  - The graph stays connected and so v would not be an articulation point in this case.



Finding Articulation Points Pseudocode

```
function dfs-visit(v)
  status[v] := gray;  time := time+1;  d[v] := time;
  low[v] := d[v];
  for each w in out(v)
    if status[w] == white
      // In this case (v,w) is a TREE edge
      dfs-visit(w);        // NOTE:   low[w] is now computed!
      if  d[v] <= low[w] then                              } Definition of low[ ]
           record that vertex v is an articulation         }
      low[v] := min(low[v], low[w]) // Note how low[ ] can propagate up to a parent.
    else if w is not the parent of v then
      // In this case (v,w) is a BACK edge
      low[v] := min(low[v], d[w])
  status[v] := black;
```

Minimum Spanning Trees

- Problem:
  - Given a connected undirected weighted graph $G = (V, E)$, find a minimum spanning tree T for G.
- Assumptions
  - Weights are nonnegative.
  - The cost of a spanning tree is the sum of all the weights of all the edges in T.
  - The Minimum Spanning Tree (MST) is the spanning tree with the smallest possible cost.
- Typical application: Connect nodes in a computer network using as little wire as possible (MST links).

Kruskal's Algorithm

```
// Sort edges in order of increasing weight
// so that w[f[1]] <= w[f[2]] <= ... <= w[f[m]]

T := empty set;
for i:=1 to m do
   let u,v be the endpoints of edge f[i]
   if there is no path between u and v in T then
      add f[i] to T
return T
```

Correctness of Kruskal's Algorithm

- Kruskal's algorithm produces a MST:
  - Kruskal's greedy algorithm produces a tree T_G. Let edges be $e_1, e_2, \ldots, e_{n-1}$ sorted by weight.
  - Then for any $0 \leq k \leq n - 1$ there exists a minimum spanning tree that contains edges $e_1, e_2, \ldots, e_k$.
- Proof by induction:
  - Base case:
    - For $k = 0$ the lemma holds trivially.
  - Induction step:

Correctness of Kruskal's Algorithm

- Suppose there is a MST T^* with edges: $e_1, e_2, \ldots, e_{k-1}$.
- Case 1: $e_k \in T^*$:
  - Then T^* contains all the edges $e_1, e_2, \ldots, e_k$ and the statement is true.
- Case 2: $e_k \notin T^*$:
  - If we remove e_k from T_G, then T_G becomes disconnected and will have two components (call them A and B).
  - Add e_k to T^*. This creates a cycle in T^* involving vertices in both A and B.
  - So, the cycle must contain an edge e' different from e_k that has one endpoint in A and one in B.
  - Remove edge e', to obtain a new graph T' ==> T' is a spanning tree.

Correctness of Kruskal's Algorithm

- Note that $w(e') \geq w(e_k)$, otherwise e' would have been chosen by Kruskal's algorithm instead of e_k.
- The cost of T' can be written as:

  $w(T') = w(T^*) + w(e_k) - w(e')$ implying

  $w(T') \leq w(T^*)$.
- Since T^* is a MST, $w(T') = w(T^*)$ and T' is also a MST.
- Moreover, T' contains each of the edges $e_1, e_2, \ldots, e_k$ which is what we wanted to prove.

- Thus, we have proved by induction that for every k there exists a MST that contains each of the edges $e_1, e_2, \ldots, e_k$.

Analysis of Kruskal's Algorithm

- Running time:
  - Sorting the edges takes $\Theta(m \log m) = \Theta(m \log n)$ time.
  - Running time for the rest of algorithm depends on implementation of the path detection statement: `"if there is no path between u and v in T"`
  - Use DFS on the edges of T selected so far:
    - There are less than n of them, so it will take $O(n)$ per check.
    - This implies a final running time that is $O(mn)$.
  - Use a Union/Find data structure (covered in CS466):
    - The check would take $O(\log n)$ (or better) for each check.
    - This implies a final running time that is $O(m \log n)$.

Prim's Algorithm

- Main idea:
  - Start from an arbitrary single vertex s and gradually "grow" a tree.
    - We maintain a set of connected vertices S.

```
S := {s};
T := empty set;
while S <> V do
   e := (u,v) such that u is in S, v is not
        in S and w(e) is smallest possible;
   add v to S;
   add e to T;
return T;
```

Correctness of Prim's Algorithm

- Prim's algorithm produces a MST:
  - Let Prim's greedy algorithm produce a tree T_G containing edges: $e_1, e_2, \ldots, e_{n-1}$ (numbered in the order they were added by the algorithm).
  - Then for any $0 \leq k \leq n - 1$ there exists a minimum spanning tree that contains edges $e_1, e_2, \ldots, e_k$.
- Proof by induction:
  - Base case:
    - For $k = 0$ the lemma holds trivially.
  - Induction step:

Correctness of Prim's Algorithm

- Suppose there is a MST T^* with edges: $e_1, e_2, \ldots, e_{k-1}$.
- Case 1: $e_k \in T^*$:
  - Then T^* contains all the edges $e_1, e_2, \ldots, e_k$ and the statement is true.
- Case 2: $e_k \notin T^*$:
  - Let S be the set of finished vertices after $k - 1$ steps of the algorithm.
  - Add e_k to T^*. This will create a cycle in T^*.
  - The cycle must contain an edge e' different from e_k that has one endpoint in S and one not in S.
  - Remove edge e' and denote the new graph by T' .
  - T' is a spanning tree.

Correctness of Prim's Algorithm

- Note that $w(e') \geq w(e_k)$, otherwise e' would have been chosen by Prim's algorithm instead of e_k.
- The cost of T' can be written as:

  $w(T') = w(T^*) + w(e_k) - w(e')$ implying

  $w(T') \leq w(T^*)$.
- Since $T^*$is a MST, $w(T') = w(T^*)$ and T' is also a MST.
- Moreover, T' contains each of the edges $e_1, e_2, \ldots, e_k$ which is what we wanted to prove.

- Thus, we have proved by induction that for every k there exists a MST that contains each of the edges $e_1, e_2, \ldots, e_k$.

Analysis of Prim's Algorithm

- Running time:
  - We can improve the algorithm by keeping for each vertex not in S its least cost neighbour in S.
    - The cost for this neighbour will be stored in $cost[v]$ and the neighbour itself in $other[v]$. (See next page).
  - We do the same set of operations with the cost as in Dijkstra's algorithm:
    (initialize a structure, decrease values m times, select the minimum $n - 1$ times).
  - Therefore we get $O(n^2)$ time when we implement cost with an array, and $O((n + m) \log n)$ when we implement it with a heap.

Pseudocode for Prim's Algorithm

Recall: **cost[x]** is least cost between **x** and "nearest" vertex in **S**.

Note: ***w(u,v)*** is defined for all possible ***u*** and ***v*** in ***V***. When there is **no** edge between ***u*** and ***v*** we have **w = infinity**.

```
S := {s};
T := empty set;
// Initialize data structure
for each u not in S
   cost[u] := w(s,u);
   other[u] := s;
// Main computation
while S<>V do
   v := vertex not in S with the smallest cost[v];
   e := (v, other[v]);
   add v to S;
   add e to T;
   // Update data structure
   for each x not in S and a neighbour of v
      if w(v,x) < cost[x] then
         cost[x] := w(v,x);
         other[x] := v;
return T;
```

Dijkstra's Algorithm

- Objective of Dijkstra's algorithm:
  - Dijkstra's algorithm finds the least cost paths from a source vertex s to **all** the other vertices in the graph.

Dijkstra's Algorithm Setup

- We maintain 2 sets of vertices:
  - The set C of "finished" vertices.
    - We can think of C as the "cloud set".
    - It will start with the source vertex and eventually expand to include all the other vertices.
    - For any vertex in the cloud we will be assured that we know its least cost path to the source.
  - The set Q of vertices that are yet to be processed.
    - They are <u>not</u> in the cloud.

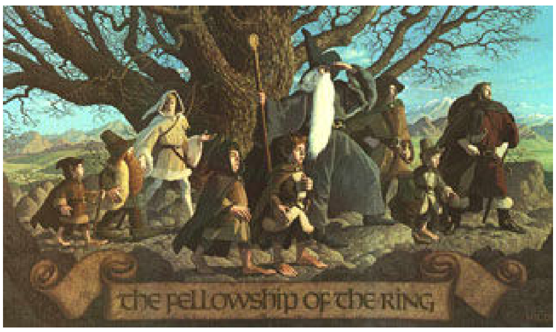
Dijkstra's Algorithm: Pseudocode

```
function least_cost(s)
   // Initialize costs
   C := ϕ; Q := V;
   for all w in V do cost[w] := infinity;
   cost[s] := 0;
   // Iterative steps:
   while Q is non-empty do        //Note: Greedy!
      v := vertex such that cost[v] is a minimum;
      move vertex v from Q to the cloud set C;
      // Update the costs
      for all t in Q that are in out(v) do
         cost_via_v := cost[v] + c[v, t];
         if(cost_via_v < cost[t]) then
            cost[t] := cost_via_v;
```

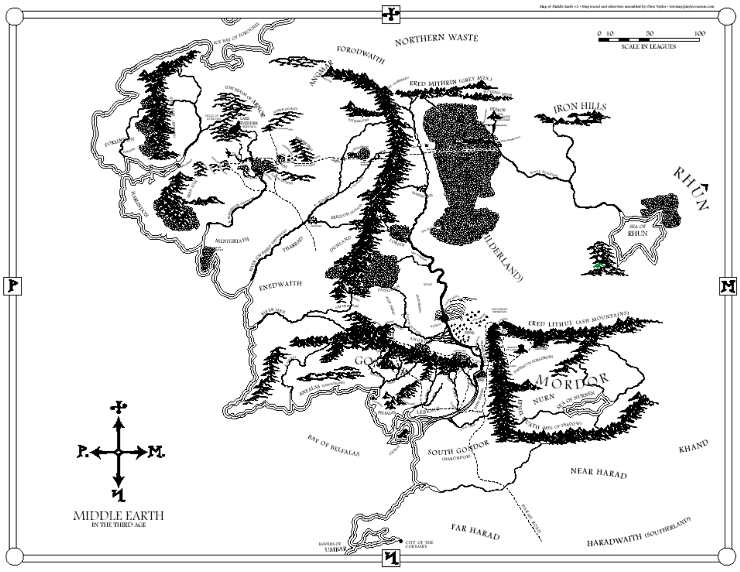
- Path reconstruction:
  - We keep the last but one vertex in the shortest path.

An Example of Dijkstra's Algorithm

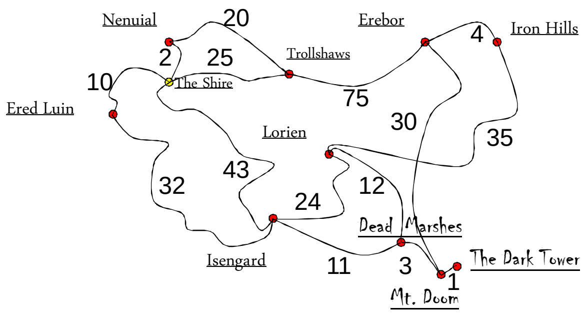
- Suppose small people must minimize travel costs because they walk around with big hairy feet and no shoes….



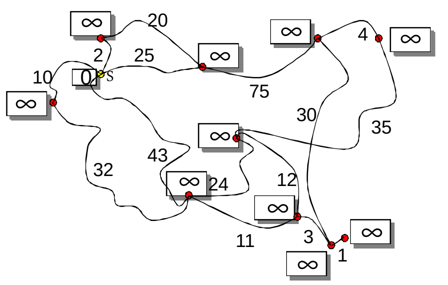
Middle Earth (In the Third Age)

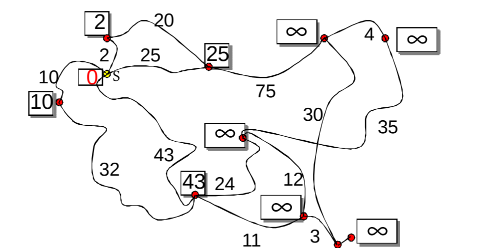


Middle Earth: Travel Cost

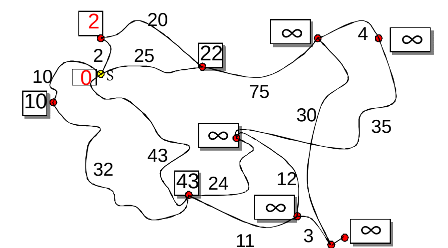


Dijkstra Clouds Middle Earth

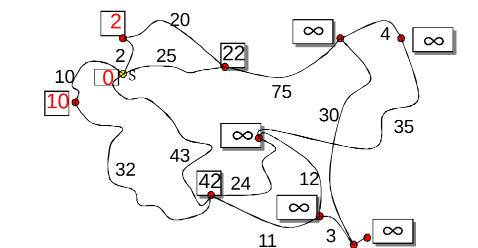




Pull s into the cloud and do relaxation

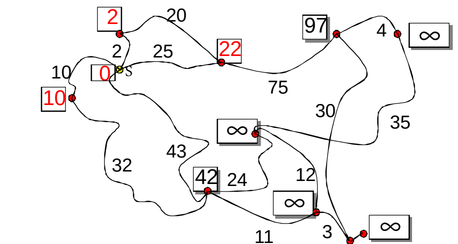


Pull Nenuial into the cloud and do relaxation

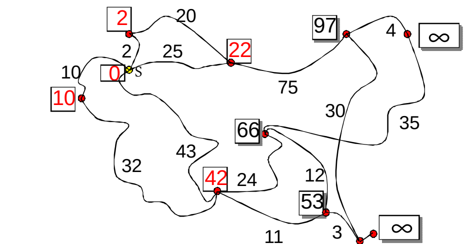


Pull Erid Luin into the cloud and do relaxation

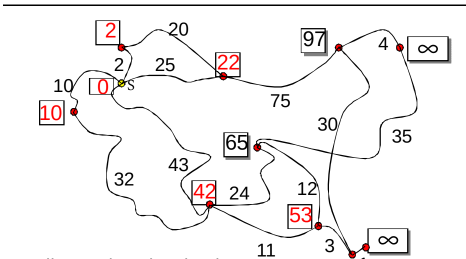
Dijkstra Clouds Middle Earth



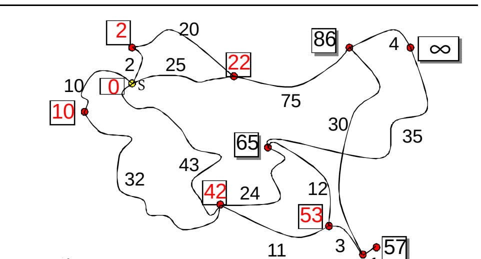
Pull Trollshaws into the cloud and do relaxation



Pull Isengard into the cloud and do relaxation

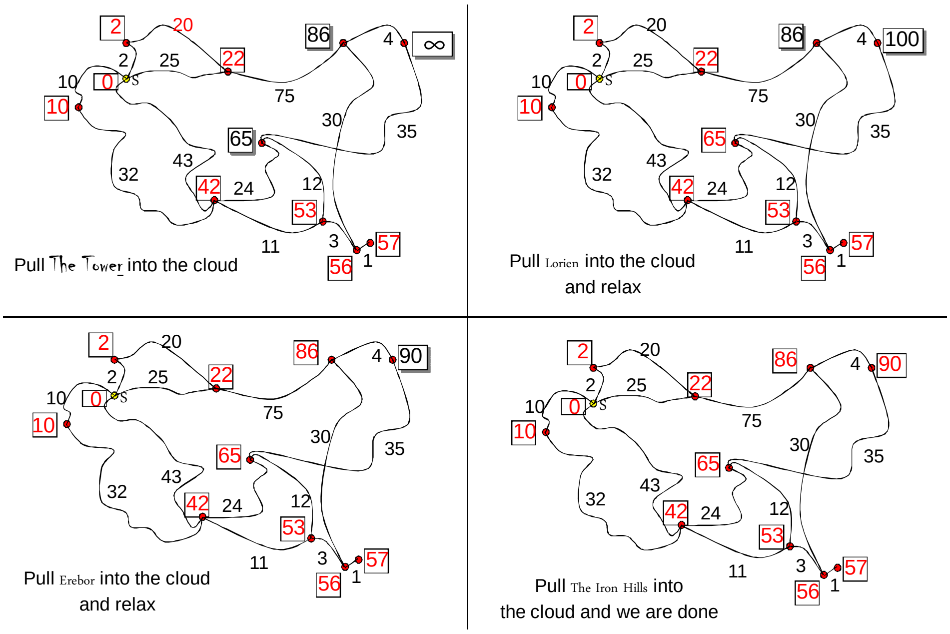


Pull Marshes into the cloud and do relaxation



Pull Mt. Doom into the cloud and do relaxation

Dijkstra Clouds Middle Earth



Running Time of Dijkstra’s Algorithm

- Running time will depend on the implementation of the data structure for $cost[s]$.

The Floyd-Warshall Algorithm

- Problem:
  - Given a graph $G = (V, E)$, directed or undirected, weighted with edge costs, find the least cost path from u to v for all pairs of vertices (u, v).
  - We assume all weights are **non-negative numbers**.
  - The cost of a path will be the sum of the costs of all edges in the path.

Floyd-Warshall: a Useful Lemma

- Lemma:
  - Let P be the least cost path from u to v.
  - Consider any two vertices x and y on this path.
  - The part of the path between vertices x and y will be the least cost path between x and y.

  Proof:
  - If there was a subpath from x to y that was not the least cost path from x to y, then we could replace this subpath with the least cost path from x to y, obtaining a lesser cost for the overall path.
  - This contradicts our statement that the path from u to v was the shortest path, so the lemma is true.

Floyd-Warshall: Extending the Cost Function

- The previous lemma suggest the possibility of using a dynamic programming strategy for our problem.
- A useful way to look at the problem:
  - It is convenient to think of the problem as having a cost $c(u, v)$ assigned to each of the pairs for **all** possible pairs u and v in the graph.
  - $c(u, v)$ = the given edge cost if edge (u, v) exists.
  - $c(u, v)$ = infinity if there is no edge (u, v) in the graph.

Floyd-Warshall: Extending the Cost Function

- With the extended definition of cost, we can go from u to v using any subset of distinct vertices (apart from u and v) as intermediate nodes in the path.
  - Of course, if the selected path uses a non-existent edge in G, the cost of the path is infinity.
  - The algorithm will discard paths with infinite cost and so we will get solutions made up from the given edges.
  - So, the algorithm will examine all possible paths without the need to check beforehand if edges actually exist in G.

Floyd-Warshall: Subproblem Definition

- Subproblem setup:
  - We assume the vertices are labeled (i.e. indexed) using integers ranging from 1 to n.
  - An adjacency matrix representation of the graph is convenient.
- Subproblem definition:
  - We let $cost[i, j, k]$ hold the cost of the least cost path between vertex i and vertex j with intermediate nodes chosen from vertices $1, 2, \ldots, k$.

Floyd-Warshall: Subproblem Definition

- Recall our subproblem definition:
  - We let $cost[i, j, k]$ hold the cost of the least cost path between vertex i and vertex j with intermediate nodes chosen from vertices $1, 2, \ldots, k$.
  - As the index k increases we have more options for discovering the shortest path between endpoints i and j.
    - Even if there is an edge from i to j, its cost might exceed that of another path running from i to j.
  - So, the least cost for the path from i to j will be $cost[i, j, n]$, that is, we have the option of selecting from **all** the other nodes different from i and j.
  - Base case: $cost[i, j, 0] = c(i, j)$. ← (Given edge costs)
    - $cost[i, j, 0]$ is for the path with **no** intermediate nodes.

Floyd-Warshall: The Recurrence

- How do we evaluate $cost[i, j, k]$?
  - Our strategy will be to evaluate all $cost[\ ]$ values starting with $k = 1$, then $k = 2$, etc.
    - Recall that the least cost path for $cost[i, j, k]$ can involve any intermediate nodes selected from $\{1, 2, \ldots, k\}$.
  - In particular, the least cost path may involve node k or it may not…
  - Case 1: The least cost path does ***not*** go through node k, then $cost[i, j, k] = cost[i, j, k\text{-}1]$.
  - Case 2: The least cost path ***does*** go through node k, then $cost[i, j, k] = cost[i, k, k\text{-}1] + cost[k, j, k\text{-}1]$.

Floyd-Warshall: The Recurrence

- Of course, we want to use the case that gives us the smaller cost:

$$\boxed{cost[i, j, k] = \min\{cost[i, j, k\text{-}1],\ \ cost[i,k,k\text{-}1] + cost[k,j,k\text{-}1]\}}$$

Some improvements:

- The value of $cost[i, j, k]$ is always dependent on the immediately previous cost values corresponding to the third parameter equal to $k\text{-}1$ (i.e. <u>not</u> dependent on $k\text{-}2$, $k\text{-}3$, etc.)
- So, we can do away with the third parameter and keep the costs in a two dimensional array that is updated n times.
- Thus, $cost[i, j, k]$ will remain as $cost[i, j, k\text{-}1]$ unless we update it with a smaller $cost[i, k, k\text{-}1] + cost[k, j, k\text{-}1]$ value.

Floyd-Warshall: Pseudocode

```
for i := 1 to n do
   for j := 1 to n do
      cost[i, j] := c[i, j];  // Let c[u, u] := 0
for k := 1 to n do
   for i := 1 to n do
      for j := 1 to n do
         sum = cost[i, k] + cost[k, j];
         if(sum < cost[i, j]) then cost[i, j] := sum;
```

- This code derives the least cost value but there is no recovery of the actual path.
- This is done by remembering the second vertex of the path found so far:

Floyd-Warshall: Pseudocode

```
for i := 1 to n do
   for j := 1 to n do
      cost[i, j] := c[i, j];
      next[i, j] := j;        // Note!
for k := 1 to n do
   for i := 1 to n do
      for j := 1 to n do
         sum := cost[i, k] + cost[k, j];
         if(sum < cost[i, j]) then
            cost[i, j] := sum;
            next[i, j] := next[i, k];    // Note!
// To write out the path from u to v:
w := u;
write w;
while w != v do
   w := next[w, v];
   write w;
```

Note: Running time $\in \Theta(n^3)$.

Formulating Problems as Graph Problems

- As a review we now look at four problems.
  - You should read the problems and as homework try to solve them without looking at the answers in the slides that follow.

Formulating Problems as Graph Problems: Problem #1

- Reliable network routing:
  - Suppose we have a computer network with many links.
  - Every link has an assigned reliability.
    - The reliability is a probability between 0 and 1 that the link will operate correctly.
  - Given nodes u and v, we want to choose a route between nodes u and v with the highest reliability.
    - The reliability of a route is a product of the reliabilities of all its links.

Problem #2

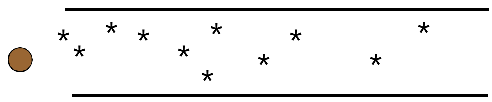
- The Greyhound bus problem:
  - Suppose we are given a bus schedule with information for several buses. A bus is characterized by four attributes:
    - the “from-city”, the “to-city”, departure time, arrival time.
  - Find buses going from city F to city T taking the fastest trip?
    - Take into account travel and wait times between bus arrivals and depatures..
  - First, we eliminate an idea that leads to an inadequate solution:
    - Use a graph that has nodes representing cities.
    - Label each edge with the travel time between cities.
    - Now go for the least cost path.
      - BUT: there is no accounting for wait times!
      - Also, travel times between two cities may vary during the day.
    - But there is another way to use a graph strategy…

Sample Bus Schedule

UW to Hamilton	15:40 17:25		
UW to Toronto	09:00 11:00	17:00 19:00	
Hamilton to Niagara Falls	17:30 18:45		
Toronto to Niagara Falls	12:30 14:05	20:30 22:10	
Niagara Falls to Buffalo	14:10 15:25	18:40 19:40	22:55 23:59

Problems #3

- The RootBear Problem:
  - Suppose we have a narrow canyon with perpendicular walls on either side of a forest.
    - We assume a north wall and a south wall.
  - Viewed from above we see the A&W RootBear attempting to get through the canyon.
    - We assume trees are represented by points.
    - We assume the bear is a circle of given diameter d.
    - We are given a list of coordinates for the trees.
  - Find an algorithm that determines whether the bear can get through the forest.



Solution to Problem #1

- Reliable network routing:
  - Suppose we have a computer network with many links.
  - Every link has an assigned reliability.
    - The reliability is a probability between 0 and 1 that the link will operate correctly.
  - Given nodes u and v, we want to choose a route between nodes u and v with the highest reliability.
    - The reliability of a route is a product of the reliabilities of all its links.

- The route will correspond to a path in the graph.
- Can we make this look like a shortest path problem?
- Yes:
  - Since reliability is computed as a product, we will want to change the weights so that an edge is assigned the logarithm of the probability.
    - Then we sum logs to work with products of probabilities.
  - To get the best reliability path we want the highest probability of operation which we can derive by finding the least weight path if the assigned weights are *negative* logarithms of the probability values.
    - Then we are able to use Dijkstra's algorithm.

Solution to Problem #2

- The Greyhound bus problem:
  - Suppose we are given a bus schedule with information for several buses. A bus is characterized by four attributes:
    - the "from-city", the "to-city", departure time, arrival time.
  - Find buses going from city F to city T with the fastest trip?
    - Take into account travel and wait times between arrival and departure times..
  - First, let's eliminate an idea leading to an inadequate solution:
    - Use a graph that has nodes representing cities.
    - Label each edge with the travel time between cities.
    - Now go for the least cost path.
      - BUT: there is no accounting for wait times!
      - Also, travel times between two cities may vary during the day.
    - But there is another way to use a graph strategy…

Sample Bus Schedule

UW to Hamilton	15:40 17:25		
UW to Toronto	09:00 11:00	17:00 19:00	
Hamilton to Niagara Falls	17:30 18:45		
Toronto to Niagara Falls	12:30 14:05	20:30 22:10	
Niagara Falls to Buffalo	14:10 15:25	18:40 19:40	22:55 23:59

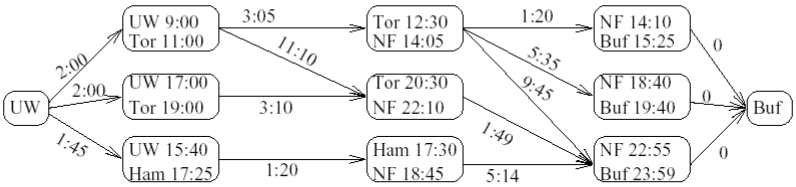
- Another approach:
  - Use a graph in which each vertex is a bus.
  - There will be an edge between busses x and y if and only if:

    $x.to\_city = y.from\_city$ and
    $y.departure\_time \geq x.arrival\_time$.

  - Our time cost for an edge will be:

    $waiting\ time + travel\ time\ on\ bus\ y =$
    $(y.departure\_time - x.arrival\_time) +$
    $(y.arrival\_time - y.departure\_time)$
    $= y.arrival\_time - x.arrival\_time.$

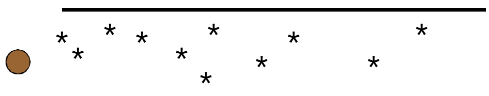
- We need two special vertices for the origin and destination cities.
- There is an edge from origin to bus *x* if and only if *x.from_city = origin*.
- Time cost of this edge is *x.arrival_time* - *x.departure_time*.
- There is an edge from bus *y* to the destination if and only if *y.to_city = destination*.
- The time cost of this edge is 0.
- We now have a shortest path problem:



- Note: the shortest trip is via Toronto with time 6:25 hours.

Solution to Problem #3

- The RootBear Problem:
  - Suppose we have a canyon with perpendicular walls on either side of a forest.
    - We assume a north wall and a south wall.
  - Viewed from above we see the A&W RootBear attempting to get through the canyon.
    - We assume trees are represented by points.
    - We assume the bear is a circle of given diameter *d*.
    - We are given a list of coordinates for the trees.
  - Find an algorithm that determines whether the bear can get through the forest.



- The graph formulation for this problem:
  - Create a vertex for each tree, and a vertex for each canyon wall.
  - Two trees are connected by an edge if and only if the RootBear cannot pass between them.

    - That is if their separation is less than d.
    - Do the same for a tree and its perpendicular distance to a canyon wall.
    - Now determine if canyon walls are in the same connected component of the graph.
    - If they are then the bear cannot pass through the canyon.
    - Otherwise the boundary of the connected component containing the northern canyon wall defines a viable path for the bear.

Conclusion

- Graphs are a very important formalism in computer science.
- Efficient algorithms are available for many important problems:
  - exploration, shortest paths, minimum spanning trees, cut links, colouring, etc.
- If we formulate a problem as a graph problem, chances are that an efficient non-trivial algorithm for solving the problem is already known.
- Some problems have a natural graph formulation.
  - For others we need to choose a less intuitive graph formulation.
- Some problems that do not seem to be graph problems at all can be formulated as such.