MONASH University
Information Technology

20th and 22nd May 2015
Monash University – Sunway Campus, Malaysia

FIT 2004
Algorithms and Data Structures

Tutorial/ Practical 11

Ian Lim
lim.wern.han@monash.edu

Tutorial11 Task01

- Transitive closure of Graph
  - Definition given in your tutorial sheet

Tutorial11 Task01

- Transitive closure of Graph
  - Definition given in your tutorial sheet
- How to get it?
  - Represent graph as a matrix
  - Perform matrix multiplication
    - > Done this before in earlier practicals

Tutorial11 Task01

- Transitive closure of Graph
  - Definition given in your tutorial sheet
- How to get it?
  - Represent graph as a matrix
  - Perform matrix multiplication
  - If you represent the graph as adjacency matrix M, then each entry (u, v) in matrix M^k would mean that it is possible to go from vertex u to vertex v through a path of k edges.

Tutorial11 Task01

- Transitive closure of Graph
- How to get it?
- Alternative: Warshall's algorithm
  - Why is this true
    - > Explanation of correctness
    - > Type of 'proving' question that Arun would love to ask for your exam
  - Relatively simple algorithm here

Tutorial11 Task01

```
Warshall(Graph G(V,W) ) {
 C = E; // Assign C to adjacency matrix of G
 for (vertex k in 1..|V|) {
   //Invariant: C[i,j] iff there is a path from i to j
   //              ...direct or via vertices in {1..k-1}
   for (vertex i in 1..|V|) {
     for (vertex j in 1..|V|) {
        C[i,j] = C[i,j] or (C[i,k] and C[k,j])
     end for
   end for
 end for
```

Tutorial11 Task01

Correctness

- The invariant in the code is the key to the algorithm's correctness.
- At the start of the kth iteration of the outer loop, $C[i,j]$ is true if there is a path from v_i to v_j, possibly via intermediate vertices in the set $\{v_1, \cdots, v_{k-1}\}$. Othewise false.
- This is certainly true initially when no intermediate vertices are allowed.
- Now there is no point in visiting any vertex more than once on any shortest path (unless negative weight cycles).
- So if v_k is to improve on the shortest known path from v_i to v_j then it can only be by going from v_i to v_k, possibly via vertices in $\{v_1, \cdots, v_{k-1}\}$, and then from v_k to v_j, possibly via vertices in $\{v_1, \cdots, v_{k-1}\}$.

Tutorial11 Task02

- Bellmann-Ford algorithm
  - Allows you to find the shortest path
  - Unlike Djikstra, you are able to find the shortest path for graph with negative edge
    - > Detect when there is a negative weight cycle but do not handle it.

Tutorial11 Task02

- Bellmann-Ford algorithm
  - 2 main components
    - > Calculate distance
    - > Check for negative cycles

Tutorial11 Task02

- Bellmann-Ford algorithm
  - Calculate distance
  - Check for negative cycles

Tutorial11 Task02

- Bellmann-Ford algorithm
  - Calculate distance
    - > Loops |V|-1 time. Why?
    - > This is close to brute force.
  - Check for negative cycles

Tutorial11 Task02

- Bellmann-Ford algorithm
  - Calculate distance
    - > Loops |V|-1 time. Why?
      - Because a vertex to itself would have a distance of 0 thus ignored.
      - Maximum traversal of a graph from a vertex u is to go through |V|-1 edges
    - > This is close to brute force.
  - Check for negative cycles

Tutorial11 Task02

- Bellmann-Ford algorithm
  - Calculate distance
    - > Loops |V|-1 time. Why?
    - > This is close to brute force.
      - Also dynamic programming
      - Breaks the distance down to number of hops, calculating minimum distance of each hop using hop-1
  - Check for negative cycles

Tutorial11 Task02

- Bellmann-Ford algorithm
  - Calculate distance
    - > Loops |V|-1 time. Why?
    - > This is close to brute force.
  - Check for negative cycles
    - > If there is a cycle, the additional traversal would reduce in a smaller distance than what have been calculated earlier

Tutorial11 Task02

- Bellmann-Ford algorithm
  - Calculate distance
    - > Loops |V|-1 time. Why?
    - > This is close to brute force.
  - Check for negative cycles
    - > If there is a cycle, the additional traversal would reduce in a smaller distance than what have been calculated earlier
    - > Tutorial question is here

Tutorial11 Task02

- Bellmann-Ford algorithm
  - Calculate distance
    - > Loops |V|-1 time. Why?
    - > This is close to brute force.
  - Check for negative cycles
    - > If there is a netagive cycle, the additional traversal would reduce in a smaller distance than what have been calculated earlier
    - > If this occur, means you can always get a shorter distance by just looping it over and over.

Tutorial11 Task02

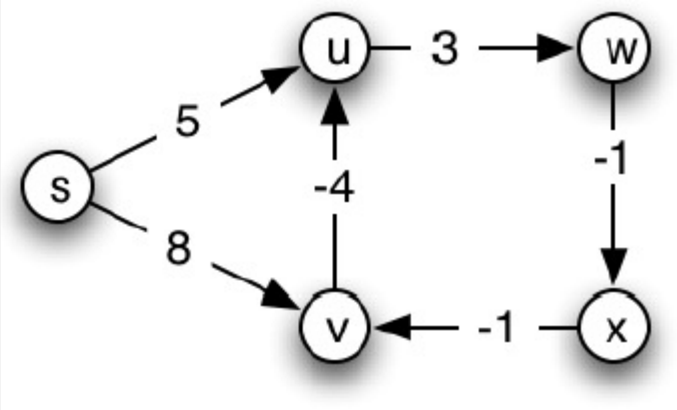
- Bellmann-Ford algorithm
  - Going through the algorithm
    - > Wikipedia have a very good explanation of the algorithm and explanation (including proof). I would recommend going through it to understand it better.

Tutorial11 Task02

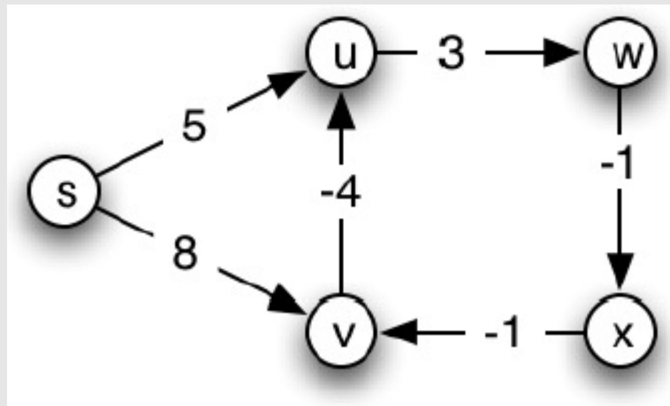
```
function BellmanFord(list vertices, list edges, vertex source)
   ::distance[],predecessor[]

   // This implementation takes in a graph, represented as
   // lists of vertices and edges, and fills two arrays
   // (distance and predecessor) with shortest-path
   // (less cost/distance/metric) information

   // Step 1: initialize graph
   for each vertex v in vertices:
       if v is source then distance[v] := 0
       else distance[v] := inf
       predecessor[v] := null

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge (u, v) with weight w in edges:
           if distance[u] + w < distance[v]:
               distance[v] := distance[u] + w
               predecessor[v] := u

   // Step 3: check for negative-weight cycles
   for each edge (u, v) with weight w in edges:
       if distance[u] + w < distance[v]:
           error "Graph contains a negative-weight cycle"
   return distance[], predecessor[]
```

Tutorial11 Task02

- Bellmann-Ford algorithm
  - Going through the algorithm



Tutorial11 Task02

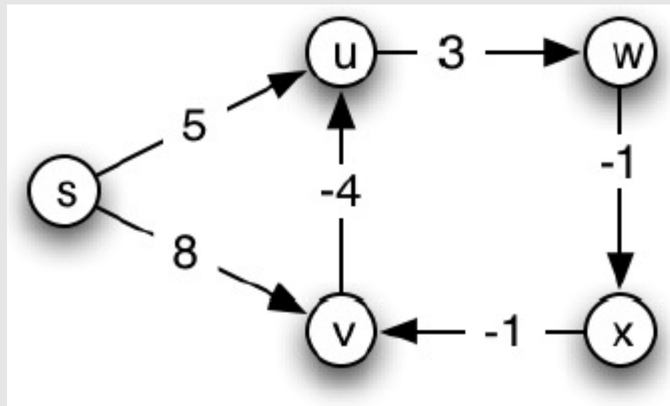


	i=0	i=1	i=2	i=3	i=4
s	0	0	0	0	0
u	inf	5s	4v	4v	4v
v	inf	8s	8s	8s	6x
w	inf	inf	8u	7u	7u
x	inf	inf	inf	7w	6w

Tutorial11 Task02

- Bellmann-Ford algorithm
  - Now we perform the relaxation

Tutorial11 Task02



	i=0	i=1	i=2	i=3	i=4	Checking	
s	0	0	0	0	0	0	
u	inf	5s	4v	4v	4v	2v	Break
v	inf	8s	8s	8s	6x	5x	Break too
w	inf	inf	8u	7u	7u	7u	
x	inf	inf	inf	7w	6w	6w	

Tutorial11 Task02

- Bellmann-Ford algorithm
  - Calculate distance
    - > Loop |V|-1 times outer
    - > Loop |E| times through all edges
    - > Total complexity = |V||E|
  - Check for negative cycles
    - > Extra 1 traversal
  - Total complexity is O(|V||E|)

Tutorial11 Task03

- Tree
  - What is a tree?

Tutorial11 Task03

- Tree
  - What is a tree? Connected undirected graph with no cycles

Tutorial11 Task03

- Tree
  - What is a tree? Connected undirected graph with no cycles
- Spanning Tree
  - What is it?

Tutorial11 Task03

- Tree
  - What is a tree? Connected undirected graph with no cycles
- Spanning Tree
  - What is it?
    - > Tree that include every vertex of graph.
    - > Tree that include maximum edges without creating a cycle
    - > Tree that include minimum edges to connect all vertices
  - Subgraph of the graph

Tutorial11 Task03

- Tree
- Spanning Tree

Tutorial11 Task03

- Remember the following definitions
  - Used for Prim's and Kruskal

Let $G = (V, E)$ be a **connected and undirected** weighted graph.
Definitions:

Cut A **cut** of G is a **partition** on its vertex set V, that separates into two groups C and $V - C$.

Cross An edge $\langle x, y \rangle \in E$ **crosses** the cut $(C, V - C)$ if x is in C, and y is in $V - C$ or vice versa.

Respect A cut **respects** a set M of edges if no edge in M crosses the cut.

Lightest edge An edge is a **lightest edge** crossing a cut if its weight is the minimum of any edge crossing the cut.

Tutorial11 Task03

- Prim’s algorithm
  - Do you want me to go through Prim’s?

Tutorial11 Task03

- Prim’s algorithm
  - Key concept

Tutorial11 Task03

- Prim's algorithm
  - Key concept
    - > Idea of growing a tree (basic idea for MST algorithms)

Tutorial11 Task03

- Prim's algorithm
  - Key concept
    - > Idea of growing a tree (basic idea for MST algorithms)
    - > Start with a vertex (now a tree of 1 vertex)

Tutorial11 Task03

- Prim's algorithm
  - Key concept
    - > Idea of growing a tree (basic idea for MST algorithms)
    - > Start with a vertex (now a tree of 1 vertex)
    - > Grows it by adding the nearest vertex (shortest edge) from that tree (think of Dijkstra)
      - Note: Only for vertices that are not in the tree yet

Tutorial11 Task03

- Prim's algorithm
  - Key concept
    - > Idea of growing a tree (basic idea for MST algorithms)
    - > Start with a vertex (now a tree of 1 vertex)
    - > Grows it by adding the nearest vertex (shortest edge) from that tree (think of Dijkstra)
      - Note: Only for vertices that are not in the tree yet
    - > Repeat till all vertices have been added

Tutorial11 Task03

```
function MST_Prim( G(V,E,W) ){
  /*Initializations*/
  C = random(V);   // Growing vertices of min spanning SUBtree
  M = null;        // Growing edges of min spanning SUBtree
  Q = V-C; // The key for each vertex y in Q is the lightest EDGE
  // ...connecting y to adjacent vertex in C. If no direct edge
  // ...key[y] = infinity
 /* Rest of the algorithm */
  while (Q not_empty) { // loops |V|-1 times
    //INV: Tree/graph made of (C,M) is a (growing) subset/subtree of MST
     y = EXTRACT_MIN(Q); // closest vertex to some x in growing MST.
     x = closest_vertex_in_C_to_y; //can be stored as property of y in Q
     C = C + {y}      // set C grows by addition of vertex y
     M = M + {<x,y>} // set M grows by addition of edge <x,y>

     // update Q
     for (each vertex z in Q adjacent to y ) {
        if ( w(<y,z>) is LESS THAN key[z] in Q ) {
           UPDATE_KEY( z, w(<y,z>) );
        }//end_if
     }//end_for
  }//end_while
  return C,M // C=vertices M=Edges of MST.
}
```

Tutorial11 Task03

- Prim's algorithm
  - Complexity?

Tutorial11 Task03

- Prim’s algorithm
  - Complexity?
    - > Go through |V|-1 vertices for O(V)

Tutorial11 Task03

- Prim’s algorithm
  - Complexity?
    - > Go through |V|-1 vertices for O(V)
    - > Extract minimum vertices (in Q for the algorithm)
      - As a priority queue for quick retrieval of the closest vertex
      - Complexity of O(log V)

Tutorial11 Task03

- Prim's algorithm
  - Complexity?
    - > Go through |V|-1 vertices for O(V)
    - > Extract minimum vertices (in Q for the algorithm)
      - As a priority queue for quick retrieval of the closest vertex
      - Complexity of O(log V)
    - > Update the remaining vertices stored as a priority queue
      - Only for vertices adjacent to the closest which is O(E)
      - Updating the priority queue would be O(log V)

Tutorial11 Task03

- Prim's algorithm
  - Complexity? O(V log V + VE log V) = O(VE log V)
    - > Go through |V|-1 vertices for O(V)
    - > Extract minimum vertices (in Q for the algorithm)
      - As a priority queue for quick retrieval of the closest vertex
      - Complexity of O(log V)
    - > Update the remaining vertices stored as a priority queue
      - Only for vertices adjacent to the closest which is O(E)
      - Updating the priority queue would be O(log V)

Tutorial11 Task03

- Prim's algorithm
  - Complexity?
    > But the two loops (line 9 and line 27) do add up to O(E) thus we replace the VE to E.

    > O(V log V + E log V)

Tutorial11 Task03

- Kruskal's algorithm
  - Minor difference from Prim's
  - Key concept
    - > Graph starts with multiple tree (size of 1 vertex each). Repeat the process now

Tutorial11 Task03

- Kruskal’s algorithm
  - Greedy algorithm
    - > Why?

Tutorial11 Task03

- Kruskal's algorithm
  - Greedy algorithm
    - > Why? Makes decision based on local optima (smallest addition of an edge)
    - > Does it work? See the correctness proof in the lecture notes. We are going through this later anyways.

Tutorial11 Task03

- Kruskal's algorithm
  - Greedy algorithm
    - > Why? Makes decision based on local optima (smallest addition of an edge)
    - > Does it work? See the correctness proof in the lecture notes. We are going through this later anyways.
    - > This also means that Prim's is greedy as well.

Tutorial11 Task03

- Kruskal's algorithm
  - Dynamic programming?
    - > Yes
    - > Breaking down into subtrees
    - > Solving subtrees by adding 1 vertex at a time
    - > So is Prim's

Tutorial11 Task03

- Kruskal’s algorithm
  - Minor difference from Prim’s
  - Key concept
    - > Graph starts with multiple tree (size of 1 vertex each).
    - > Find 2 nearest tree (according to edge<u,v>) and turn them into a forestRepeat the process now

Tutorial11 Task03

- Kruskal's algorithm
  - Minor difference from Prim's
  - Key concept
    - > Graph starts with multiple tree (size of 1 vertex each).
    - > Find 2 nearest tree (according to edge<u,v>) and turn them into a forest
      - Forest are disjoint spanning trees in a graph
      - Note: Ensure that both trees are not from the same forest! This would cause a cycle
    - > Repeat the process now

Tutorial11 Task03

```
/*Initializations*/
for (each vertex x in V) {
    MAKE_DIJOINT_SET(x)
}

SORT_EDGES(E) // ...into an increasing order by weight
M = null      // stores edges of MST
for (each edge <x,y> in E in sorted order) {
 // INV: M is always a minimal spanning (sub-)tree
   if ( FIND_SET(x) NOT EQUALS FIND_SET(Y) ) {
      M = M + {<x,y>}; // add <x,y> edge to set M
      UNION_SETS( FIND_SET(x), FIND_SET(y) );
   }
}//end_for
return M
```

Tutorial11 Task03

- Kruskal's algorithm
  - Complexity?
    > Making disjoint set would be O(V). Yes I am lazy to write |V|

Tutorial11 Task03

- Kruskal's algorithm
  - Complexity?
    - > Making disjoint set would be O(V). Yes I am lazy to write |V|
    - > Sorted of edges would be O(E log E) from any sorting algorithm which you have learnt.

Tutorial11 Task03

- Kruskal's algorithm
  - Complexity?
    - > Making disjoint set would be O(V). Yes I am lazy to write |V|
    - > Sorted of edges would be O(E log E) from any sorting algorithm which you have learnt.
    - > Go through each edge<u,v> for O(E)
      - Check if the vextex u and vertex v is in the same forest. This would take around O(log V) though can be O(V).
      - Perform union of the 2 forest. This would be roughly O(log V) to O(V)

Tutorial11 Task03

- Kruskal's algorithm
  - Complexity?
    - > Making disjoint set would be O(V). Yes I am lazy to write |V|
    - > Sorted of edges would be O(E log E) from any sorting algorithm which you have learnt.
    - > Go through each edge<u,v> for O(E)
      - – Check if the vextex u and vertex v is in the same forest. This would take around O(log V) though can be O(V).
      - – Perform union of the 2 forest. This would be roughly O(log V) to O(V)
    - > O(V + E log E + E log V) = O(E log E)

Tutorial11 Task04

- Kruskal’s algorithm correctness
  - I have gone through the proof by contradiction in the tutorial class. Try to recall what I talk about, it is similar to the general proof in your lecture notes.
  - Generally the outline is as in the following:

Tutorial11 Task04

- Kruskal’s algorithm correctness
  - Consider a minimum spamming tree of T from graph G(U, V). This is the basis for the prove.
  - I shall now demonstrate the entire proof on the whiteboard. Try to remember it.

Tutorial11 Task04

- Kruskal’s algorithm correctness
  - In the following slides, the proofs are the ones similar to your lecture notes.
  - Note how they differ from the one I just gone through on the whiteboard

Tutorial11 Task04

- Kruskal’s algorithm correctness
  - Consider a minimum spamming tree of T from graph G(U, V)

Tutorial11 Task04

- Kruskal’s algorithm correctness
  - Consider a minimum spamming tree of T
  - Assume two forest T1 and T2 are joined with edge e<u,v> to get T

Tutorial11 Task04

- Kruskal’s algorithm correctness
  - Consider a minimum spamming tree of T
  - Assume two forest T1 and T2 are joined with edge e<u,v> to get T
  - If we have the vertices in T1 and T2 connected by another edge e<x,y> to obtain a final minimum spanning tree of T’. Then:

Tutorial11 Task04

- Kruskal's algorithm correctness
  - Consider a minimum spamming tree of T
  - Assume two forest T1 and T2 are joined with edge e<u,v> to get T
  - If we have the vertices in T1 and T2 connected by another edge e<x,y> to obtain a final minimum spanning tree of T'. Then:
    - > As we know e<u,v> if part of the minimum spanning tree T, then we add e<u,v> to T' that would create a cycle.
    - > We break the cycle by removing e<x,y> after that giving us a new T' which consist of T1, T2 and e<u,v>

Tutorial11 Task04

- Kruskal’s algorithm correctness
  - Continue to join subtrees until a single minimum cost tree (spanning all vertices) remains.

Practical11 Task01

- Dijkstra algorithm
  - Implement it
  - Gone through it last tutorial

Practical11 Task01

- Dijkstra algorithm
  - Implement it
  - Gone through it last tutorial
- Goal of this question is to see if it would function for negative edges (more to negative cycles)

Practical11 Task02

- Prim’s algorithm
  - Gone through it earlier already

Practical11 Task02

- Prim's algorithm
  - Gone through it earlier already
  - Would it work with negative edges?

Practical11 Task02

- Prim's algorithm
  - Gone through it earlier already
  - Would it work with negative edges? Yes in fact it would (so is Kruskal's) because it would take the minimum weight edge first over the other edges without caring overall weight (unlike Dijkstra) because it just takes the single minimum weight edge without caring about the overall total weights

Practical11 Task03

- Kruskal’s algorithm
  - Implement this too
  - Union-Find data structure
    - > How to store the subsets (forests)
    - > How to merge the forests
    - > Complexity of the algorithm depends on this

Practical11 Task04

- Problem given in the link
  - https://projecteuler.net/problem=107

Practical11 Task04

- Problem given in the link
  - https://projecteuler.net/problem=107
- How do you solve it?
  - Minimum Spanning Tree (MST) via
    - > Prim’s
    - > Kruskal’s

Practical11 Task04

- Problem given in the link
  - https://projecteuler.net/problem=107
- How do you solve it?
  - Minimum Spanning Tree (MST) via
    - > Prim’s
    - > Kruskal’s
  - You can Google for Project Euler solutions
  - http://www.mathblog.dk/project-euler-107-efficient-network/

MONASH University
Information Technology

20th and 22nd May 2015
Monash University – Sunway Campus, Malaysia

Thank You