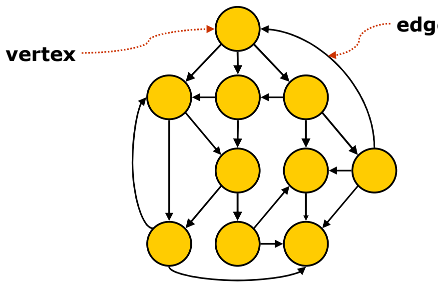
Week 12: Graphs

Readings

- Optional
  - [Weiss] ch20
  - [CLR] ch5.4
- Exercise
  - 20.5

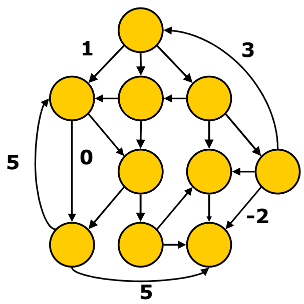
[CLR]: Cormen, Leiserson and Rivest, "*Introduction to Algorithms*" QA76.6 Crm RBR

Graph

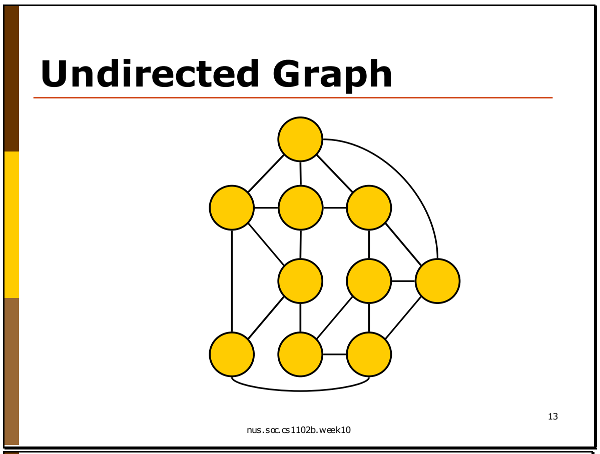


A graph consists of a set of vertices and a set of edges between the vertices. In a tree, there is a unique path between any two nodes. In a graph, there may be more than one path between two nodes.

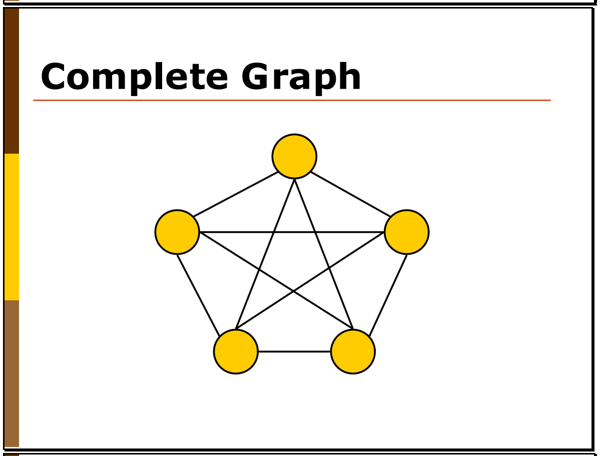
Weighted Graph



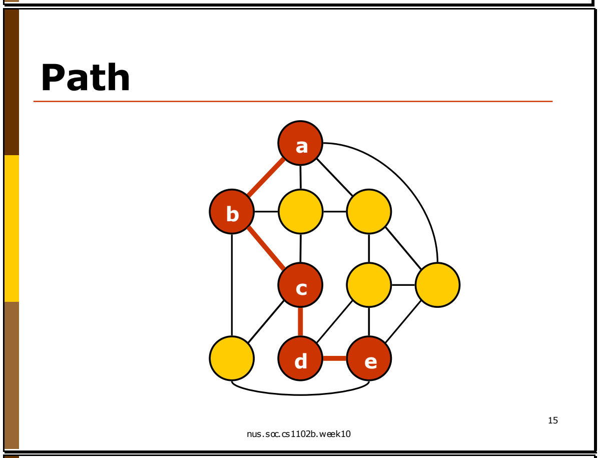
In a weighted graph, edges have a weight (or cost) associated with it. Not all weights are labeled in this slides for simplicity.



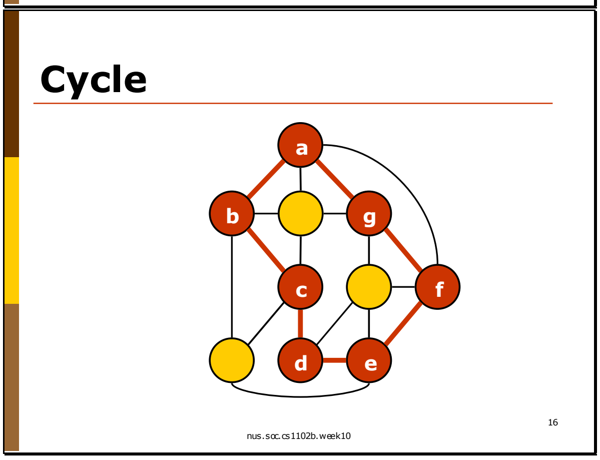
In an undirected graph, edges are bidirectional.



In a complete graph, a node is connected to every other nodes. The number of edges in a complete graph is n(n-1)/2, where n is the number of vertices. (Why is it so?). Therefore, the number of edges is $O(n^2)$.

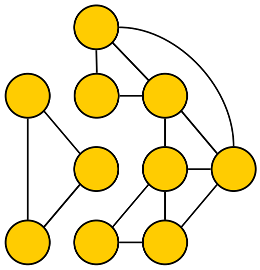


A path is a sequence of vertices $v_0, v_1, v_2, .. v_n$ where there is an edge between v_i and v_{i+1}. The length of a path p is the number of edges in p.



A path $v_0, v_1, v_2, \ldots v_n$ is a cycle if $v_n = v_0$ and its length is at least 1. Note that the definition of path and cycle applies to directed graph as well.

Disconnected Graph



In a connected graph, there is a path between every nodes. A graph does not have to be connected. The above graph has two connected components.

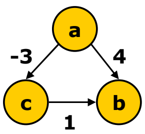
Formally

A graph G = (V, E, w), where

- V is the set of vertices
- E is the set of edges
- w is the weight function

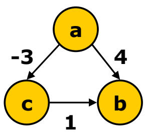
Example

V = { a, b, c }
E = { (a,b), (c,b), (a,c) }
w = { ((a,b), 4), ((c, b), 1), ((a,c),-3) }



Adjacent Vertices

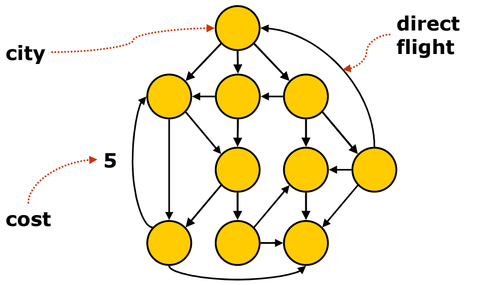
- **adj(v)** = set of vertices adjacent to v
  adj(a) = {b, c}
  adj(b) = {}
  adj(c) = {b}
- $\sum_v |adj(v)| = |E|$
- adj(v): Neighbours of v



Interested students may refer to [CLR] Sec 5.4 for more precise definition of graph terminologies.

Applications

Travel Planning



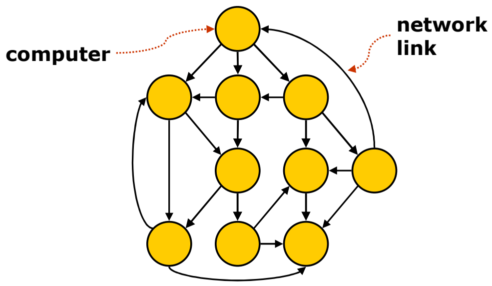
What is the shortest way to travel between A and B?

"SHORTEST PATH PROBLEM"

How to mimimize the cost of visiting n cities such that we visit each city exactly once, and finishing at the city where we start from?

"TRAVELLING SALESMAN PROBLEM"

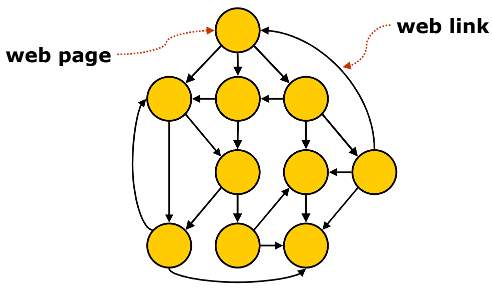
Internet



What is the shortest route to send a packet from A to B?

"Shortest Path Problem"

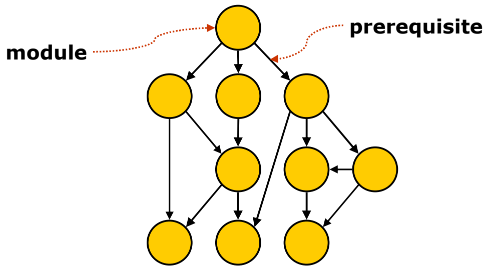
The Web



Which web pages are important?

Which group of web pages are likely to be of the same topic?

Module Selection



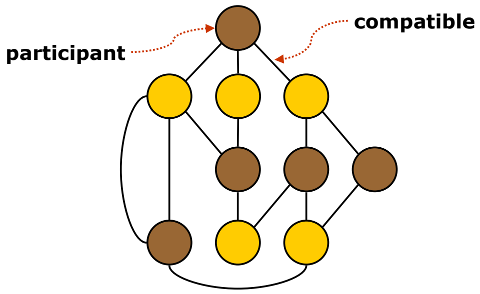
30
nus.soc.cs1102b.week10

Find a sequence of modules to take such that the prerequisite requirements are satisfied.

"Topological Sort"

This is an example of a directed, acyclic graph, or dag for short.

SDU Matchmaking



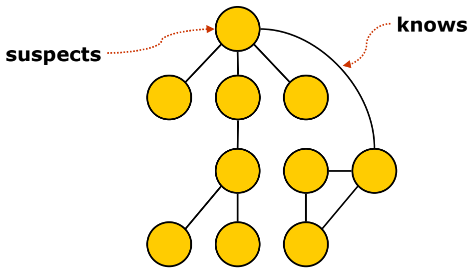
32
nus.soc.cs1102b.week10

How to match up as many pairs as possible?

"Maximum matching problem"

This is an example of a bipartite graph. A bipartite graph is a graph where we can partition the vertices into two sets V and U. No edges exists between two vertices in the same partition.

Terrorist



34
nus.soc.cs1102b.week10

Who are the important figures in a terrorist network?

http://www.orgnet.com/hijackers.html

Other Applications

- Biology
- VLSI Layout
- Vehicle Routing
- Job Scheduling
- Facility Location

:
:

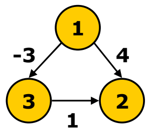
37
nus.soc.cs1102b.week10

Implementation

Adjacency Matrix

double vertex[][];

	1	2	3
1	∞	4	-3
2	∞	∞	∞
3	∞	1	∞

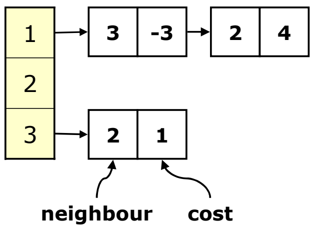


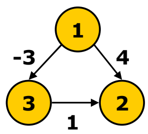
This requires $O(N^2)$ memory, and is not suitable for sparse graph. (Only 1/3 of the matrix in this example contains useful information).

How about undirected graph? How would you represent it?

Adjacency List

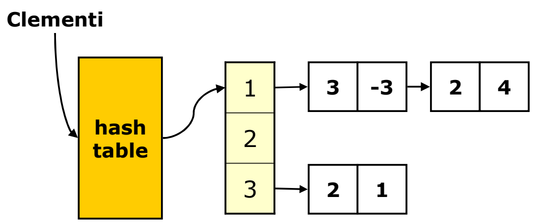
EdgeList vertex[];





This requires only $O(V + E)$ memory.

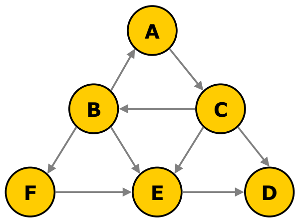
Vertex Map



Since vertices are usually identified by names (person, city), not integers, we can use a hash table to map names to indices in our adjacency list/matrix.

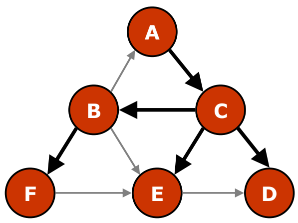
Breadth-First Search

Breadth-First Search

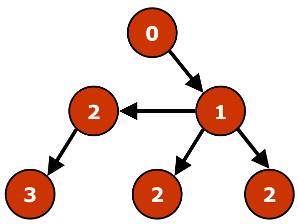


Given a source node, we like to start searching from that source. The idea of BFS is that we visit all nodes that are of distance i away from the source before we visits nodes that are of distance i+1 away from the source. The order of searches is not unique and depends on the order of neighbours visited.

Breadth-First Search

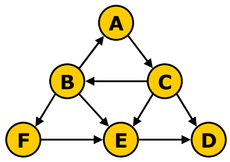


Breadth-First Search



After BFS, we get a tree rooted at the source node. Edges in the tree are edges that we followed during searching. We call this BFS tree. Vertices in the figure are labeled with their distance from the source.

BFS(v)



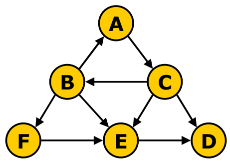

```
Q = new Queue
Q.enq (v)
while Q is not empty
   curr = Q.deq()
   if curr is not visited
        print curr
        mark curr as visited
        foreach w in adj(curr)
                if w is not visited
                        Q.enq(w)
```

53

nus.soc.cs1102b.week10

The pseudocode for BFS is very similar to level-order traversal of trees. The major difference is that, now we may visit a vertex twice (since unlike a tree, there may be more than one path between two vertices). Therefore, we need to remember which vertices we have visited before.

Building the BFS Tree



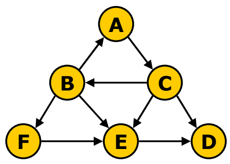

```
Q = new Queue
Q.enq (v)
v.parent = null
while Q is not empty
   curr = Q.deq()
   if curr is not visited
        mark curr as visited
        foreach w in adj(curr)
                if w is not visited
                        w.parent = curr
                        Q.enq(w)
```

54

nus.soc.cs1102b.week10

We can represent the BFS tree by maintaining the parent of a vertex during searching. (This is called "prev" in the textbook)

Calculating Level




```
Q = new Queue
Q.enq (v)
v.level = 0
while Q is not empty
   curr = Q.deq()
   if curr is not visited
        mark curr as visited
        foreach w in adj(curr)
                if w is not visited
                        w.level = curr.level + 1
                        Q.enq(w)
```

55

nus.soc.cs1102b.week10

Similarly, we can maintain the distance of a vertex from the source. (level is equivalent to dist in the textbook)

Search All Vertices

```
Search(G)
   foreach vertex v
        mark v as unvisited
   foreach vertex v
        if v is not visited
            BFS(v)
```

56

nus.soc.cs1102b.week10

BFS guarantees that if there is a path to a vertex v from the source, we can always visit v. But since some vertices maybe unreachable from the source, we can call BFS multiple times from multiple sources.

Running Time

```
Q = new Queue
Q.enq (v)
while Q is not empty
  curr = Q.deq()
  if curr is not visited
   print curr
   mark curr as visited
   foreach w in adj(curr)
     if w is not visited
        Q.enq(w)
```

Main Loop

$$\Theta(\sum_{w \in V} adj(w)) = \Theta(E)$$

Initialization

$$\Theta(V)$$

Total Running Time

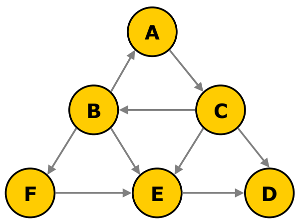
$$\Theta(V + E)$$

Each vertex is enqueued exactly once. The for loop runs through all vertices in the adjacency list. Therefore the running time is $O(\sum_v adj(v)) = O(E)$.

(Note that technically, it should be $O(|E|)$, but we will abuse the notation for E and V to mean the number of edges and vertices as well).

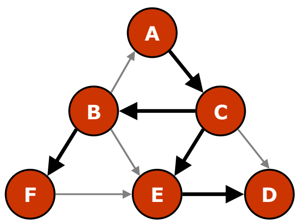
Depth-First Search

Depth-First Search

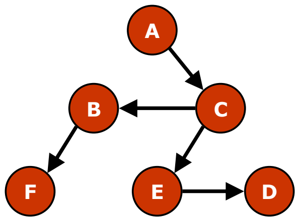


Idea for DFS is to go as deep as possible. Whenever there is an outgoing edge, we follow it.

Depth-First Search

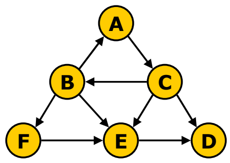


Depth-First Search



DFS(v)

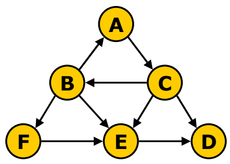
```
S = new Stack
S.push (v)
while S is not empty
   curr = S.pop()
   if curr is not visited
        print curr
        mark curr as visited
        foreach w in adj(v)
             if w is not visited
                S.push(w)
```



In DFS, we use a stack to "remember" where to backtrack to.

Recursive Version: DFS(v)

```
print v
marked v as visited
foreach w in adj(v)
   if w is not visited
     DFS(w)
```



We can write DFS() recursively. (Trace through this code using the example above!)

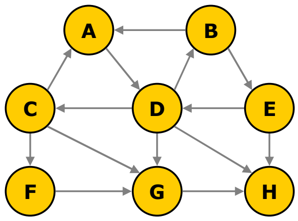
Search All Vertices

```
Search(G)
   foreach vertex v
        mark v as unvisited
   foreach vertex v
        if v is not visited
           DFS(v)
```


Just like BFS, we may want to call DFS() from multiple vertices to make sure that we visit every vertex in the graph.

The running time for DFS is O(V + E). (Why?)

Two more times!



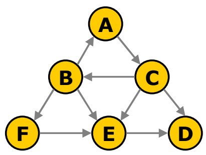
For practice, trace through the above graph using BFS and DFS.

Single-Source Shortest Path

Definition

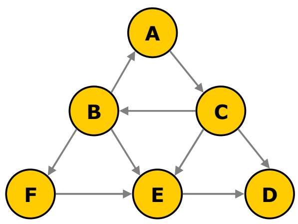
A path on a graph G is a sequence of vertices v_0, v_1, v_2, .. v_n where $(v_i,v_{i+1}) \in E$

The cost of a path is the sum of the cost of all edges in the path.



In Single-source shortest path problem, we are given a vertex v, and we want to find the path with minimum cost to every other vertex. The term "distance" and "length" of the path will be used interchangeably with the "cost" of a path.

Unweighted Shortest Path



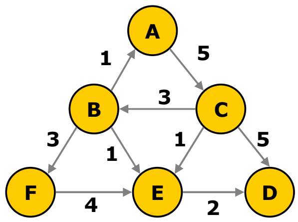
If a graph is unweighted, we can treat the cost of each edge as 1.

ShortestPath(s)

- Run BFS(s)
  - w.level: shortest distance from s
  - w.parent: shortest path from s

The shortest path for an unweighted graph can be found using BFS. To get the shortest path from a source s to a vertex v, we just trace back the parent pointer from v back to s. The number of edges in the path is given by the level of a vertex in the BFS tree. (Why does BFS guarantee that the paths are shortest?)

Positive Weighted Shortest Path



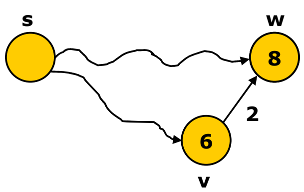
Next, we look at another version of the problem, where the edges have positive cost function.

BFS(s) does not work

- Must keep track of smallest distance so far.
- If we found a new, shorter path, update the distance.

Convince yourself the BFS does not solve our shortest path problem here. Distance here refers to the cost of the path, not the number of edges as in BFS.

Idea 1



In the following figures, we label a node with the shortest distance discovered so far from the source. Here is the basic idea that will help us solve our shortest path problem. If the current shortest distance from s to w is 10, to v is 6, and the cost of edge (v,w) is 2, then we have discovered a shorter path from s to w (through v).

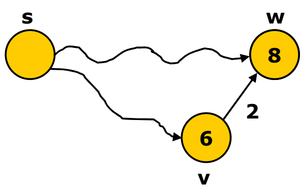
Definition

distance(v) : shortest distance so far from s to v

parent(v) : previous node on the shortest path so far from s to v

cost(u, v) : the cost of edge from u to v

Example



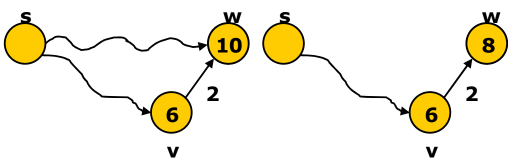
distance(w) = 8

cost(v,w) = 2

parent(w) = v

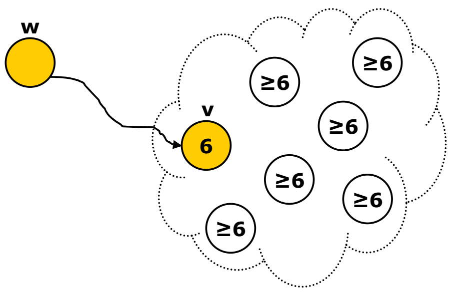
Relax(v,w)

d = distance(v) + cost(v,w)
if distance(w) > d **then**
  distance(w) = d
  parent(w) = v



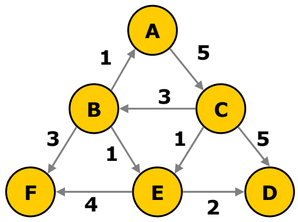
We now look at the pseudocode for a RELAX operation, based on our first idea.

Idea 2

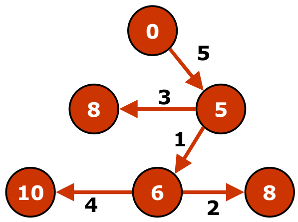


The second idea is that if we know the shortest distance so far from w to v is 6, and the shortest distances so far from w to other nodes are bigger or equal to 6, then there cannot be a shorter path to v through the other white nodes. (This is only true if costs are positive!)

Dijkstra's Algorithm



Dijkstra's Algorithm



Dijkstra's Algorithm

color all vertices yellow
foreach vertex w
  distance(w) = INFINITY
distance(s) = 0

Dijkstra's Algorithm

while there are yellow vertices
  v = yellow vertex with min distance(v)
  color v red
  foreach neighbour w of v
    relax(v,w)

Now we are ready to describe our single source, shortest path algorithm for graphs with positive weights. The algorithm is called Dijkstra's algorithm.

Running Time $O(V^2 + E)$

```
color all vertices yellow
foreach vertex w
  distance(w) = INFINITY
distance(s) = 0
while there are yellow vertices
  v = yellow vertex with min distance(v)
  color v red
  foreach neighbour w of v
     relax(v,w)
```

nus.soc.cs1102b.week10 96

Initialization takes O(V) time. Picking the vertex with minimum distance(v) can take O(V) time, and relaxing the neighbours take O(adj(v)) time. The sum of these over all vertices is $O(V^2+E)$. We can improve this, if we can improve the running time for picking the minimum.

Using Priority Queue

```
foreach vertex w
  distance(w) = INFINITY
distance(s) = 0
pq = new PriorityQueue(V)

while pq is not empty
  v = pq.deleteMin()
  foreach neighbour w of v
     relax(v,w)
```

nus.soc.cs1102b.week10 97

Since priority queue supports efficient minimum picking operation, we can use a priority queue here to improve the running time. Note that we no longer color vertices here. Yellow vertices in the previous pseudocode are now vertices that are in the priority queue.

Initialization O(V)

```
foreach vertex w
  distance(w) = INFINITY
distance(s) = 0
pq = new PriorityQueue(V)
```

nus.soc.cs1102b.week10 98

Initialization still takes O(V)

Main Loop

```
while pq is not empty
  v = pq.deleteMin()
  foreach neighbour w of v
     relax(v,w)
```

nus.soc.cs1102b.week10 99

But we have to be more careful with the analysis of the main loop. We know that each deleteMin() takes O(log V) time. But relax(v,w) is no longer O(1).

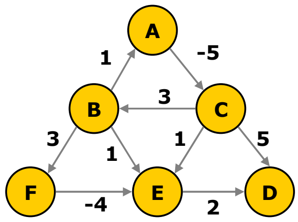
Main Loop O((V+E) log V)

```
while pq is not empty
  v = pq.deleteMin()
  foreach neighbour w of v
     d = distance(v) + cost(v,w)
     if distance(w) > d then
          // distance(w) = d
          pq.decreaseKey(w, d)
          parent(w) = v
```

If we expand the code for relax(), we will see that we cannot simply update distance(v), since distance(v) is a key in pq. Here, we use an operation called decreaseKey() that updates the key value of distance(v) in the priority queue. decreaseKey() can be done in O(log V) time. (How?).

The running time for this new version of Dijkstra's algorithm takes O((V+E)log V) time.

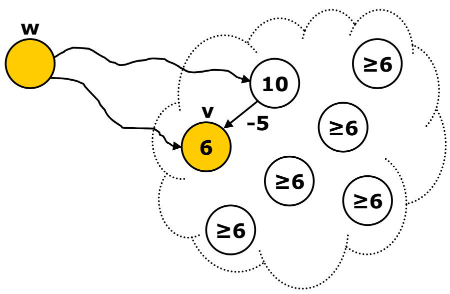
cost(u,v) < 0?



nus.soc.cs1102b.week10 101

Dijkstra's does not work for graphs with negative weights. There are two problems.

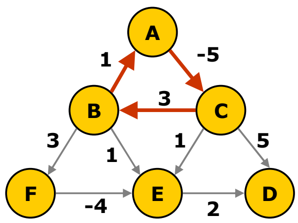
Problem 1



nus.soc.cs1102b.week10 102

Even if we know the shortest path from w to v is 6, there may be a shorter path through the other white nodes as the weight can be negative.

Problem 2



nus.soc.cs1102b.week10 103

If a cycle with negative weights (1 + 3 – 5 = -1) exists in the graph, the shortest path is not well defined, as we can keep going in the negative weighed cycle to get a path with smaller cost.

Basic Idea

foreach edge (u,v)
    relax(u , v)

We will get the shortest paths of length 1 between s and all other vertices.

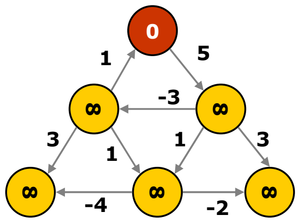
Repeat the above pseudocode |V|-1 times.

nus.soc.cs1102b.week10 105

Here is the idea behind the algorithm for solving the general case shortest path problem.

We repeat |V| - 1 times since a path between two vertices has at most |V| - 1 edges. (Note that we consider only *simple* path, i.e., path with no cycles.)

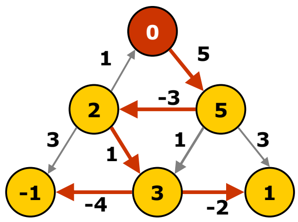
Bellman-Ford Algorithm



nus.soc.cs1102b.week10 106

The algorithm to solve this is called Bellman-Ford Algorithm. Trace through the pseudocode given below, and check your answer against the next slide.

Bellman-Ford Algorithm



nus.soc.cs1102b.week10 110

Bellman-Ford Algorithm

do |V|-1 times
  foreach edge (u,v)
      relax(u,v)

// check for negative weight cycle
foreach edge (u,v)
  if distance(u) < distance(v) + cost(v,u)
      ERROR: has negative cycle

nus.soc.cs1102b.week10 111

I claim that the running time for Bellman-Ford algorithm is O(VE). Verify this claim.