202: Computer Science II

Northern Virginia Community College

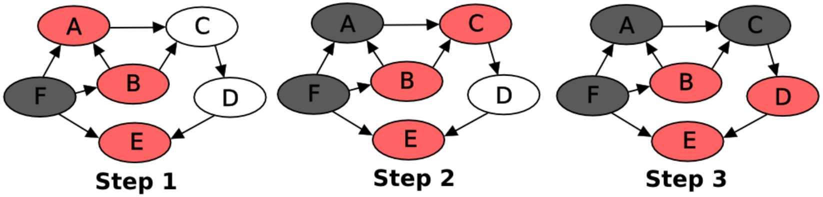
Graphs: Algorithms

Cody Narber, M.S.
November 18, 2017

Depth First Search

Problem: Does there exist a path from vertex v_i to vertex v_j.
Solution 1: Begin looking at an adjacent vertex from v_i, then look at an adjecent vertex to that one and so on as far as you can from the original vertex, backtracking when you have reached a visited vertex or there are no more edges

Depth First Search is when vertices of distance d are visited then a vertex $d + 1$ distance away, and so on until cannot go further, back up and try a different path.



Depth First Search Implementation

By looking at deeper and deeper we are wanting to explore the latest vertex examined. So the last one in should be the first one out. **LIFO = use of stacks!**
First let us define our graph data structure (Let us say we are using a Adjacency Matrix Structure so we only have to define our graph):

Graph<T extends Comparable<T>>
- vertices : ArrayList<T>
- edges : int[][]
- addVertex(datum : T)
- addEdge(ind1 : int, ind2 : int)
- BFS(ind : int, data : T) : boolean
- DFS(ind : int, data : T) : boolean

Depth First Search Implementation

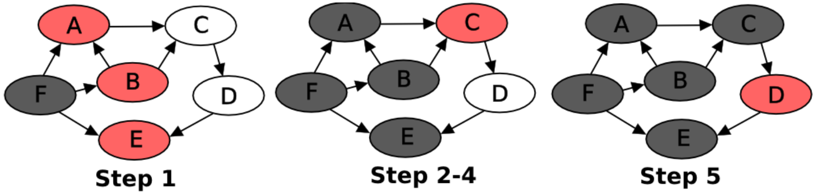
Depth First Search (DFS):

```
public void DFS(int ind, T data){
     //defaults to 0, but just for coverage
     int[] visited = new int[vertices.size()];
     for(int i=0; i<visited.length; i++) visted[i] = 0;

     IntegerStack discoveredVerts = new IntegerStack();
     discoveredVerts.push(ind);

     while( !discoveredVerts.isEmpty() ){
          int cInd = discoveredVerts.pop();

          if (vertices.get(cInd).compareTo(data)==0)
               return true;
          visted[cInd]=1; //visit our ind first
          for(int j = 0; j < vertices.size(); j++)
               if(edges[cInd][j]==1 && visited[j]!=1)
                    discoveredVerts.push(j);
     }
     return false;
}
```

Breadth First Search

Problem: Does there exist a path from vertex v_i to vertex v_j.
Solution 2: Begin looking at all adjacent vertices from v_i, then look at all vertices that have path size 2, then path size 3, then....

Breadth First Search is when all vertices of distance d are visited before looking at any vertices $> d$ distance away. We visit the vertices we discovered first before moving on sounds like a job for **FIFO = Queues!**



Breadth First Search Implementation

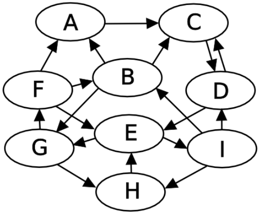
Breadth First Search (BFS):

```
public void BFS(int ind, T data){
      //defaults to 0, but just for coverage
      int[] visited = new int[vertices.size()];
      for(int i=0; i<visited.length; i++) visted[i] = 0;

      IntegerQueue discoveredVerts = new IntegerQueue();
      discoveredVerts.enqueue(ind);

      while( !discoveredVerts.isEmpty() ){
            int cInd = discoveredVerts.dequeue();

            if (vertices.get(cInd).compareTo(data)==0)
                  return true;
            visted[cInd]=1; //visit our ind first
            for(int j = 0; j < vertices.size(); j++)
                  if(edges[cInd][j]==1 && visited[j]!=1)
                        discoveredVerts.enqueue(j);
      }
      return false;
}
```

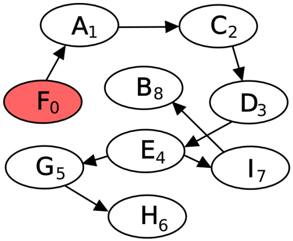
BFS vs DFS

Using these searches: we never go to the same vertex twice...i.e. we only follow one and only one path from each vertex to another. Essentially we are creating a tree when we only look at the paths taken:

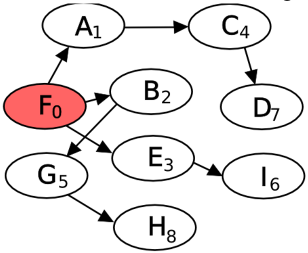
Original Graph



DFS Created Tree - starting at F



BFS Created Tree - starting at F

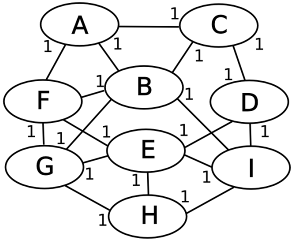


Minimum Spanning Tree

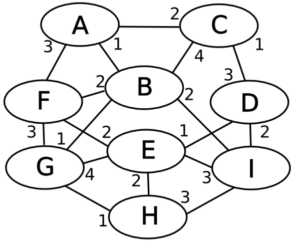
Minimum Spanning Tree - A spanning tree Is a subtree such that the edges of the tree such that every vertex is contanied within the tree. Minimum refers minimizing the weight of edges when added (So far we have assumed all weights to be equal...1 in all previous examples).

When the weights are equal the solution is trivial as the previous trees found by DFS and BFS both are minimum spanning trees (Number of edges equals $|V| - 1$)

Uniform Weights



Differing Weights



Prim's Algorithm

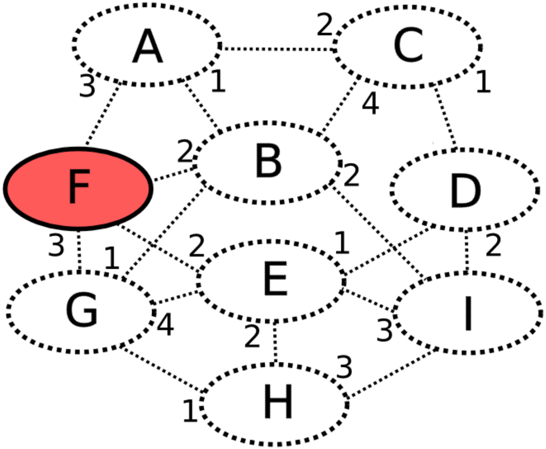
Prim's Algorithm:

1. Initialize a tree with a single vertex, chosen arbitarily from the graph
2. Grow the tree by one edge; of the possible edges that connect the tree to the vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree
3. Repeat until all vertices have been added

This is a **Greedy** algorithm - in that it only chooses the best possible solution given limited knowledge (proof that this works here is beyond this course - greedy algorithms do not always work)
NOTE: Only guarenteed to work with **undirected** edges...to be seen...can work with directed as shown in class, but must meet specific requirements.

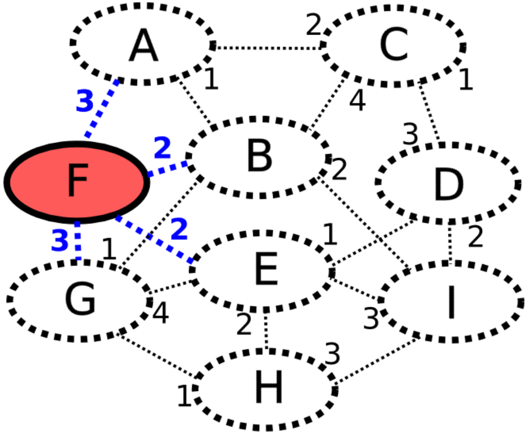
Prim's Algorithm Example - 1

Prim's Algorithm: Initialize a tree with a single vertex, chosen arbitarily from the graph



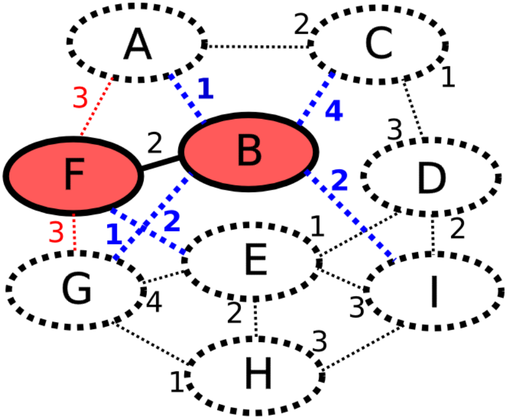
Prim's Algorithm Example - 2

Prim's Algorithm: Examine the costs of the adjecent edges leading to not yet visited nodes (highlighted in Blue).



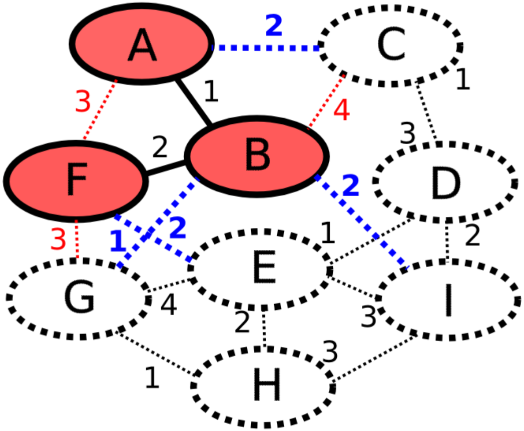
Prim's Algorithm Example - 3

Prim's Algorithm: Follow the smallest edge to visit an unknown vertex (B). Examine the adjacent vertices of B: (A,C,G,I). Consider all edges to built MST. (Red edges are there to highlight when a cheaper or equal edge exists to a vertex, or an edge between to visited nodes)



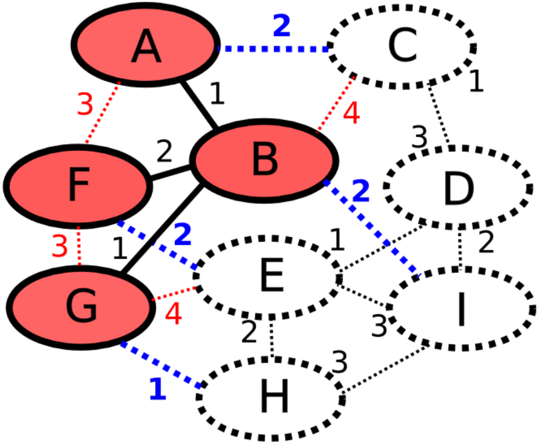
Prim's Algorithm Example - 4

Prim's Algorithm: Continue to follow smallest edge to vertices adjacent to the MST. Updating available adjacent edges as we go.



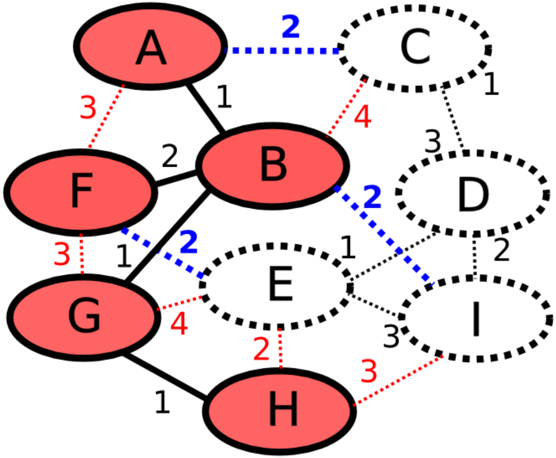
Prim's Algorithm Example - 5

Prim's Algorithm: Continue to follow smallest edge to vertices adjacent to the MST. Updating available adjacent edges as we go.



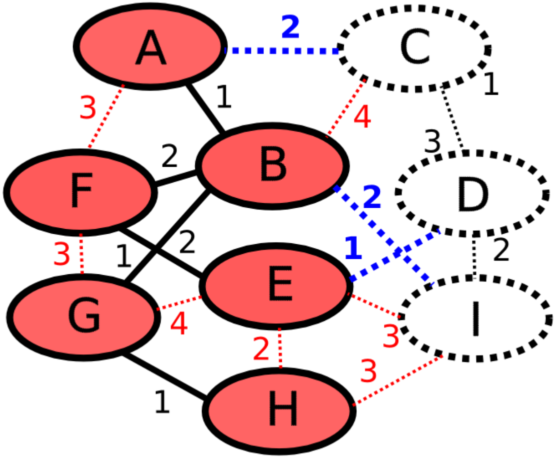
Prim's Algorithm Example - 6

Prim's Algorithm: Continue to follow smallest edge to vertices adjacent to the MST. Updating available adjacent edges as we go.



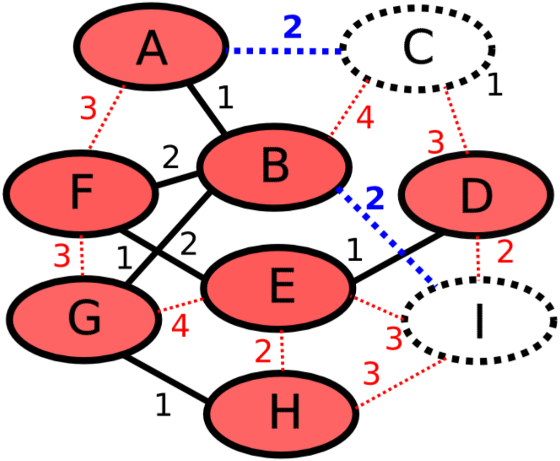
Prim’s Algorithm Example - 7

Prim’s Algorithm: Continue to follow smallest edge to vertices adjacent to the MST. Updating available adjacent edges as we go.



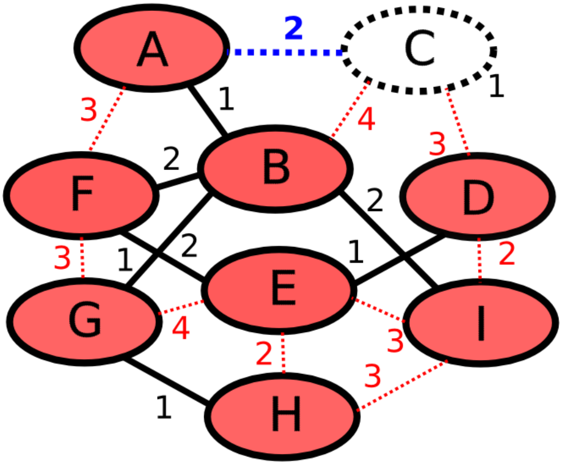
Prim's Algorithm Example - 8

Prim's Algorithm: Continue to follow smallest edge to vertices adjacent to the MST. Updating available adjacent edges as we go.



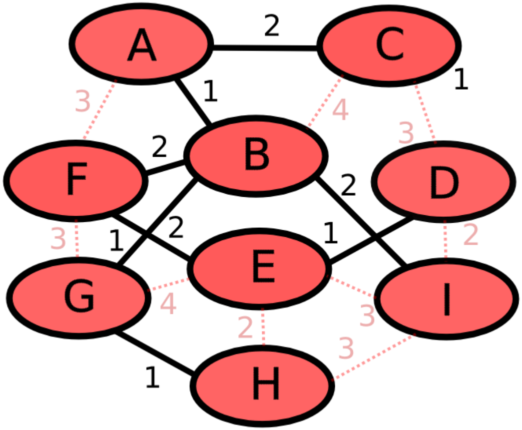
Prim's Algorithm Example - 9

Prim's Algorithm: Continue to follow smallest edge to vertices adjacent to the MST. Updating available adjacent edges as we go.



Prim's Algorithm Example - 10

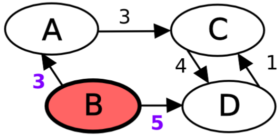
Prim's Algorithm: Continue to follow smallest edge to vertices adjacent to the MST. Updating available adjacent edges as we go.



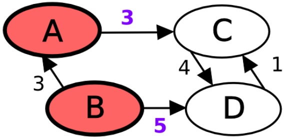
Prim's Directed - ERROR!

Prim's Algorithm: Does not work on directed graphs. It will sometimes, but many times it will fail because a local minimum does not guarenteed global minimum...see the example graph below with steps completed with Prim (Starting at B):

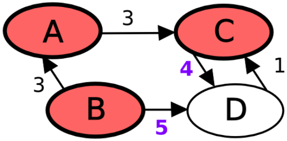
Step 1: B as source



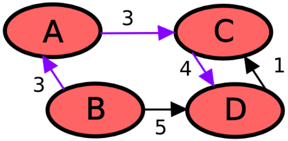
Step 2: B→A (3) less than 5



Step 3: A→C still less than 5



Step 4: C→D still less than 5



Prim results in a tree of total weight 10, Optimal is 9: $\{$***(B,A),(B,D),(D,C)***$\}$

Shortest Path

Shortest Path - Is the shortest path between two vertices. Dijkstra's algorithm is one notable algorithms for computing the shortest path between vertex v_i and v_j:

1. Assign an infinte distance to all other vertices from the source.
2. Mark the source as current. Mark all other vertices unvisited. Create a set of all the unvisited vertices called the unvisited set (Priority Queue).
3. For the current vertex, consider all of its unvisited neighbors:
   - Calculate their tentative distances.
   - Compare the newly calculated tentative distance to the current assigned value and assign the smaller one (May need to reorganize PQ).
4. Mark current vertex as visited and remove it from the unvisited.
5. If the destination vertex has been marked visited or the smallest tentative distance among the vertices in the unvisited set is infinity - Done, found it or cannot find path respectively.

Dijkstra's algorithm - 1

Using our existing adjacency matrix:

```
public int Dijkstra(int sourceInd, int destInd):
      int[] dists = new int[vertices.size()];
      int[] parents = new int[vertices.size()];
      boolean[] visited = new visited[vertices.size()];

      dists[sourceInd] = 0;
      MinHeap<T> PQ = new MinHeap();

      for(int i =0; i<vertices.size(); i++){
          if(i!=sourceInd){
              dists[i] = Integer.MAX_VALUE;
              parents[i] = -1;
          }
          visited[i]=false;
          PQ.enqueue(i, dists[i])
      }

     //...Iterate the Priority Queue (greedy)
}
```

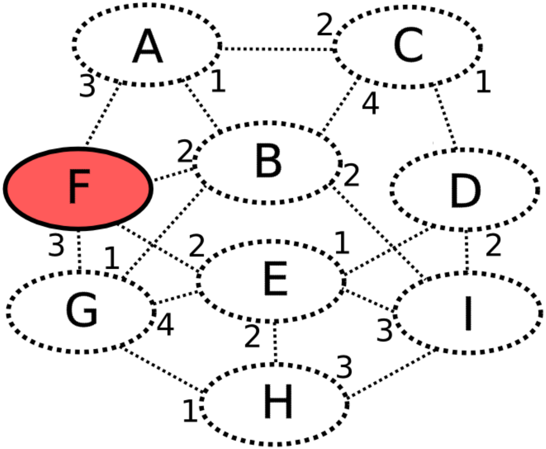
Dijkstra's algorithm - 2

Using our existing adjacency matrix:

```
public int Dijkstra(int sourceInd, int destInd):
     //...End Initialiation

     while(!PQ.isEmpty()){
         int u = PQ.dequeue();
         //Iterate the edges of u
         for(int i=0; i<vertices.size(); i++){
               //edge exists and not visited already
               if(edges[u][i]!=0 && !visited[i]){
                  int alt = dists[u] + edges[u][i]
                  if(alt < dists[i]){
                       dists[i] = alt;
                       parents[i] = u;
                       PQ.setPriority(i, alt); // New Method
                  }
             }}//iterated through all neighbors
             visited[u]=true;
     } //end a PQ is empty
     return dists[destInd];
}
```

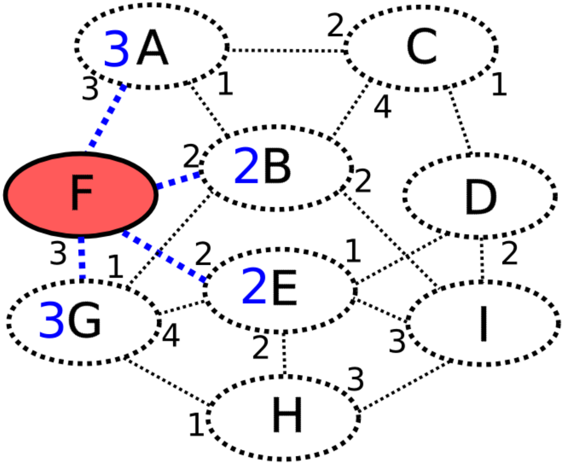
Dijkstra's Algorithm Example - 1

Dijkstra Example: Shortest Path from F - D
Begin with the starting vertex (a.k.a. source).



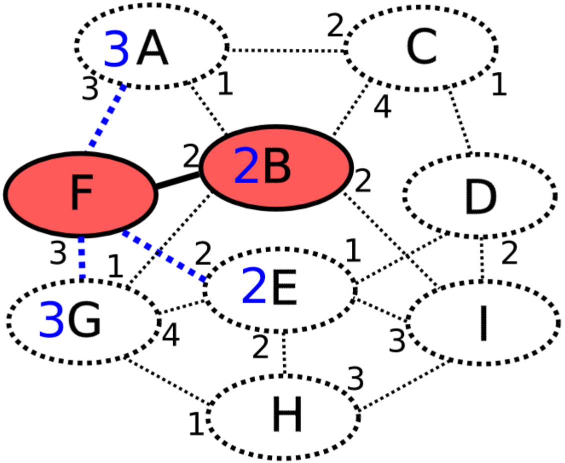
Dijkstra's Algorithm Example - 2

Dijkstra Example: Shortest Path from F - D
Generate costs to move to adjacent vertices (blue number on vertex), showing possible edges in blue.



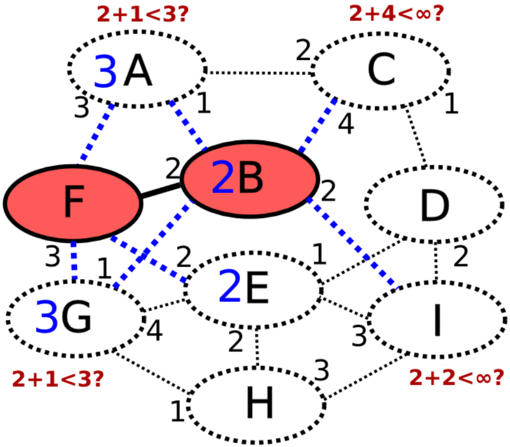
Dijkstra's Algorithm Example - 3

Dijkstra Example: Shortest Path from F - D
Choose the minimal non-visted vertex. (Using Priority Queue)



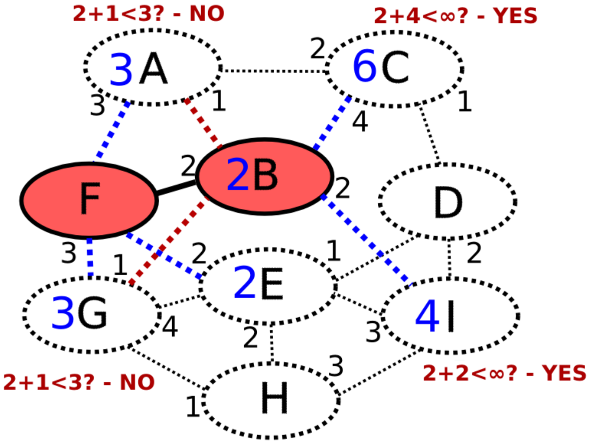
Dijkstra's Algorithm Example - 4

Dijkstra Example: Shortest Path from F - D
Examine new edges and check to see if costs from vertex result in a cheaper path?



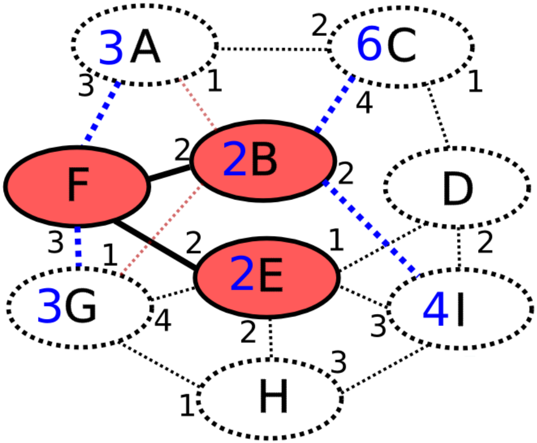
Dijkstra's Algorithm Example - 5

Dijkstra Example: Shortest Path from F - D
Update costs and possible edges to follow (i.e. parents of nodes)



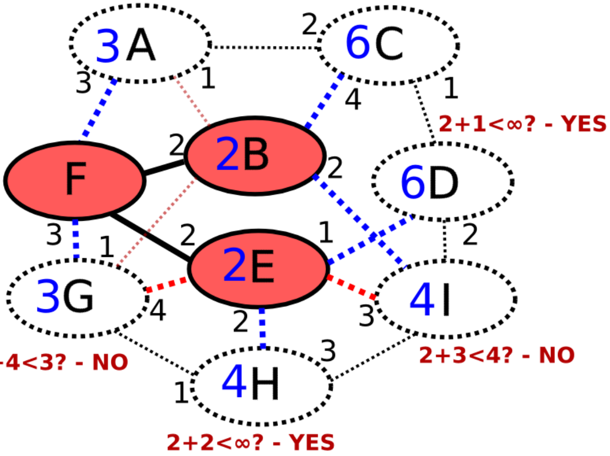
Dijkstra's Algorithm Example - 6

Dijkstra Example: Shortest Path from F - D
Select the next vertex to visit



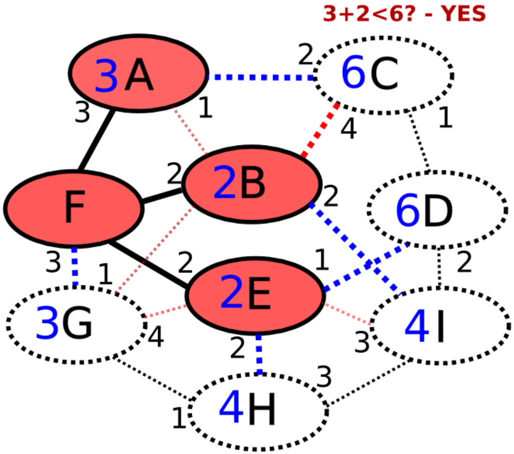
Dijkstra's Algorithm Example - 7

Dijkstra Example: Shortest Path from F - D
Update Costs and Links from newly visted vertex



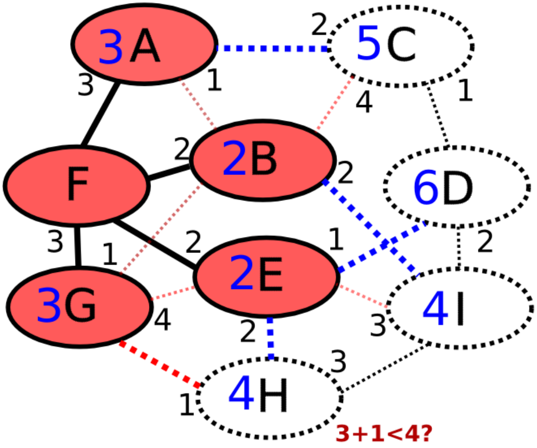
Dijkstra's Algorithm Example - 8

Dijkstra Example: Shortest Path from F - D
Visit next vertex (A) and update costs



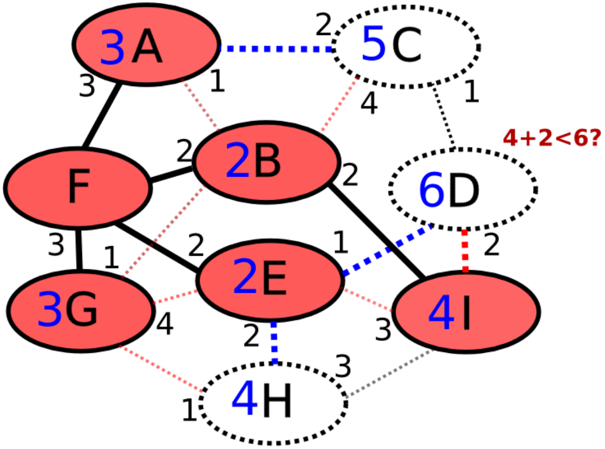
Dijkstra's Algorithm Example - 9

Dijkstra Example: Shortest Path from F - D
Visit next vertex (G) and update costs



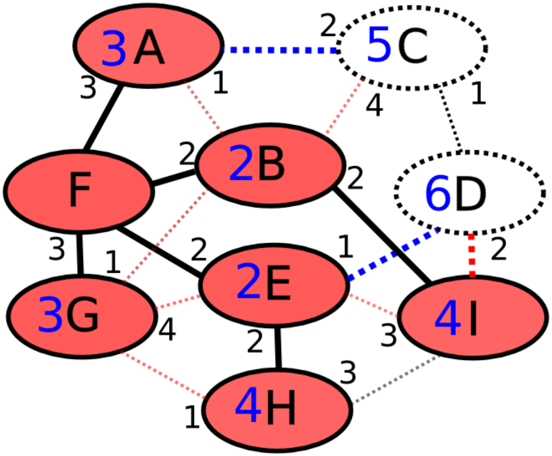
Dijkstra's Algorithm Example - 10

Dijkstra Example: Shortest Path from F - D
Visit next vertex (I) and update costs



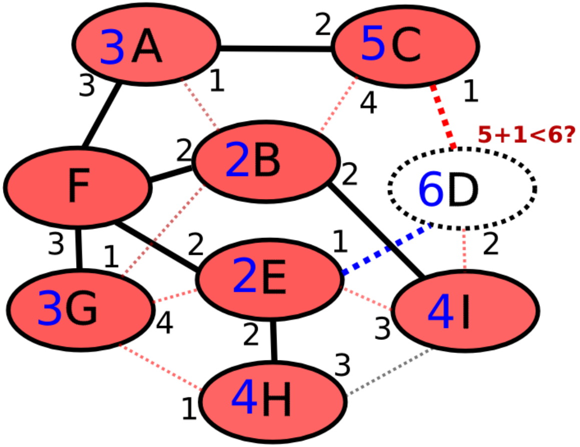
Dijkstra's Algorithm Example - 11

Dijkstra Example: Shortest Path from F - D
Visit next vertex (H) and update costs



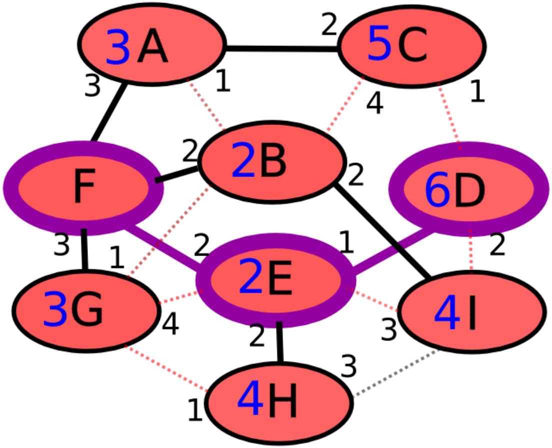
Dijkstra's Algorithm Example - 12

Dijkstra Example: Shortest Path from F - D
Visit next vertex (C) and update costs



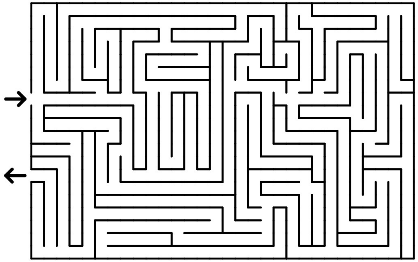
Dijkstra's Algorithm Example - 13

Dijkstra Example: Shortest Path from F - D
Visit next vertex (D) and all vertices visited, and a path exists from F (created by backtracking through links.)



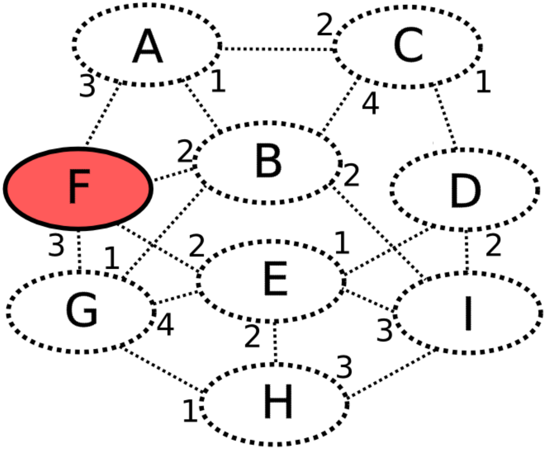
A* Path Search

A* is a generalization of Dijkstra's algorithm in which each path has a cost like before, but also a **heuristic** cost associated with getting to the source (an estimate, which could be as simple as assumed number of edges)



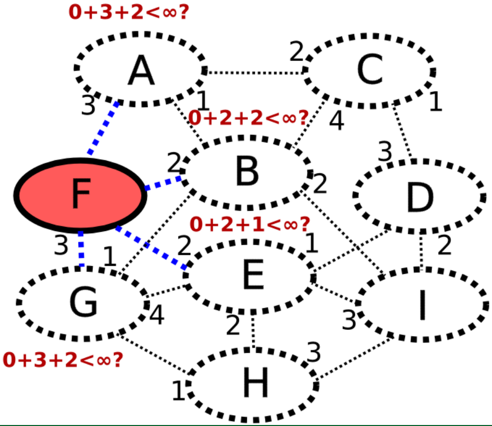
A* Search Algorithm Example - 1

A* Search Example: Path from F - D
Begin with the starting vertex (a.k.a. source).



A* Search Algorithm Example - 2

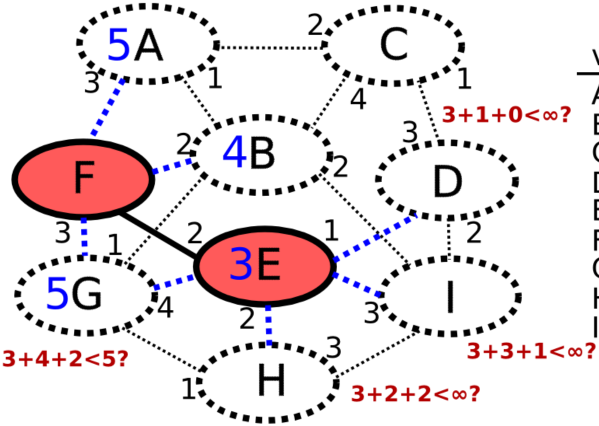
A* Search Example: Path from F - D
Generate costs to move to adjacent vertices: Cost is the sum of the cost to get to current vertex (at source it is zero) plus the cost of the edge plus the heurstic value of the target vertex.



v	h(v)
A	2
B	2
C	1
D	0
E	1
F	3
G	2
H	2
I	1

A* Search Algorithm Example - 3

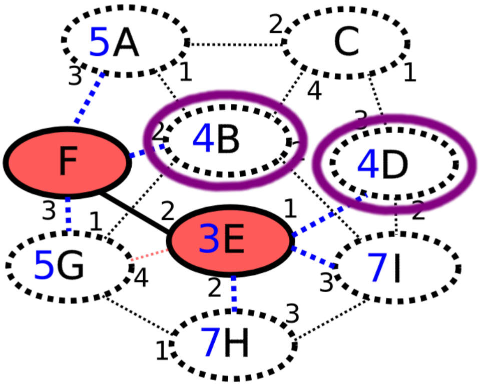
A* Search Example: Path from F - D
Like in Dijkstra's we visit the cheapest vertex next, and generate the costs as before.



v	h(v)
A	2
B	2
C	1
D	0
E	1
F	3
G	2
H	2
I	1

A* Search Algorithm Example - 4

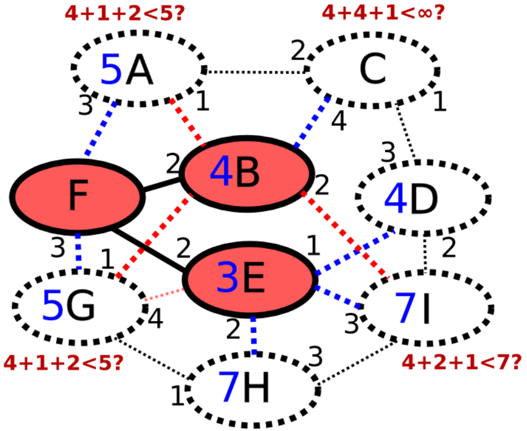
A* Search Example: Path from F - D
At this point we could chose D as we know that is our destination and be done (as it is a search algorithm = done when found; we are not looking for optimal path like in Dijkstra's). But just to illustrate the next step I'll visit B instead.



v	h(v)
A	2
B	2
C	1
D	0
E	1
F	3
G	2
H	2
I	1

A* Search Algorithm Example - 5

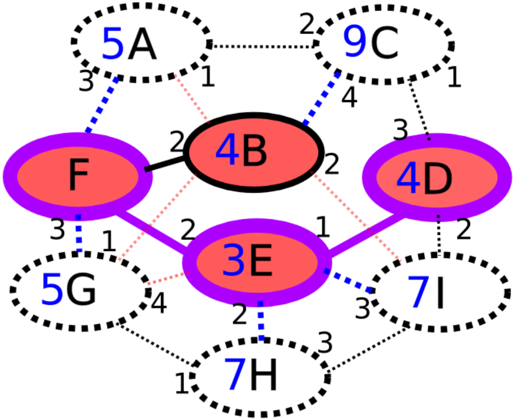
A* Search Example: Path from F - D
Visiting B just to illustrate one more step.



v	h(v)
A	2
B	2
C	1
D	0
E	1
F	3
G	2
H	2
I	1

A* Search Algorithm Example - 6

A* Search Example: Path from F - D
Completed search; backtrack as before to get the path to target.



v	h(v)
A	2
B	2
C	1
D	0
E	1
F	3
G	2
H	2
I	1