Theory Lecture 7

Algorithms for Graphs

Learning Objectives

- Look at some algorithms for graphs in depth
- Study some algorithms for shortest path and min spanning tree
- Discuss some hard graph problems

Graph paths

Graph Refresher

We looked at the basic terminology for graphs in theory lecture 4.

You also studied graph representation in SOF1 (week 8, seminar 4), along with the Depth-First and Breadth-First search or traversal of graphs.

In SOF1 seminar 4 (exercise 5), you looked at the problem of finding the number of links between two vertices in a connection network.

How many ways are there of getting between two vertices in a graph?

Walk and Path

In a walk, we move from vertex to vertex following edges, starting at v_0 and ending at v_k. The length is the number of edges used.

A path forbids us from returning to the same vertex again, so the edges and vertices are all different from each other.

Walk

A walk is a sequence of vertices $(v_0, v_1, \dots, v_k)$ such that $(v_i, v_{i+1}) \in E$. The length of the walk is k.

Path

A path is a walk where all the vertices are distinct, i.e. no vertex is visited more than once.

Number of paths

The number of walks and paths in a graph depends on the configuration of edges, but consider the **complete graph**, where there is an edge between all pairs of edges.

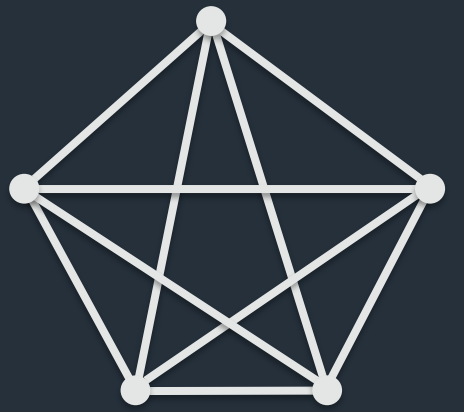
Any sequence of vertices starting at u and ending at v is a walk so there are an infinite number of walks.

Any sequence of vertices starting at u, ending at v and of length $k+1$ is a walk of length k so there are n^{k-1} such walks.

For a path, there are $n-2$ choices for the second vertex, $n-3$, for the third etc, so the number of paths is

$$(n-2)(n-3)\ldots(n-k) = \frac{(n-2)!}{(n-k-1)!}$$

The number of walks and paths is potentially exponentially large.



Complete graph on *n*=5 vertices

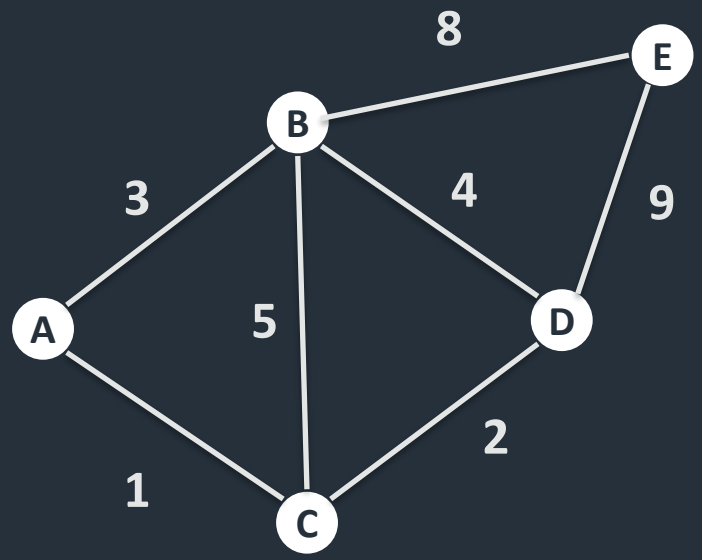
Shortest Path

The graph on the right is a weighted graph. The number in this case represents the length of an edge. The length of a walk (path) is the sum of the lengths of edges used, e.g. $l(ABC) = 3 + 5 = 8$.

How can we find the shortest path between two vertices among all the possible paths?

Notation: let $\mathrm{sp}(u, v)$ be the shortest path between u and v, and $l(\mathrm{sp}(u, v))$ be its length.

Key property of shortest paths. If $w \in \mathrm{sp}(u, v)$ then $sp(u, v) = sp(u, w) + sp(w, v)$



Dijkstra’s algorithm

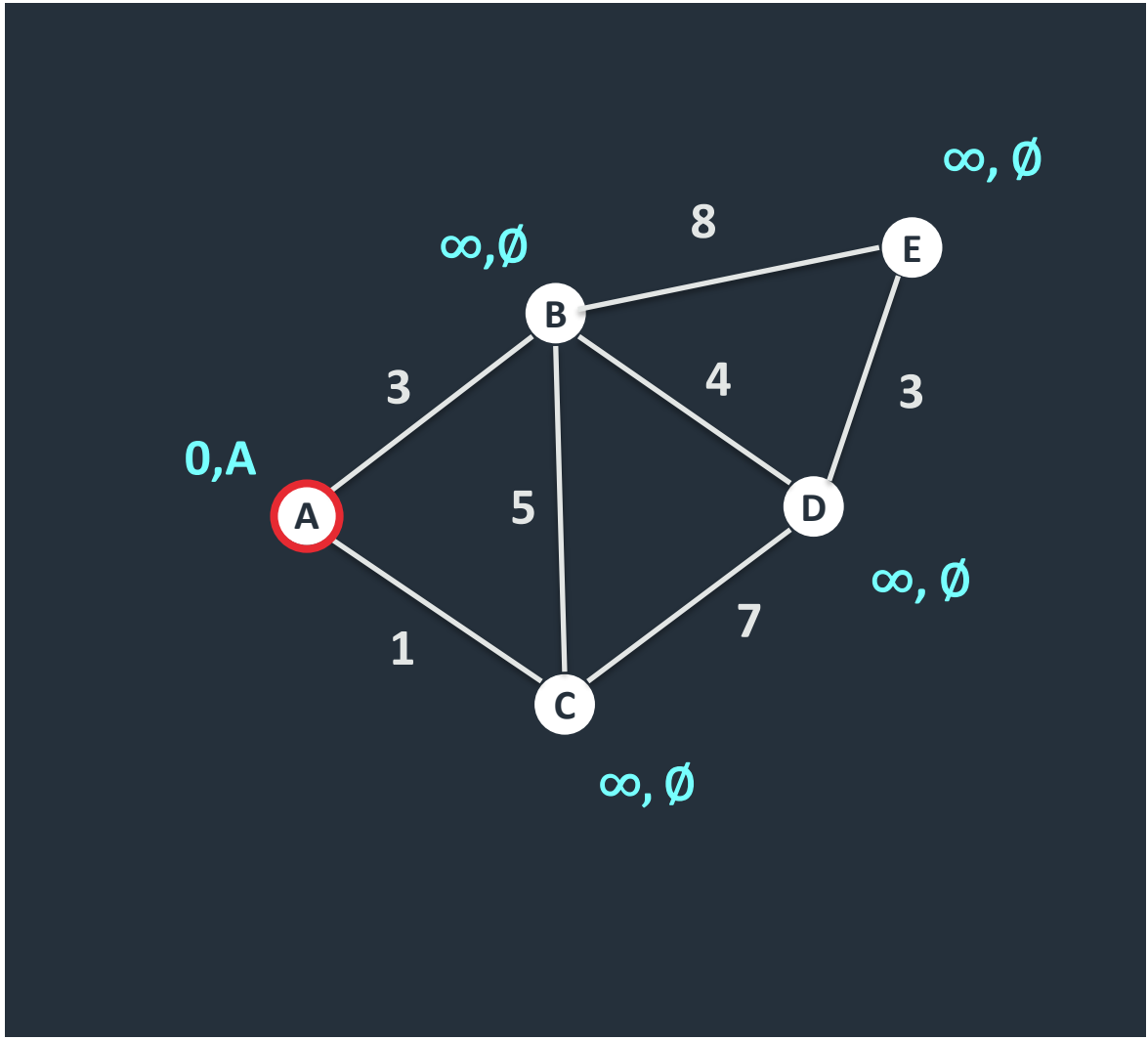
Dijkstra’s algorithm

```
ShortestPath(G : graph, source : node)

Mark all nodes unvisited, infinite distance, empty path
Mark source node visited, 0 distance, path=source node
u=source node
while u has unvisited neighbour
    for v in unvisited neighbours
        t=distance(u)+length(u,v)
        if t<distance(v)
            distance(v)=t
            path(v)=path(u)+v
        endif
    endfor
    visited(u)=true
    u=closest unvisited node
endwhile
Return path(destination)
```

Dijkstra’s algorithm

Dijkstra’s algorithm

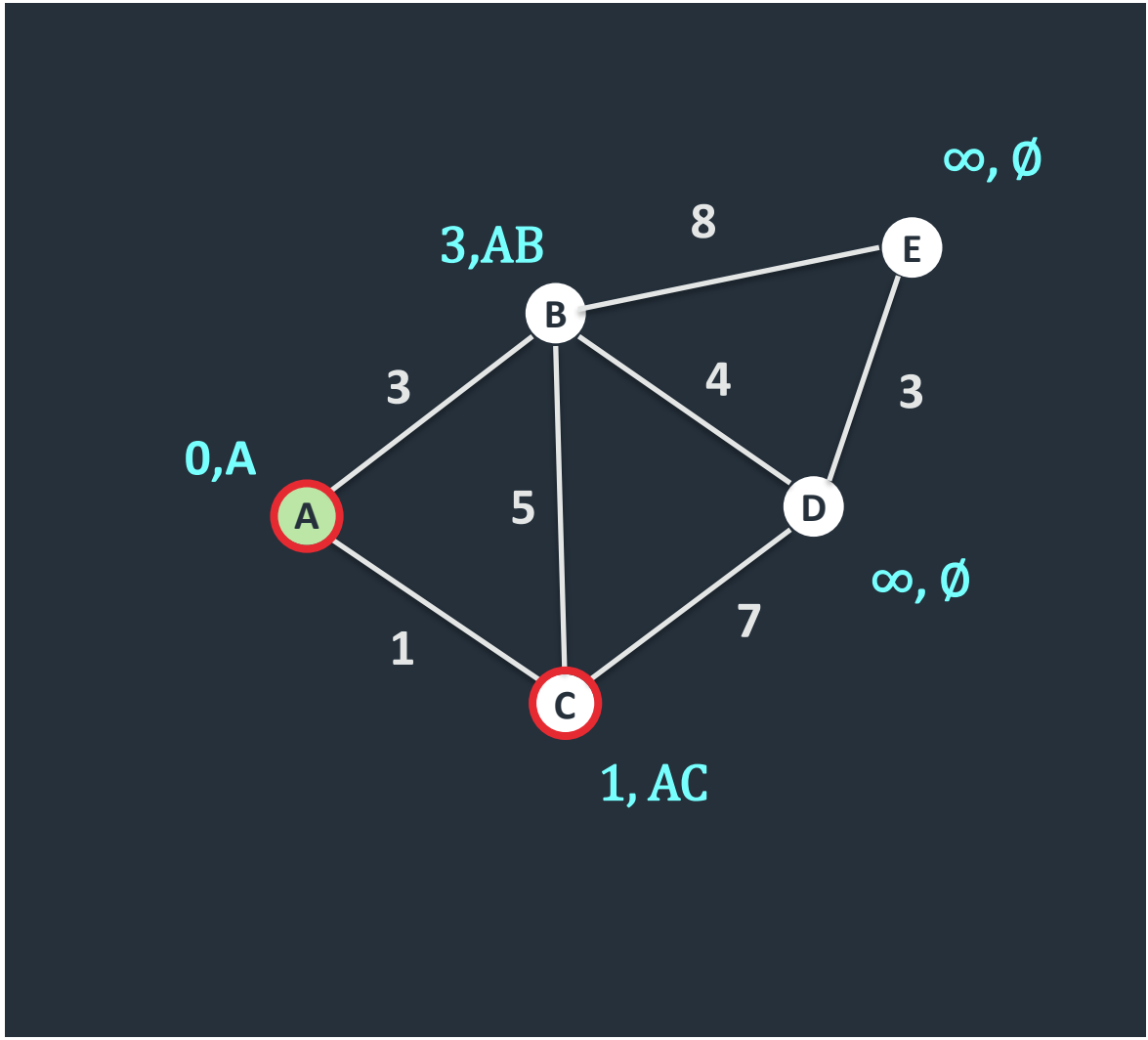
```
ShortestPath(G : graph, source : node)

Mark all nodes unvisited, infinite distance, empty path
Mark source node visited, 0 distance, path=source node
u=source node
while u has unvisited neighbour
    for v in unvisited neighbours
        t=distance(u)+length(u,v)
        if t<distance(v)
            distance(v)=t
            path(v)=path(u)+v
        endif
    endfor
    visited(u)=true
    u=closest unvisited node
endwhile
Return path(destination)
```



Dijkstra's algorithm

Dijkstra's algorithm

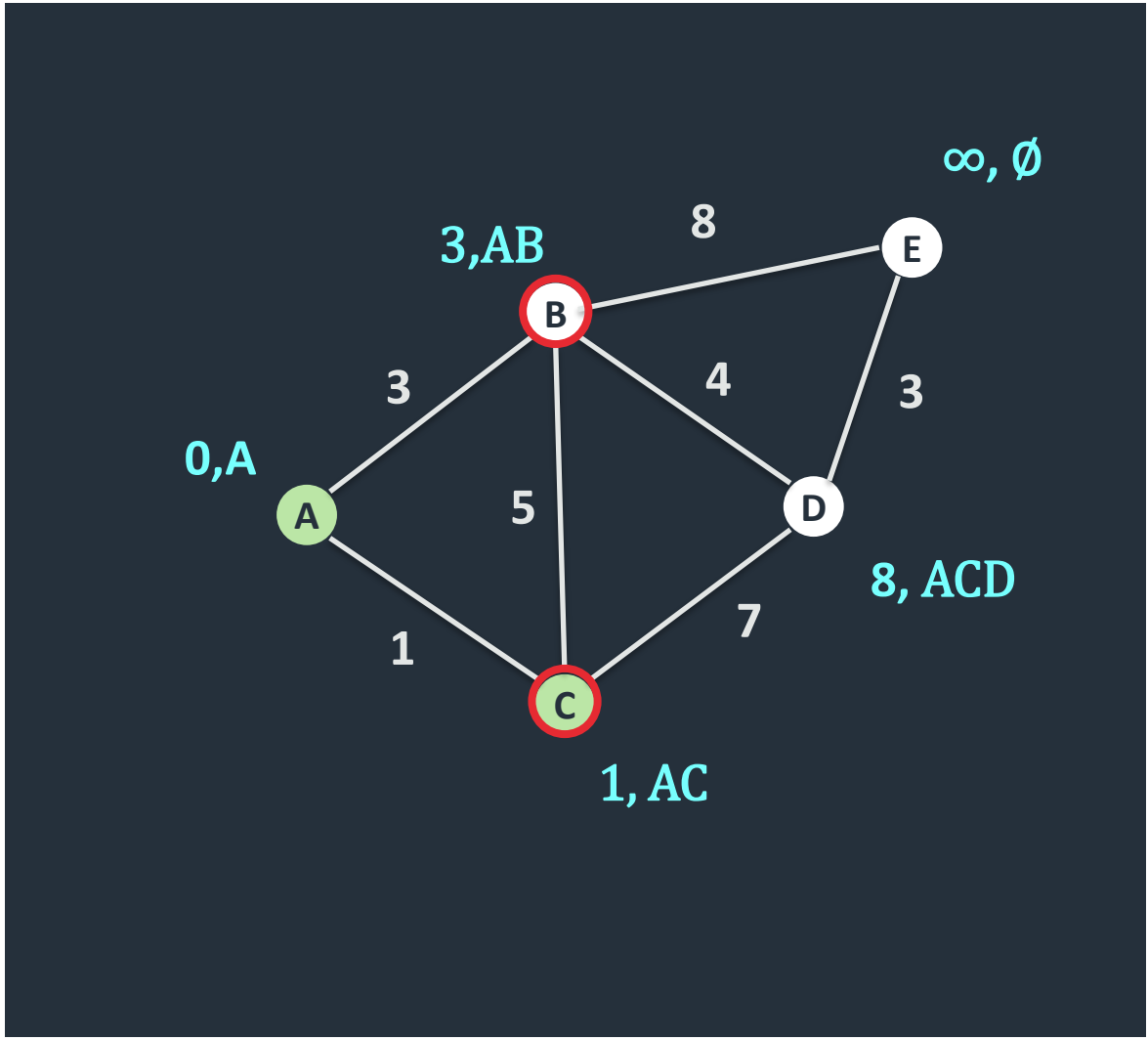
```
ShortestPath(G : graph, source : node)

Mark all nodes unvisited, infinite distance, empty path
Mark source node visited, 0 distance, path=source node
u=source node
while u has unvisited neighbour
    for v in unvisited neighbours
        t=distance(u)+length(u,v)
        if t<distance(v)
            distance(v)=t
            path(v)=path(u)+v
        endif
    endfor
    visited(u)=true
    u=closest unvisited node
endwhile
Return path(destination)
```



Dijkstra's algorithm

Dijkstra's algorithm

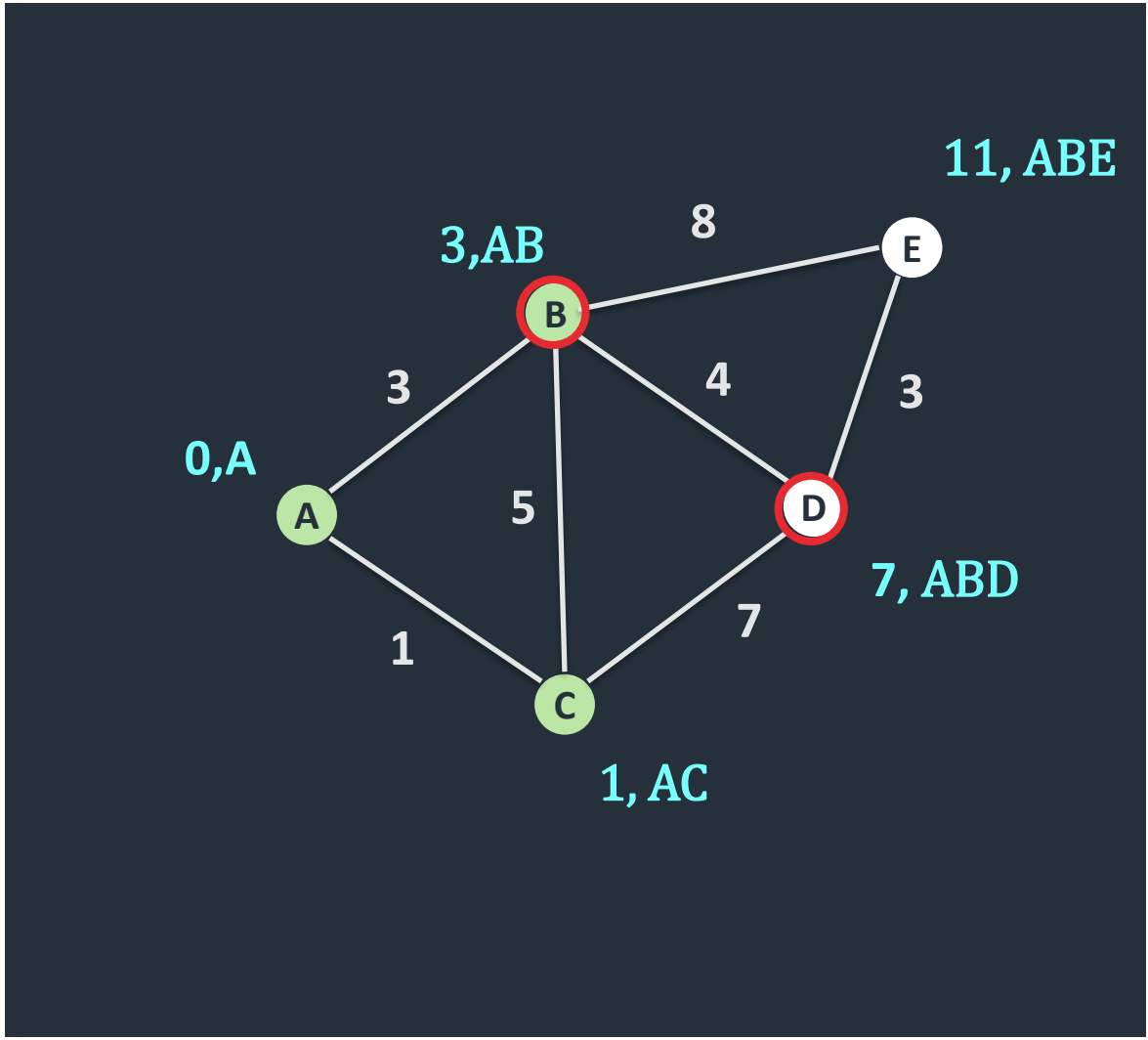
```
ShortestPath(G : graph, source : node)

Mark all nodes unvisited, infinite distance, empty path
Mark source node visited, 0 distance, path=source node
u=source node
while u has unvisited neighbour
    for v in unvisited neighbours
        t=distance(u)+length(u,v)
        if t<distance(v)
            distance(v)=t
            path(v)=path(u)+v
        endif
    endfor
    visited(u)=true
    u=closest unvisited node
endwhile
Return path(destination)
```



Dijkstra's algorithm

Dijkstra's algorithm

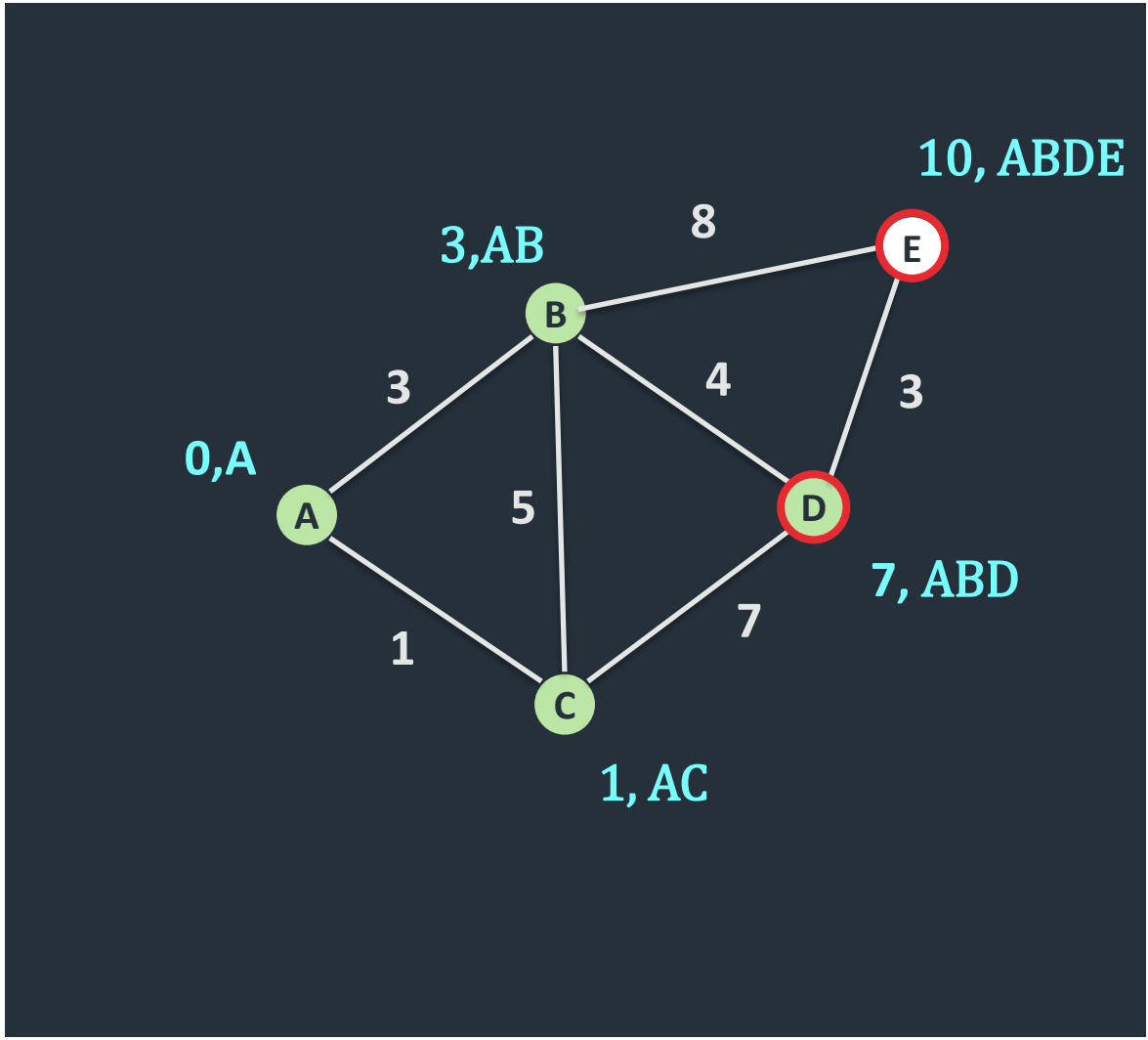
```
ShortestPath(G : graph, source : node)

Mark all nodes unvisited, infinite distance, empty path
Mark source node visited, 0 distance, path=source node
u=source node
while u has unvisited neighbour
    for v in unvisited neighbours
        t=distance(u)+length(u,v)
        if t<distance(v)
            distance(v)=t
            path(v)=path(u)+v
        endif
    endfor
    visited(u)=true
    u=closest unvisited node
endwhile
Return path(destination)
```



Dijkstra's algorithm

Dijkstra's algorithm

```
ShortestPath(G : graph, source : node)

Mark all nodes unvisited, infinite distance, empty path
Mark source node visited, 0 distance, path=source node
u=source node
while u has unvisited neighbour
    for v in unvisited neighbours
        t=distance(u)+length(u,v)
        if t<distance(v)
            distance(v)=t
            path(v)=path(u)+v
        endif
    endfor
    visited(u)=true
    u=closest unvisited node
endwhile
return path(destination)
```

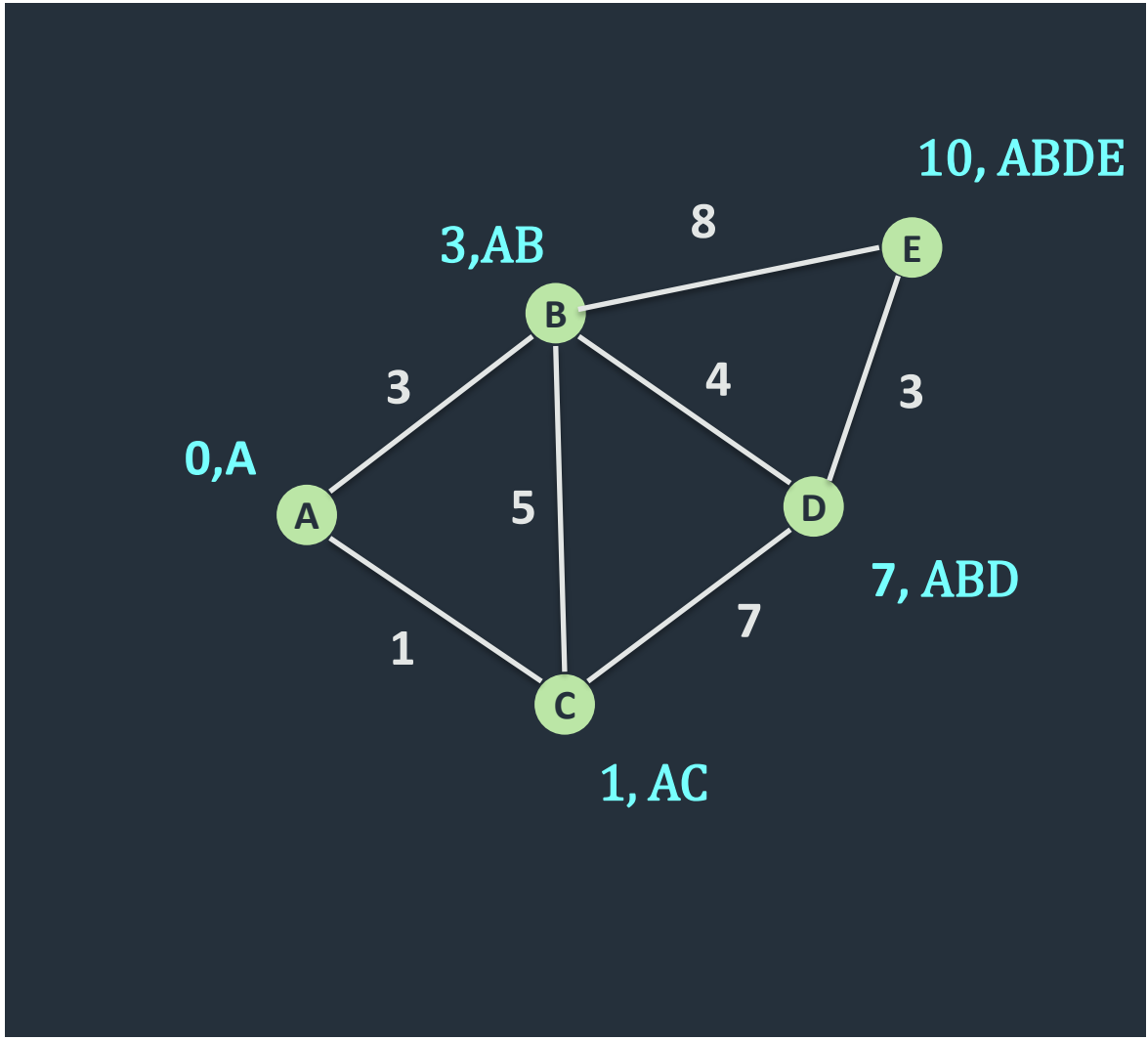


Dijkstra's algorithm

Dijkstra's algorithm is a single-source all-destination method.

The complexity is $O(n^2)$

For all-source all-destination shortest paths, the Floyd-Warshall algorithm can be used which is $O(n^3)$ (see problem set).



Spanning tree

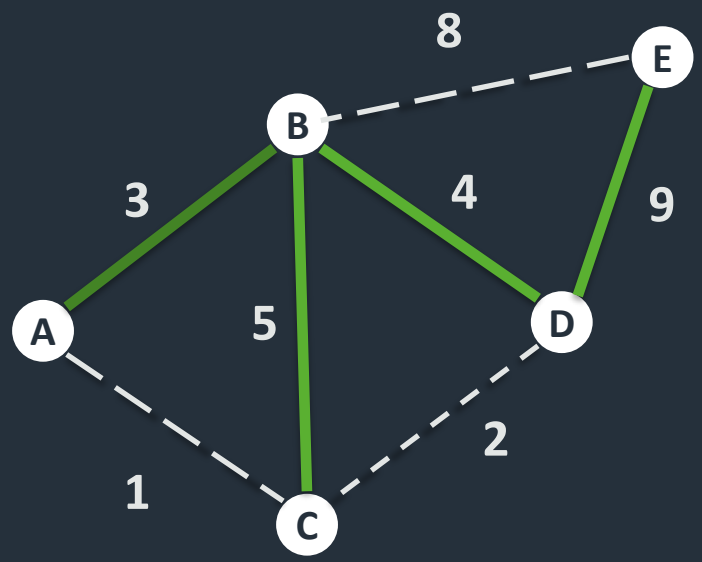
Spanning tree

A **spanning tree** of a graph is a tree which includes all vertices and uses only edges in the graph. All connected graphs have at least one spanning tree.

Since a spanning tree is a tree, it has exactly $n - 1$ edges, and uses the minimum number of edges to connect all the vertices.

The tree on the right has weight $w(T) = 3 + 5 + 4 + 9 = 21$

In a weighted graph, the **minimum spanning tree** is the tree with the smallest weight.



Prim's algorithm

Prim's algorithm

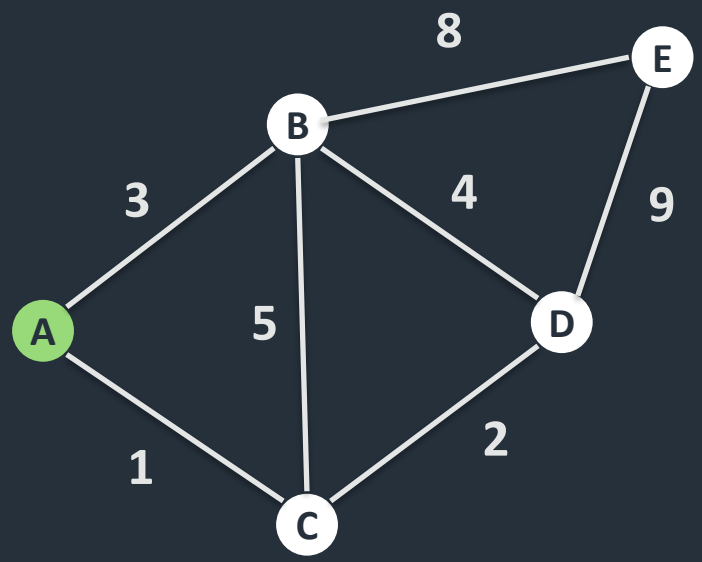
```
MinimumSpanningTree(G : graph)
Choose a starting node u
V(T)={u}, E(T)=Ø
while V(T)≠V(G)
    find the edge (u,v) with the minimum weight
                    such that one of u,v is in V(T) and the other is not
    add v to V(T) and (u,v) to E(T)
endwhile
Return V(T), E(T)
```

This is an example of a greedy algorithm – we take the best possible step at each iteration. In general a greedy algorithm does not find an optimal solution, but Prim's algorithm does.

Prim’s algorithm

Prim’s algorithm

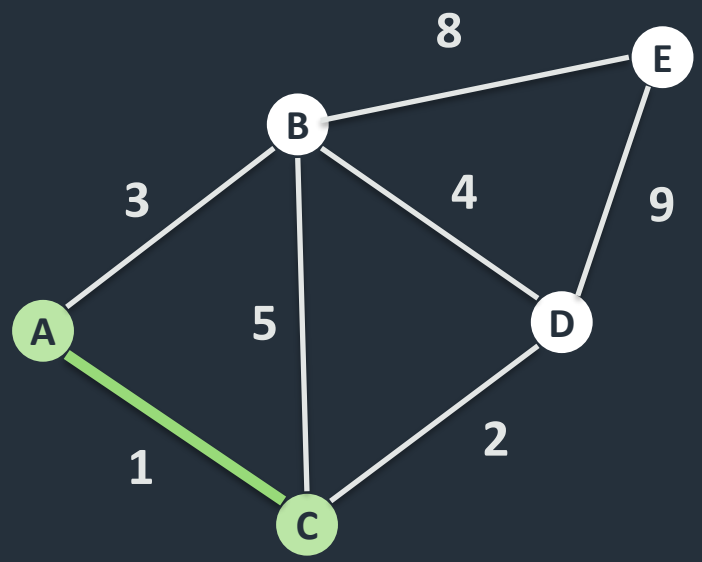
```
MinimumSpanningTree(G : graph)
Choose a starting node u
V(T)={u}, E(T)=Ø
while V(T)≠V(G)
    find the edge (u,v) with the minimum weight
                    such that one of u,v is in
                    V(T) and the other is not
    add v to V(T) and (u,v) to E(T)
endwhile
Return V(T), E(T)
```



Prim’s algorithm

Prim’s algorithm

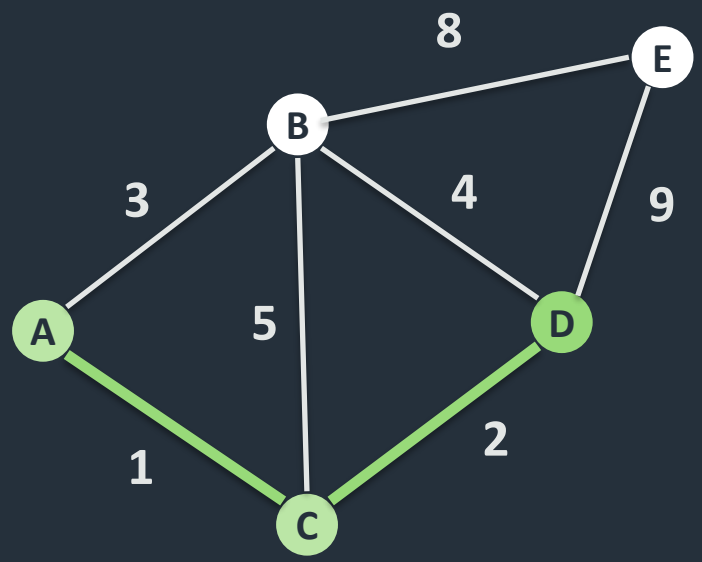
```
MinimumSpanningTree(G : graph)
Choose a starting node u
V(T)={u}, E(T)=Ø
while V(T)≠V(G)
    find the edge (u,v) with the minimum weight
                    such that one of u,v is in
                    V(T) and the other is not
    add v to V(T) and (u,v) to E(T)
endwhile
Return V(T), E(T)
```



Prim’s algorithm

Prim’s algorithm

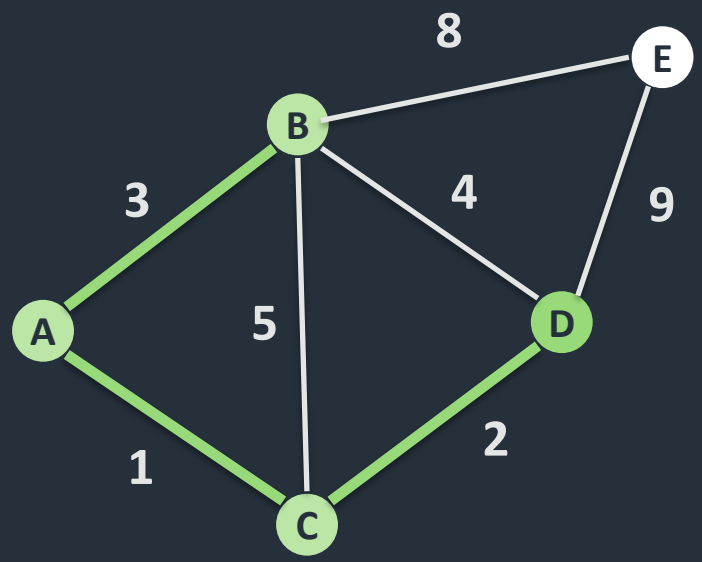
```
MinimumSpanningTree(G : graph)
Choose a starting node u
V(T)={u}, E(T)=Ø
while V(T)≠V(G)
    find the edge (u,v) with the minimum weight
                    such that one of u,v is in
                    V(T) and the other is not
    add v to V(T) and (u,v) to E(T)
endwhile
Return V(T), E(T)
```



Prim's algorithm

Prim's algorithm

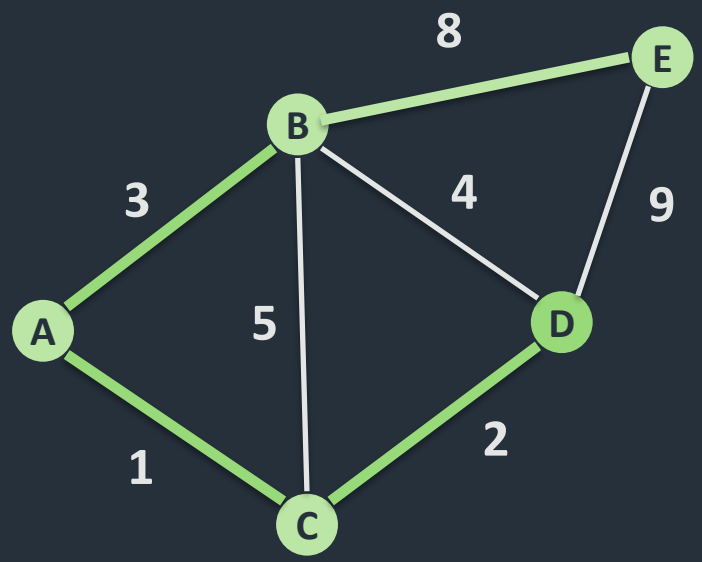
```
MinimumSpanningTree(G : graph)
Choose a starting node u
V(T)={u}, E(T)=Ø
while V(T)≠V(G)
    find the edge (u,v) with the minimum weight
                    such that one of u,v is in
                    V(T) and the other is not
    add v to V(T) and (u,v) to E(T)
endwhile
Return V(T), E(T)
```



Prim’s algorithm

Prim’s algorithm

```
MinimumSpanningTree(G : graph)
Choose a starting node u
V(T)={u}, E(T)=Ø
while V(T)≠V(G)
    find the edge (u,v) with the minimum weight
                    such that one of u,v is in
                    V(T) and the other is not
    add v to V(T) and (u,v) to E(T)
endwhile
Return V(T), E(T)
```



Prim's algorithm

Since there are $O(n^2)$ potential edges, as written the algorithm takes $O(n^3)$ time.

As we have seen with heaps, the right data structure can improve the complexity of finding the min element. The edge-find can be improved to $O(n)$ by the method discussed in e.g. Skiena section 8.1, for a overall complexity of $O(n^2)$.

The correctness proof of Prim's algorithm is also outlined in Skiena.

Prim's algorithm

```
MinimumSpanningTree(G : graph)
Choose a starting node u
V(T)={u}, E(T)=∅
while V(T)≠V(G)
    find the edge (u,v) with the minimum
        weight such that one of u,v is in
        V(T) and the other is not
    add v to V(T) and (u,v) to E(T)
endwhile
Return V(T), E(T)
```

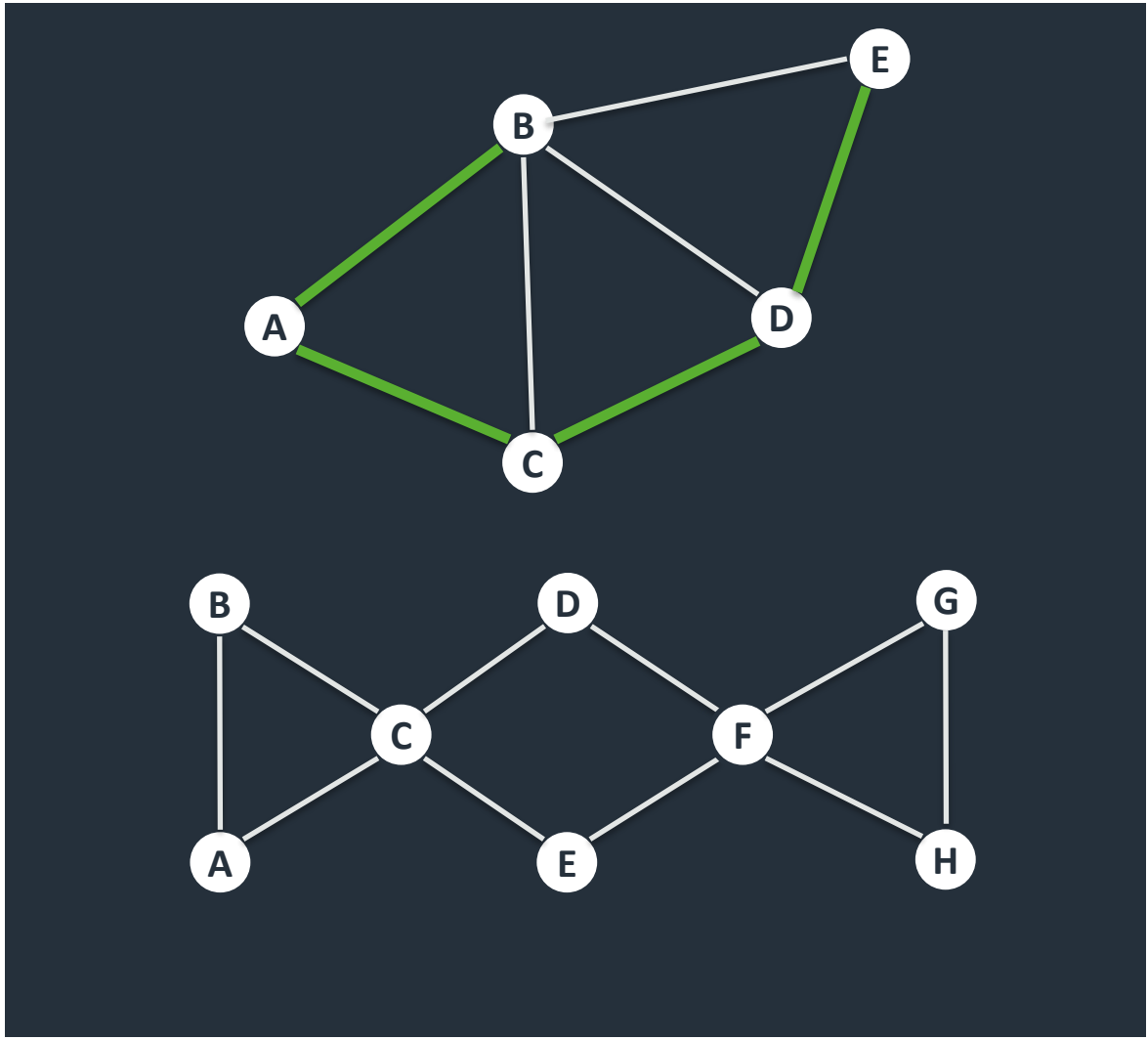
Other problems

Hamiltonian Path

A Hamiltonian path is a path in the graph which visits all vertices exactly once.

The right-bottom graph does not have a Hamiltonian path

Find the Hamiltonian paths, or even determining whether one exists, is a hard problem (NP-hard technically). The best known algorithms are $\Omega(n^c)\forall c$, i.e. there is no polynomial time algorithm.



Travelling Salesman Problem

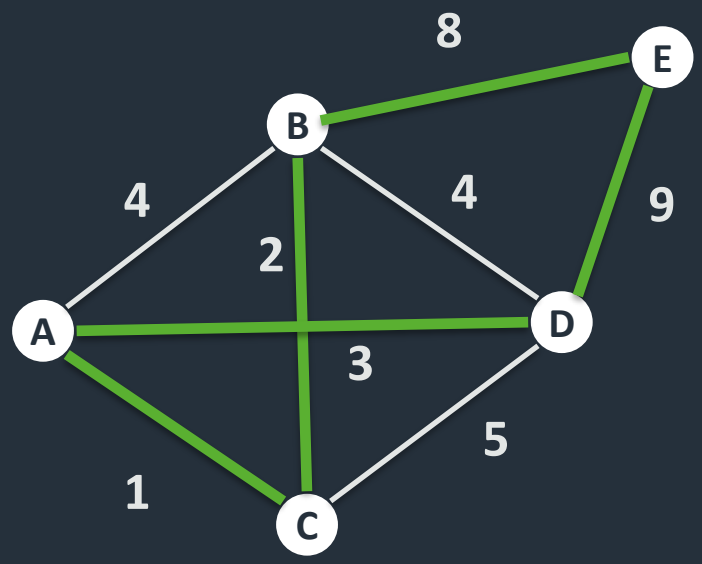
In the travelling salesman problem (TSP), a travelling salesman must visit a set of cities and return to the start, while minimising the distance travelled.

This problem can be represented on a graph – the cities are the vertices and the edge weights represent travelling distances between cities.

The problem is then to find a closed path (called a cycle) with minimum cost.

This is another NP-hard problem with all known solutions having super-polynomial complexity.

We will discuss solutions to this problem later in the module.



Intractable problems

A problem where there is an algorithm which takes polynomial time, i.e. it takes $O(n^c)$ time for some constant c, is called a tractable problem.

Shortest path and minimum spanning tree are tractable problems.

A problem where the best algorithm is $\Omega(n^c) \forall c$ is superpolynomial in complexity and is called intractable.

Hamiltonian path and TSP are currently intractable.

If we were to discover a polynomial time algorithm for one of these, they would become tractable, but this is considered unlikely.

Summary

Understand:

- Shortest path and Dijkstra's algorithm
- Minimum spanning tree and Prim's algorithm
- Some problems, for example Hamiltonian path and TSP, are difficult solve
- The difference between tractable and intractable problems

Read

- Skiena, Sections 8.1, 8.3, 11.3.1
- Try exercises 8.7, 8.15, 8.19

Next

- Algorithm design