The Traveling Salesperson Problem (TSP)

Given the cities and cost to travel between cities, obtain a route R such that:

1. All cities are in R
2. Every city is visited only once in R
3. R has the minimum cost. That is, for any other route R' meeting 1-2; cost(R') $\geq$ cost(R).

The Travelling Salesperson:

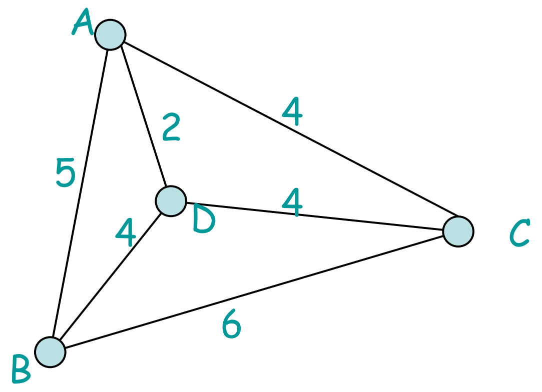
Typically a travelling salesperson wants to cover all the towns in their area using the minimum driving distance.

Travelling salesperson problems tackle this by first turning the towns and roads into a network.

Definition:

A *Hamiltonian cycle* is a tour which contains every node once

Consider the Hamiltonian cycles for this graph

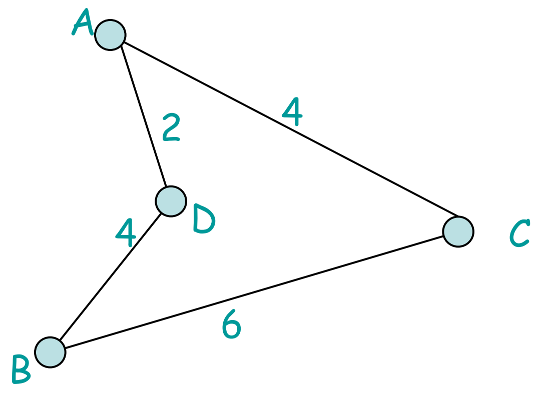


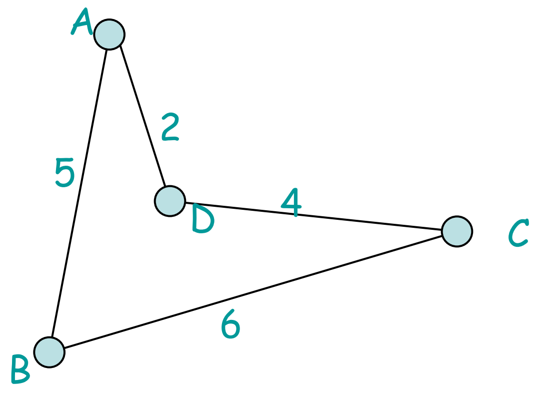
There are just three essentially different Hamiltonian cycles:

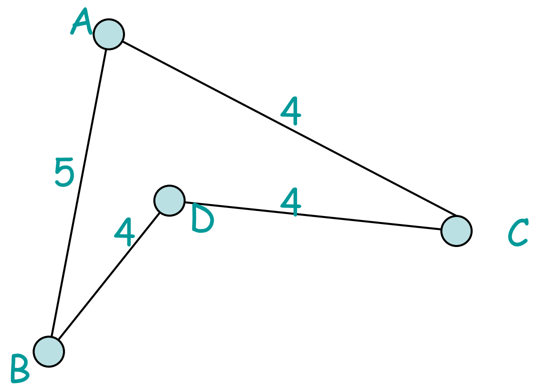
ACBDA with weight 16

ABCDA with weight 17

ABDCA with weight 17

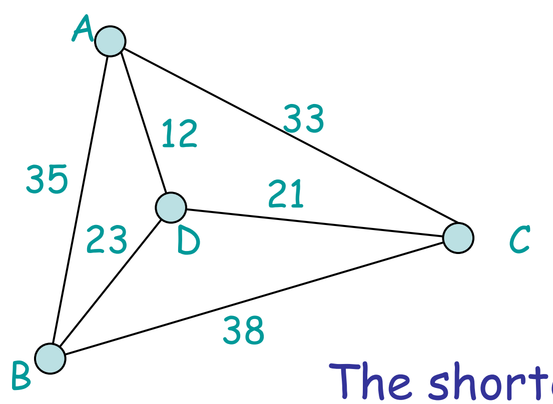






However, not all graphs have Hamiltonian cycles.

Consider this graph:



A salesperson living in town B, say, may still want to find the shortest round trip visiting every town.

To enable you to use the next three algorithms you can replace any network like this one by the complete network of shortest distances.

The shortest distance between A and C is 33 (via D: 12 + 21)

The shortest distance between A and B is 35 (via D: 12 + 23)

Adding direct arcs AC and AD does not change the problem but does produce a graph with Hamiltonian cycles.

The rest of this module assumes the problem is always the classical one of finding a Hamiltonian cycle of minimum weight with no repetition of nodes.

Our next algorithm is called the "Nearest Neighbour" – it will find a solution to a travelling salesperson problem but not necessarily the best (optimum).

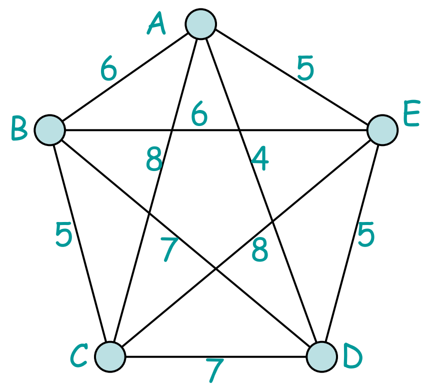
So we will then look at finding limits between which the optimum solution must lie and how to improve any solution we find.

The Nearest Neighbour Algorithm

To find a (reasonably good) Hamiltonian cycle i.e. a closed trail containing every node of a graph

Step 1	Choose any starting node
Step 2	Consider the arcs which joins the node just chosen to nodes as yet unchosen. Pick the one with minimum weight and add it to the cycle
Step 3	Repeat step 2 until all nodes have been chosen
Step 4	Then add the arc that joins the last-chosen node to the first-chosen node

To find a (reasonably good) Hamiltonian cycle i.e. a closed trail containing every node of this graph using the Nearest Neighbour Algorithm:



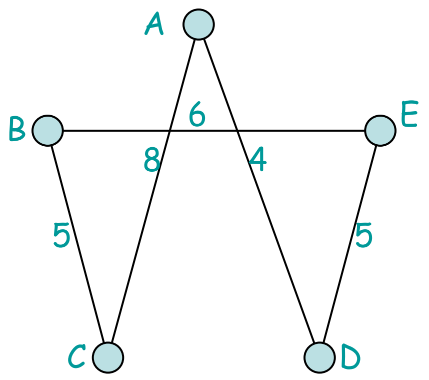
Choose any starting node – let's say A

Consider the arcs which join the node just chosen to nodes as yet unchosen. Pick the one with minimum weight and add it to the cycle

The first arc is AD as 4 is the least of 6, 8, 4 and 5

From D, DE is chosen as 5 is the least of 7, 7, and 5

Then EB, BC, and finally CA



The cycle is ADEBCA which has weight 28 (5 + 5 + 6 + 4 + 8)

The Nearest Neighbour Algorithm is 'greedy' because at each stage the immediately best route is chosen, without a look ahead, or back, to possible future problems or better solutions.

We do know that the optimum solution will have an equal or lower weight than it (by definition) and so the algorithm has produced an *upper bound*. ??

This then begs the question:

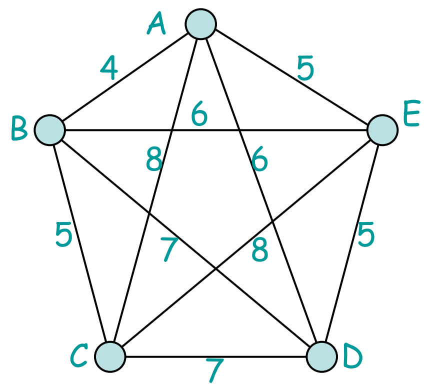
Can we find a lower bound to see how much potential there is for improvement?

The Lower Bound Algorithm

To find a lower bound for a travelling salesperson problem

Step 1	Pick any node and remove the two connecting arcs with least weight
Step 2	Find the minimum spanning tree for the other nodes using Prim's algorithm
Step 3	Add back in the two arcs removed previously
Step 4	The weight of the resulting graph (which may not be a cycle) is a lower bound i.e. any optimum solution must have at least this weight

To find a lower bound for a Hamiltonian cycle using the Lower Bound Algorithm:



Pick any node and remove the two connecting arcs with least weight

Let's pick A and remove arcs AB and AE

Find the minimum spanning tree for BCDE using Prim's algorithm

Pick any starting node, say, B.

The shortest arc is BC, then BE and finally ED.

Now we add in the two arcs we removed earlier

This tree has weight 25 (4 + 6 + 5 + 5 + 5)

The optimum solution must have at least this weight

How do we know this?

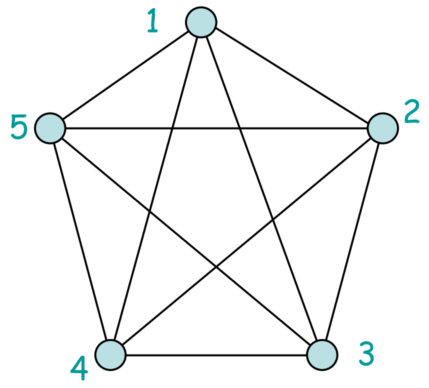
Well, *any* Hamiltonian cycle **must** consist of 5 arcs (because there are 5 nodes): 2 arcs into and out of A and the other three arcs connecting B, C, D and E.

The two arcs from A must have weight of at least 4 + 5, since these are the smallest available.

By applying Prim's algorithm to B, C, D and E we can see that the three arcs connecting B, C, D and E must have at least weight 5 + 5 + 6 (the minimum connector or spanning tree). **Hence, it is *impossible* for any Hamiltonian cycle to have weight less than 4 + 5 + 5 + 5 + 6**

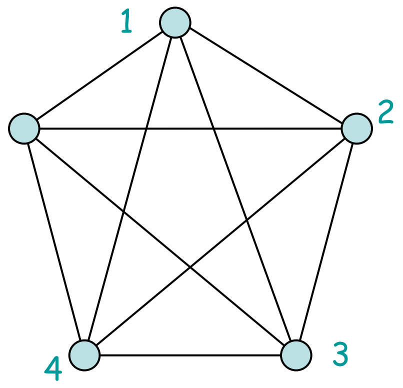
The Tour Improvement Algorithm

To look for possible improvement in a tour found by the nearest neighbour algorithm:



Step 1 Number the nodes in the order of the tour; start at node 1

Step 2 Consider just the part of the tour 1 - 2 - 3 - 4



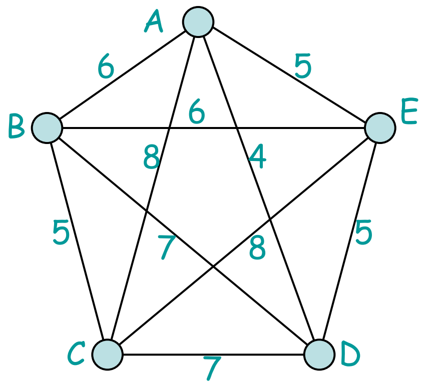
Step 3 Swap the middle nodes to change the order to 1 - 3 - 2 - 4

Step 4 Compare the two and keep the order with the lower weight

Step 5 Move on to node 2 and repeat until each node has been the start node once

To look for possible improvement in a tour using the tour improvement algorithm:

Here is a network:



And here is a tour:
ACEDBA

It has weight 34

Let's see if we can improve on that:

We have now had each node at the front and so we stop.

The final tour is ADCBEA.

Pick A as node 1 and consider the tour from A-C-E-D
This has weight 8 + 8 + 5 = 21
Now swap the middle two nodes
And you get A-E-C-D

This has weight 5 + 8 + 7 = 20

So swap E and C and the tour becomes AECDBA
Now look at ECDB

ECDB = 8 + 7 + 7 = 22 EDCB = 5 + 7 + 5 = 17
So swap C and D and the tour becomes AEDCBA

Now look at DCBA
DCBA = 7 + 5 + 6 = 18 DBCA = 7 + 5 + 8 = 20

So leave the tour as AEDCBA

Now look at CBAE

CBAE = 5 + 6 + 5 = 16 CABE = 8 + 6 + 6 = 20

So leave the tour as AEDCBA

Now look at BAED

BAED = 6 + 5 + 5 = 16 BEAD = 6 + 5 + 4 = 15

So swap A and E and the tour becomes ADCBEA

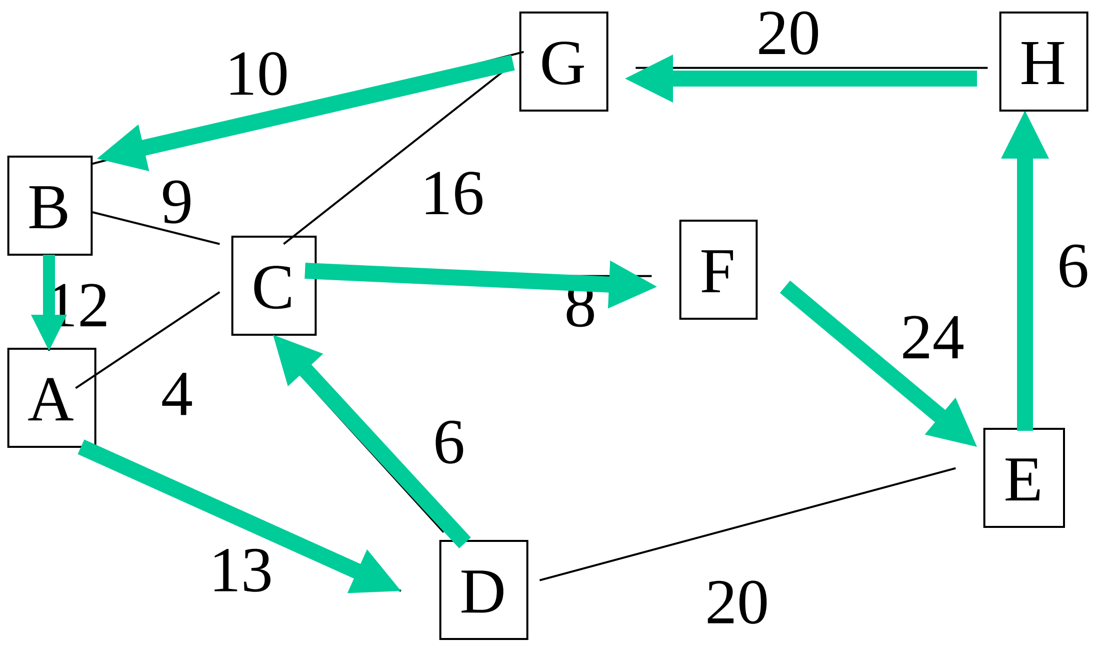
NP Complexity

The class **NP** consists of all problems that can be solved in polynomial time by **nondeterministic algorithms**

Nondeterministic algorithms are done in two phases:

- **Phase I**: algorithm that makes a guess (solutions must be included among the possible guesses)
- **Phase II**: algorithm that checks if the guess is an actual solution or not

Example: Traveling Salesman Problem (TSP) is in NP



Why do We Care About NP Problems?

Network Problems	•Traveling Salesperson •Longest path •Graph coloring
Data Storage	•Minimum Bin Packing
Scheduling	•Minimum Job Scheduling •Minimum Multiprocessor Scheduling
Mathematical Programming	•Knapsack

Proof

We can show that Traveling Salesman Problem (TSP) is in NP

```
phaseI(G: Graph)
//input: G a graph with n nodes
//output: C a guess for TSP

v ← randomNode(G)
C ← ()
While random(0,1) = 1
        and |C| < n do {
  w ← PickNeighbourRandomly(v)
  C ← C + (v,w)
  v ← w
}
return C
```

```
phaseII(C: path, min: int )
//input: C a guessed solution
//output: true iff C is a TSP

If |C| < n then return false
Visited ← {}
for i =1 to n do {
    (u,v) = C[i]
    if v in Visited then
            return false
    else Visited ← Visited + {v}
}
return cost(C) = min
```