Weighted Graphs

Many networks have a cost associated with each arc. Such costs are often referred to as weights.

A weighted graph is a simple graph G together with a weight function $W : arcs(G) \rightarrow R^+$

Minimum Spanning Trees

When dealing with weighted graphs we would like to find a minimum spanning tree: a spanning tree where the sum of the weights of its arcs is as small as possible.

Let G be a weighted graph. The weight of a spanning tree T for G is the sum of the weights of the arcs of T. T is a minimum spanning tree (MST) for G if T is a spanning tree for G and no other spanning tree for G has smaller weight.

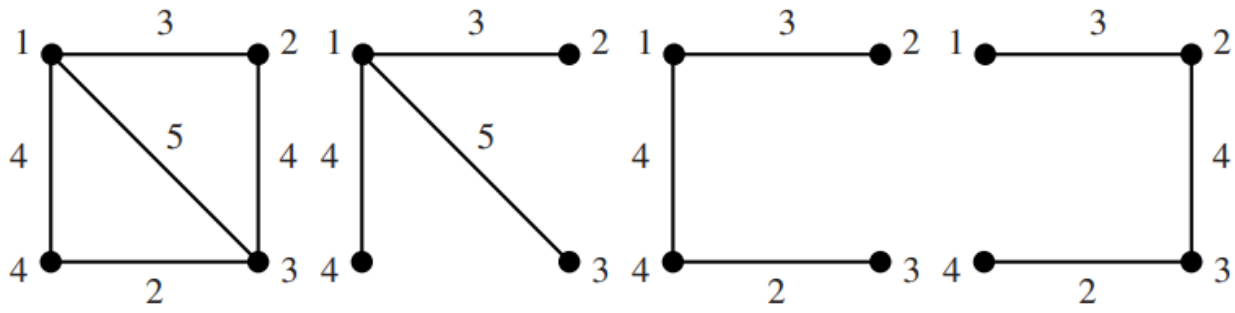


Figure 1.21: Minimum and non-minimum spanning trees

Consider the graph drawn on the left. We show three spanning trees. The first has a weight of 12, while the other two both have a weight of 9. The last two clearly have minimum weight, since any spanning tree must have three arcs, and they have selected the three arcs of least weight.

Minimum spanning trees are not necessarily unique.

Prim’s Algorithm

Suppose we want to grow an MST starting from a root node. At each stage we could add the shortest arc which will extend the tree. This is the “greedy” approach, where we do what gives a short-term advantage, even if it may not be the best overall.

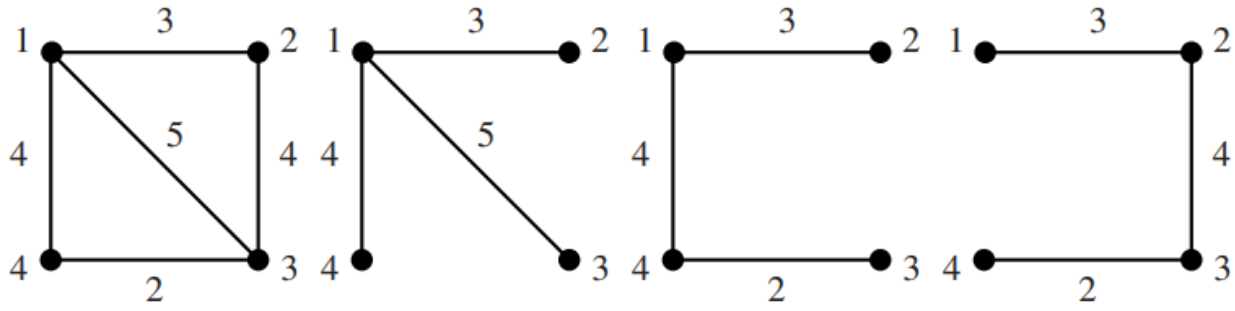


Figure 1.21: Minimum and non-minimum spanning trees

Let us apply this approach to this graph, choosing 1 as our start node. Since the arc to 2 is the shortest, we add it to our tree. Now we look for the shortest arc which joins either 1 or 2 to one of the remaining nodes. We can choose either the arc (1,4) or the arc (2,3). Suppose we choose (1,4). We now look for the shortest arc which joins node 3 to the tree. This will be the arc (4,3). We end up with the first of the two MSTs in Figure 1.21.

This greedy algorithm is due to Prim.

At an arbitrary stage in Prim’s MST algorithm, there are three kinds of nodes:

- Those which are in the tree constructed so far (tree nodes)
- Those which are candidates to join at the next stage (fringe nodes)
- The rest (unseen nodes)

Initially all nodes are unseen.

```
Choose any node start as the root
Reclassify start as tree
Reclassify all nodes adjacent to start as fringe
while fringe nonempty:
        Select an arc of minimum weight between a tree node t and a
fringe node f (*)
        Reclassify f as tree
```

```
        Add arc (t, f) to the tree
        Reclassify all unseen nodes adjacent to f as fringe
```

Each time the while loop is executed another node is added to the tree. Hence the while loop is executed $O(n)$ times.

The line marked (*) involves finding the shortest arc among all possible arcs between $O(n)$ tree nodes and $O(n)$ fringe nodes. This is therefore $O(n + m)$, which makes the whole algorithm $O(n(n + m))$.

We can improve the performance of the algorithm if we keep track of which arcs might be used.

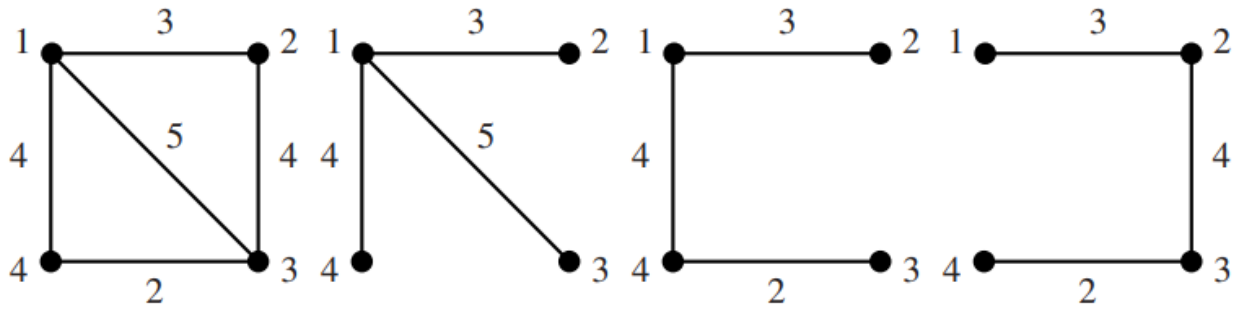


Figure 1.21: Minimum and non-minimum spanning trees

Consider this graph when the tree so far is just the arc (1,2). The fringe nodes are 3 and 4. The arc (1,3) of weight 5 was an option before 2 was added, but now it is longer than (2,3) and therefore no longer a candidate. The candidate arcs are (1,4) and (2,3).

It is convenient to look at this using the parent array we adopted for graph traversal. We let the parent of a node f in the fringe be the node in the tree t such that (t, f) has least weight. So initially in our example $parent[3] = 1$, but after 2 is added we change to $parent[3] = 2$, making (2,3) a candidate arc.

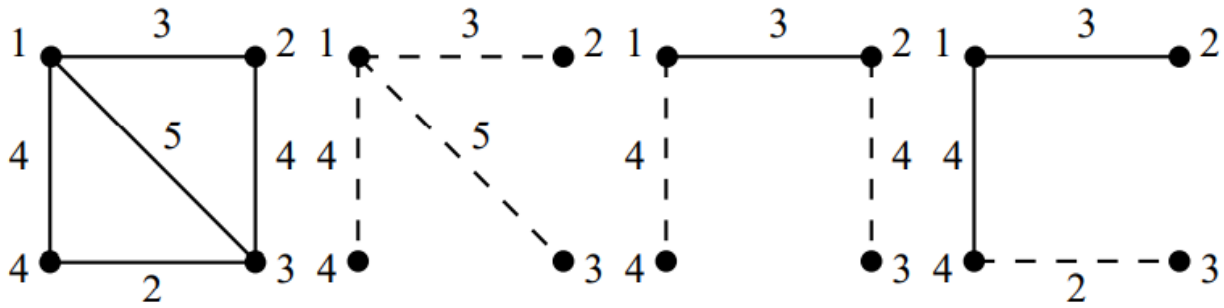


Figure 1.22: Candidate arcs (dashed lines)

This leads us to recast Prim's Algorithm to:

```
Choose any node start as the root
tree[start] = true
for x in adj[start]:
  # add x to fringe
  fringe[x] = true
  parent[x] = start
  weight[x] = W[start, x]
while fringe nonempty:
  Select a fringe node f such that weight[f] is minimum
  fringe[f] = false
  tree[f] = true
  for y in adj[f]:
    if not tree[y]:
      if fringe[y]:
        # update candidate arc
        if W[f, y] < weight[y]:
          weight[y] = W[f, y]
          parent[y] = f
      else:
        # y is unseen
            fringe[y] = true
        weight[y] = W[f, y]
        parent[y] = f
```

Clearly we can obtain the weight of the MST by just adding the weights of the nodes (apart from `start`).

Analysing this:

- As before there are $\mathrm{O}(\mathrm{n})$ executions of the while loop
- Testing whether the fringe is empty is $\mathrm{O}(\mathrm{n})$
- Finding the fringe node f such that $\mathrm{weight}[\mathrm{f}]$ is minimum is $\mathrm{O}(\mathrm{n})$
- The updating of the candidate arc for each y in $\mathrm{adj}[\mathrm{f}]$ is $\mathrm{O}(1)$
- Hence the for loop is $\mathrm{O}(\mathrm{n})$
- Therefore, the algorithm is $\mathrm{O}(\mathrm{n}^2)$
- Better than before: $\mathrm{O}(\mathrm{n}(\mathrm{n}+\mathrm{m}))$

Let G be a connected weighted graph. Then Prim's algorithm constructs an MST for G.

Proof. For simplicity we concentrate on the schematic version of the algorithm (page 29) rather than the optimised version of Figure 1.23.

Let G have n nodes. Each stage of the algorithm adds an arc to the subgraph constructed so far. Let the subgraphs constructed at each stage be $T_0, \ldots, T_k, \ldots$. We start with T_0 having just the node start. Let T_{k+1} be got from T_k by adding arc a_{k+1}. Since a new node from the fringe is added at each stage, clearly T_k has $k+1$ nodes, and so there are $n-1$ stages, with T_{n-1} being returned by the algorithm.

We shall show by induction on k that each T_k is a subgraph of an MST T' of G.

Base case $k = 0$. T_0 has one node and no arcs. Clearly $T_0 \subseteq T'$ for any MST T' of G.

Induction step. Assume that $T_k \subseteq T'$, some MST T' of G. Let the new arc a_{k+1} join node x of T_k to a new fringe node y not in nodes(T_k).

If $a_{k+1} \in \text{arcs}(T')$ then $T_{k+1} \subseteq T'$ as required. So suppose $a_{k+1} \notin \text{arcs}(T')$. Since T' is a spanning tree, there must be a path P in T' from x to y. So $P \cup \{a_{k+1}\}$ forms a cycle. There must be an arc a in P which joins a node x' of T_k to a fringe node y' not in nodes(T_k). See Figure 1.24. We can form a new spanning tree T'' from T' by removing a and adding a_{k+1}. Since the algorithm chose a_{k+1} rather than a, we have $W(a_{k+1}) \leq W(a)$. Hence $W(T'') \leq W(T')$ and so T'' is an MST (since all MSTs have the same weight, we must actually have $W(a_{k+1}) = W(a)$). Also, $T_{k+1} \subseteq T''$ as required.

Now T_{n-1} has $n-1$ arcs, and $T_{n-1} \subseteq T'$ for some MST T'. Since all spanning trees for G have $n-1$ arcs, we must have $T_{n-1} = T'$. Hence T_{n-1} is an MST, as required.

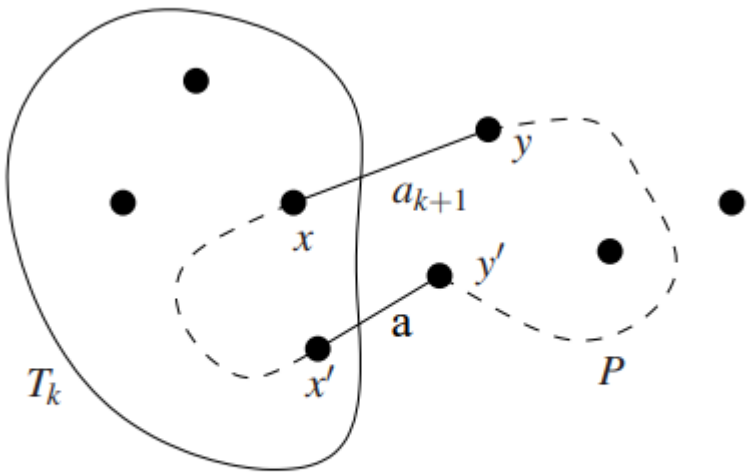


Figure 1.24: Correctness of Prim's algorithm

For a sometimes more efficient version of Prim's algorithm we can use priority queues. Unlike ordinary FIFO queues, each item x of the queue has a priority key[x] (usually a natural number). In a min PQ, as considered here, this key represents cost. Items are removed from the queue lowest key first.

The PG abstract data type has the following operations:

- `Q = PQcreate()` : create an empty PQ
- `isEmpty(Q)` : tests whether empty
- `insert(Q, x)` : add a new item x to the queue. It will have a key value `key[x]`
- `getMin(Q)` : returns an item with lowest key value
- `deleteMin(Q)` : deletes the item returned by `getMin(Q)`
- `decreaseKey(Q,x,newkey)` : updates `key[x] = newkey`

We can use a priority queue to store the candidate arcs in order, by storing the fringe nodes and their associated parents and using the weight of the candidate arc as the key. It is convenient to also place the unseen nodes in the queue, with key ∞, which can be taken to be any number greater than the weight of any arc.

```
Q = PQcreate()
for x in Nodes(G):
  key[x] = ∞;
  parent[x] = nil
  insert(Q, x)
decreaseKey(Q,start,0)
while not isEmpty(Q):
  f = getMin(Q); deleteMin(Q)
  tree[f] = true
  for y in adj[f]:
    if not tree[y]: # so y in Q
      if W[f, y] < key[y]:
        decreaseKey(Q, y,W[f, y])
        parent[y] = f
```

With n nodes and m arcs the number of PQ operations in the above algorithm is:

- insert: $O(n)$
- isEmpty: $O(n)$

- getMin: $O(n)$
- deleteMin: $O(n)$
- decreaseKey: $O(m)$

For a PQ of length N all operations are $\log N$, apart from `isEmpty` and `getMin` which are $O(1)$.

So Prim with PQ has overall complexity $O(m \log n)$, assuming that $n < m$ as is usually the case.

If the graph is sparse, say $m \leq n \log n$, we have:

$O(m \log n) = O(n \log^2 n)$, which is better than $O(n^2)$, so Prim with PQ is better.

On the other hand, if the graph is dense, with $O(n^2)$ arcs, we have $O(m \log n) = O(n^2 \log n)$, which is worse than $O(n^2)$. So classic Prim is better.

Kruskal's Algorithm

An even greedier strategy.

At each stage choose the shortest arc not yet included, except when this would give a cycle.

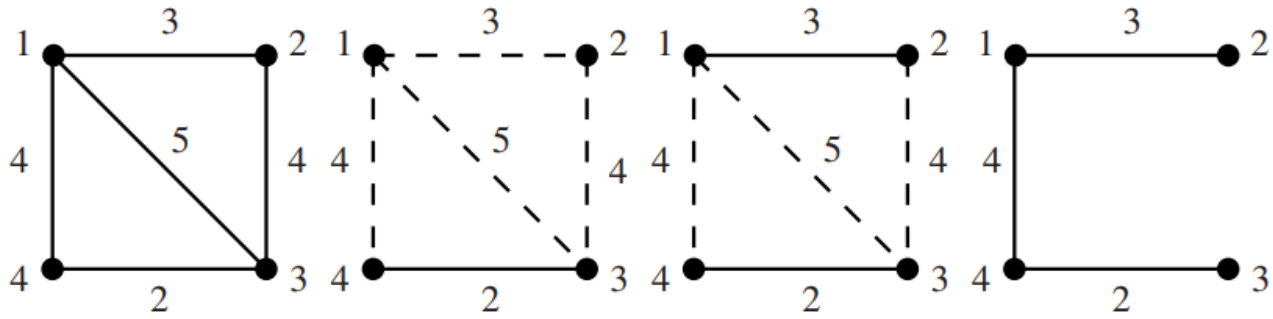


Figure 1.25: MST constructed by Kruskal's algorithm

First we choose (3,4) as it has the least weight. Then we choose (1,2). This shows that during the construction process we have a forest (i.e. an acyclic graph) rather than a tree, since the graph need not be connected. Finally we choose one of (1,4) or (2,3). The process is finished, since adding any other arc would give a cycle.

```
F = Empty Set # forest being constructed
R = arcs(G) # remaining arcs
while R nonempty:
  remove a of smallest weight from R
  if a does not make a cycle when added to F:
    add a to F
return F
```

Notice that arcs are added in increasing order of weight. So a good implementation strategy could be to work on a list of arcs sorted by weight in increasing order.

Let G be a connected weighted graph. Then Kruskal's algorithm constructs an MST for G.

Proof. Our strategy is much the same as for Prim's algorithm (Theorem 1.7.3). Let G have n nodes. Each stage of the algorithm adds an arc to the subgraph constructed so far. Let the subgraphs constructed at each stage be $F_0, \ldots, F_k, \ldots$. We start with F_0 empty. Let F_{k+1} be got from F_k by adding arc a_{k+1}. So each F_k has k arcs.

Clearly there should be $n-1$ stages, since any spanning tree must have $n-1$ arcs. But we need to check this, since the algorithm executes the while loop as long as it can add a new arc without creating a cycle. If there is any unused node or F_k is not connected then a further stage is possible. Any connected acyclic graph with n nodes must have $n-1$ arcs (Proposition 1.4.2). Hence there are at least $n-1$ stages. Since we then have a spanning tree we stop at stage $n-1$, with F_{n-1} being returned by the algorithm.

We shall show by induction on k that each F_k is a subgraph of an MST T' of G.

Base case $k = 0$. F_0 is empty. Clearly $F_0 \subseteq T'$ for any MST T' of G.

Induction step. Assume that $F_k \subseteq T'$, some MST T' of G. Let the new arc a_{k+1} join node x to node y. If $a_{k+1} \in \text{arcs}(T')$ then $T_{k+1} \subseteq T'$ as required. So suppose $a_{k+1} \notin \text{arcs}(T')$. Since T' is a spanning tree, there must be a path P in T' from x to y. So $P \cup \{a_{k+1}\}$ forms a cycle. There must be an arc a in P which does not belong to F_k, since otherwise the algorithm could not add a_{k+1} to F_k as it would form a cycle. Suppose that a joins x' to y'. There cannot be a path from x' to y' in F_k or else there would be a cycle in T' (recall that $F_k \subseteq T_k$). Hence a is a candidate for the algorithm to add to F_k. Since the algorithm chose a_{k+1} rather than a, we must have $W(a_{k+1}) \leq W(a)$. We can form a new spanning tree T'' from T' by removing a and adding a_{k+1}. We have $W(T'') \leq W(T')$ and so T'' is an MST (since all MSTs have the same weight, we must actually have $W(a_{k+1}) = W(a)$). Also, $T_{k+1} \subseteq T''$ as required.

Now T_{n-1} has $n-1$ arcs, and $T_{n-1} \subseteq T'$ for some MST T'. Since all spanning trees for G have $n-1$ arcs, we must have $T_{n-1} = T'$. Hence T_{n-1} is an MST, as required.

For the implementation of Kruskal's algorithm, we have to:

1. Look at each arc in ascending order of weight. We can use a priority queue here (or just sort the arcs at the start).
2. Check whether adding the arc to the forest so far creates a cycle.
    1. Here we use dynamic equivalence classes.
    2. Put nodes in the same equivalence class if they belong to the same connected component of the forest constructed so far. Map each node to the representative of its equivalence class. An arc (x, y) can be added if x and y belong to different equivalence classes. If (x, y) is added, then merge the equivalence classes of x and y

Dynamic equivalence classes can be handled using the Union-Find data type. Each set has a leader element which is the representative of that set.

- find: find the leader of the equivalence class
- union: merge two classes

Operations:

- `sets = UFcreate(n)`: creates a family of singleton sets $\{1\}, \{2\}, \ldots, \{n\}$ with `find(sets,x) = x`
- `x' = find(sets,x)`: finds the leader x' of x within sets
- `union(sets,x, y)`: merge the sets led by x and y and use one of x or y as the new leader
- NB x and y must be the leaders of their sets, so that `x = find(sets,x)` and `y = find(sets,y)`

```
Let G have n nodes numbered from 1 to n
Build a priority queue Q of the edges of G with the weights as keys
sets = UFcreate(n) # initialise Union-Find with singletons {1},...,
{n}
F = Empty Set # forest being constructed
while not isEmpty(Q):
  (x, y) = getMin(Q); deleteMin(Q)
  x' = find(sets, x); y' = find(sets, y)
  if x' != y': # no cycle
    add (x, y) to F
    union(sets, x', y') # merge the two components
```

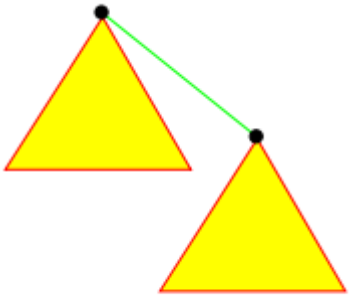
We now consider how to implement union-find. A naïve implementation would be as follows:

Maintain an array leader of nodes:

- $\mathrm{leader}[x]$ stores the leader of the set to which node x belongs
- Initially $\mathrm{leader}[x] = x$ for all nodes.

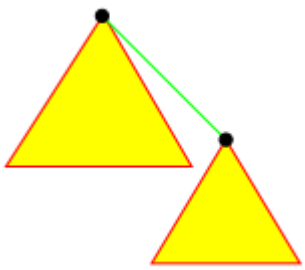
Find is now $O(1)$. However union takes $O(n)$. That means that it takes $O(n^2)$ to perform the $O(n)$ unions required for Kruskal.

Instead, each set is stored as a tree. The root node is the leader of the set. We merge two sets by appending one tree to the other, so that the root of one tree is the child of the root of the other tree.



To store the tree structure, for each node x we maintain $\mathrm{parent}[x]$, where $\mathrm{parent}[x] = x$ if x is the root (the leader).

Initially $\mathrm{parent}[x] = x$ for any node x. Then $\mathrm{union}(sets, x, y)$ just involves setting $\mathrm{parent}[y] = x$ (or vice versa) — constant time $O(1)$. However find involves following $\mathrm{parent}[x]$ up to the root. The time taken is bounded by the depth of the tree. With naïve merging of trees as above, this can be as much as n, so that find becomes $O(n)$. We clearly need to keep the depth as low as possible. A possible strategy is to use weighted union: since we can append trees in either order, always append the tree of lower size to the one of greater size.



This requires us to store the size of the tree and update this — easy to do.

Using weighted union, the depth of a tree of size k is $\leq \lfloor \log k \rfloor$.

Proof. Proved by (strong) induction on k.

For $k = 1$ depth is $0 = \lfloor \log 1 \rfloor$.

Suppose true for all $k' < k$. Let tree T of size k and depth d be got from T_1 of size k_1 and depth d_1 and T_2 of size k_2 and depth d_2.

Suppose that $k_2 \leq k_1$ and T_2 is appended to T_1 to form T. By induction hypothesis, $d_1 \leq \lfloor \log k_1 \rfloor$ and $d_2 \leq \lfloor \log k_2 \rfloor$. Also $d = \max(d_1, d_2 + 1)$.

Now $k_1 \leq k$. So $d_1 \leq \lfloor \log k \rfloor$. Also $k_2 \leq k/2$. So $d_2 \leq \lfloor \log k/2 \rfloor = \lfloor \log k \rfloor - 1$.

Combining: $d \leq \lfloor \log k \rfloor$ as required.

Using the Lemma, with weighted union, each find takes $O(\log n)$, and each union takes $O(1)$.

For Kruskal, there will be:

- $O(m)$ inserts to build the PQ: time taken $O(m \log m)$
- $O(m)$ `getMins` and $O(m)$ `deleteMins`: time taken $O(m \log m)$
- $O(m)$ finds: time taken $O(m \log n)$
- $O(n)$ unions: time taken $O(n)$

So overall time taken is $O(m \log m)$ (assuming $m \geq n$, as is normally the case).

The number of arcs m is bounded by n^2. So $O(m \log m) = O(m \log n)$. Hence the overall complexity for Kruskal is $O(m \log n)$. This is the same as Prim with priority queues.

We can improve union-find using path compression. The complexity for the union-find part of Kruskal then reduces to

$$O((n + m) \log^* n)$$

Where $\log^* n$ is an extremely slow-growing function.

Path compression works as follows: when finding the root (leader) for a node x, if this is not $parent[x]$ then make $parent[y] = root$ for all y on the path from x to the root.

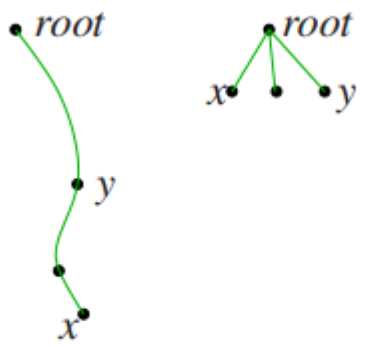


Figure 1.26: Path Compression.

We have extra work in updating parent, but we keep the depth of the nodes lower so that future finds are faster.

This procedure combines well with weighted union (on size) since size is unchanged by path compression.

Normal find:

```
Algorithm proc find(x)::
    y = parent[x]
    if y == x:      # x is the root
        root = x
    else:
        root = find(y)
    return root
```

We can modify to perform the path compression:

```
Algorithm proc cfind(x)::
    y = parent[x]
    if y == x:      # x is the root
        root = x
    else:
        root = cfind(y)
        if root != y:
            parent[x] = root
    return root
```

Comparing the various algorithms for finding MSTs:

- Kruskal: O(mlogn)
- Prim with PQ (binary heap): O(mlogn)
- Classic Prim: $O(n^2)$

Which is better? As when comparing classic Prim and Prim with PQ:

- On dense graphs where m is large ($order\ n^2$) then Kruskal gives $O(n^2 \log n)$ and classic Prim is best
- On sparse graphs, where m is small (say $O(n \log n)$), then Kruskal (or Prim with PQ) give better results than classic Prim:

  $$O(m \log n) = O(n \log^2 n)$$

  Priority queues can also be implemented with Fibonacci heaps rather than binary heaps. All operations are $O(1)$ apart from `deleteMin`, which is $O(\log n)$.

Complexity of Prim with PQ (Fibonacci heap):

$$O(m + n \log n)$$

In practice the memory usage and constant factors can be high.

The Shortest Path Problem

Given a weighted graph (G, W), and two nodes *start* and *finish*, find the shortest path from *start* to *finish* in the graph.

Different to the **All Pairs Shortest Path Problem**: Given a weighted graph G, find the shortest paths between all pairs of nodes of G.

Dijkstra's algorithm ($O(n^2)$ time)

Call the two nodes start and finish. Build up a spanning tree starting from the start node. We classify nodes into:

- tree nodes: already included
- fringe nodes: not in the tree yet, but adjacent to a tree node
- unseen nodes: the rest

The new idea in the shortest path algorithm is that we have already computed the shortest path from start to all the tree nodes: it is the path given by the tree.

For the fringe nodes, we know the shortest path using the tree constructed so far. This path might be improved as the tree grows.

We know nothing about the shortest path to an unseen node. We store two values for each tree or fringe node: its parent node in the tree, and the length of the shortest path known.

At each stage the next node to be added is the fringe node with the smallest length. We can obtain the shortest path to a node x in reverse order from the parent function:

$$x, \mathrm{parent}[x], \mathrm{parent}[\mathrm{parent}[x]], \ldots, \mathrm{start}$$

Let $\mathrm{path}(x)$ denote this path in the forwards direction:
$\mathrm{start}, \ldots, \mathrm{parent}[\mathrm{parent}[x]], \mathrm{parent}[x]$

This is defined for all tree or fringe nodes.

Dijkstra’s algorithm:

Algorithm Dijkstra's Shortest Path Algorithm:
Input: Weighted graph (G, W) together with a pair of nodes start, finish
Output: Length of shortest path from start to finish

```
tree[start] = true
for x in adj[start]:
    # add x to fringe
    fringe[x] = true
    parent[x] = start
    distance[x] = W[start, x]
while not tree[finish] and fringe nonempty:
    Select a fringe node f such that distance[f] is minimum
    fringe[f] = false
    tree[f] = true
    for y in adj[f]:
        if not tree[y]:
            if fringe[y]:
                # update distance and candidate arc
                if distance[f] + W[f, y] < distance[y]:
                    distance[y] = distance[f] + W[f, y]
                    parent[y] = f
            else:
                # y is unseen
                fringe[y] = true
                distance[y] = distance[f] + W[f, y]
                parent[y] = f
return distance[finish]
```

The algorithm does not need to complete computing the entire spanning tree, since it can stop as soon as *finish* joins the tree. When the algorithm terminates, the length of the shortest path is distance[finish] and we can read off the path through the tree using the parent function.

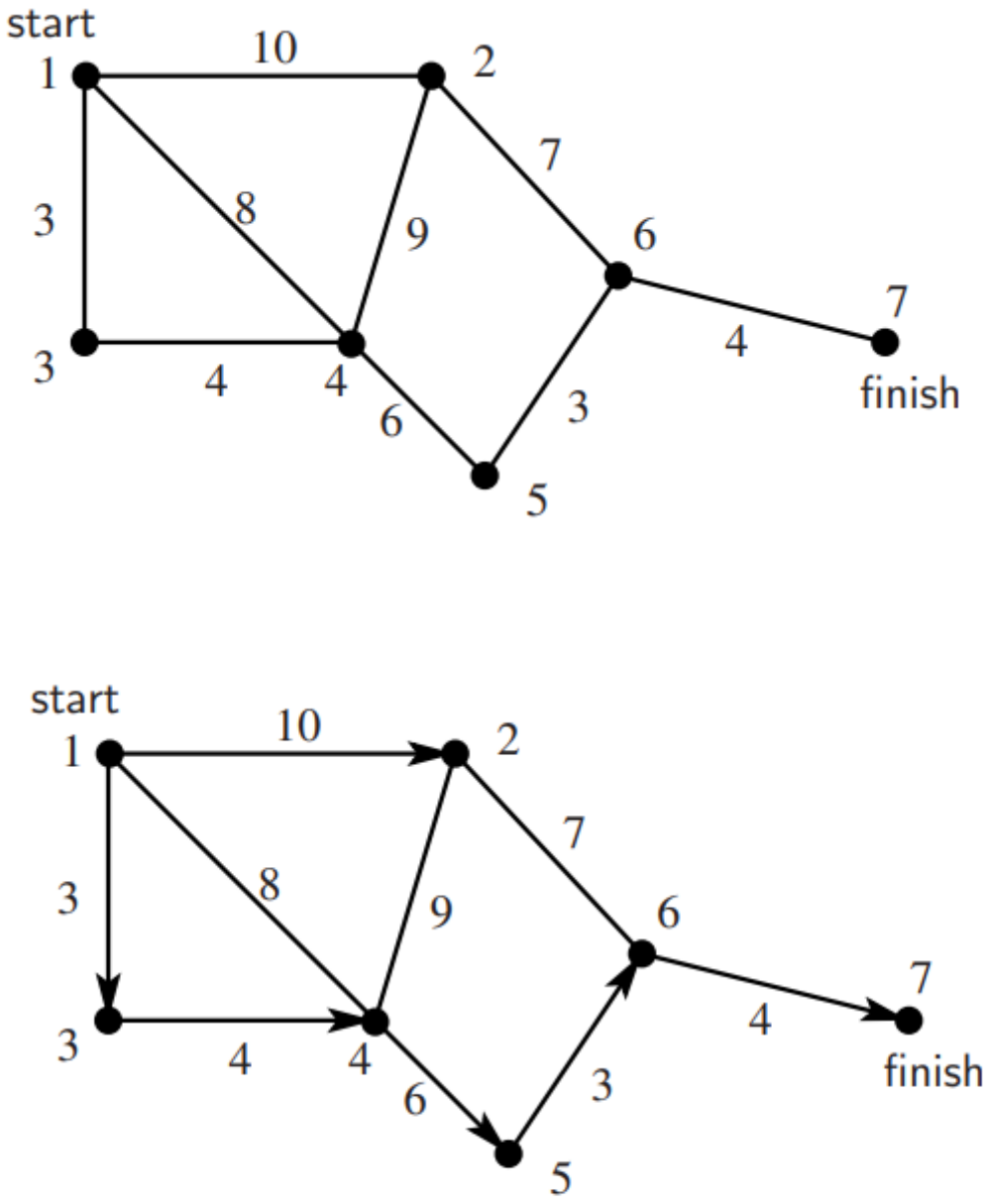


Figure 1.29: Solution to the shortest path problem

In this case we got a spanning tree for the entire graph, since the finish node was the last to be added to the tree. It is not a minimum spanning tree.

The algorithm will work for directed graphs.

The algorithm terminates, since we clearly increase the tree each time we execute the while loop. To see why the algorithm is correct we need to formulate an invariant. It has three conditions:

- If x is a tree or fringe node (other than start) then parent[x] is a tree node

- If x is a tree node (other than start) then distance[x] is the length of the shortest path, and parent[x] is its predecessor along that path.
- The fringe nodes are precisely those non-tree nodes f reachable from start by a path in the tree (except for the final arc to f).
- If f is a fringe node then distance[f] is the length of the shortest path where all nodes except f are tree nodes. Furthermore, parent[f] is its predecessor along that path.

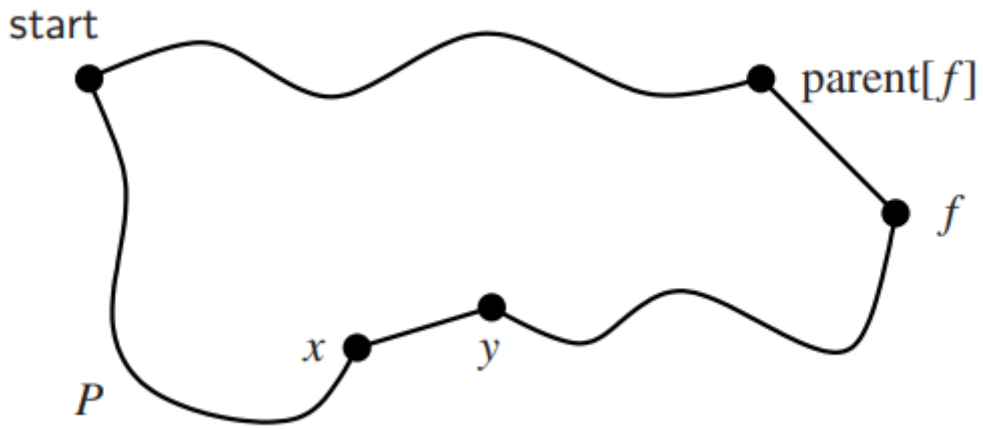
When the program terminates, finish is a tree node, and by (2) we then have the required shortest path.

It is easy to see that the initialisation establishes the invariant. It is also easy to see that (1) is always true. It guarantees that when we find a shortest path it will lie wholly inside the tree. The main work is in seeing that parts (2) and (3) of the invariant are maintained by the while loop.

To see that (2) is maintained when f is added to the tree, we need to check that distance[f] is the length of the shortest path, and that parent[f] is the predecessor along that path. Let path(f) be the path given by the algorithm:

$$\mathrm{start}, \ldots, \mathrm{parent[f]}, \mathrm{f}$$

Suppose we have a different and shorter path P (not necessarily in the tree). Let x be the last node on P to belong to the tree, and let y be the next node along P (possibly f itself). In the diagram, $\mathrm{path(f)}$ is along the top and P is along the bottom:



Now by (3) we know that $\mathrm{distance[y]} \leq \mathrm{the\,distance\,from\,start\,to\,y\,using\,P}$. Hence the $\mathrm{length\,of\,P} \geq \mathrm{distance[y]}$. But $\mathrm{distance[y]} \geq \mathrm{distance[f]}$ by our choice of f .

Hence P is at least as long as $\mathrm{path(f)}$. We leave the proof that (3) is maintained as a (not so easy) exercise.

If (G,W) is a connected weighted graph with nodes *start* and *finish*, then Dijkstra's algorithm finds the shortest path from *start* to *finish*.

We can implement Dijkstra's algorithm using priority queues:

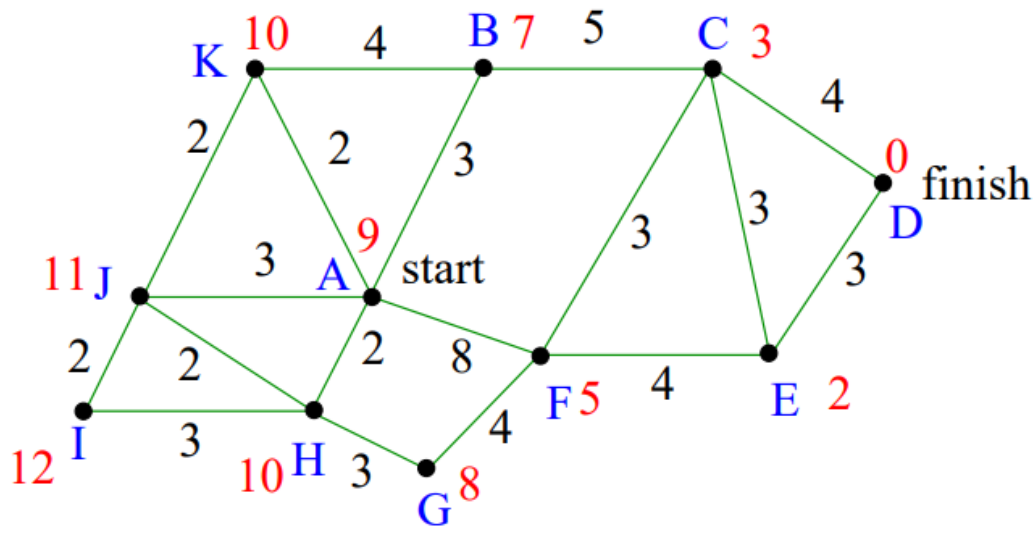
```
Q = PQcreate()
for x in Nodes(G):
        key[x] = ∞;
        parent[x] = nil
        insert(Q, x)
decreaseKey(Q,start,0)
while not tree[finish] and not isEmpty(Q):
        f = getMin(Q); deleteMin(Q)
        tree[f] = true
        for y in adj[f]:
                if not tree[y]: # so y in Q
                        if key[f]+W[f, y] < key[y]:
                                decreaseKey(Q, y, key[f] +W[f, y])
                                parent[y] = f
```

- Dijkstra with PQ (binary heap) is overall $\mathrm{O(m \log n)}$ assuming that $\mathrm{n < m}$ as is usually the case.
- Dijkstra with PQ (Fibonacci heap) is overall $\mathrm{O(m + n \log n)}$

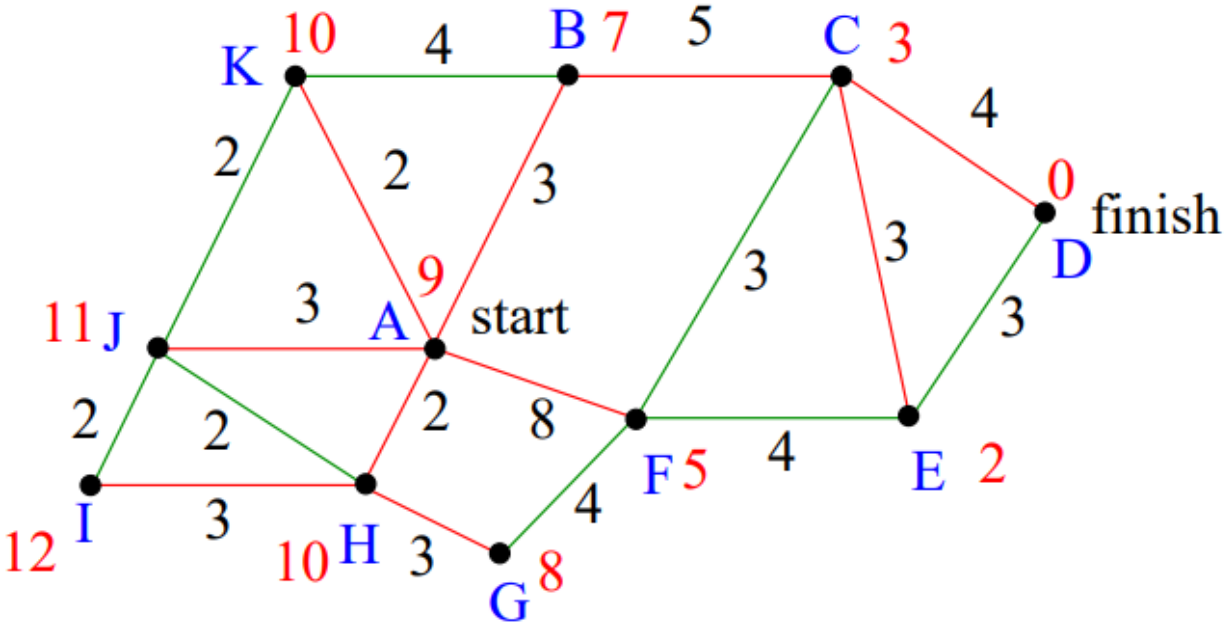
A* algorithm

We assume that we have a heuristic function $\mathrm{h(x)}$ which **underestimates** the distance from any node x to the finish node.

Example 1.7.11 *Each node x has heuristic value h(x) shown in red.*

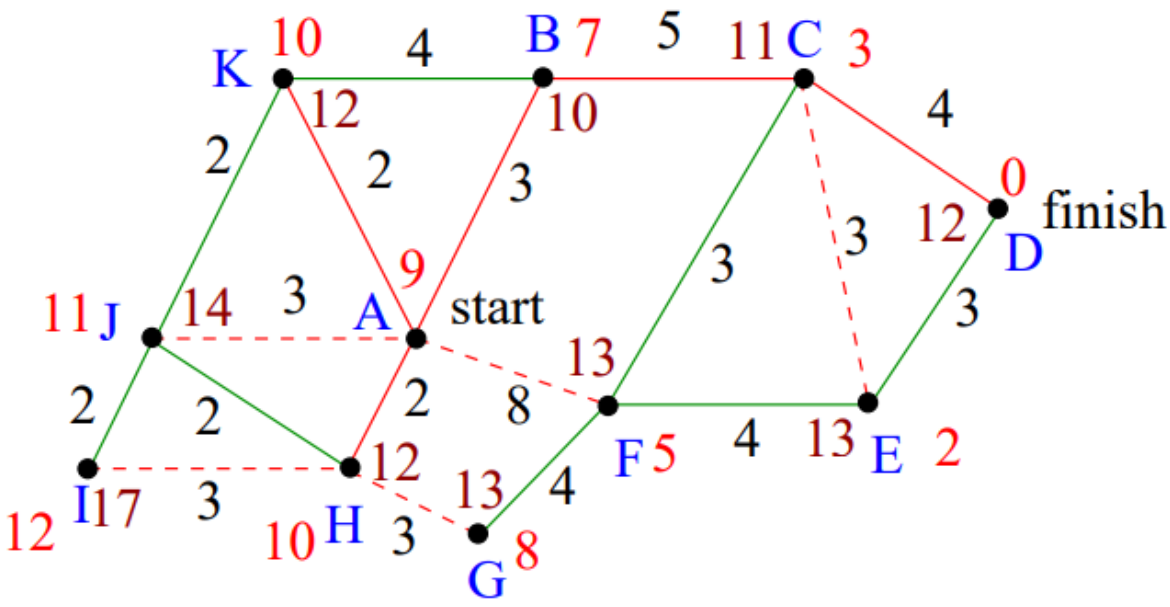


Using Dijkstra's algorithm we have to compute the entire shortest path tree before reaching node D.



We had to find the shortest paths to all the nodes which are closer to start than D.

With A*:



A heuristic function is consistent if

- for any adjacent nodes x, y we have $h(x) \leq W(x, y) + h(y)$
- h(finish) = 0

This is satisfied by the Euclidean distance heuristic function if $W(x, y)$ represents distances on a map.

Let h be a consistent heuristic function, and let P be a path from node x to node y. Then $h(x) \leq \text{len}(P) + h(y)$

A heuristic function is admissible if for any node x we have $h(x) \leq$ the weight of the shortest path from x to the goal finish.

If h is consistent then h is admissible.

Algorithm A* algorithm:
Input: Weighted graph (G, W) together with a pair of nodes start, finish and consistent heuristic function h
Output: Length of shortest path from start to finish

```
tree[start] = true
g[start] = 0
f[start] = g[start] + h[start]
for x in adj[start]:
    # add x to fringe
    fringe[x] = true
    parent[x] = start
    g[x] = W[start, x]
    f[x] = g[x] + h[x]
while finish not a tree node and fringe non-empty:
    Select a fringe node x s.t. f[x] is minimum
    fringe[x] = false
    tree[x] = true
    for y in adj[x]:
        if not tree[y]:
            if fringe[y]:    # update g(y), f(y) and candidate arc
                if g[x] + W[x,y] < g[y]:
                    g[y] = g[x] + W[x,y]
                    f[y] = g[y] + h[y]
                    parent[y] = x
            else:    # y is unseen
                fringe[y] = true
                g[y] = g[x] + W[x,y]
                f[y] = g[y] + h[y]
                parent[y] = x
return g[finish]
```

- The set of tree nodes is often called the closed set and the set of fringe nodes is the open set
- If we set $h(x) = 0$ for all nodes x then h is consistent, and the A* algorithm is just Dijkstra's algorithm
- We deduce that the running time for A* is the same as for Dijkstra in the worst case, though we hope to do better on average, depending on h
- We have presented A* for consistent heuristics, for simplicity and for its closeness to Dijkstra's algorithm
- There is a more general version of A* which is guaranteed to give the correct solution for admissible heuristics.

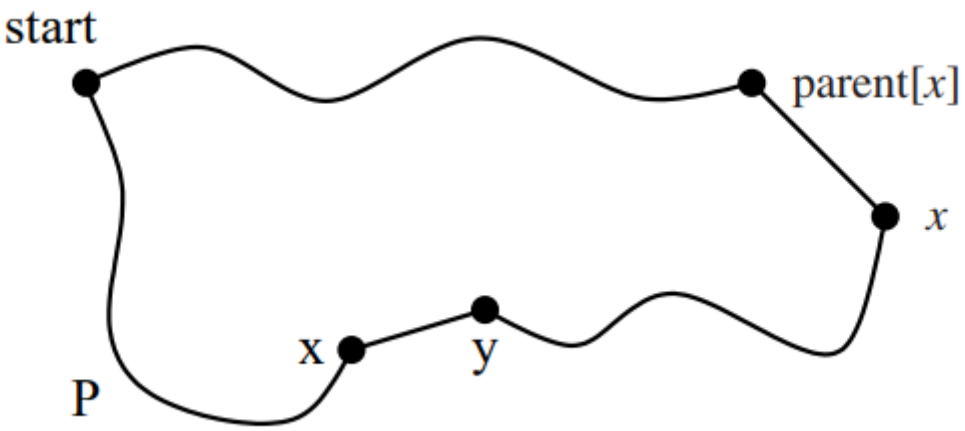
- The difference is that we may have to re-examine nodes that are already in the closed set (the tree).

Proof that A* correctly computes the shortest path:

- The algorithm terminates, since we clearly increase the tree each time we execute the while loop.
- To see why the algorithm is correct we need to formulate an invariant

Invariant:

- If x is a tree or fringe node (other than *start*) then $\mathrm{parent}[\mathrm{x}]$ is a tree node
- If x is a tree node (other than start) then $\mathrm{g}[\mathrm{x}]$ is the length of the shortest path, and $\mathrm{parent}[\mathrm{x}]$ is its predecessor along that path
- If x is a fringe node then $\mathrm{g}[\mathrm{x}]$ is the length of the shortest path where all nodes except x are tree nodes
- Furthermore, $\mathrm{parent}[\mathrm{x}]$ is its predecessor along that path

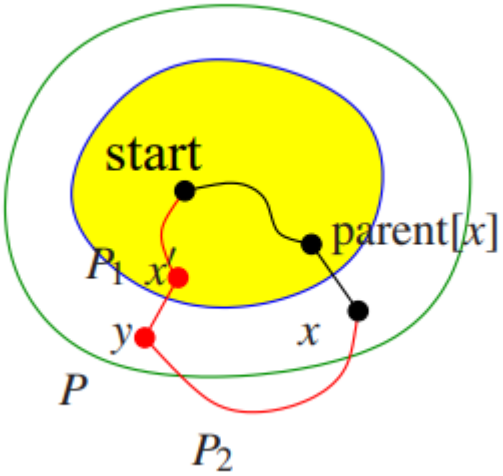


When the program terminates, *finish* is a tree node, and by (2) we then have the required shortest path.

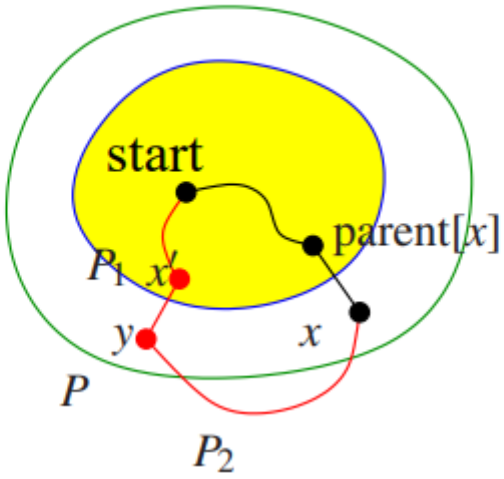
So it remains to show that the invariant is

- established before the while loop
- maintained during the while loop

We just show that (2) is maintained, and omit the rest of the proof. Suppose x is added to the tree. We need to check that we have found the shortest path. The path given by the algorithm: $\mathrm{start}, \ldots, \mathrm{parent}[\mathrm{x}], \mathrm{x}$ has length $\mathrm{g}[\mathrm{x}]$. Suppose we have a different and shorter path P (not necessarily in the tree).



Then $\mathrm{len(P)} < \mathrm{g[x]}$. Let y be the first node on P not to belong to the tree. Let $\mathrm{P_1}$ be P from start to y, and $\mathrm{P_2}$ be P from y to x.



$$
\begin{array}{lcll}
f[y] & = & g[y]+h[y] & \\
 & \leq & g[y]+\mathrm{len}(P_2)+h[x] & \text{consistency of } h \\
 & \leq & \mathrm{len}(P_1)+\mathrm{len}(P_2)+h[x] & \text{by (3) for } y \\
 & = & \mathrm{len}(P)+h[x] & \\
 & < & g[x]+h[x] & \text{assumption} \\
 & = & f[x] &
\end{array}
$$

But $\mathrm{f[x]} \leq \mathrm{f[y]}$ by our choice of x. Contradiction.

A* algorithm with priority queues:

```
Q = PQcreate()
for x in nodes(G):
```

```
        g[x] = ∞; key[x] = ∞; parent[x] = nil
        insert(Q, x)
g[start] = 0; decreaseKey(Q, start, g[start] + h[start]) while not
tree[finish] and not isEmpty(Q):
        x = getMin(Q); deleteMin(Q)
        tree[x] = true
        for y in adj[x]:
                if not tree[y]: # so y in Q
                        if g[x] + W[x, y] < g[y]:
                                g[y] = g[x] +W[x, y];
decreaseKey(Q, y,g[y] +h[y])
                                parent[y] = x
```

Warshall's Algorithm

Definition 1.7.14 *Let $R \subseteq X^2$ be a binary relation.*

1. *We say that R is* transitive *if for all $x, y, z \in X$, if $R(x,y)$ and $R(y,z)$ then $R(x,z)$ holds.*
2. *The* transitive closure *of R, denoted R^+, is the smallest relation $S \subseteq X^2$ such that $R \subseteq S$ and S is transitive.*

If R is transitive then $\mathrm{R}^+ = \mathrm{R}$.

An equivalent definition is that R^+ is the intersection of all transitive relations on X which include R as a subset.

The intersection of a family of transitive relations is also transitive.

An alternative characterisation of R being transitive is that $\mathrm{R} \circ \mathrm{R} \subseteq \mathrm{R}$ where $\circ$ is relational composition.

Let R^k be $R \circ \cdots \circ R$ (k times). Then we can show that

$$R^+ = \bigcup_{k=1}^{\infty} R^k$$

If X is finite and $|X| = n$ then

$$R^+ = \bigcup_{k=1}^{n} R^k$$

We can interpret R as a directed graph G. The nodes of G are just the members of X, and there is an arc from x to y iff $\mathrm{R}(\mathrm{x}, \mathrm{y})$. Note that there are no parallel arcs; we could have loops where $\mathrm{R}(\mathrm{x}, \mathrm{x})$.

It is easy to see that $\mathrm{R}^{\mathrm{k}}(\mathrm{x}, \mathrm{y})$ iff there is a path of length k from x to y. So $\mathrm{R}^{+}(\mathrm{x}, \mathrm{y})$ iff there is a path of length ≥ 1 from x to y.

Suppose $X = \{1, \ldots, n\}$. Clearly if we set

$$A[i, j] = \begin{cases} 1 & \text{if } R(i, j) \\ 0 & \text{otherwise} \end{cases}$$

then A is the adjacency matrix of G.

We can compute R^{k} using matrix multiplication:

$R^k(i, j)$ iff $A^k[i, j] > 0$

Let:

$$B = \sum_{k=1}^{n} A^k$$

Then:

$$R^+(i,j) \text{ iff } B[i,j] > 0$$

So far we have been building up the transitive closure by looking for paths of length k for $k = 1, \ldots, n$. We now look at a quite different and more efficient method.

Suppose that the nodes are $\{1, \ldots, n\}$. Consider a path $p = x_1, x_2, \ldots, x_k$ from x_1 to x_k. Nodes $x_2, \ldots, x_{k-1}$ are intermediate nodes of p.

We look for paths which use nodes $\leq k$ as intermediate nodes.

Let $B_k[i,j] = 1$ iff there is a path from i to j which uses intermediate nodes $\leq k$ (set $B_k[i,j] = 0$ otherwise).

Clearly $B_0[i,j] = A[i,j]$ since we only have paths of length 1, as there can be no intermediate nodes ≤ 0.

Also $R^+(i,j)$ iff $B_n[i,j] = 1$, since B_n allows all possible intermediate nodes, and so all possible paths.

Now we just need to calculate B_k from B_{k-1} (for $k = 1, \ldots, n$).

Suppose we have a path p from i to j using intermediate nodes $\leq k$.

There are two cases:

1. k is not an intermediate node of p. Then $B_{k-1}[i,j]$ already
2. k is an intermediate node of p
    - We can assume that k occurs only once, since if it occurs multiple times we can shorten the path by removing the cycle(s) from k to k
    - But then we have paths i to k and k to j which just use intermediate nodes $\leq k - 1$. So $B_{k-1}[i,k]$ and $B_{k-1}[k,j]$

This is the idea behind Warshall's algorithm.

```
input A
copy A into B (array of Booleans) # B = B0
for k = 1 to n:
        # B = B_{k-1}
        for i = 1 to n:
                for j = 1 to n:
                        b_{ij} = b_{ij} || (b_{ik} && b_{kj})
```

```
        # B = B_k
# B = B_n
return B
```

The complexity is $O(n^3)$

To show correctness of the algorithm we formulate an invariant

$B = B_{k-1}$

We can recast the outer for loop as a while loop and show that this invariant is established initially and then maintained by each iteration of the while loop.

Warshall's algorithm with while loop:

```
copy A into B # B = B_0
k = 1 # B = B_{k-1}
while k ≤ n:
        # B = B_{k-1}
        for i = 1 to n:
                for j = 1 to n:
                        b_{ij} = b_{ij} || (b_{ik} && b_{kj})
        k = k + 1
# k = n + 1, B = B_n
return B
```

Floyd's Algorithm

The All Pairs Shortest Path Problem: given a weighted directed graph G, find the shortest paths between all pairs of nodes of G.

This can be solved efficiently using a simple modification of Warshall's algorithm.

Let G be a weighted directed graph with nodes $\{1, \ldots, n\}$ and adjacency matrix A.

Let $B_k[i, j]$ be the length of the shortest path from i to j which uses intermediate nodes $\leq k$. If there is no such path set $B_k[i, j] = \infty$

$$\text{Clearly } B_0[i,j] = \begin{cases} A[i,j] & \text{if } A[i,j] \\ \infty & \text{otherwise} \end{cases}$$

Also $B_n[i,j]$, will be the length of the shortest path from i to j.

Now we just need to calculate B_k from B_{k-1} (for $k = 1, \ldots, n$). Suppose we have a shortest path p from i to j using intermediate nodes $\leq k$ of length d.

There are two cases:

1. k is not an intermediate node of p. Then $B_{k-1}[i,j] = d$ already
2. k is an intermediate node of p.
   - Clearly k occurs only once, since p is shortest path
   - But then we have paths i to k and k to j which just use intermediate nodes $\leq k - 1$
   - These must be shortest paths just using intermediate nodes $\leq k - 1$ (or else p could be shorter)
   - So $d = B_{k-1}[i,k] + B_{k-1}[k,j]$

We see that $B_k[i,j] = \min(B_{k-1}[i,j], B_{k-1}[i,k] + B_{k-1}[k,j])$

This also works if there is no shortest path just using nodes $\leq k$

Floyd's algorithm

input A

$$\text{set } B[i,j] = \begin{cases} 0 & \text{if } i = j \\ A[i,j] & \text{if } i \neq j \text{ and there is an arc } (i,j) \\ \infty & \text{otherwise} \end{cases}$$

$\# B = B_0$
for $k = 1$ to n:
  for $i = 1$ to n:
    for $j = 1$ to n:
      $b_{ij} = \min(b_{ij}, b_{ik} + b_{kj})$
return B

The complexity is $O(n^3)$ as for Warshall's algorithm.

Dynamic Programming

Warshall's algorithm and Floyd's algorithm are both examples of dynamic programming.

In dynamic programming:

- break the main problem down into sub-problems
- the sub-problems are ordered (e.g. increasing size) and culminate in the main problem

To solve the main problem:

- move through the sub-problems in order
- solve each sub-problem using the stored solutions of the previous sub-problems and storing the new solution for later use
- solve the main problem as the final sub-problem

The Travelling Salesman Problem

The Travelling Salesman Problem (TSP): Given a complete weighted graph (G,W), find a way to tour the graph visiting each node exactly once and travelling the shortest possible distance.

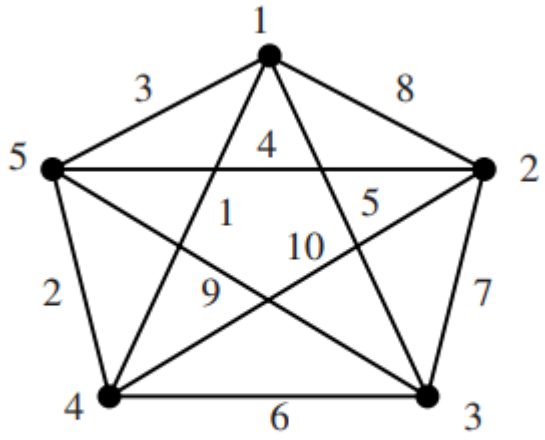


Figure 1.30: The travelling salesman problem

If arcs are missing from the graph we could make it complete by adding fictitious arcs with weights made high enough to ensure that they would never be chosen.

TSP is related to both the Hamiltonian Circuit Problem (HCP) and the Shortest Path Problem. We have to find a Hamiltonian circuit (HC) which is of minimum

weight.

The difficulty is not in finding an HC, since we have assumed that the graph is complete. But to find the shortest HC involves potentially checking $n!$ different tours if G has n nodes.

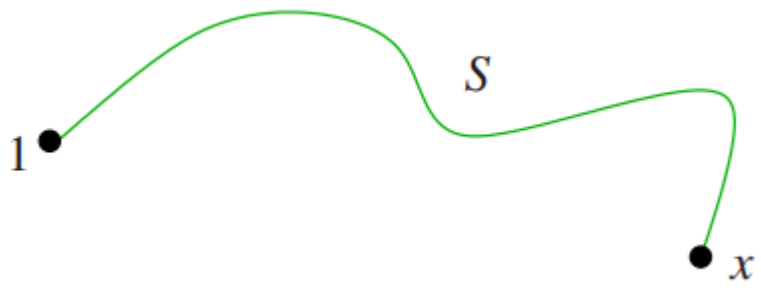
TSP is NP-complete and so is unlikely to have a polynomial solution.

A better algorithm than checking every possible tour is the Bellman-Held-Karp algorithm ($O(n^2 2^n)$), which is another example of dynamic programming.

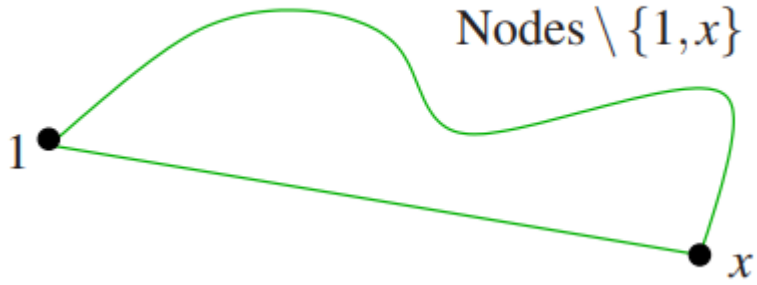
Let (G,W) have Nodes = $\{1, \ldots, n\}$.
Idea: Fix a start node, say 1.

For each $x \neq 1$ and each $S \subseteq Nodes \setminus \{1, x\}$:

- Find and store the minimum cost $C(S, x)$ of a path from node 1 to node x using set of intermediate nodes precisely S.



A TS tour can start at 1 wlog (without loss of generality). Let the last node before returning to 1 be x. The least cost of such a tour is $C(Nodes \setminus \{1, x\}, x) + W(x, 1)$



So the solution to TSP is:

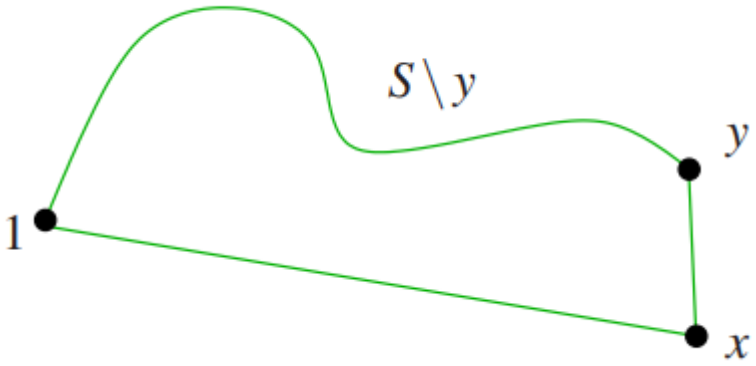
$$\min_{x \neq 1} C(\text{Nodes} \setminus \{1, x\}, x) + W(x, 1)$$

We calculate C(S, x) in increasing order of size: do all S of size 0, then 1, up to $n - 2$

Clearly $C(\emptyset, x) = W(1, x)$ as no intermediate nodes are allowed.

Assume we know C(S, x) for all S of size k.
Suppose $|S| = k + 1$.

Consider the last intermediate node y in a least cost path from 1 to x using intermediate nodes S (including y).



The cost must be $C(S \setminus y, y) + W(y, x)$. So

$$C(S, x) = \min_{y \in S} C(S \setminus y, y) + W(y, x)$$

The Bellman-Held-Karp algorithm in pseudocode:

```
Input (G,W)
Choose start ∈ Nodes(G)
for x ∈ Nodes \ {start}:
        C[/0, x] = W[start, x] # Process sets S in increasing order
of size.
for S ⊆ Nodes \ {start} with S /= /0:
        for x ∈ Nodes \ (S ∪ {start}):
                # Find C[S, x]
                C[S, x] = ∞
                for y ∈ S:
```

```
                        C[S, x] = min(C[S \ {y}, y] + W[y, x], C[S,
x])
# Now have calculated and stored all values of C[S, x]
opt = ∞
for x ∈ Nodes \ {start}:
        opt = min(C[Nodes \ {start, x}, x] +W[x,start], opt)
return opt
```

For each subset of Nodes (roughly speaking) we do $O(n^2)$ work with the two for loops. Overall, this gives $O(n^2 2^n)$

The Bellman-Held-Karp algorithm can be adapted to solve the Hamiltonian circuit problem. The complexity is still $O(n^2 2^n)$

Exact solutions take too long, but we still want to solve TSP. Instead, we use approximate methods.

For instance, we could try a “greedy” algorithm which always chooses the shortest available arc, the so-called *nearest neighbour heuristic (NNH)*.

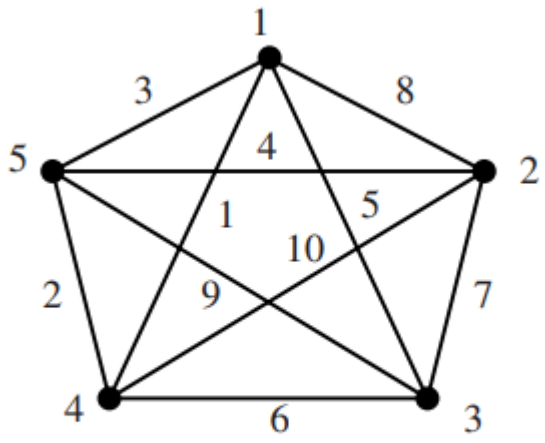


Figure 1.30: The travelling salesman problem

Starting from node 1 we get the tour 1,4,5,2,3,1 with a total weight of 1 + 2 + 4 + 7 + 5 = 19 which is quite good. However such a short-sighted method can also fail dramatically. Suppose that we change the weight of (1,3) from 5 to 500. Clearly this arc should be avoided, but NNH is forced to choose it on the last step.