Graphs and Algorithms

Iain Phillips

Spring 2024

Imperial College

Overview

Introduction

- Part I: Graphs
- Part II: Graph Algorithms
- Part III: Algorithm Analysis
- Part IV: Introduction to Complexity

Outline

Overview of Part I: Graphs

- Graphs
  - Basics
  - Representations
- Isomorphism, Planarity and Colouring
  - Isomorphism
  - Planar Graphs
  - Colouring
- Paths and Cycles
  - Basics
  - Euler Paths
  - Hamiltonian Paths
- Trees
  - Basics
  - Spanning Trees
- Directed Graphs

Overview of Part II: Graph Algorithms

- Graph Traversal
  - Preliminaries
  - Depth-First Search and Breadth-First Search
  - Applications
- Minimum Spanning Trees
  - Basics
  - Prim's Algorithm
  - Kruskal's Algorithm
- Shortest Path Problem
  - Dijkstra's Algorithm
  - A* algorithm
  - Warshall's Algorithm
  - Floyd's Algorithm
- The Travelling Salesman Problem

Overview of Part III: Algorithm Analysis

- Introduction
- Searching a list
  - Unordered lists
  - Ordered Lists (Binary Search)
- Orders
- Strassen's Algorithm
- Sorting
  - Insertion Sort
  - Lower Bounds
  - MergeSort
  - Master Theorem
  - QuickSort
  - Heapsort
- Dynamic Programming

Overview of Part IV: Introduction to Complexity

Tractable problems and P

NP

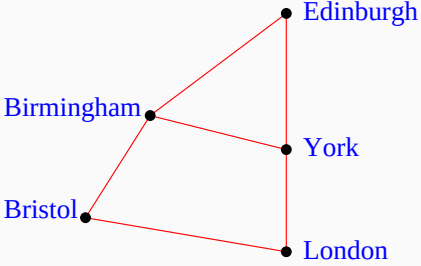
Problem reduction

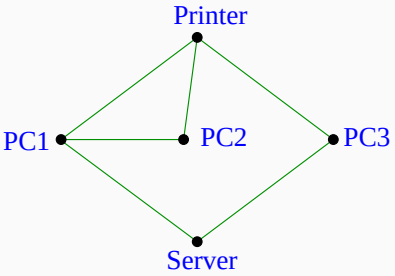
NP-completeness

Part I

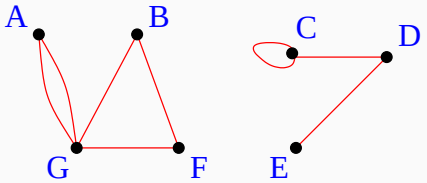
Graphs

Examples


Edinburgh
Birmingham
York
Bristol
London


Printer
PC1
PC2
PC3
Server

Graphs in General



- parallel arcs
- loops
- disconnected components

Definitions

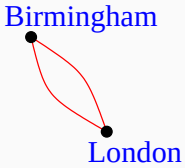
An (undirected) graph is a set N of nodes and a set A of arcs such that each $a \in A$ is associated with an unordered pair of nodes (the endpoints of a)

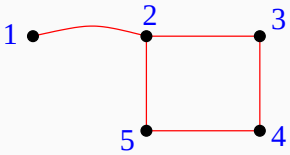
Notation

- G
- $\mathrm{nodes}(G)$
- $\mathrm{arcs}(G)$

A graph is simple if it has no parallel arcs and no loops.

Why Parallel Arcs?





Multiple connections for robustness against failures

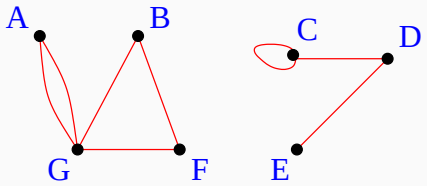
- Is our network robust?
- How many failures can it tolerate before becomes disconnected?

Degrees

The degree of a node is the number of arcs incident on it.
- the no of arcs which have that node as an endpoint

- Count loops twice

Each arc contributes twice to the total of all the degrees
- once for each endpoint

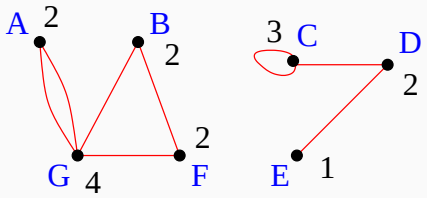


Degrees

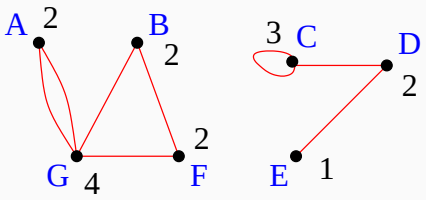
The degree of a node is the number of arcs incident on it.
- the no of arcs which have that node as an endpoint

- Count loops twice

Each arc contributes twice to the total of all the degrees
- once for each endpoint



Sum of Degrees



There are 8 arcs.

Sum of degrees is

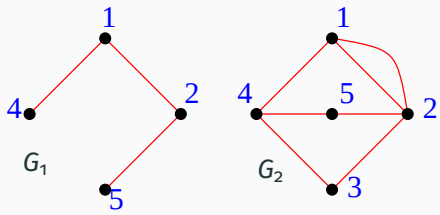
$2 + 2 + 3 + 2 + 1 + 2 + 4 = 2 \times 8$.

Theorem

(1) The sum of the degrees of all the nodes of a graph is twice the number of arcs, and therefore even.

(2) The number of nodes with odd degree is even.

Subgraphs



Here G_1 is a subgraph of G_2:

- $\text{nodes}(G_1) \subseteq \text{nodes}(G_2)$
- $\text{arcs}(G_1) \subseteq \text{arcs}(G_2)$

Of course G_1 and G_2 don't have to be drawn the same way.

- the connectivity is what counts

Full and Spanning Subgraphs

Full (Induced) Subgraphs

- Any subset $X \subseteq \mathrm{nodes}(G)$ induces a subgraph $G[X]$ of G, where $G[X]$ has nodes X and $G[X]$ contains all arcs of G which join nodes in X.
- G' is a full (or induced) subgraph of G if $G' = G[X]$ for some $X \subseteq \mathrm{nodes}(G)$.

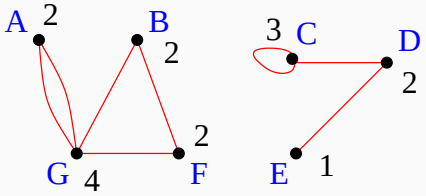
Spanning Subgraphs

If G' is a subgraph of G and $\mathrm{nodes}(G') = \mathrm{nodes}(G)$, we say that G' spans G.

Representations

- Adjacency matrices
- Adjacency lists

Adjacency Matrices



$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Create adjacency matrix.

i, *j* entry:

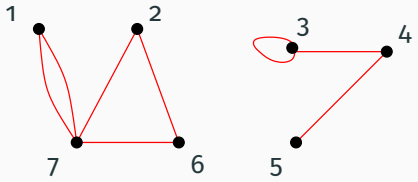
how many arcs connect *i* to *j*.

- Symmetric because arcs are undirected
- If we order the nodes differently the matrix gets rearranged.

Diagonal entries:

- count each loop twice
- to be consistent with our definition of degree
- so every arc contributes twice to the adjacency matrix

Adjacency Lists



1	→	7	→	7				
2	→	6	→	7				
3	→	3	→	4				
4	→	3	→	5				
5	→	4						
6	→	2	→	7				
7	→	1	→	1	→	2	→	6

Create adjacency list:
Array of linked lists.

- For each node list nodes which are adjacent.
- If multiple arcs then multiple entries.
- Each arc gets entered twice (apart from loops).

Comparison

The adjacency matrix gives efficient access to any arc in the graph.

However certain algorithms rely on looking at all arcs incident on a given node.

For these algorithms it can be faster to use adjacency lists.

This is particularly the case if the matrix has a lot of zeroes, i.e. the graph is sparse.

Suppose that there are n nodes and m arcs:

- The adjacency matrix has size n^2.
- The adjacency list has size $\leq n + 2m$.

Say that a graph is sparse if m is much smaller than n^2.

- Storage space will be less with adjacency list.

Big-Oh Notation

Multiplication of $n \times n$ matrices:

$$\begin{pmatrix} 1 & 0 & 2 \\ 3 & 1 & 0 \\ 4 & 2 & 3 \end{pmatrix} \begin{pmatrix} 3 & 6 & 1 \\ 2 & 0 & 5 \\ 1 & 4 & 2 \end{pmatrix} = \begin{pmatrix} 5 & 14 & 5 \\ 11 & 18 & 8 \\ 19 & 36 & 20 \end{pmatrix}$$

Each entry takes n multiplications and $n - 1$ additions.

So total is

- n^3 multiplications
- $n^3 - n^2$ additions
- $2n^3 - n^2$ arithmetical operations

Often prefer not to be so precise: $O(n^k)$ means bounded by n^k.

- $O(n^3)$ multiplications
- $O(n^3)$ additions
- $O(n^3)$ arithmetical operations

Big-Oh Notation

We can ignore constant factors and less important terms.

Terminology

- $O(1)$ constant (e.g. 371)
- $O(n)$ linear (e.g. $25n + 4$)
- $O(n^2)$ quadratic (e.g. $2n^2 + 2500n - 7$)

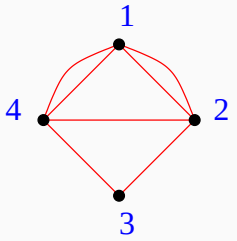
We shall use Big-Oh notation when calculating the space and time usage of data structures and algorithms.

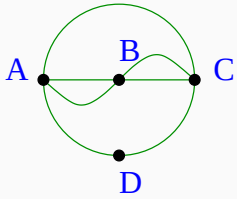
Advantages

- abstract away from implementation-dependent specifics
- concentrate on factor which determines growth

We give precise definitions later.

Graph Isomorphism





What does it mean for a graph to be the "same" as another?

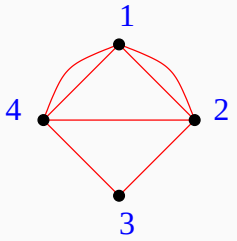
The graphs look different, but they have the same shape, in the sense that the nodes are connected in the same way.

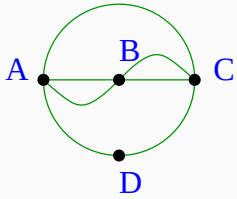
- Degrees

1	2	3	4	A	B	C	D
4	4	2	4	4	4	4	2

- parallel arcs: 14, 12 and AB, BC.
- Which nodes correspond?

Graph Isomorphism





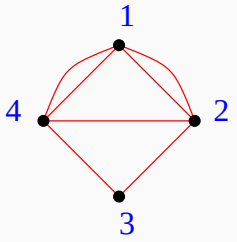
3 must correspond to D - only nodes with degree 2.
1 must correspond to B, since
1 only node not adjacent to 3 and
B only node not adjacent to D
We now can either have $2 \longmapsto A, 4 \longmapsto C$ or
$2 \longmapsto C, 4 \longmapsto A$

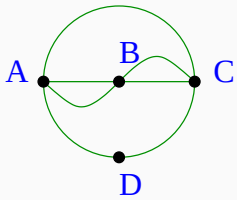
Suppose we take

1	2	3	4
B	A	D	C

This is a bijection between nodes.
But also must check that connections are the same.
Fairly obvious by mentally transforming the graphs so they look the same.
To check rigorously we need to find the adjacency matrices.

Check





In the standard order (1234 and *ABCD*) the adjacency matrices are:

$$\begin{pmatrix} 0 & 2 & 0 & 2 \\ 2 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 2 & 1 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 2 & 1 & 1 \\ 2 & 0 & 2 & 0 \\ 1 & 2 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

To make them match, reorder RH matrix to correspond to mapping

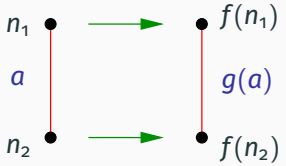
1	2	3	4
B	*A*	*D*	*C*

	B	*A*	*D*	*C*
B	0	2	0	2
A	2	0	1	1
D	0	1	0	1
C	2	1	1	0

Definition

Let G, G' be graphs.

An isomorphism from G to G' is a bijection $f : \mathrm{nodes}(G) \to \mathrm{nodes}(G')$
together with a bijection $g : \mathrm{arcs}(G) \to \mathrm{arcs}(G')$
such that if $a \in \mathrm{arcs}(G)$ has endpoints n_1 and n_2 then
the endpoints of $g(a)$ are $f(n_1)$ and $f(n_2)$.



G is isomorphic to G'

This amounts to seeing that the adjacency matrices of G and G' are
the same, except that the rows and columns may have been reordered.

Testing for Isomorphism

When testing whether two graphs are isomorphic, it is simplest to start with the obvious checks:

- # nodes
- # arcs
- # loops
- degrees

If this turns up a difference, then the graphs can't be isomorphic.

If they are the same on these tests, then attempt to find a bijection on nodes.

Check that it works using adjacency matrices.

Complexity

In general, discovering whether two graphs are isomorphic has quite a high complexity.

We need to see whether the adjacency matrices are rearrangements of each other.

There are $n!$ rearrangements if there are n nodes.

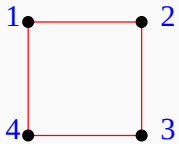
This is worse than exponential (2^n).

Remark:

László Babai (2015): Graph isomorphism can be solved in quasi-polynomial time, i.e. $\exp(\log^k n)$ for some k.

Better than exponential though worse than polynomial.

Automorphisms



An automorphism on a graph G is an isomorphism from G to itself.

Possible automorphism: $\begin{matrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{matrix}$

This is a 90 degree clockwise rotation.

Another one is $\begin{matrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 1 & 4 \end{matrix}$

This is a flip across the diagonal $2-4$.

How many other automorphisms?

Method

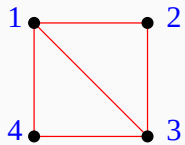
1 can map to 4 different places.

Fix 1. Then 2 can map to 2 places.

Fix 2. Then 3 can map to 1 place. 4 is also fixed.

Total: $4.2.1.1 = 8$ (including the identity).

Example



How many automorphisms?
By method:
1 can map to 2 different places.
Fix 1. Then 2 can map to 2 places.
Fix 2. Then 3 and 4 are fixed.
Total: $2.2.1.1 = 4$ (including the identity).

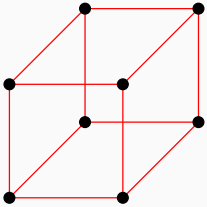
Planar Graphs

Sometimes it is desirable to avoid arcs crossing.

e.g. microchips

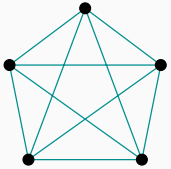
Definition

A graph is planar if it can be drawn so that no arcs cross.



Planar

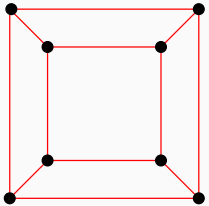
K_5 - the complete graph on 5 nodes.



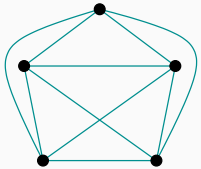
Not planar

Cube and K_5

The cube graph is planar:



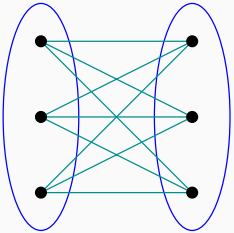
The best we can do with K_5:



There is still one crossing.

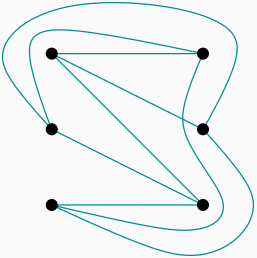
$K_{3,3}$

A further non-planar graph:



$K_{3,3}$ is the complete bipartite graph on two sets of 3 nodes.

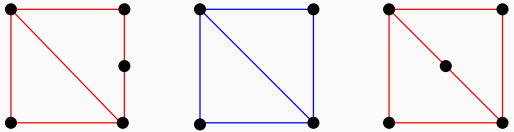
Again we cannot remove all crossings:



Kuratowski's Theorem

It turns out that any non-planar graph contains K_5 or $K_{3,3}$ as a subgraph, in a sense.

Two graphs are homeomorphic if they can both be obtained from the same graph by a series of operations where an arc $x - y$ is replaced by two arcs $x - z - y$.



Theorem

(Kuratowski, 1930) A graph is planar iff it does not contain a subgraph homeomorphic to K_5 or $K_{3,3}$.

Is a Graph Planar?

As we have seen, it is not obvious whether a given graph is planar.

But there is a linear time algorithm due to Hopcroft & Tarjan (1976).

> *"Linear time" here means $O(n + m)$ where n is the number of nodes and m is the number of arcs.*

So in a sense, testing planarity is "easy".

- "low complexity"

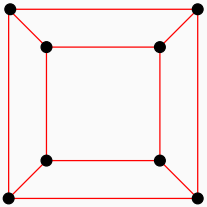
Fact

If a (simple) graph is planar it can always be redrawn so that all arcs are straight lines which don't cross.

Euler's Formula

Any planar graph splits the plane into regions, called faces.

The cube graph has 6 faces, including the outside.



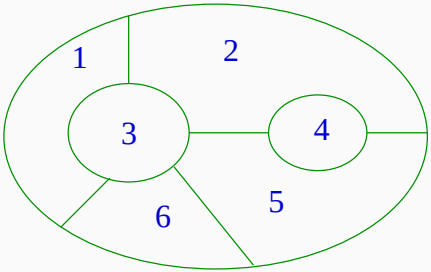
Let G have N nodes, A arcs and F faces.

Euler's formula: $F = A - N + 2$ for a connected planar graph

For the cube: $F = 6$, $A = 12$ and $N = 8$.

Map Colouring

A map with six countries:

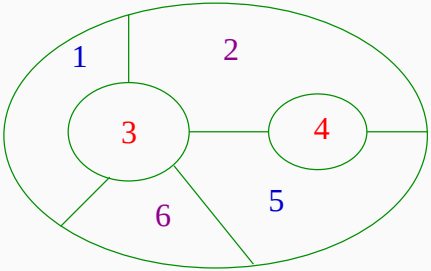


Problem: Colour the map so that if two countries share a border then they have different colours.

Question: How many different colours do we need?

Map Colouring

A map with six countries:



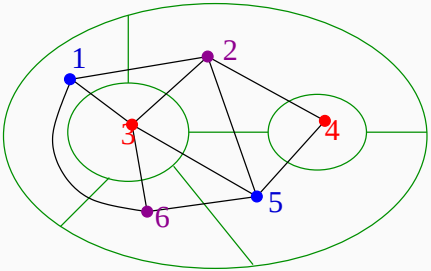
Problem: Colour the map so that if two countries share a border then they have different colours.

Question: How many different colours do we need?

The Dual Graph

We can turn a map into a planar graph by letting the countries be the nodes, and joining them if they are neighbours on the map.

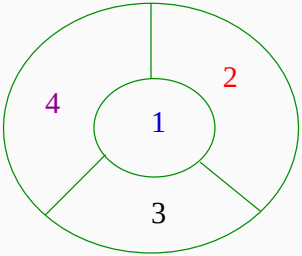
This graph is called the dual graph.



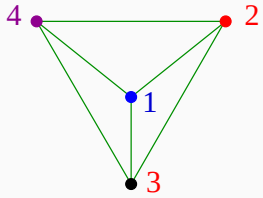
The problem now is to colour the nodes of a planar graph in such a way that if two nodes are adjacent then they have different colours.

The Dual Graph

A map that needs four colours:



The dual graph:



This is just K_4.

Maps and simple planar graphs

- Dual graphs of maps are always simple and planar.
- Conversely, any simple planar graph is the dual graph of some map.

The Four Colour Theorem

Four Colour Theorem

Every map can be coloured using at most four colours.

- 1976. Original proof by Appel and Haken. Very long. 1482 cases generated and checked by computer. But how to check it?
- 1994. A different computer proof.
- 2004. Gonthier and Werner check 1994 proof using a general-purpose proof assistant.

Graph Colouring

Definition

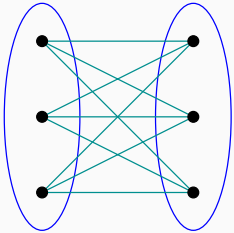
A graph G is k-colourable if the nodes of G can be coloured using no more than k colours.

From the Four Colour Theorem:

- Every simple planar graph is 4-colourable.

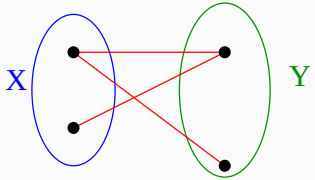
Bipartite Graphs

We already saw an example of a bipartite graph - $K_{3,3}$.



Definition

G is bipartite if $\mathrm{nodes}(G)$ can be partitioned into sets X and Y in such a way that no two nodes of X are joined and no two nodes of Y are joined.



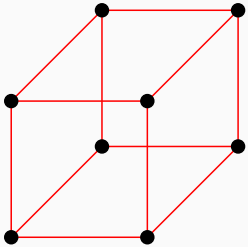
Proposition

A graph is bipartite iff it is 2-colourable.

Cube

Not always obvious.

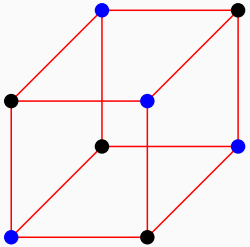
For example, the cube graph is 2-colourable.



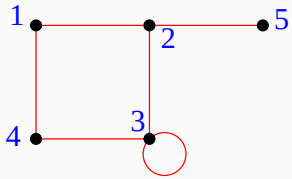
Cube

Not always obvious.

For example, the cube graph is 2-colourable.



Paths and Connectedness



A path in a graph is a sequence of adjacent arcs.
Usually describe paths by nodes passed through.

Paths in example include $1, 2, 3, 4$ and $2, 5, 2, 3, 3, 4$.

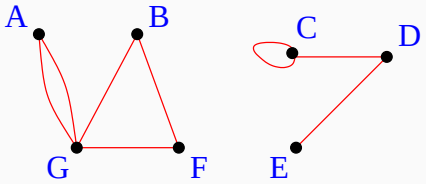
Definition

A path is simple if it has no repeated nodes.

Definition

A graph is connected if there is a path joining any two nodes.

Connected Components



The graph is not connected.
It has two connected components, with nodes A, B, F, G and C, D, E.

Connected Components

For any G we can define a relation on $\mathrm{nodes}(G)$ by

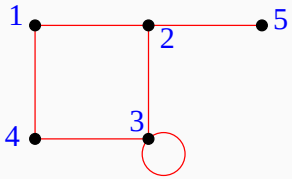
$x \sim y$ *iff there is a path from* x *to* y

We can check that this is an equivalence relation:

- reflexive
- symmetric
- transitive

The equivalence classes of $\sim$ are the nodes of the connected components.

Cycles



A cycle (or circuit) is a path which

- finishes where it starts
- has at least one arc
- does not use the same arc twice

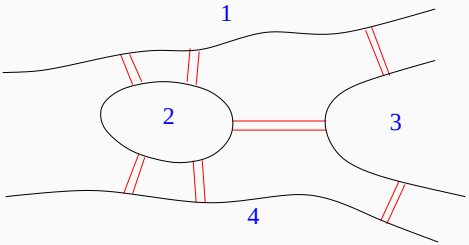
Cycles in example include $1, 2, 3, 4, 1$ and $3, 3$.

However $2, 5, 2$ is not a cycle.

Definition

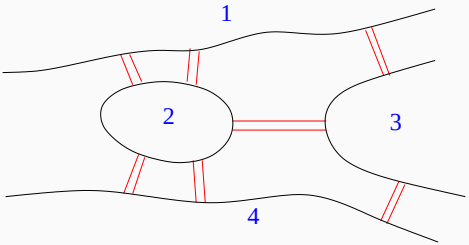
A graph with no cycles is called acyclic.

The Königsberg Bridge Problem

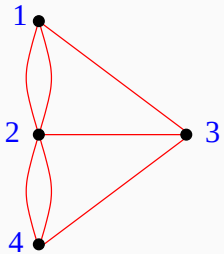


Problem: Start anywhere and cross each bridge exactly once.

The Königsberg Bridge Problem



Problem: Start anywhere and cross each bridge exactly once.



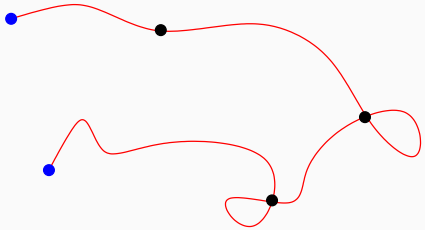
Notice the degrees of the nodes.

Definitions

- An Euler Path is a path which uses each arc exactly once.
- An Euler Circuit (or Euler Cycle) is a cycle which uses each arc exactly once.

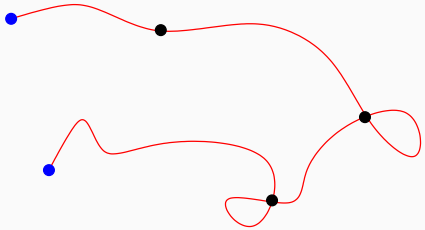
(So an EC is an EP which finishes where it starts)

Euler Paths



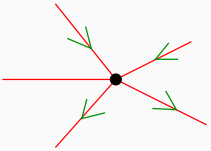
An intermediate node must be entered and left the same number of times.

Euler Paths

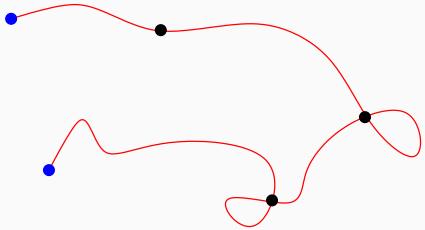


An intermediate node must be entered and left the same number of times.

If an intermediate node n has odd degree, then cannot use all the arcs joined to n.

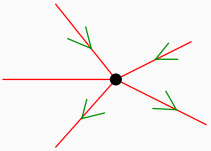


Euler Paths



An intermediate node must be entered and left the same number of times.

If an intermediate node *n* has odd degree, then cannot use all the arcs joined to *n*.



- If a graph has an Euler path, then the number of odd nodes must be 0 or 2.
- If a graph has an Euler circuit, then every node must be even.

Euler's Theorem

Theorem (Euler 1736)

- *A connected graph has an Euler path iff there are 0 or 2 odd nodes.*
- *A connected graph has an Euler circuit iff every node has even degree.*

Proof. We have shown "$\Rightarrow$".

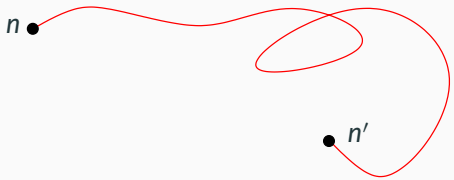
For "$\Leftarrow$", assume that exactly two nodes n, n' have odd degree.

(Case where all nodes have even degree is similar.)

Proof (cont.)

Start at n and carry on until can go no further.

We must have got to n':

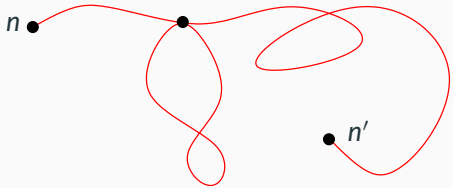


- if stop at n then there will be a spare arc
  (n is odd and have used even no of arcs)
- if stop at $n'' \neq n, n'$ then there will be a spare arc
  (n'' is even and have used odd no of arcs)

Notice that the no of unused arcs at any node is now even.

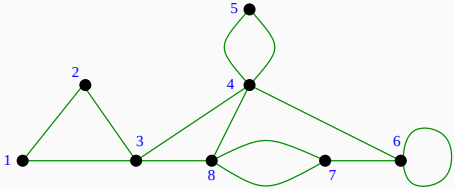
Proof (cont.)

Now do a side journey:

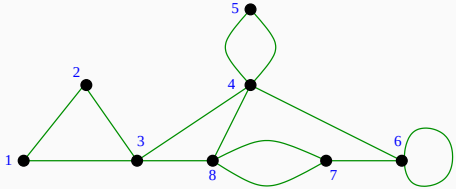


Keep on adding until all arcs used up.

Example

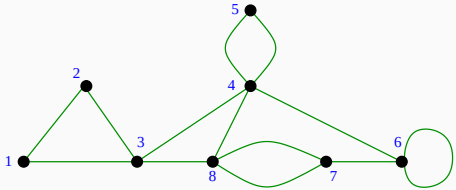


Example

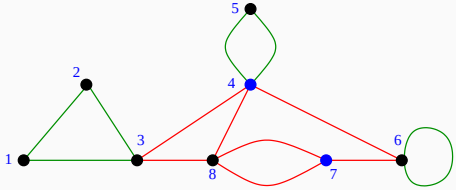


Identify endpoints of EP (4,7).

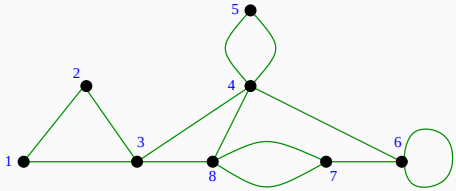
Example



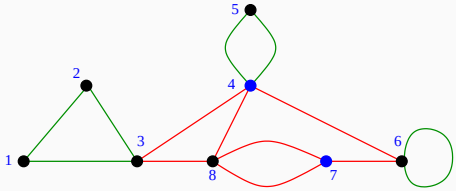
Identify endpoints of EP (4,7).
Start from 4 (say):
Path so far: 4,8,7,6,4,3,8,7



Example



Identify endpoints of EP (4,7).
Start from 4 (say):
Path so far: 4,8,7,6,4,3,8,7



Now do side journeys:

- 3,2,1,3
- 4,5,4
- 6,6

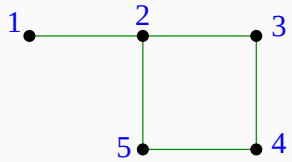
Complete path: 4, 5,4, 8,7,6, 6, 4,3, 2,1,3, 8,7

Hamiltonian Paths

Can we find a path which visits every node exactly once?

Such a path is called a Hamiltonian path.

Similarly Hamiltonian circuit: HP which returns to start node.



Graph has HP but not HC.

We may as well consider simple graphs, since we can never follow a loop, and we can never follow more than one arc between two nodes.

Hamiltonian Circuit Problem (HCP)

Given a graph *G* determine whether *G* has a HC.

Conditions

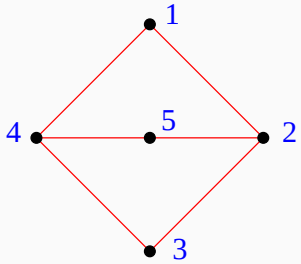
For HP to exist:

graph must be connected

For HC to exist:

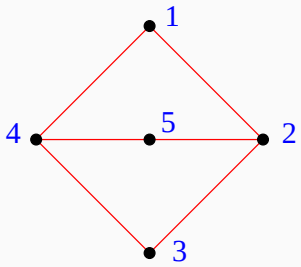
connected and each node must have degree ≥ 2

But this is not sufficient:



The graph has no HC, despite every node degree ≥ 2.

Brute Force Attack



We can solve by “brute force”:
check every possible circuit

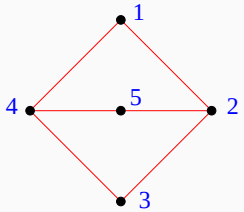
12345(1) - no arc from 5 to 1

21345(2) - no arc from 1 to 3

etc.

How long does this take with n nodes?
Possible circuit corresponds to a permutation of nodes.
A permutation is a bijection $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$.
To be an actual circuit we must have
$\pi(i)$ adjacent to $\pi(i+1)$ for $i = 1, \ldots, n$
(take $n+1$ to be 1 for convenience).

Brute Force Attack



The possible circuit 5,4,3,1,2,(5) corresponds to the permutation

$\pi(1) = 5, \pi(2) = 4, \pi(3) = 3, \pi(4) = 1, \pi(5) = 2$

Not a circuit since $\pi(3)$ not adjacent to $\pi(4)$.

- Each possible circuit can be checked in $O(n)$.
- There are $n!$ possible circuits, corresponding to $n!$ permutations of the nodes.

This is much too slow
- worse than exponential ($> 2^n$).

Improvement

A ‘dynamic programming’ method can solve HCP in $O(n^2 2^n)$ - still exponential.

Due to Bellman, Held, and Karp (1962)

Will be discussed later in the course.

Complexity

The complexity of HCP is much greater than that of the Euler Path Problem (EPP):

Can solve EPP in polynomial time – $O(n^2)$

HCP seems to require $O(n^2 2^n)$

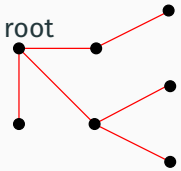
In fact HCP has been shown to be NP-complete.

The NP-complete problems are believed not to be solvable in polynomial time.

i.e. they cannot be solved in $O(n^k)$ for any k

NP-completeness will be discussed later in the course.

Trees



- A rooted graph is a graph G together with a distinguished node (the root of G).
- A tree is an acyclic, connected, rooted graph.
- A nonrooted tree is an acyclic, connected graph.

In a tree there is a unique (non-repeating) path between any two nodes:

two different paths would give a cycle

The depth of a node x is the distance (along the unique path) from the root to x.

If x is not the root, define the parent of x to be the unique node adjacent to x on the path from x to the root.

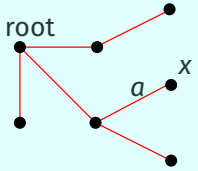
The depth of a tree is the maximum of the depths of all its nodes.

Nodes and Arcs

Theorem

Let T be a tree with n nodes. Then T has $n - 1$ arcs.

Proof.



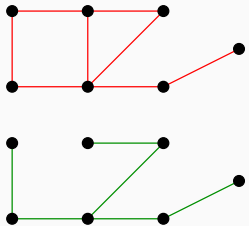
Note that there is a 1-1 correspondence between arcs and non-root nodes:

- Any arc a joins a unique non-root node $f(a)$ to its parent.
- Also, if x is a non-root node then there is a unique arc $g(x)$ which joins x to its parent.

Clearly f and g are mutual inverses. □

Also holds for nonrooted trees—just make any node into the root.

Spanning Trees



Find a subgraph which is a tree and which “spans” all the nodes.

The tree has 6 arcs of course.

Spanning tree: lower-cost network which still connects the nodes.

Definition

Let G be a graph.
A nonrooted tree T is said to be a spanning tree for G if T spans G, i.e.

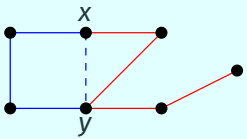
- T is a subgraph of G and
- $\text{nodes}(T) = \text{nodes}(G)$.

Existence

Proposition

Let G be a connected graph. Then G has a spanning tree.

Proof.

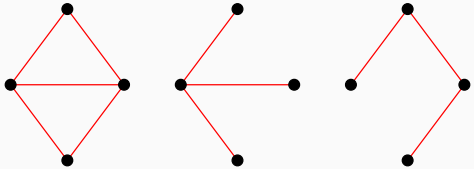


Obtain a spanning tree as follows:
If G has a cycle C then remove any arc x to y of C.
Still a path from x to y using remainder of C.
Hence still connected.
All nodes still present.
Repeat until all cycles removed.
This must terminate—why?
We have a spanning tree.

□

Uniqueness

Spanning trees are not necessarily unique.



Any two spanning trees for the same graph with *n* nodes must have the same number of arcs.

Directed Graphs

So far we have only considered undirected graphs.

For many applications it makes sense for arcs/edges to be directed.

Definition

A directed graph is a set N of nodes and a set A of arcs such that each $a \in A$ is associated with an ordered pair of nodes (the endpoints of a)

- In diagrams the arcs are shown with arrows from source node to target node.
- In a path $a_1, \ldots, a_n$ in a directed graph the source of a_{i+1} must match the target of a_i (for $i = 1, \ldots, n-1$)
- If for any pair of nodes x, y there is at most one arc from x to y then we can refer to this arc as (x, y).

Directed Graphs

Definition

The indegree of a node x is the number of arcs entering x.

The outdegree of a node x is the number of arcs leaving x.

For any directed graph:

sum of indegrees of all nodes = sum of outdegrees = number of arcs

Definition

A directed graph is strongly connected if for any $x, y \in \mathrm{nodes}(G)$ there is a path from x to y.

So we need paths both from x to y and from y to x.

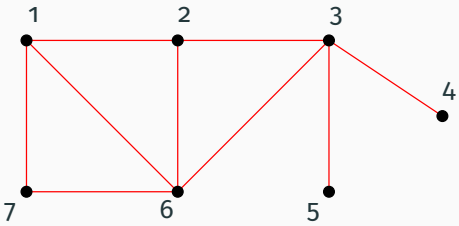
Convention

In this course (though not necessarily elsewhere) by default
a graph is undirected unless we state that it is directed.

Part II

Graph Algorithms

Traversing a Graph



Start at some node.

Visit every node, using some path.

Allowed to backtrack.

Will reach every node if graph connected.

In a network may need to process each node

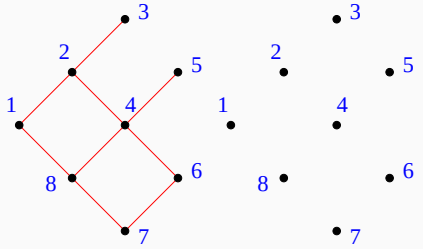
- check if working properly
- calculate some function (time since last failure)

Traversal Algorithms

Two standard traversal procedures are

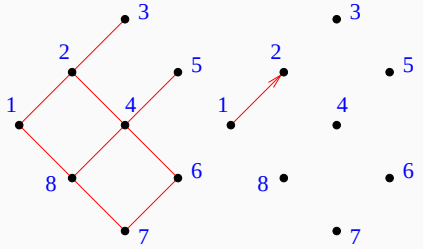
- Depth-first search
- Breadth-first search

Depth-First search



Start and finish at 1.

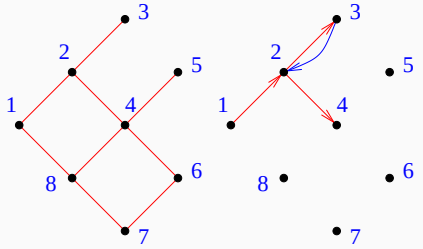
Depth-First search



Start and finish at 1.

Go to 2 and start DFS from 2.

Depth-First search

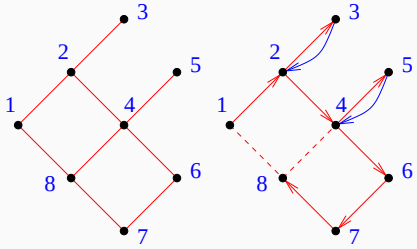


Start and finish at 1.

Go to 2 and start DFS from 2.

Backtrack from 3 to 2 and then go to 4.

Depth-First search



Start and finish at 1.

Go to 2 and start DFS from 2.

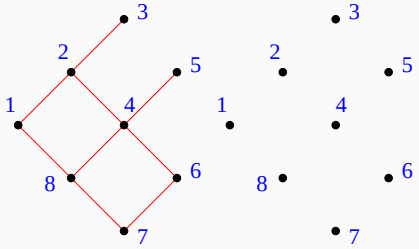
Backtrack from 3 to 2 and then go to 4.

At 8 all adjacent nodes are visited: backtrack to start.

Order of visiting: 1, 2, 3, 4, 5, 6, 7, 8

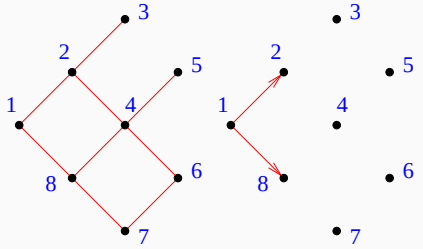
We get a spanning tree. Notice that 8 is distance 5 from start.

Breadth-First Search



Start at 1.

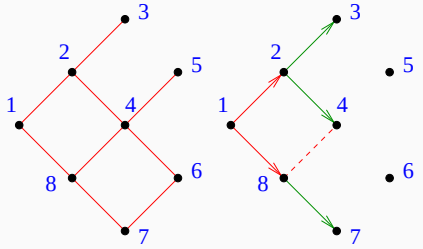
Breadth-First Search



Start at 1.

Fan out to visit adjacent nodes at distance 1: 2, 8

Breadth-First Search

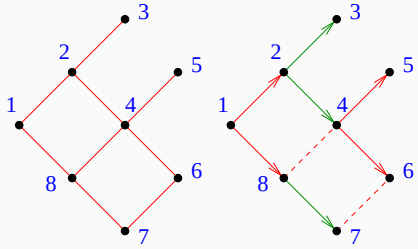


Start at 1.

Fan out to visit adjacent nodes at distance 1: 2, 8

Then visit nodes at distance 2: 3, 4, 7

Breadth-First Search



Start at 1.

Fan out to visit adjacent nodes at distance 1: 2, 8

Then visit nodes at distance 2: 3, 4, 7

Finally visit 5, 6

Again get a (different) spanning tree.

Depth of any node is its distance from start.

Comparison

Both DFS and BFS traverse all nodes in a connected graph, but the order of visiting is different.

Sometimes either procedure will do; other times one procedure is to be preferred.

Formalising DFS and BFS

Graph to be traversed is given as an adjacency list.

- visited: Boolean array of nodes
- parent: parent node in search tree
- output nodes in order visited
- initialise: no nodes visited

DFS Algorithm

DFS uses recursion. Each time we visit a new node *y*, we perform DFS completely at *y* before backtracking to parent node *x*.

```
procedure dfs(x)
    visited[x] = true
    print x
    for y in adj[x]:
        if not visited[y]:
            parent[y] = x
            dfs(y)
            # backtrack to x
```

Once DFS is completed every node has a parent except for the start node. Of course, the parent information can be omitted.

Analysis

- dfs(x) is applied to each node at most once
  (in fact exactly once if the graph is connected).
- Also each application of dfs(x) runs through the arcs incident on x exactly once.

Therefore the running time of DFS is $O(n + m)$, where n is the number of nodes and m is the number of arcs.

- Note that we are just counting accesses to the graph
  (stored as adjacency lists).
- Each access counts as 1 unit.
- We are ignoring overheads due to recursion.

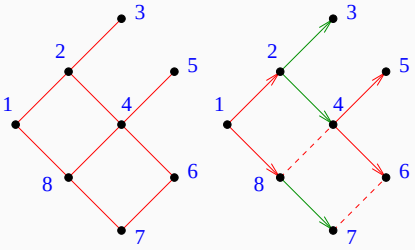
Breadth-First Search

BFS is naturally expressed using a queue of nodes.
FIFO (first in, first out)
Queue is initialised with start node x.
We then process nodes from the front of the queue.
For each node we visit its immediate neighbours, adding them to the back of the queue.

Algorithm: Breadth-first Search

```
visited[x] = true ; print x
enqueue(x, Q)
while not isempty(Q):
    y = front(Q)
    for z in adj[y]:
        if not visited[z]:
            visited[z] = true
            print z
            parent[z] = y
            enqueue(z, Q)
    dequeue(Q)
```

Example



Head of the queue to the left; nodes added to the right.
Initially the queue is [1].
Process 1, and add 2, 8 to get [2,8].
Process 2 to get [8,3,4].
etc.
Nodes are added to the queue in the order 1,2,8,3,4,7,5,6
Removed in the same order.

Discussion

Queue grows and shrinks during the computation.

Size of the queue represents “breadth” of the front on which BFS working.

Analysis

As with DFS each node is processed once and each adjacency list is processed once.

Again $O(n + m)$.

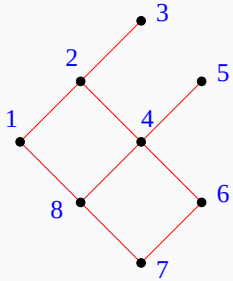
Is a Graph Connected?

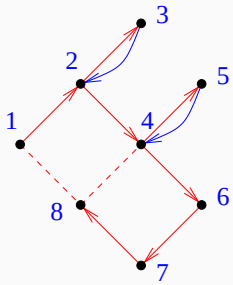
- So far we have assumed that the graph to be traversed is connected.
- But we can traverse non-connected graphs as well.
- Of course we will only visit nodes which are in the same connected component as the start node.
- We can easily adapt either DFS or BFS to return a list of visited nodes.
- Clearly the graph is connected iff this list is the same (up to reordering) as the complete list of nodes.
- This gives us an $O(n + m)$ algorithm to determine whether a graph is connected.

Does a Graph Have a Cycle?

- Suppose a connected graph has n nodes.
- If it has $\geq n$ arcs then it contains a cycle
  (exercise: in fact true even if graph not connected).
- We can use this to check easily whether a graph has a cycle.
- But this method does not *find* a cycle.
- As an alternative, use DFS.

Does a Graph Have a Cycle?





Suppose that we are using DFS to traverse a graph.

Example

When reach 8 we find that 1 (and 4) already visited.

Backtrack from 8 to get cycle 8, 7, 6, 4, 2, 1, (8).

If we encounter a node which we have already visited (except by backtracking), this tells us that the node can be approached by two different routes from the start node.

Hence there is a cycle in the graph.

Conversely, if we never encounter an already visited node, the graph is in fact a tree, with no cycles.
(We omit the proof.)

Does a Graph Have a Cycle?

Therefore we can adapt DFS to test whether a graph has a cycle.

Fact (exercise)

Let G be a connected graph, and let T be a spanning tree of G obtained by DFS starting at node start.

If a is any arc of G (not necessarily in T), with endpoints x and y, then either x is an ancestor of y in T or y is an ancestor of x in T.

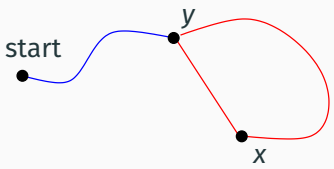
Here 'x is an ancestor of y in T' means that
x lies on the (unique) path from start to y in T.

Does a Graph Have a Cycle?

Return x, y if from node x we find that y (not the parent of x) has already been visited.

In this case there must be a cycle.

By the Fact, y must be an ancestor of x.



Cycle is x, parent[x], . . ., y, x.

Distances from the Start Node

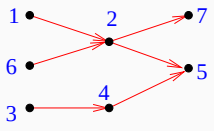
BFS finds the shortest path from the start node to any reachable node, while DFS may well give a longer distance than necessary.

- Easily adapt BFS to calculate the distance from the start node.
- Record the depth of each visited node in the BFS tree.
- The shortest path from a node *y* to the start node can be read off from the parent function as:
  y, *parent[y]*, *parent[parent[y]]*, . . ., start

Topological Sorting

Suppose that we have a list of tasks to be completed and some tasks have to be completed before others.

Can view as a directed graph:



This graph must be acyclic or else we cannot find an order in which to complete the tasks.

Given a directed acyclic graph (DAG) G with n nodes, find a total ordering of the nodes $x_1, \ldots, x_n$ such that for any $i, j \leq n$, if $j > i$ then there is no path from x_j to x_i in G.

Such a total ordering is called a topological sort of G.
It could be presented as a list or array of nodes.
For the example diagram a TS could be
$1, 6, 3, 2, 4, 7, 5$ or $6, 1, 2, 7, 3, 4, 5$, etc.

DAGs and partial orderings

Given a DAG G, let $x \leq y$ iff there is a path from x to y.

Then $\leq$ is a (weak) partial ordering:

1. reflexive
2. transitive
3. antisymmetric

Conversely, if $(X, \leq)$ is a partial ordering, let G be the directed graph with nodes X and arcs $\{(x, y) : x \leq y\}$. Then G is acyclic.

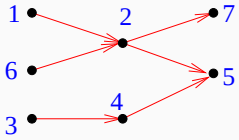
So a topological sorting of a DAG amounts to a linearisation of a partial ordering, i.e. a linear order which extends the partial ordering.

Topological Sorting using DFS

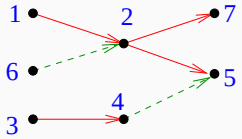
Can perform topological sorting using DFS.

The idea is that when we have finished processing a node *x* we must have finished with all nodes which are reachable from *x*, and which must come after *x* in the sorting.

So we can add a node to the sorted list (starting from the top end) once we have finished processing it (once we exit the node).



Topological Sorting using DFS



In the example we enter (visit) the nodes in the order

$$1, 2, 5, 7, 3, 4, 6$$

assuming that the adjacency lists are given in numerical order.
We exit the nodes in the order

$$5, 7, 2, 1, 4, 3, 6$$

We get the sort $6, 3, 4, 1, 2, 7, 5$

Topological Sorting using DFS

Given: a directed graph G with n nodes.

Return: topological sort of G as array ts of nodes if G acyclic (else abort).

procedure dfsts(x)

```
entered[x] = true
for y in adj[x]:
    if entered[y]:
        if not exited[y]: abort      # cycle
    else:
        parent[y] = x
        dfsts(y)
exited[x] = true
ts[index] = x ; index = index −1
```

Topological Sorting using DFS

index = $n - 1$

for x in $\mathrm{nodes}(G)$:

  if not entered[x]:

    dfsts(x)

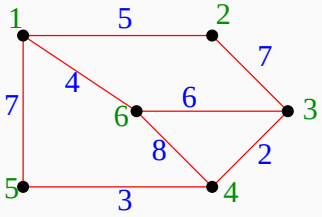
Note that we add nodes to the array as they are exited.

Correctness follows from

Proposition

When performing DFS on a DAG, when we exit a node x we have already exited all nodes reachable from x.

Weighted Graphs



Many networks have a cost associated with each arc.
Transport network: distance, time or fuel
Call these costs weights.

Definition

A weighted graph (G, W) is a simple graph G together with a weight function $W : \mathrm{arcs}(G) \to \mathbb{R}^+$ (reals $\geq$ 0).

We shall investigate these problems:

- finding a minimum spanning tree
- finding shortest paths
- finding a shortest circuit

Why Simple Graphs?

We are trying to minimise cost, and so the restriction to simple graphs is sensible:

- if there are two parallel arcs then we will always choose the cheaper
- if there is a loop then we will never wish to use it.

With simple graphs an arc can be specified uniquely by giving its endpoints: $(1, 2)$

Can regard the weight function as acting on pairs of nodes: $W(1, 2)$ ($= W(2, 1)$ of course)

Minimum Spanning Trees

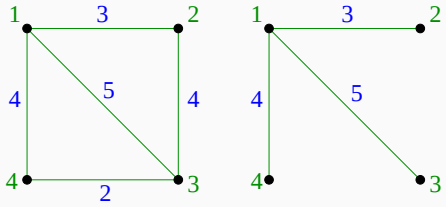
Recall that every connected graph has a spanning tree

- connects all nodes with least number of arcs

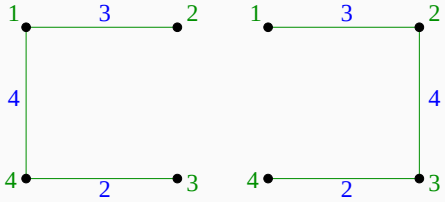
When dealing with weighted graphs we want to find a minimum spanning tree, that is, a spanning tree where the sum of the weights of its arcs is as small as possible.

For instance, if we had to build a road network joining some cities, a minimum spanning tree would represent the cheapest network which would connect all the cities.

Example



We have a spanning tree above with weight 12, but we can do better:



These both have weight 9.
They are minimum—easy to check.

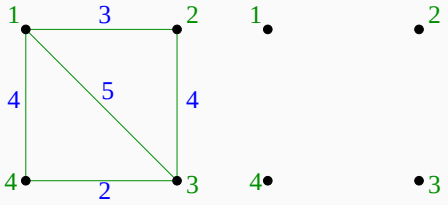
Shows MSTs need not be unique.

Definition

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.

Prim's Algorithm



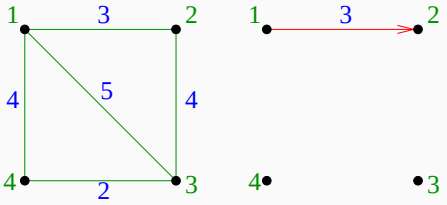
Build MST starting from root node 1.

Add the shortest arc which will extend the tree.

So-called greedy approach

- do what gives a short-term advantage, even if it may not be the best overall.

Prim's Algorithm



Build MST starting from root node 1.

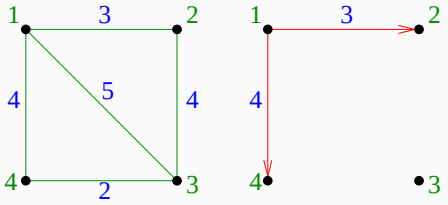
$(1, 2)$

Add the shortest arc which will extend the tree.

So-called greedy approach

- do what gives a short-term advantage, even if it may not be the best overall.

Prim's Algorithm



Build MST starting from root node 1.

$(1, 2)$

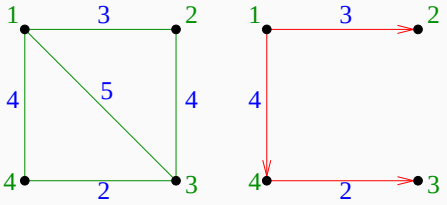
Then $(1, 4)$ or $(2, 3)$.

Add the shortest arc which will extend the tree.

So-called greedy approach

- do what gives a short-term advantage, even if it may not be the best overall.

Prim's Algorithm



Build MST starting from root node 1.

$(1, 2)$

Then $(1, 4)$ or $(2, 3)$.

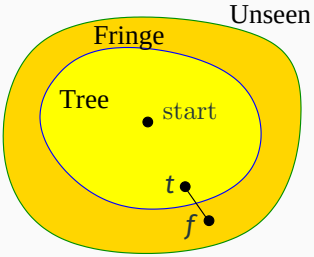
Finally $(4, 3)$.

Add the shortest arc which will extend the tree.

So-called greedy approach

- do what gives a short-term advantage, even if it may not be the best overall.

Fringe Nodes



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

- tree nodes
- candidates to join at the next stage: fringe nodes adjacent to a tree node
- the rest—unseen nodes

Initially all nodes are unseen.

Prim's Algorithm—Scheme

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

Analysis

n nodes, m arcs

Each time the while loop is executed another node is added to the tree.

Hence the while loop is executed $O(n)$ times.

(*) involves finding the shortest arc among all possible arcs between $O(n)$ tree nodes and $O(n)$ fringe nodes.

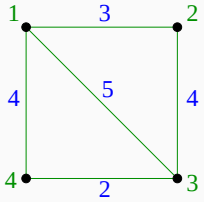
Therefore $O(n + m)$.

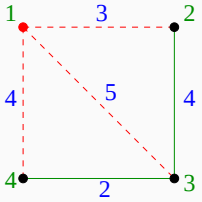
So the whole algorithm $O(n(n + m))$.

Improvement

Keep track of which arcs might be used—candidate arcs.

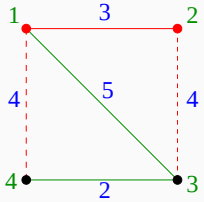
Example. Initially just 1 in tree.





Fringe nodes 2,3,4.

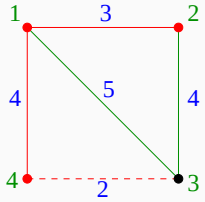
Now $(1, 2)$ is added:



Fringe nodes 3,4.
Candidate arc for 3 has changed.

Improvement

Now $(1, 4)$ added:



Fringe node 3.
Candidate arc for 3 has changed again.

Implement using parent function (as for graph traversal).

Initially parent[3] = 1.

Then parent[3] = 2.

Finally parent[3] = 4.

Defined for tree and fringe nodes.

Let the parent of a fringe node f be the tree node t such that (t, f) has least weight.

Prim's MST Algorithm ('classic' version)

Choose any node start as the root

initialise tree

tree[start] = true

initialise fringe

for *x* in adj[start]:

  # add *x* to fringe

  fringe[*x*] = true

  parent[*x*] = start

  weight[*x*] = $W[\mathrm{start}, x]$

end of initialisation

[continued

continued

while fringe nonempty:

Select fringe node f s.t. weight[f] is minimum

fringe[f] = false

tree[f] = true

[continued

continued

```
for y in adj[f]:
    if not tree[y]:
        if fringe[y]:
            # update candidate arc if lower weight poss.
            if W[f, y] < weight[y]:
                weight[y] = W[f, y]
                parent[y] = f
        else:
            # y is unseen—add to fringe
            fringe[y] = true
            weight[y] = W[f, y]
            parent[y] = f
```

Discussion

Weight of the MST: add weights of nodes (apart from start).

Analysis

- As before there are $O(n)$ executions of the while loop.

Executing the while loop:

- Testing whether the fringe is empty is $O(n)$.
- Finding the fringe node f such that weight[f] is minimum is $O(n)$.
- The updating of the candidate arc for each y in adj[f] is $O(1)$.
  Hence the for loop is $O(n)$.

We conclude that the algorithm is $O(n^2)$.

Improves our earlier estimate of $O(n(n+m))$.

(Recall that m can be as large as n^2, so that $O(n(n+m))$ is $O(n^3)$.)

Correctness

Theorem

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

Let G have n nodes.

Let the trees constructed at each stage be $T_0, \ldots, T_k, \ldots$.

T_0: just the node start

T_{k+1} got from T_k by adding arc a_{k+1}.
Then T_k has $k+1$ nodes.

There are $n-1$ stages, with T_{n-1} being returned by the algorithm.

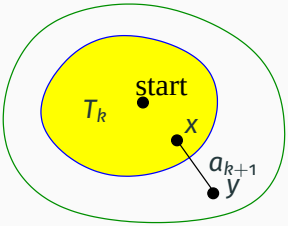
We show by induction on k that

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.

Correctness

Induction step. Assume that $T_k \subseteq T'$, some MST T' of G.

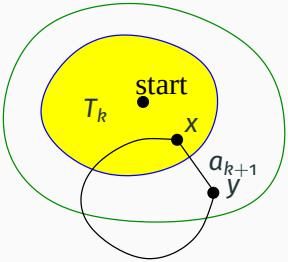


If $a_{k+1} \in \text{arcs}(T')$ then $T_{k+1} \subseteq T'$ as required.

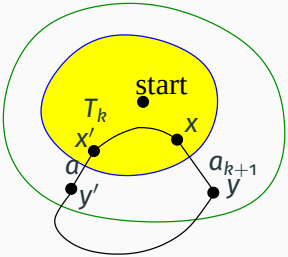
So suppose $a_{k+1} \notin \text{arcs}(T')$.

Since T' is a spanning tree, there must be a path in T' from x to y.

Correctness



Get a cycle.



Must cross from T_k to fringe.

Correctness

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, actually $W(a_{k+1}) = W(a)$.)

Also, $T_{k+1} \subseteq T''$ as required.

This completes the induction step.

Now T_{n-1} has $n-1$ arcs, and $T_{n-1} \subseteq T'$ 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.

Remarks

1. We can regard the induction hypothesis as an invariant.
   - Established initially
   - Maintained through each execution of the while loop of the code
2. Each T_k constructed by Prim's algorithm is connected.
3. Also, in fact T_k is an MST for the subgraph of G induced by $\mathrm{nodes}(T_k)$:
   the subgraph with nodes $\mathrm{nodes}(T_k)$ and all arcs of G which join nodes in $\mathrm{nodes}(T_k)$

We did not require either (2) or (3) for the proof.

Priority Queues

- Each item x of the queue has a priority key[x]—usually a natural number.
- Key represents cost
- Items removed lowest key first.

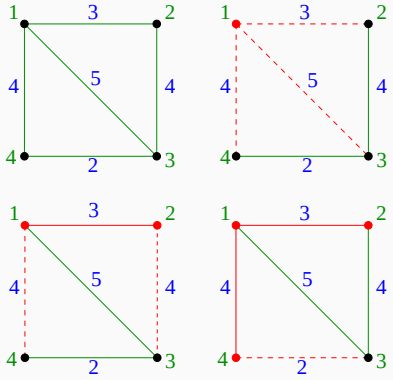
Operations:

- Q = PQcreate()
- isEmpty(Q)
- insert(Q, x)
- getMin(Q)
- deleteMin(Q)
- decreaseKey(Q,x,newkey) — updates key[x] = newkey

Prim's algorithm with priority queues

```
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
```

Example



node	1	2	3	4
key	∞	∞	∞	∞
	0	∞	∞	∞
	—	3	5	4
	—	—	4	4
	—	—	2	—
	—	—	—	—

Analysis

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

- $O(n)$ inserts
- $O(n)$ isEmptys
- $O(n)$ getMins
- $O(n)$ deleteMins
- $O(m)$ decreaseKeys

Good implementation of priority queue via Binary Heap (later in these lectures).

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

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

Analysis

Which is better—'classic' Prim with candidate arcs or Prim with PQ?

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

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

Better than $O(n^2)$

So Prim with PQ is better.

If graph is dense:

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

Worse than $O(n^2)$

So classic Prim is better.

Kruskal's MST Algorithm

Prim's algorithm is greedy:

always chooses the shortest candidate arc, pursuing short-term advantage.

Remarkably, this turned out to give optimal results:

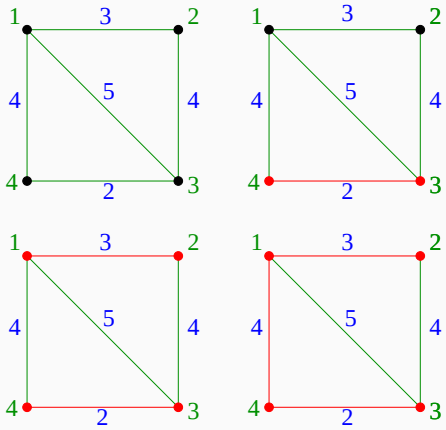
the algorithm is guaranteed to construct an MST.

An even "greedier" strategy:

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

This is Kruskal's MST algorithm.

Example



At intermediate stages get a forest (acyclic graph) rather than a tree (connected acyclic graph).

Kruskal's MST algorithm

Scheme

$F = \emptyset$ # forest being constructed

$R = \mathrm{arcs}(G)$ # 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

- Arcs are added in increasing order of weight.
- Possible implementation strategy: work on a list of arcs sorted by weight in increasing order.

Correctness

Theorem

Let G be a connected weighted graph.

Then Kruskal's algorithm constructs an MST for G.

Proof strategy much the same as Prim's algorithm.

We show that stage k we have forest $F_k \subseteq T'$, some MST T' of G.

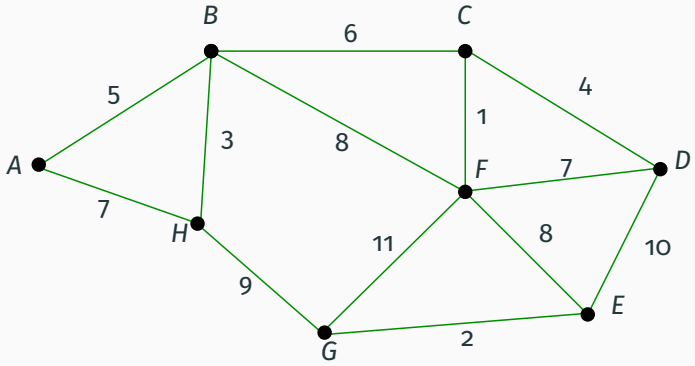
See lecture notes for the details.

Implementation

We have to do two things:

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.
   Use dynamic equivalence classes.
   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.

Example



Union-Find

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:
  x = find(sets,x)
  y = find(sets,y)

Implementation scheme for Kruskal

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 $\{1\}, \ldots, \{n\}$

$F = \emptyset$ # 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

Implementing union-find

A naive implementation:

Maintain an array leader of nodes:

- leader[x] stores the leader of the set to which node x belongs.
- Initially leader[x] = x for all nodes.

Find is now $O(1)$.

However union takes $O(n)$.

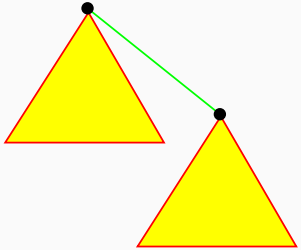
That means that it takes $O(n^2)$ to perform the $O(n)$ unions required for Kruskal.

Non-binary trees

Instead: each set is stored as a (non-binary) tree.

The root node is the representative (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.



Non-binary trees

To store the tree structure,
for each node x we maintain parent[x],
where parent[x] = x if x is the root (the leader)

Initially parent[x] = x for any node x.

union(sets,x, y) just involves setting parent[y] = x (or vice versa) — constant time $O(1)$

However find involves following parent[x] up to the root.

Time taken is bounded by the depth of the tree.

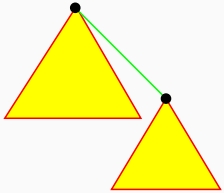
With naive merging of trees as above, this can be as much as n,
so that find becomes $O(n)$.

Reducing the depth

We clearly need to keep the depth as low as possible.

Weighted union:

since 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.

Bound on depth

Lemma

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

Proved by (strong) induction on k.

See the lecture notes.

Complexity

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 $O(m \log m) = O(m \log n)$.

Overall complexity for Kruskal: $O(m \log n)$.

Same as Prim with PQ.

Complexity

Remark

In fact can build the PQ in time $O(m)$ rather than $O(m \log m)$. Does not bring down the overall complexity here.

Can improve union-find using path compression.

The complexity for the union-find part of Kruskal then reduces to

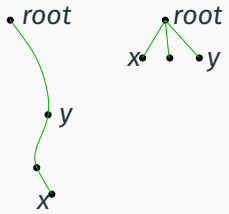
$$O((n + m) \log^* n)$$

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

$\log^* n \leq 5$ for any conceivable n that might be used.

Path compression

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.



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

Combines well with weighted union (on size) since size is unchanged by path compression.

Path compression

```
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
```

The highlighted lines perform the path compression

— omitting them gives normal find

Comparison

Kruskal: $O(m \log n)$

Prim with PQ (binary heap): $O(m \log n)$

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 to be preferred.
- On sparse graphs, where m is small (say $O(n \log n)$), then Kruskal (or Prim with PQ) give better results than classic Prim:

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

Fibonacci heaps

Priority queues can also be implemented with Fibonacci heaps rather than binary heaps.

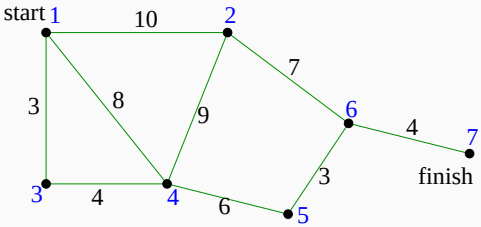
All operations are $O(1)$ apart from deleteMin, which is $O(\log n)$.

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



Network of cities joined by roads.
Wish to find the shortest route between two cities.
Find shortest path from 1 to 7.
The path 1,2,6,7 has length 10+7+4 = 21.
Can we do better?

Single Pair Shortest Path Problem

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

All Pairs versus Single Pair

Contrast with

All Pairs Shortest Path Problem

Given a weighted graph G, find the shortest paths between all pairs of nodes of G.

Algorithm due to Floyd later in these lectures.

Excellent algorithm for finding all shortest paths between any pairs of nodes.

It runs in $O(n^3)$ time.

Suppose instead that we only want to find the shortest distance from a single start node to a single finish node.

Seems to be no way to speed up Floyd.

But there is an $O(n^2)$ algorithm due to Dijkstra.

It is very closely related to Prim's MST algorithm.

Idea

As in Prim's algorithm, we build up a spanning tree starting from the start node.

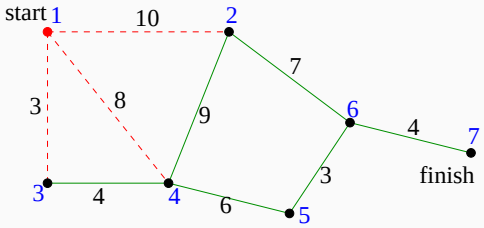
We classify nodes into

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

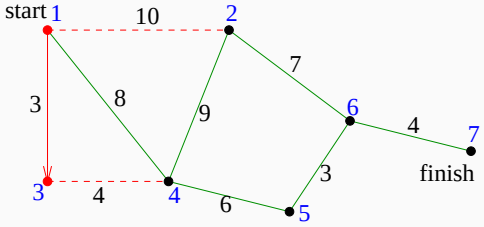
The new idea:

- We have already computed the shortest path from start to all the tree nodes: it is the path given by the tree.
- As far as the fringe nodes are concerned, we know the shortest path using the tree constructed so far. This path might be improved as the tree grows.

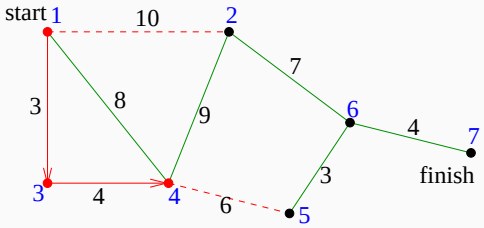
Example



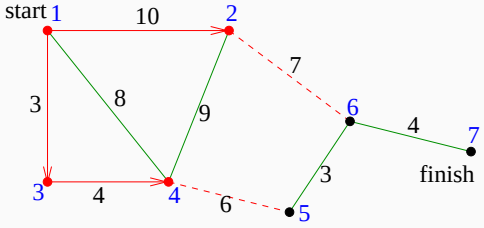
Add 3. Update candidate arcs.



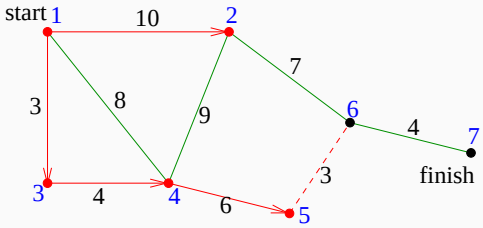
Example



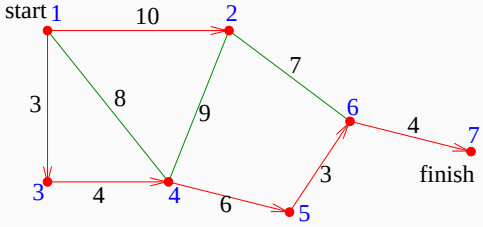
Now we add 2 rather than 5 since distance[5] = 13.



Example



Now distance[6] has reduced to 16. Add 6 and 7.



Example

Here we computed shortest paths from start for all nodes, but in general we stop as soon as finish joins the tree.

We got a spanning tree, but it is not an MST.

Implementation

We store two values for each tree or fringe node:

- its parent node in the tree
- the distance of the shortest path known.

At each stage the next node to be added to the tree is the fringe node with the smallest distance.

We then update the fringe, possibly improving the current shortest path.

We can obtain the shortest path to a node x in reverse order from the parent function:

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

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]
# end of initialisation
```

[continued

Continued

while not tree[finish] and fringe nonempty:

Select a fringe node f s.t. distance[f] is minimum

fringe[f] = false

tree[f] = true

[continued

Continued

```
    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]
```

Discussion

When terminates, read off the path using parent.

Running time $O(n^2)$, just as Prim's algorithm.

Correctness

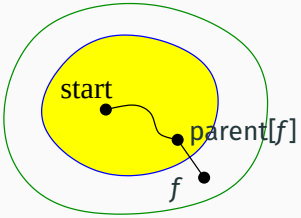
The algorithm terminates, since we clearly increase the tree each time we execute the while loop.

To see why the algorithm is correct we need to formulate an invariant.

Correctness

Invariant

1. If x is a tree or fringe node (other than start) then parent[x] is a tree node.
2. If x is a tree node (other than start) then distance[x] is the length of shortest path, and parent[x] is its predecessor along that path.
3. 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.



Correctness

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.

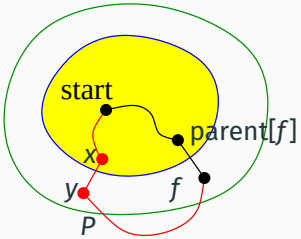
(2) Maintained

Suppose f is added to the tree.

We need to check that we have found the shortest path.

The path given by the algorithm: $\mathrm{start}, \ldots,$ parent$[f], f$

Suppose we have a different and shorter path P
(not necessarily in the tree).



Let y be the first node on P not to belong to the tree.
By (3) distance$[y] \leq$ distance from start to y using P.
Hence length of $P \geq$ distance$[y]$.
But distance$[y] \geq$ distance$[f]$ by our choice of f.
Hence P is at least as long as path(f).

Dijkstra's algorithm with 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
```

Analysis

Very much the same as for Prim's algorithm.

Dijkstra with PQ (binary heap) overall $O(m \log n)$ assuming that $n < m$ as is usually the case.

Dijkstra with PQ (Fibonacci heap) overall $O(m + n \log n)$.

A* algorithm

We now consider a different algorithm for the single pair shortest path problem.

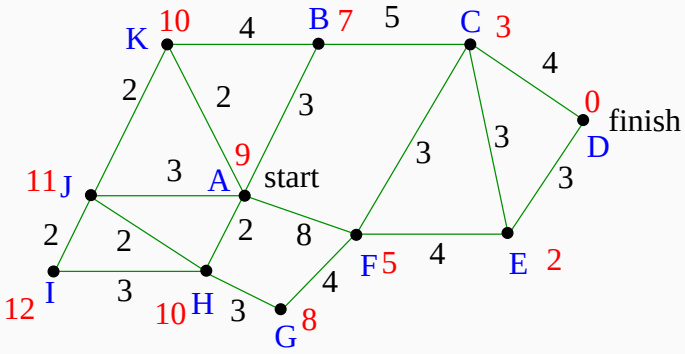
The A* algorithm is due to Hart, Nilsson and Raphael (1968).

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

If we are dealing with cities on a map, h could be the Euclidean distance (as the crow flies).

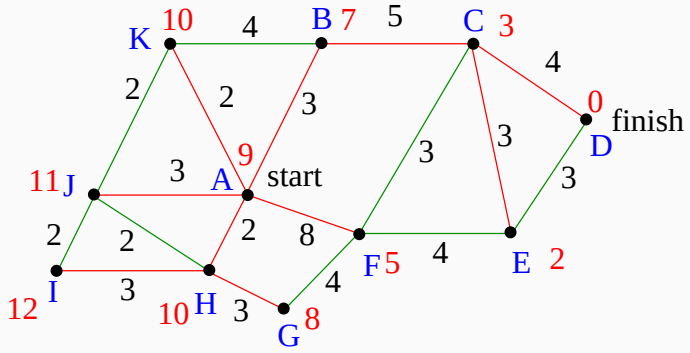
Example

Each node x has heuristic value $h(x)$ shown in red.



Example (continued)

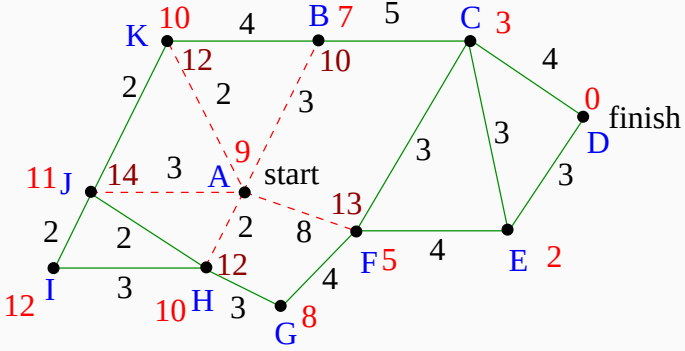
Using Dijkstra's algorithm we have to compute the entire shortest path tree before reaching node *D*.



We had to find the shortest paths to all the nodes which are closer to start than *D*. Of course we did not use the heuristic information.

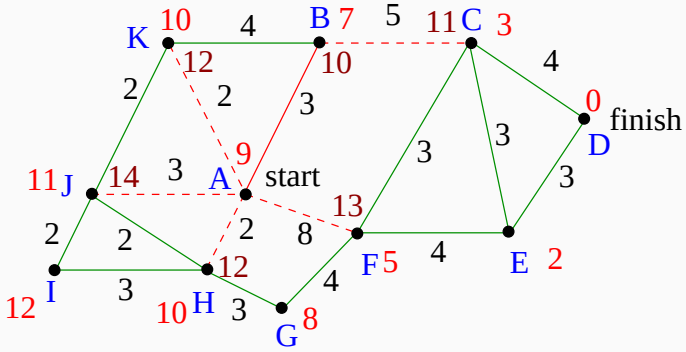
Example (continued)

With A*:



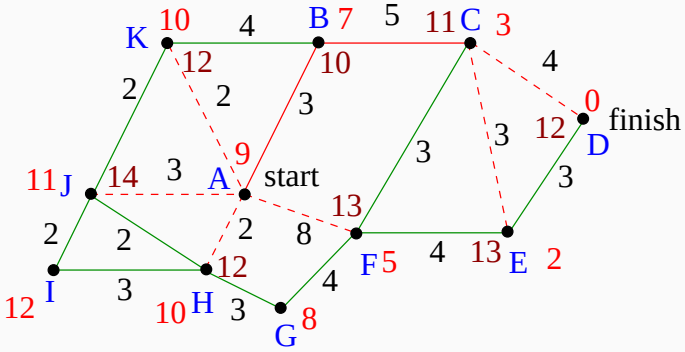
Example (continued)

With A*:



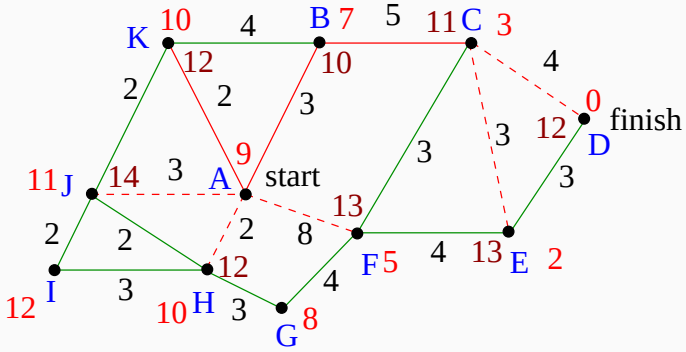
Example (continued)

With A*:



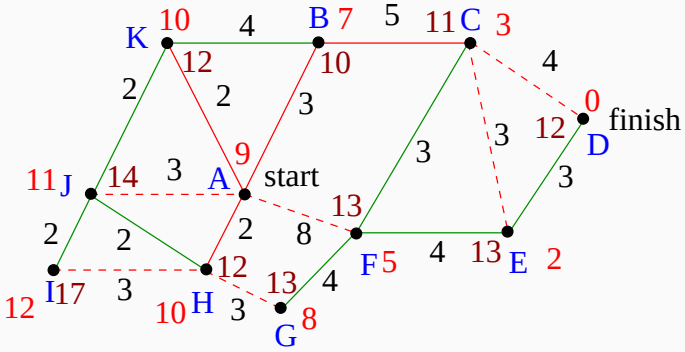
Example (continued)

With A*:



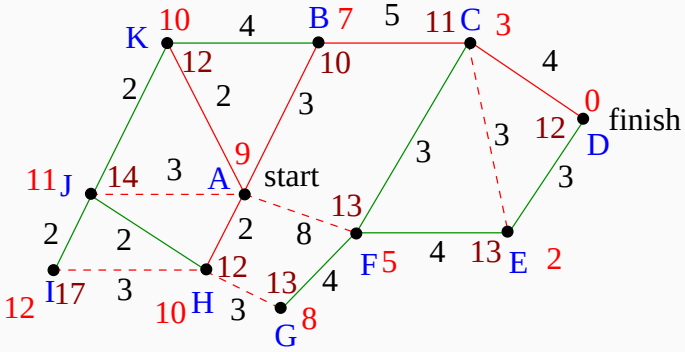
Example (continued)

With A*:



Example (continued)

With A*:



Consistent heuristic

A heuristic function is consistent if

1. for any adjacent nodes x, y we have $h(x) \leq W(x, y) + h(y)$
2. $h(\mathrm{finish}) = 0$

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

Proposition

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

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.

It is easy to check that if h is consistent then h is admissible (exercise).

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 ; $g[\mathrm{start}] = 0$

$f[\mathrm{start}] = g[\mathrm{start}] + h[\mathrm{start}]$

for x in adj[start]:

    # add x to fringe

    fringe[x] = true

    parent[x] = start

    $g[x] = W[\mathrm{start}, x]$

    $f[x] = g[x] + h[x]$

[continued

Continued

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

[continued

Continued

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 $g[\text{finish}]$

Remarks

1. The set of tree nodes is often called the closed set and the set of fringe nodes is the open set.
2. If we set $h(x) = 0$ for all nodes x then h is consistent, and the A* algorithm is just Dijkstra's algorithm.
3. 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.
4. We have presented A* for consistent heuristics, for simplicity and for its closeness to Dijkstra's algorithm
5. 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).

Correctness

We now show that A* correctly computes the shortest path.

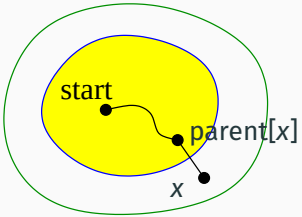
The algorithm terminates, since we clearly increase the tree each time we execute the while loop.

To see why the algorithm is correct we need to formulate an invariant, which is essentially the same as for Dijkstra's algorithm.

Correctness

Invariant

1. If x is a tree or fringe node (other than start) then parent[x] is a tree node.
2. If x is a tree node (other than start) then $g[x]$ is the length of shortest path, and parent[x] is its predecessor along that path.
3. If x is a fringe node then $g[x]$ is the length of the shortest path where all nodes except x are tree nodes. Furthermore, parent[x] is its predecessor along that path.



Correctness

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.

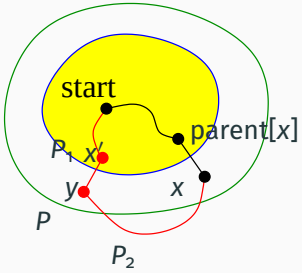
(2) Maintained

Suppose x is added to the tree.

We need to check that we have found the shortest path.

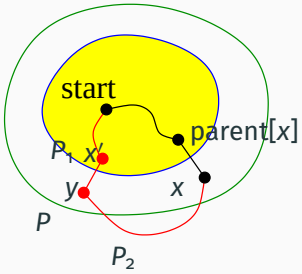
The path given by the algorithm: $\mathrm{start}, \ldots,$ parent$[x], x$ has length $g[x]$.

Suppose we have a different and shorter path P
(not necessarily in the tree).



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

(2) Maintained



$f[y] = g[y] + h[y]$
$\leq g[y] + \text{len}(P_2) + h[x]$ (consistency of h)
$\leq \text{len}(P_1) + \text{len}(P_2) + h[x]$ (by (3) for y)
$= \text{len}(P) + h[x]$
$< g[x] + h[x]$ (assumption)
$= f[x]$
But $f[x] \leq f[y]$ 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
```

Transitive closure

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

Recall from Discrete Mathematics that the transitive closure of R is

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

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. Arc from x to y iff $R(x, y)$.

NB No parallel arcs. We could have loops where $R(x, x)$.

It is easy to see that $R^k(x, y)$ iff there is a path of length k from x to y.

So $R^+(x, y)$ iff there is a path of length ≥ 1 from x to y.

Transitive closure

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:

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

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.

Warshall's algorithm

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.

We say that nodes $x_2, \ldots, x_{k-1}$ are intermediate nodes of p.

We look for paths which use nodes $\leq i$ 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 one, 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.

Warshall's algorithm

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.

Warshall's algorithm

input A

copy A into B (array of Booleans) # $B = B_0$

for k = 1 to n:

  # $B = B_{k-1}$

  for i = 1 to n:

    for j = 1 to n:

      $b_{ij} = b_{ij}$ or (b_{ik} and b_{kj})

  # $B = B_k$

$B = B_n$

return B

Complexity is clearly $O(n^3)$.

Floyd's algorithm

We return to:

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 $B_k[i,j] = \infty$.

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.

Floyd's algorithm

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

(Also works if there is no shortest path just using nodes $\leq k$)

Floyd's algorithm

input A

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

Complexity is clearly $O(n^3)$.

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:

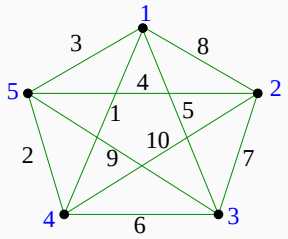
- move through the sub-problems in order
- solve each sub-problem using the stored solutions of the previous sub-problems and storing the new solution for later use
- solve the main problem as the final sub-problem

The Travelling Salesman Problem

Given some cities and roads between them, a travelling salesman wants to find a tour of all the cities with the shortest distance.

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.



The restriction to complete graphs is not that strong: if arcs were 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.

Complexity

TSP is clearly related to

- Hamiltonian Circuit Problem (HCP)
- 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.

Just like HCP, TSP is NP-complete and so unlikely to have a polynomial solution.

A better algorithm than just checking every possible tour yields a running time of $O(n^2 2^n)$ (Bellman, Held & Karp).

- still worse than exponential

Bellman-Held-Karp algorithm

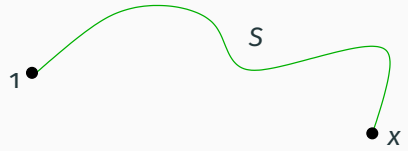
Another example of dynamic programming.

Let (G, W) have $\mathrm{Nodes} = \{1, \ldots, n\}$.

Idea: Fix a start node, say 1.

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

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.



Bellman-Held-Karp algorithm

A TS tour can start at 1 wlog.

Let the last node before returning to 1 be x.

The least cost of such a tour is $C(\text{Nodes} \setminus \{1, x\}, x) + W(x, 1)$.

So solution to TSP is

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

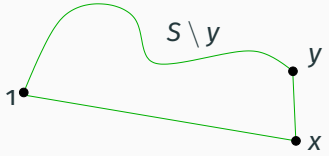
Calculating $C(S, x)$

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).



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

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

Bellman-Held-Karp algorithm in pseudocode

Input (G, W)

Choose $\text{start} \in \text{nodes}(G)$

for $x \in \text{Nodes} \setminus \{\text{start}\}$:
  $C[\emptyset, x] = W[\text{start}, x]$

Process sets S in increasing order of size.
for $S \subseteq \text{Nodes} \setminus \{\text{start}\}$ with $S \neq \emptyset$:
  for $x \in \text{Nodes} \setminus (S \cup \{\text{start}\})$:
    # Find $C[S, x]$
    $C[S, x] = \infty$
    for $y \in S$:
      $C[S, x] = \min(C[S \setminus \{y\}, y] + W[y, x],\ C[S, x])$

Now have calculated and stored all values of $C[S, x]$

[continued

continued

opt = ∞
for $x \in \text{Nodes} \setminus \{\text{start}\}$:
  opt = min($C[\text{Nodes} \setminus \{\text{start}, x\}, x] + W[x, \text{start}]$, opt)
return opt

Complexity

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

Overall $O(n^2 2^n)$

Hamiltonian circuit problem

The Bellman-Held-Karp algorithm can be adapted to solve the HCP.

Complexity is still $O(n^2 2^n)$.

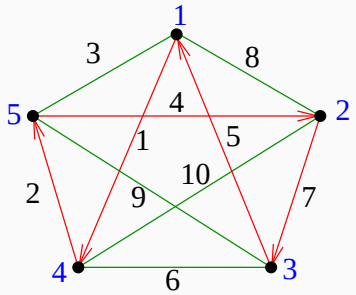
Approximate Methods for TSP

Despite exact solutions taking too long, we still want to solve TSP—there are applications in circuit design, etc.

Approximate methods.

For instance, we could try a “greedy” algorithm which always chooses the shortest available arc,

the so-called nearest neighbour heuristic (NNH).



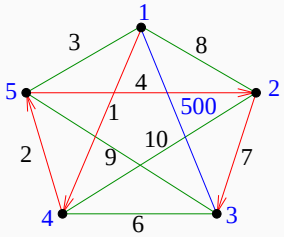
Starting from node 1 we get the tour $1, 4, 5, 2, 3, 1$ with a total weight of

$1 + 2 + 4 + 7 + 5 = 19$

which is quite good.

Approximate Methods

However such a short-sighted method can also fail dramatically.



Suppose that we change the weight of $(1, 3)$ from 5 to 500.

Then clearly this arc should be avoided,
but NNH is forced to choose it on the last step.

Part III

Algorithm Analysis

Algorithm Analysis

Problem *P*

- sort a list
- shortest path in a graph
- multiply two matrices

S the set of all possible solutions for *P*

- MergeSort
- QuickSort, etc.

Which of the available algorithms is best?

We shall rank algorithms according to how fast they run:
their time complexity

Lower Bounds

A harder question:

> *Can we improve our existing algorithms, or have we already found the best possible algorithm for P?*

- Can improve $\Longrightarrow$ go on looking
- Found best $\Longrightarrow$ no need to waste time looking

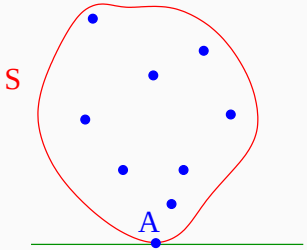
We must reason about all possible members of S,
not just the ones we know already.

- a minimum amount of work which every member of S must do
  (e.g. inspect every element in a list)
- provides a lower bound on the time complexity of
  any algorithm for P.

Which Problems?

We examine searching and sorting:

1. Frequently used
   - efficiency is particularly important
2. Analysis is well worked out
   - Optimal (best possible) algorithms are known



Searching a list

L a list of elements of type D

$$L[0], \ldots, L[n-1] \qquad (n \geq 1)$$

Random access $L[k]$

Problem: Searching an unordered list L

For $x : D$

- if x in L return k such that $L[k] = x$
- if x not in L return "not found"

A particular algorithm:

Linear Search

Inspect $L[0], L[1], \ldots, L[n-1]$ in turn.

Stop and return index if x found. Otherwise return "not found".

Analysis

How long does LS take?

Count the number of comparisons $x = L[k]$

This is general:

will work for all algorithms for the problem

Number of comparisons varies.

With $n = 4$, $D = \mathbb{N}$, $x = 5$:

- $[5, 4, 3, 6]$ 1 comparison
- $[3, 4, 6, 2]$ 4 comparisons

With input size n ranges between

1 best case
n worst case

comparisons.

Worst and Average Case

Worst case analysis

$W(n) =$ largest number of comparisons for input size n

[Here $W(n) = n$]

Average case analysis

$A(n) =$ average number of comparisons for input size n

NB We need to know (or make a reasonable assumption about) the probability distribution:

- how likely is x to be in L?
- how likely is $x = L[k]$?

Which is better?

Average case might seem superior.

But worst case has two advantages:

1. Can guarantee on input size n that never takes longer than $W(n)$
2. $W(n)$ is easier to compute than $A(n)$ and often gives similar results (same order of complexity)

Best-case analysis is not a good idea:
can "tune up" a slow algorithm to look fast on particular inputs.

LS is optimal

LS can be varied, but it cannot be improved.

- LS is optimal (in worst case)

Justification:

Take any A which solves the search problem.

Claim: If A returns "not found" then must have inspected every entry of L.

Proof: Suppose for a contradiction that A did not inspect $L[k]$.

$$\begin{array}{lccccccc} L: & L[0] & L[1] & \ldots & L[k] & \ldots & L[n-1] \\ L': & L[0] & L[1] & \ldots & x & \ldots & L[n-1] \end{array}$$

On input L', A will return "not found", which is wrong. Contradiction

Hence in worst case n comparisons are needed
— *lower bound*

Ordered Lists

Problem

Search an ordered list.

LS will solve this, but can do better.

Modified LS

Inspect $L[0], L[1], \ldots$ as before.

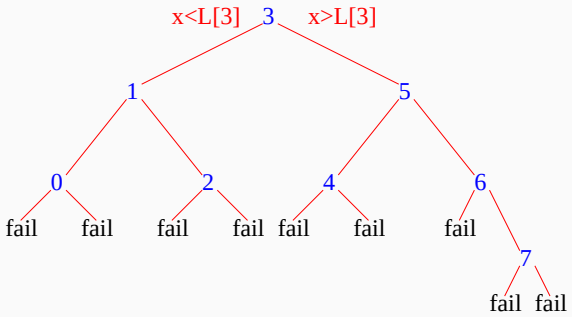
Stop if find x, or if $L[k] > x$ (meaning that x not in L).

$W(n) = n$ still.

Expect that $A(n)$ has improved.

Binary Search

Decision tree for $n = 8$



$W(8) = 4$

(e.g. when $x = L[7]$)

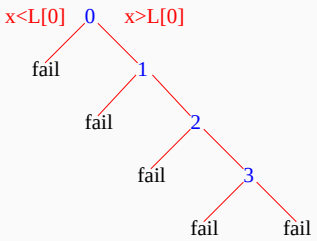
Big improvement on MLS.

BS is optimal.

- Keeps the tree depth low for the same number of nodes.

Modified Linear Search Tree

MLS decision tree for $n = 4$



$$W(4) = 4$$

(look at depth of tree)

The decision tree represents the algorithm.

Floor and Ceiling

Definition

The floor of $x \in \mathbb{R}$ is the greatest $n \in \mathbb{Z}$ such that $n \leq x$.

Notation $\lfloor x \rfloor$

For instance, $\lfloor 23.18 \rfloor = 23, \lfloor \pi \rfloor = 3$

Definition

The ceiling of $x \in \mathbb{R}$ is the least $n \in \mathbb{Z}$ such that $n \geq x$.

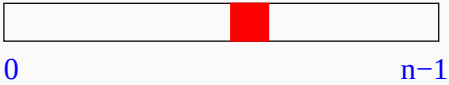
Notation $\lceil x \rceil$

For instance, $\lceil 0.001 \rceil = 1$.

Worst-case analysis for Binary Search

$W(1) = 1$

$W(n) = 1 + W(\lfloor n/2 \rfloor)$



Each half has length $\leq \lfloor n/2 \rfloor$.

Solve by repeated expansion:

$$
\begin{aligned}
W(n) &= 1 + W(\lfloor n/2 \rfloor) \\
&= 1 + [1 + W(\lfloor n/4 \rfloor)] \\
&= 1 + 1 + 1 + W(\lfloor n/8 \rfloor) \\
&\quad = 3 + W(\lfloor n/2^3 \rfloor) \\
&\quad \cdots \\
&= 1 + \cdots + 1 + W(1)
\end{aligned}
$$

How many ones?

Examples

$$W(8) = 1 + 1 + 1 + W(1)$$

$$\begin{aligned} W(9) &= 1 + W(4) \\ &= 1 + 1 + W(2) \\ &= \underbrace{1 + 1 + 1}_{3} + W(1) \end{aligned}$$

Number of 1s is number of times divisible by 2.

k where $2^k \leq n < 2^{k+1}$

$k = \lfloor \log n \rfloor$ — logarithm base 2

$$W(n) = 1 + \lfloor \log n \rfloor$$

Can check this against depth of trees.

Binary Search is Optimal

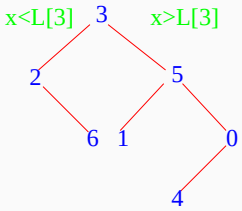
We now show that Binary Search is optimal.

Represent any possible search algorithm *A* by a decision tree.

The tree must have *n* nodes at least.

The tree will be binary.

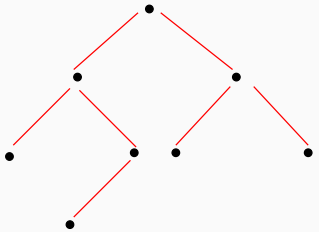
e.g.



So it must have at least a certain depth.

Binary Search is Optimal

e.g. $n = 8$



Depth at least 3.

Depth vs. Nodes

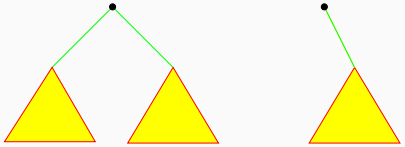
Proposition

If a binary tree has depth d, then it has $\leq 2^{d+1} - 1$ nodes.

Proof by induction:

Base case. $d = 0$. $2^{0+1} - 1 = 1$. Checked.

Induction Step. Assume true for d.



Suppose tree has depth $d + 1$.
Children have depth $\leq d$,
and $\leq 2^{d+1} - 1$ nodes (by inductive hypothesis).

Total number of nodes $\leq 1 + (2^{d+1} - 1) + (2^{d+1} - 1) = 2^{(d+1)+1} - 1$

Minimality of Binary Search

The tree for algorithm A has $\geq n$ nodes.

If the depth is d then

$$n \leq 2^{d+1} - 1 \qquad (*)$$

The worst-case performance of A is $d + 1$.

From (*):

$$d + 1 \geq \log(n + 1)$$

In fact

$$d + 1 \geq \lceil \log(n + 1) \rceil$$

For Binary Search,

$$W(n) = 1 + \lfloor \log n \rfloor$$

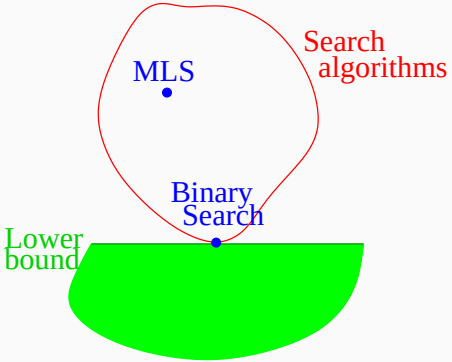
We can show

$$\lceil \log(n + 1) \rceil = 1 + \lfloor \log n \rfloor$$

(see notes)

So any algorithm must do as many comparisons as Binary Search.

Binary Search is optimal.



Orders

Suppose for some algorithm

$$W(n) = 8n^2 + 300n + 70$$

As n gets large, $8n^2$ is the most important term.

If we ignore the constant, $W(n)$ is order n^2.

For a different algorithm

$$W'(n) = 2n^3 + n^2 + 4$$

This is order n^3.

So the first algorithm is of lower order:

- should be preferred for large n, even though possibly $W'(n) < W(n)$ for small n.

Constant Factors can be Ignored

When calculating worst-case complexity, it is often desirable to ignore constant factors.

Example: Matrix Multiplication

$$A.B = C$$

$n \times n$ matrices

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

This requires n multiplications and $n - 1$ additions
$(= 2n - 1$ operations)

$$W(n) = n^2(2n - 1)$$

But if just count multiplications

$$W(n) = n^3$$

However the orders are the same: n^3

A Hierarchy of Orders

Polynomial

$$1 \quad n \quad n^2 \quad n^3 \quad \ldots$$

Exponential

$$2^n \quad 3^n \quad 4^n \quad \ldots$$

We are also interested in logarithms.

e.g. for Binary Search

$W(n)$ is order $\log n$

$$1 \quad \log n \quad n \quad n \log n \quad n^2 \quad n^2 \log n \quad \ldots$$

Definitions

$$\mathbb{R}^+ = \{x \in \mathbb{R} : x \geq 0\}$$

Let $f, g : \mathbb{N} \to \mathbb{R}^+$

1. f is $O(g)$ iff $\exists m \in \mathbb{N}\ \exists c \in \mathbb{R}^+$ such that

$$\forall n \geq m\ \ f(n) \leq c.g(n)$$

2. f is $\Theta(g)$ iff f is $O(g)$ and g is $O(f)$.

Example

We know that $8n^2 + 300n + 70$ is $\Theta(n^2)$.

We prove this just to illustrate the definition.

Clearly n^2 is $O(8n^2 + 300n + 70)$.

We need to find $m \in \mathbb{N}$, $c \in \mathbb{R}^+$ such that

$$8n^2 + 300n + 70 \leq cn^2$$

(all $n \geq m$)

Guess $c = 9$:

$$9n^2 - (8n^2 + 300n + 70) = n^2 - 300n - 70$$

This is ≥ 0 for n sufficiently large

$$m = 1000$$

So $8n^2 + 300n + 70$ is $O(n^2)$.

Order as an Upper Bound

Sometime the exact order is unknown.

Matrix Multiplication

We have seen an $\Theta(n^3)$ algorithm.

But since there may be a faster algorithm, we say that the problem of matrix multiplication is $O(n^3)$.

- A lower bound of $\Theta(n^2)$ is known.
- Strassen 1969: $O(n^{\log 7}) = O(n^{2.807})$
- Coppersmith-Winograd 1987: $O(n^{2.376})$
- Stothers 2010: $O(n^{2.373})$
- Williams 2012: $O(n^{2.3729})$
- Le Gall 2014: $O(n^{2.3728639})$
- Alman, Williams 2021: $O(n^{2.3728596})$
- Daun, Wu, Zhou 2022: $O(n^{2.37188})$

Exercise

List the following from lowest to highest order, indicating any which have the same order:

- $n^3/2$
- $\log n$
- 2^n
- $n^2 - n^3 + n^5/2$
- $\log(n^2)$
- 2^{n-1}
- $n/5$
- $6n + \log n$
- $2^{\sqrt{n}}$

Strassen's Algorithm

Assume we are multiplying two $n \times n$ matrices:

$$AB = C$$

Start with $n = 2$.

Then

$$C = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

This takes 8 multiplications (and 4 additions).

Strassen's Algorithm

Strassen: can do $n = 2$ in only 7 multiplications (and 18 additions).

$$C = \begin{pmatrix} x_1 + x_4 - x_5 + x_7 & x_3 + x_5 \\ x_2 + x_4 & x_1 + x_3 - x_2 + x_6 \end{pmatrix}$$

where

$$\begin{aligned} x_1 &= (a_{11} + a_{22}) * (b_{11} + b_{22}) & x_5 &= (a_{11} + a_{12}) * b_{22} \\ x_2 &= (a_{21} + a_{22}) * b_{11} & x_6 &= (a_{21} - a_{11}) * (b_{11} + b_{12}) \\ x_3 &= a_{11} * (b_{12} - b_{22}) & x_7 &= (a_{12} - a_{22}) * (b_{21} + b_{22}) \\ x_4 &= a_{22} * (b_{21} - b_{11}) \end{aligned}$$

Note that commutativity of multiplication is not used.

Hence we can generalise to matrices.

Strassen's Algorithm

Suppose that $n = 2^k$.

Divide up matrices into four quadrants each $n/2 \times n/2$:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Compute C_{ij} using the formulas for c_{ij}.

Recursively compute each multiplication by further subdivision until bottom out at $n = 2$.

Strassen's Algorithm

Need to add/subtract 18 matrices of dimension $n/2 \times n/2$.
Recursively perform 7 multiplications of $n/2 \times n/2$ matrices.

Number of arithmetic operations $A(k)$ for $n = 2^k$:

$$A(0) = 1$$
$$A(k) = 7A(k-1) + 18(n/2)^2$$

Can solve by repeated expansion and then summing the resulting geometric progression.

Solution:

$$A(k) = 7 \cdot 7^k - 6 \cdot 4^k = 7 \cdot 7^{\log n} - 6n^2 = 7 \cdot n^{\log 7} - 6n^2 \approx 7 \cdot n^{2.807} - 6n^2$$

So Strassen's matrix multiplication is $\Theta(n^{2.807})$.

What if $n \neq 2^k$? Can add extra row and column
to keep the dimension even to allow subdivision.

Divide and Conquer Algorithms

Strassen's algorithm is an example of a divide and conquer algorithm.

- Divide problem into a subproblems of size n/b (here $a = 7$ and $b = 2$):
  May take work to set up the subproblems (here matrix addition).
- Solve each subproblem recursively.
- Then combine to get the result.
  Again, this may take work (here matrix addition).

We shall see further examples: MergeSort, QuickSort.

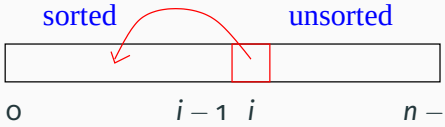
Sorting

1. Measure time complexity of sorting algorithms.
2. Obtain lower bounds for the amount of work required.

Techniques

1. Recurrence relations
2. Decision trees
3. Finding orders of functions

Insertion Sort



Insert $L[i]$ into $L[0\,..\,i-1]$ in correct position.

Then $L[0\,..\,i]$ is sorted.

Insertion performed by letting $L[i]$ filter downwards by successive swaps.

This takes between 1 and i comparisons.

- worst case when $L[i]$ below $L[0]$

Perform insertion for $i = 1$ to $n-1$:

$$W(n) = \sum_{i=1}^{n-1} i$$

Solution

Fact

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Hence

$$W(n) = \frac{n(n-1)}{2}$$

The worst case actually arises when the list is in reverse order.

$W(n)$ is $\Theta(n^2)$.

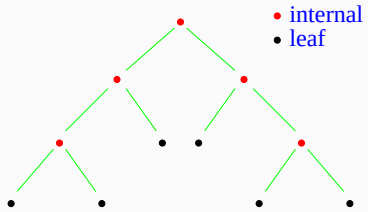
We wish to see if this can be improved upon.

Lower Bounds

Idea: express sorting algorithm as decision tree.

The internal nodes are the comparisons.

The leaves are the results (the rearranged lists).



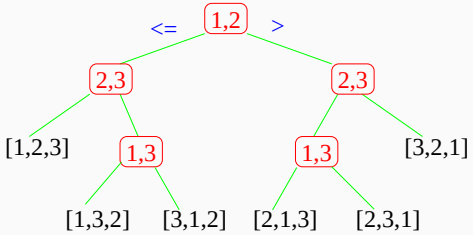
We then argue that to have a certain number of leaves, the tree must have sufficient depth.

Depth = worst-case number of comparisons

Example: Algorithm 3-Sort

Sorts a list of length 3.

Simply given by a decision tree.



Each leaf is a possible permutation of L.

Worst-case number of comparisons is 3.

Exercise

Find a decision tree for Insertion Sort on $n = 3$.

Optimality for $n = 3$

Any decision tree for sorting a list of length 3 must have $3! = 6$ leaves.

Cannot have depth ≤ 2, as all binary trees of depth ≤ 2 have ≤ 4 leaves.

So depth at least 3.

Worst-case number of comparisons at least 3.

Hence 3-Sort (and Insertion Sort) are optimal for $n = 3$.

General Case

Sorting a list of length n.

There are $n!$ permutations.

So the decision tree must have $n!$ leaves (at least).

How deep must a binary tree be to have $n!$ leaves?

e.g. $n = 4$?

Bound on Leaves

Proposition

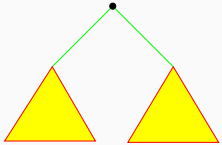
If a binary tree has depth d then it has $\leq 2^d$ leaves.

Proof: By induction on d.

$d = 0$: $2^0 = 1$. Checked.

Assume if depth d then $\leq 2^d$ leaves.

Consider a tree of depth $d + 1$:



Subtrees have depth $\leq d$ and so $\leq 2^d$ leaves.

So tree of depth $d + 1$ has $\leq 2^d + 2^d = 2^{d+1}$ leaves.

Lower Bound for Sorting

Decision tree has $\geq n!$ leaves and depth d.

So

$$\begin{aligned} 2^d &\geq n! \\ d &\geq \log(n!) \\ d &\geq \lceil \log(n!) \rceil \end{aligned}$$

Lower bound for sorting in worst case

Any algorithm for sorting by comparisons must perform at least

$$\lceil \log(n!) \rceil$$

comparisons in worst case.

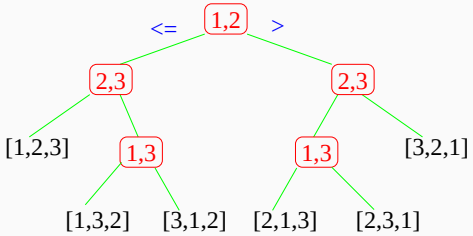
How Good is Insertion Sort?

We compare the lower bound $\lceil \log(n!) \rceil$ with $W(n)$ for Insertion Sort.

n	1	2	3	4	5	6	7	8	9	10	11	12
$\lceil \log(n!) \rceil$	0	1	3	5	7	10	13	16	19	22	26	29
$\frac{n(n-1)}{2}$	0	1	3	6	10	15	21	28	36	45	55	66

Average Case

Example: 3-Sort.



6 permutations.

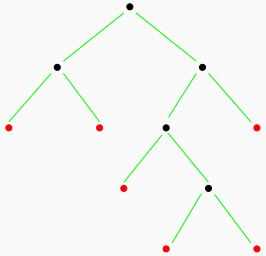
Assume each equally likely. Probability 1/6.

Average number of comparisons $2\frac{2}{3}$.

Not much less than worst case (3).

Total Path Length

Consider this tree:



6 leaves.
Depths are $2, 2, 3, 4, 4, 2$
Average depth

$$\frac{1}{6}(2 + 2 + 3 + 4 + 4 + 2)$$

Definition

The total path length of a tree is the sum of the depths of all leaf nodes.

Minimising Total Path Length

Suppose T is a decision tree for sorting a list of length n.

Suppose T has $n!$ leaves and total path length b.

Then average number of comparisons is

$$\frac{b}{n!}$$

Given a fixed number of leaves ($n!$) we want to find a lower bound on the total path length.

It turns out that the total path length is lowest when leaves are at roughly equal depth.

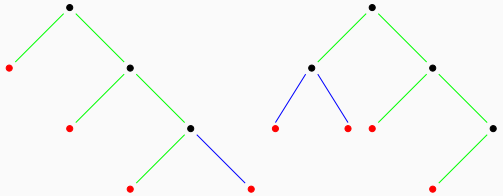
Balanced Trees

Definition

A tree of depth d is balanced if every leaf is at depth d or $d-1$.

Proposition

If a tree is unbalanced then we can find a balanced tree with the same number of leaves without increasing the total path length.



Has the total path length reduced?

Lower Bound for Average Case

So when looking for a lower bound on total path length,
we can just consider balanced trees.

But in a balanced tree of depth d, average is between d and $d - 1$.

Hence:

Lower bound for average case

Any algorithm for sorting a list of length n must perform at least

$$\lfloor \log(n!) \rfloor$$

comparisons in average case.

Room for Improvement

We have a lower bound on the number of comparisons needed in worst/average case to sort a list of length n.

e.g. for $n = 10$ require:

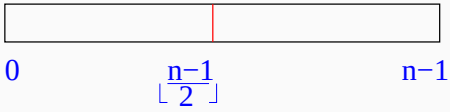
- worst case 22
- average case 21.84

A good sorting algorithm will use not much more than this number.

Insertion Sort is poor, since it uses 45 comparisons in worst case.

So we look for a better sorting algorithm.

MergeSort



1. Divide roughly into two.
2. Sort each half separately (by recursion)
3. Merge the two halves.

We need to know how many comparisons must be done to merge the two halves.

The merging will be done by comparing the current
least elements of the lists, and outputting the smaller.

Examples

How many comparisons?

1. $L_1 : [2, 3, 5, 7]$ with $L_2 : [4, 6, 9, 11]$
2. $L_3 : [2, 4, 6, 8]$ with $L_4 : [3, 5, 7, 9]$

Worst-Case Analysis

Adding an element to the merged list takes one comparison until one of the lists is exhausted.

In the worst case, only the last element is transferred “for free”.

So worst case is $n - 1$ comparisons.

$$\begin{aligned} W(1) &= 0 \\ W(n) &= n - 1 + W(\lceil \tfrac{n}{2} \rceil) + W(\lfloor \tfrac{n}{2} \rfloor) \end{aligned}$$

MergeSort Performs Well

We compare the lower bound $\lceil \log(n!) \rceil$ with $W(n)$ for MergeSort.

n	1	2	3	4	5	6	7	8	9	10	11	12
$\lceil \log(n!) \rceil$	0	1	3	5	7	10	13	16	19	22	26	29
$W(n)$	0	1	3	5	8	11	14	17	21	25	29	33

Does MergeSort remain close to optimal as n gets large?

Solving the Recurrence Relation

$$\begin{aligned} W(1) &= 0 \\ W(n) &= n - 1 + W(\lceil \tfrac{n}{2} \rceil) + W(\lfloor \tfrac{n}{2} \rfloor) \end{aligned}$$

Assume $n = 2^k$.

$$W(n) = n - 1 + 2W(n/2)$$

Use repeated expansion.

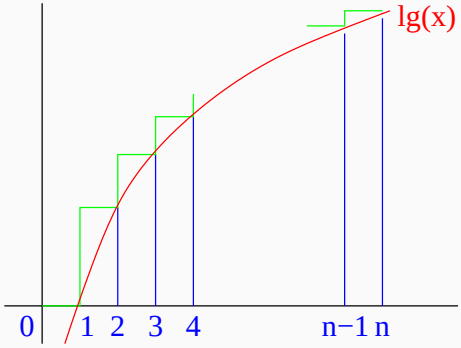
$$\begin{aligned} W(n) &= kn - n + 1 \\ &= n \log(n) - n + 1 \end{aligned}$$

This is $\Theta(n \log n)$.

How does $n \log(n) - n + 1$ compare with $\lceil \log(n!) \rceil$?

$\log(n!)$

$$\log(n!) = \log(1) + \log(2) + \cdots + \log(n)$$



$\log(n!)$ is the area under the green line.
Roughly the same as area under red curve.

$$\int_1^n \log x \, dx$$

Now $\log x = c \ln x$, and

$$\int_1^n \ln x \, dx = [x \ln(x) - x]_1^n = n \ln(n) - n + 1$$

But

$$W(n) = n \log(n) - n + 1$$

So $W(n)$ and $\lceil \log(n!) \rceil$ are of the same order.

Recurrence relations

Worst case comparisons for Binary Search:

$$W(1) = 1$$
$$W(n) = W(n/2) + 1$$

Worst case comparisons for MergeSort:

$$W(1) = 0$$
$$W(n) = 2W(n/2) + (n-1)$$

Number of arithmetic operations for Strassen's Algorithm:

$$A(1) = 1$$
$$A(n) = 7A(n/2) + 18(n/2)^2$$

All examples of Divide and Conquer algorithms.

Divide and Conquer

General form for Divide and Conquer

Work $T(n)$ for input size n

Split into a sub-problems of size n/b.

Non-recursive work (split and combine) is $f(n)$.

$$T(n) = aT(n/b) + f(n)$$

(plus base cases)

We shall show how to solve such a recurrence relation up to Θ.

Recursion Trees

Example:

$$T(1) = 1$$
$$T(n) = aT(n/2) + n$$

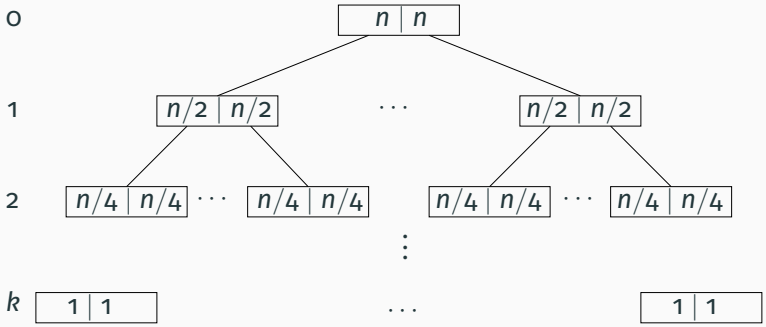
(like MergeSort if $a = 2$)

Create a recursion tree:

- Start with input of size n
- Each unfolding of the recursion takes us down one level to a subproblems with size halved
- Each node of tree records size and non-recursive work done

Suppose $n = 2^k$.

Recursion Trees



The total work done is the sum of the work at each of the $k+1$ levels.

$$n + a(n/2) + a^2(n/2^2) + \cdots + a^{k-1}n/(2^{k-1}) + a^k$$
$$= n + (a/2)n + (a/2)^2 n + \cdots + (a/2)^{k-1}n + a^k$$

Geometric series.

Geometric Series

Proposition

$$\sum_{i=o}^{k} ar^i = \frac{a(r^{k+1}-1)}{r-1} \text{ provided } r \neq 1$$

Clearly sum is $(k+1)a$ if $r = 1$.

Corollary

Let $t(n)$ be the largest term in the geometric progression

$$a, ar, ar^2, \ldots, ar^k$$

where r is non-negative, $r \neq 1$ and r does not depend on n (though a and k can depend on n). Then

$$\sum_{i=o}^{k} ar^i = \Theta(t(n))$$

Three cases

Apply Corollary to

$$n + (a/2)n + (a/2)^2 n + \cdots + (a/2)^{k-1} n + a^k$$

Here $r = a/2$.

- If $a < 2$ greatest term is n.

  $$T(n) = \Theta(n)$$

  Non-recursive work at level 0 dominates.
- If $a = 2$ then

  $$T(n) = (k+1)n = \Theta(n \log n)$$

  Work is (roughly) evenly spread at all levels. (cf. MergeSort)
- If $a > 2$ greatest term is $a^k = a^{\log n} = n^{\log a}$.

  $$T(n) = \Theta(n^{\log a})$$

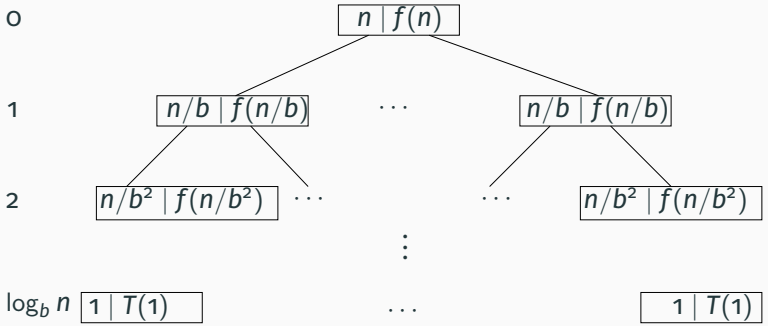
  Base cases (leaves of recursion tree) dominate.

Three cases depending on a.

Towards the general case

$$T(n) = aT(n/b) + f(n)$$

(plus base cases) Recursion tree will have $1 + \log_b n$ levels.



Towards the general case

Level 0: work $f(n)$
Level 1: work $af(n/b)$
Level 2: work $a^2 f(n/b^2)$
$\vdots$
Level $\log_b n$: work $\Theta(a^{\log_b n})$

For the bottom level we just need to know the number of leaves, as the work for each is constant.

Suppose that $f(n) = n^c$.
Then ratio $r = a/b^c$.

Let the critical exponent be $E = \log_b a = \log a / \log b$.
Then $r > 1$ iff $a > b^c$ iff $\log_b a > c$ iff $E > c$.

Three cases

Three cases:

- $E < c$:
$$T(n) = \Theta(f(n))$$
- $E = c$:
$$T(n) = \Theta(f(n) \log_b n) = \Theta(f(n) \log n)$$
- $E > c$:
$$T(n) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}) = \Theta(n^E)$$

Master Theorem

Master Theorem

$$T(n) = aT(n/b) + f(n)$$

has solutions as follows,
where $E = \log a / \log b$ is the critical exponent:

1. If $n^{E+\epsilon} = O(f(n))$ for some $\epsilon > 0$ then $T(n) = \Theta(f(n))$.
2. If $f(n) = \Theta(n^E)$ then $T(n) = \Theta(f(n) \log n)$.
3. If $f(n) = O(n^{E-\epsilon})$ for some $\epsilon > 0$ then $T(n) = \Theta(n^E)$.

Examples

Worst case comparisons for Binary Search:

$$W(n) = W(n/2) + 1$$

Here $a = 1$ and $b = 2$ and $f(n) = \Theta(n^0)$.

Then $E = \log a / \log b = 0$. So

$$W(n) = \Theta(n^0 \log n) = \Theta(\log n)$$

Worst case comparisons for MergeSort:

$$W(n) = 2W(n/2) + (n - 1)$$

Here $a = 2$ and $b = 2$ and $f(n) = \Theta(n^1)$.

Then $E = \log a / \log b = 1$. So

$$W(n) = \Theta(n \log n)$$

Examples

Number of arithmetic operations for Strassen's Algorithm:

$$A(n) = 7A(n/2) + 18(n/2)^2$$

Here $a = 7$ and $b = 2$, $f(n) = \Theta(n^2)$.

Then $E = \log a / \log b = \log 7 > 2$. So

$$A(n) = \Theta(n^{\log 7})$$

Note that any improvement to $f(n)$ here will not help with the order of $A(n)$.

QuickSort

Split the list around the first element.

e.g.

$$L = [7, 2, 10, 12, 3, 1, 8]$$

Split around 7.

Get

$$[3, 2, 1, 7, 12, 8, 10]$$

Now sort the two sides recursively.

The list is then sorted:

$$[1, 2, 3, 7, 8, 10, 12]$$

- Clearly split takes $n - 1$ comparisons.

QuickSort may well not split L evenly.

Split

Algorithm Split(left,right):
pre-condition: left $<$ right
$d = L[\text{left}]$ # pivot
$i = \text{left} + 1$; $j = \text{right}$
# Invariant:
# left $< i \leq j + 1$
# $j \leq$ right
# if left $\leq k < i$ then $L[k] \leq d$
# if $j < k \leq$ right then $L[k] > d$
while $i \leq j$:
  if $L[i] \leq d$:
    $i = i + 1$
  else:
    Swap(i, j) ; $j = j - 1$
$i = j + 1$
Swap(left, j) ; return j

Worst Case Analysis

Suppose L was $[1, 7, 2, 10, 12, 3, 8]$.

Split around 1.

The list becomes $[1, 2, 10, 12, 3, 8, 7]$.

Must then sort $[2, 10, 12, 3, 8, 7]$.

A particularly bad case is when L is already sorted, e.g. $[1, 2, 3, 7, 8, 10, 12]$.

This is in fact the worst case.

$$\begin{aligned} W(1) &= 0 \\ W(n) &= n - 1 + W(n-1) \end{aligned}$$

So

$$W(n) = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

No better than Insertion Sort.

Average Case

But QuickSort is good in practice.

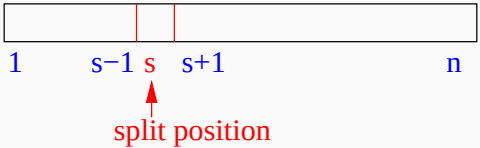
This is because the split is unlikely to occur at one end.

The average number of comparisons $A(n)$ is of lower order than $W(n)$.

We calculate $A(n)$.

Analysis

After splitting, the "split position" can be anywhere from 1 to n.



Then call QuickSort on lists of length $s-1$ and $n-s$,

taking $A(s-1)+A(n-s)$ comparisons.

$$A(n) = n - 1 + \frac{1}{n}\sum_{s=1}^{n}(A(s-1)+A(n-s))$$

(assuming each position equally likely)

Analysis

$$A(n) = n - 1 + \frac{1}{n}\sum_{s=1}^{n}(A(s-1) + A(n-s))$$

After simplifying:

$$\begin{aligned} A(1) &= 0 \\ A(n) &= n - 1 + \tfrac{2}{n}\textstyle\sum_{i=2}^{n-1} A(i) \end{aligned}$$

Fact

$A(n)$ *is* $\Theta(n \log n)$.

Performance

QuickSort gives good performance in average case.

e.g. $n = 10$:

- Average case lower bound 21.84
- $A(n)$ for QuickSort 24.44

For comparison:

- Worst case lower bound 22
- $W(n)$ for MergeSort 25

It might seem that MergeSort is better than QuickSort.

However:

- Can improve chances of a good split.
- QuickSort uses less space
  (merging requires extra space).

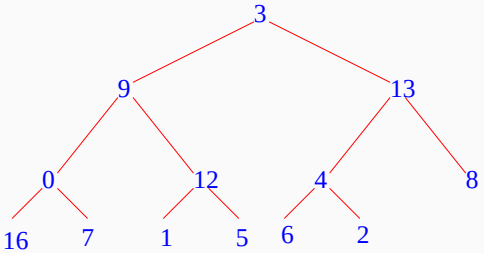
Heapsort

We now discuss Heapsort.

A heap structure is a left-complete binary tree.

Left-complete means that if the tree has depth d then

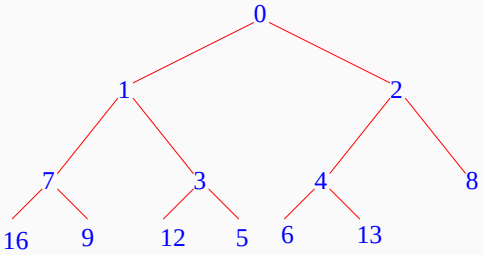
- all nodes are present at depth $0, 1, \ldots, d-1$
- and at depth d no node is missing to the left of a node which is present.



Call the rightmost node at depth d the last node.

Min Heap

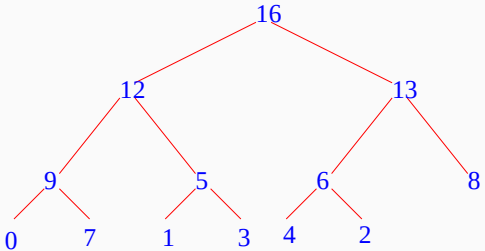
- A tree T is a minimising partial order tree if the key at any node $\leq$ the keys at each child node (if any).
- A min heap is a heap structure with the min partial order tree property.



Note that for any node of a heap, the left and right subtrees below the node are also heaps.

Max Heap

- A tree T is a maximising partial order tree if the key at any node $\geq$ the keys at each child node (if any).
- A max heap is a heap structure with the max partial order tree property.



Heapsort

Heapsort scheme

Build max heap H out of an array E of elements
for $i = n$ to 1:
    max = getMax(H)
    deleteMax(H)
    $E[i] =$ max

getMax(H) - just read the root node of H

deleteMax(H) scheme

copy element at last node into root node
remove last node
fixMaxHeap(H)

We use fixMaxHeap(H) to restore the heap property.

fixMaxHeap

Starting from a heap structure H where the left and right subtrees of the root are heaps, make H into a heap (again).

fixMaxHeap(H) scheme

```
if H not a leaf:
    largerSubHeap = the left or right subheap with the larger root
    if root(H).key < root(largerSubHeap).key:
        swap elements at root(H) and root(largerSubHeap)
        fixMaxHeap(largerSubHeap)
```

If the heap has depth d we see that fixMaxHeap takes at most $2d$ comparisons.

Since a heap with n elements has depth $\lfloor \log n \rfloor$,
fixMaxHeap takes $O(\log n)$ comparisons.

Building a heap

Starting from a heap structure H that does not necessarily have the partial order property, build a heap.

buildMaxHeap(H) scheme

if H not a leaf:
    buildMaxHeap(left subtree of H)
    buildMaxHeap(right subtree of H)
    fixMaxHeap(H)

Divide and conquer algorithm.

Analysis of buildMaxHeap

Suppose for simplicity $n = 2^k - 1$ so that the heap structure is a complete binary tree with depth $k - 1$.

Let $W(n)$ be the worst-case number of comparisons for buildMaxHeap.

$$W(n) = 2W((n-1)/2) + 2\log n$$

Apply Master Theorem with $a = 2$, $b = 2$ and $f(n) = 2\log n$.

Critical exponent $E = 1$, and so

$$W(n) = \Theta(n^E) = \Theta(n)$$

So can build the heap in linear time.

Analysis of Heapsort

Heapsort scheme

Build max heap H out of an array E of elements $O(n)$
for $i = n$ to 1:
  max = getMax(H) $O(1)$
  deleteMax(H) $O(\log n)$
  $E[i] = \max$

Overall $O(n \log n)$ comparisons.

Heaps as arrays

We can implement heaps using arrays.

Store the heap level by level in an array starting at index 1.

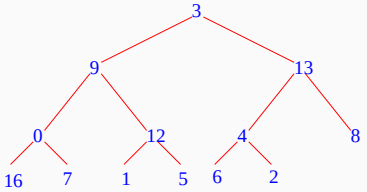
Left and right children of node i are at $2i$ and $2i + 1$.

Parent node is at $\lfloor i/2 \rfloor$.

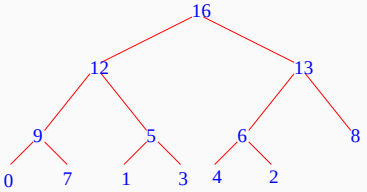
No need for pointers.

It turns out that Heapsort can be carried out entirely in place (like Quicksort).

Heaps as arrays



3	9	13	0	12	4	8	16	7	1	5	6	2
1	2	3	4	5	6	7	8	9	10	11	12	13



16	12	13	9	5	6	8	0	7	1	3	4	2
1	2	3	4	5	6	7	8	9	10	11	12	13

Heaps as arrays

Algorithm Heapsort(E, n)

```
# perform Heapsort on elements 1..n of an array E of elements
BuildMaxHeap(n)
heapsize = n
# Invariant:
# max heap in E[1..heapsize] of first heapsize-many elements
# of sorted list
# elements in heapsize+1 to n are correctly sorted
while heapsize> 1:
   swap(1,heapsize)
   heapsize = heapsize−1
   fixMaxHeap(1,heapsize)
```

Heaps as arrays

Algorithm fixMaxHeap(root,heapsize)

```
left = 2*root
right = 2*root+1
if left ≤ heapsize:
   # root is not a leaf
   if left = heapsize:
      # no right subheap
      largerSubHeap = left
   elif E[left].key > E[right].key:
      # favours right subheap if equal
      largerSubHeap = left
   else:
      largerSubHeap = right
   if E[root].key < E[largerSubHeap].key:
      swap(root,largerSubHeap)
      fixMaxHeap(largerSubHeap,heapsize)
```

Heaps and priority queues

Priority queues can be implemented as binary heaps.

Min PQ as used for Prim's algorithm:

- Each item x of the queue has a priority key[x]
- Items removed lowest key first.

Operations:

- Q = PQcreate()
- isEmpty(Q)
- insert(Q, x)
- getMin(Q)
- deleteMin(Q)
- decreaseKey(Q,x,newkey) — updates key[x] = newkey

Plainly we should use a min binary heap.

Heaps and priority queues

We sketch how to perform PQ operations using an array implementation of min binary heaps.

Array E plus heapsize parameter with the heap in $E[1..\text{heapsize}]$ (inclusive).

- Q = PQcreate()
  Create an empty array E of a suitable size with heapsize = 0.
- isEmpty(Q)
  Check if heapsize = 0. Time $O(1)$.
- getMin(Q)
  Return $E[1]$. Time $O(1)$.
- deleteMin(Q)
  $E[1] = E[\text{heapsize}]$
  decrement heapsize
  fixMinHeap(1,heapsize)
  Time $O(\log n)$.

Insert

insert(Q, x)

```
heapsize = heapsize+1
E[heapsize] = x
# allow x to percolate towards the root until heap property is restored
percolateup(heapsize)
```

procedure percolateup(c):

```
if c > 1:
    parent = ⌊c/2⌋
    if E[c].key < E[parent].key:
        swap(c,parent)
        percolateup(parent)
```

Time $O(\log n)$.

It may be possible to build the queue in one go (time $O(n)$) rather than inserting elements individually (time $O(n \log n)$.

Decrease Key

decreaseKey(Q,x,newkey)

If we know the location c of x in the heap then we can change its key to newkey and use percolateup(c) to restore the heap.

The problem is to locate x efficiently – time $O(\log n)$ rather than $O(n)$.

Solution:

Suppose that each element has an identifier id.

Suppose that identifiers are integers in a compact range [1..maxid]

Use a supplementary array xref to store the location of id:

xref[id] $= k$ means that element is at location k in heap.

Need to add code to percolateup to keep xref up to date as swaps occur.

Dynamic Programming

We have seen several examples of dynamic programming.

We now consider an example to illustrate:

- top-down versus bottom-up solutions
- memoisation

Word break problem

As an example consider the following problem:

Given a string of characters s, can s be split into words occurring in a dictionary?

Example: $s =$ ‘windown’ can be split as ‘win down’ (or ‘wind own’)

But ‘trcarlenz’ cannot be split into (English) words.

Looking at all possible splits would take too long—exponentially many.

‘Top Down’ Recursive solution

procedure wb1(s)

```
if len(s) == 0:
    return true
else:
    for i = 0 to len(s) − 1:
        if indict(s[i :]):
            if wb1(s[: i]):
                return true
return false
```

indict checks if a string is a word in the dictionary.

Slice notation:

- $s[i : j]$ string s from index i to index $j - 1$
- $s[: i]$ string s from start index 0 to index $i - 1$
- $s[i :]$ string s from index i to the end index $\text{len}(s) - 1$

‘Top Down’ Recursive solution

Recurrence relation for worst case on strings of length n:

$$\begin{aligned} W_1(0) &= 0 \\ W_1(n) &= n + W_1(0) + \cdots + W_1(n-1) \quad (n \geq 1) \end{aligned}$$

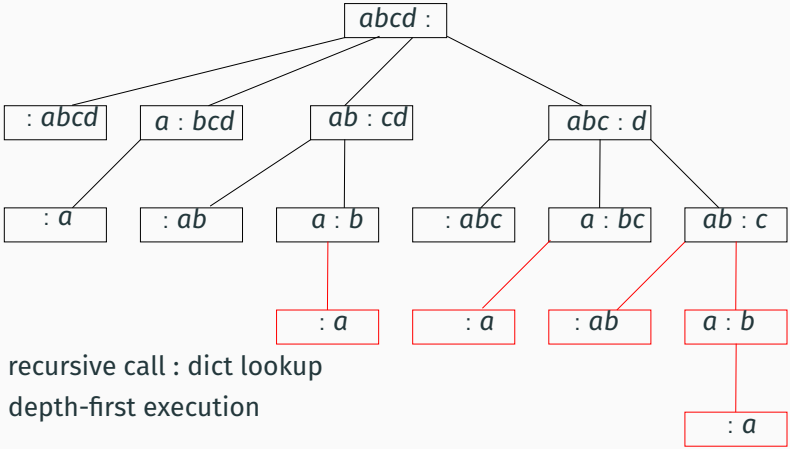
Solution

$$W_1(n) = 2^n - 1$$

Exponential!

Inefficiency from computing wb1(s[: i]) repeatedly.

Recursion tree



recursive call : dict lookup

depth-first execution

Memoised recursive solution

Use memo to store previously computed results of recursive calls.

memo = {} # empty associative array

procedure wb2(s)

```
if len(s) == 0:
    return true
else:
    for i = 0 to len(s) − 1:
        if indict(s[i :]):
            if memo[s[: i]] undefined:
                memo[s[: i]] = wb2(s[: i])
            if memo[s[: i]]:
                return true
return false
```

Note that the transformation is generic.

Memoised recursive solution

Recursion tree now cut off at depth two (red nodes removed).

Level 1: n nodes
Level 2: $\sum_{i=0}^{n-1} i$

Worst case on strings of length n: $\sum_{i=0}^{n} i = n(n+1)/2$

Complexity is $O(n^2)$.

Non-recursive solution

'bottom up'

Idea: solve increasing sub-problems culminating in main problem:

$$s[:0], s[:1], \ldots, s[:n] = s$$

Note that these problems overlap.

Store result for $s[:i]$ in array wb[i].

Non-recursive solution

Algorithm wb3(s)

```
n = len(s)
wb[0] = true
if n > 0:
    for i = 1 to n:
        wb[i] = false
        for j = 0 to i − 1:
            if wb[j] and indict(s[j : i]):
                wb[i] = true
                break
return wb[n]
```

Complexity is again $O(n^2)$.

Comparison

Which is better?

Top-down with memoisation:

- perhaps easier to develop
- may be faster if not all subproblems need computing

Bottom-up non-recursive

- avoids overheads due to recursion

Discussion

Dynamic programming typically involves taking a problem with exponentially large solution space and finding a path to a solution in polynomial time.

Find whether a solution exists (Hamiltonian Circuit, Word Break) or (very often) find best solution (TSP).

Overlapping subproblems (contrast with divide and conquer)

Storing the results of computations of subproblems

Here ‘programming’ means planning a suitable order/plan of computation.

In many cases the subproblems to consider will depend on
the previously obtained results (hence ‘dynamic’).

Part IV

Introduction to Complexity

Overview

Tractable problems and P

NP

Problem reduction

NP-completeness

Outline

Tractability

We wish to identify which problems are tractable (aka feasible)

- efficiently computable, i.e. can be computed in a reasonable amount of time

We focus on worst-case:
We want $W(n)$ to be not too large, where n is the input size.

Sorting a list by comparisons

We have seen $W(n) = n \log n$ comparisons where n is the length of the list (MergeSort).

We can agree that sorting is tractable, since comparing
two elements in a list can be done easily.

Euler path

Problem EulerPath

Given a graph G, does G have a Euler path?

Suppose G has n nodes and m arcs. Input size $|G|$ depends on the representation of G:

- adjacency matrix has size $O(n^2)$
- linked list has size $O(n + m)$

G has Euler path iff G has 0 or 2 nodes of odd degree.

This is tractable, since we can count the odd degree nodes by making
a single pass through either representation,
keeping track of the degree count for the current node
and the number of odd degree nodes found so far.

Hamiltonian path

By contrast:

Problem HamPath

Given a graph G, does G have a Hamiltonian path?

To see whether a graph with n nodes has a Ham path seems to require $O(n^2 2^n)$ time - this takes too long.

Decision problems

Decision problems are those which have a yes/no answer, e.g. $\mathrm{EulerPath}$, $\mathrm{HamPath}$.

Definition

- A decision problem D is decided by an algorithm A if for any input x, A returns 'yes' or 'no' depending on whether $D(x)$
  (and in particular A always terminates).
- A decision problem D is decidable in polynomial time iff it is decided by some algorithm A which runs within polynomial time, i.e.
  on all inputs of size n, A takes $\leq p(n)$ steps for some $p(n)$.

We abbreviate polynomial time to poly time or p-time.

E.g. $\mathrm{EulerPath}$ is decidable in p-time, but $\mathrm{HamPath}$ (apparently) not.

Note: We use D rather than P to range over decision problems as
P will be the class of polynomial-time problems.

Tractability

Cook-Karp Thesis

A problem is tractable iff it can be computed within polynomially many steps in worst case ($W(n) \leq p(n)$ for some polynomial $p(n)$).

More succinctly

Slogan

Tractable = polynomial time.

According to the thesis, sorting a list is tractable, as is $\mathrm{EulerPath}$,
but $\mathrm{HamPath}$ is (apparently) not.

Different models

We have been talking about $p(n)$ steps in a computation on input size n. Clearly

- input size
- computation step

have different meanings depending on the model we are using.

- For sorting a list, we took input size to be the number of list items, and we counted comparisons, ignoring other computation steps (swaps, copying, recursive procedure calls, etc.).
- For $\mathrm{EulerPath}$ we measured input size either using adjacency matrices or adjacency lists, and computation steps would involve inspecting the input and incrementing counters.

However it turns out that all reasonable models of computation
and measures of input size give essentially the same results.

Polynomial Invariance Thesis

If a problem can be solved in polynomial time $p(n)$ in some model, then if we change the model the problem can still be solved in p-time $q(n)$.

We may get a different polynomial, but the concept of p-time is robust.

Polynomial invariance thesis

If a problem can be solved in p-time in some reasonable model of computation, then it can be solved in p-time in any other reasonable model of computation.

Thus for sorting a list, a different model would be to take the sizes of the items to be sorted into account in measuring input length, and to take into account the fact that comparisons can take different amounts of time depending on the sizes of the items and the length of the list.

We would still get p-many steps but not necessarily $O(n \log n)$.

P and reasonable models

Definition

A decision problem $D(x)$ is in the complexity class P (polynomial time) if it can be decided within time $p(n)$ in some reasonable model of computation, where n is the input size $|x|$.

By the Invariance Thesis this definition is model-independent.

Note that sorting a list does not belong to the class P, since it is not a decision problem.

It is technically convenient to define complexity classes for decision problems only, at least to start with.

What models would be unreasonable?

Unreasonable models

- Superpolynomial parallelism is unreasonable. If we could carry out more than polynomially many operations in parallel in a single step, then we might be able to solve exponential time problems in p-time. So this model is unreasonable.
- Unary numbers (writing 11111 for 5) are unreasonable. When dealing with numbers we do not allow unary representation (use base 2 or greater). This is because unary gives input size which is exponentially larger than binary, and so an exponential time algorithm can appear to be p-time.

Example

With unary we can check whether a number n is prime by looking at all $m < n$ and seeing whether m divides n. This takes n divisions. This is p-time if input size is n.

However the true input size $|n|$ is actually $\log n$ and so we have an exp-time algorithm.

Arithmetical operations

Arithmetical operations (addition, subtraction, multiplication, division) are p-time i.e. they are polynomial in $|n| = \log n$ rather than in n.

Input size and output size

Suppose that f is a p-time function.

The output size $|f(x)|$ is polynomially bounded in the input size $|x|$:

$$|f(x)| \leq p(|x|)$$

for some polynomial $p(n)$.

The reason is that any program which computes f has only p-time in which to build the output.

Function Composition

The next result shows that poly time is well-behaved.

It will be useful when discussing reduction and NP-completeness.

Proposition

Suppose that f and g are functions which are p-time computable. Then the composition $g \circ f$ is also p-time computable.

Suppose

- $f(x)$ is computed by algorithm A within time $p(n)$ where $n = |x|$
- $g(y)$ is computed by algorithm B within time $q(m)$ where $m = |y|$

Function Composition (continued)

Take input x with $|x| = n$.

We compute $g(f(x))$ by first running A on x to get $f(x)$ and then running B on $f(x)$ to get $g(f(x))$.

Running A on x takes $\leq p(n)$ steps.

To see how long running B takes we need a bound on the size of the input $f(x)$.

But A runs for $\leq p(n)$ steps to build $f(x)$. So $|f(x)|$ must be poly bounded in n - there is no time to build a larger output.

Say $|f(x)| \leq p'(n)$ for some polynomial $p'(n)$.

Then B runs within $q(p'(n))$ steps.

Total running time (A followed by B) is $p(n) + q(p'(n))$.

This is polynomial in n. Hence result.

Outline

Guessing a certificate

Consider the $\mathrm{HamPath}$ problem. Given a graph G, if we guess a list π then it is easy to check whether π is a Ham path of G.

- check that the items of π are a permutation of $\mathrm{nodes}(G)$;
- check that successive nodes of π are adjacent in G.

It is pretty clear that these checks can be carried out in p-time.

Thus $\mathrm{HamPath}$ becomes easy (p-time) if we guess the path.

The Ham path π acts as a certificate that HamPath(G).

Of course if we guess π and we discover that π is not a Ham path of G, then we are none the wiser, since it might be that G has a (different) Ham path, or that G has no Ham path.

Nevertheless, it remains the case that if G has a Ham path then some guess will prove correct.

Problem $\mathrm{Ver\text{-}HamPath}$

Given a graph G and a list π, is π a Ham path of G?

Note that the verification problem $\mathrm{Ver\text{-}HamPath}(G, \pi)$ is in P.

Clearly $\mathrm{HamPath}(G)$ iff $\exists \pi . \mathrm{Ver\text{-}HamPath}(G, \pi)$.

Definition

A decision problem $D(x)$ is in NP (non-deterministic polynomial time) if there is a problem $E(x, y)$ in P and a polynomial $p(n)$ such that

- $D(x)$ iff $\exists y . E(x, y)$
- if $E(x, y)$ then $|y| \leq p(|x|)$ (E is poly balanced)

We require that the certificate y is poly bounded in x
since otherwise it would take too long to guess y.

Clearly the guess for the Ham path can be p-bounded in size of G.

Slogan

- P class of decision problems which can be efficiently solved
- NP class of decision problems which can be efficiently verified

Satisfiability

A formula ϕ of propositional logic is in conjunctive normal form (CNF) if it is of the form

$$\bigwedge_i (\bigvee_j a_{ij})$$

where each a_{ij} is either a variable x or its negation $\neg x$.

- Terms a_{ij} are called literals
- Terms $\bigvee_j a_{ij}$ are called clauses

Problem SAT

Given a formula ϕ in CNF, is ϕ satisfiable?
(i.e. is there an assignment v to the variables of ϕ
which makes ϕ true?)

$\mathrm{SAT} \in \mathrm{NP}$

It seems that SAT is not decidable in p-time: we have to try all possible truth assignments.

If ϕ has m variables there are 2^m assignments - exponential.

We can let $|\phi|$ be the number of symbols in ϕ and $|v|$ be m (size of the domain of v).

Notice that m can be of similar size to $|\phi|$ - every literal could be a different variable.

However SAT does belong to NP:

- guess a truth assignment v
- verify in p-time that v satisfies ϕ

Let $\mathrm{Ver\text{-}SAT}(\phi, v)$ iff ϕ is in CNF and v satisfies ϕ.

Then:

- $\mathrm{SAT}(\phi)$ iff $\exists v.\mathrm{Ver\text{-}SAT}(\phi, v)$
- if $\mathrm{Ver\text{-}SAT}(\phi, v)$ then $|v| \leq |\phi|$ ($\mathrm{Ver\text{-}SAT}$ is p-balanced)

Proposition

If a decision problem is in P *then it is in* NP*, i.e.* $\mathrm{P} \subseteq \mathrm{NP}$*.*

Proof.

Suppose that problem D is in P.

Idea: to verify that $D(x)$ holds we don't need to guess a certificate y - we can decide $D(x)$ directly.

More formally, we define $E(x, y)$ iff $D(x)$ and $y = \epsilon$ (the empty string - a dummy guess). Then clearly

$$D(x) \text{ iff } \exists y.E(x, y) \text{ and } |y| \leq p(|x|)$$

□

The $\mathrm{P} = \mathrm{NP}$ question

It remains unknown whether $\mathrm{P} = \mathrm{NP}$ despite many researchers' attempts.

This is the most important open problem in computer science. It is arguably one of the most important open problems in mathematics.

The Clay Mathematics Institute has offered a prize of one million dollars for a solution (either equal or not equal).

It is fair to say that most researchers believe that $\mathrm{P} \neq \mathrm{NP}$.

Outline

Motivation

Since $\mathrm{P} \subseteq \mathrm{NP}$, if we show that a problem such as $\mathrm{HamPath}$ belongs to NP we do not know whether it is tractable (in P) or not.

We want to identify the hard (high complexity) problems in NP.

We start by saying what it means for one problem to be harder than another, using the concept of reduction.

Many-one reduction

Suppose that D and D' are two decision problems. We say that D (many-one) reduces to D' ($D \leq D'$) if there is a p-time computable function f such that

$$D(x) \text{ iff } D'(f(x))$$

Note that f can be a many-one function (hence the name).

The idea is that we reduce a question about D (the easier problem)
to a question about D' (the harder problem).

P and reduction

Suppose that we have an algorithm A' which decides D' in time $p'(n)$. Then if $D \leq D'$ via reduction function f running in time $p(n)$ we have an algorithm A to decide D:

Algorithm A (input x)

1. Compute $f(x)$
2. Run A' on input $f(x)$ and return the answer (yes/no)

Now A runs in p-time - same argument as when composing p-time functions.

Step 1 takes $p(n)$ steps.

$|f(x)| \leq q(n)$ for some poly q. Step 2 takes $p'(q(n))$ steps.

Hence:

Proposition

Suppose $D \leq D'$ and $D' \in \mathrm{P}$. Then $D \in \mathrm{P}$.

NP and reduction

Proposition

Suppose $D \leq D'$ and $D' \in \mathrm{NP}$. Then $D \in \mathrm{NP}$.

Assume that $D \leq D'$ and $D' \in \mathrm{NP}$.

Then there is $E'(x, y) \in \mathrm{P}$ and a $p'(n)$ such that

$D'(x)$ iff $\exists y.E'(x, y)$

and if $E'(x, y)$ then $|y| \leq p'(|x|)$.

Also we have $D(x)$ iff $D'(f(x))$.

Combining:

$D(x)$ iff $\exists y.E'(f(x), y)$

Define $E(x, y)$ iff $E'(f(x), y)$.

Then $D(x)$ iff $\exists y.E(x, y)$.

Also $E \in \mathrm{P}$ (same argument as previous slide).

NP and reduction (continued)

We check that E is p-balanced:

Suppose $E(x, y)$.

Then $E'(f(x), y)$, so that $|y| \leq p'(|f(x)|)$

As before we have $|f(x)| \leq q(n)$.

Hence $|y| \leq p'(q(|x|))$.

Hence $D \in \mathrm{NP}$.

Properties of reduction

The reduction order is reflexive and transitive:

- $D \leq D$
- if $D \leq D' \leq D''$ then $D \leq D''$

The proofs are left as an exercise.

If both $D \leq D'$ and $D' \leq D$ we write $D \sim D'$.
Here D and D' are as hard as each other.

Outline

NP-completeness

We want to identify problems in NP which are unlikely to be in P.

Definition

A decision problem D is NP-hard if for all problems $D' \in \mathrm{NP}$ we have $D' \leq D$.

- Note that NP-hard problems do not necessarily belong to NP. They could be harder.
- If D is NP-hard and $D \leq D'$ then D' is also NP-hard (consequence of transitivity of reduction)

Definition

A decision problem D is NP-complete (NPC) if

1. $D \in \mathrm{NP}$
2. D is NP-hard

NP-complete problems are the hardest problems in NP.

Cook-Levin Theorem

It is not clear from the definition that NPC problems exist. However:

Theorem (Cook-Levin 1971)

SAT *is* NP*-complete.*

Many other problems have been shown to be NPC.

Rather than proving this directly as for SAT, we use reduction as follows:

Method

To see that D is NPC show:

1. $D \in \mathrm{NP}$
2. $D' \leq D$ for some known NPC problem D'

It is clear that item 2 establishes that D is NP-hard,
since D' is NP-hard.

$\mathrm{HamPath} \in \mathrm{NP}$

As an example take $\mathrm{HamPath}$. We have already seen that $\mathrm{HamPath} \in \mathrm{NP}$ by guessing and verifying in p-time.

If we can show $\mathrm{SAT} \leq \mathrm{HamPath}$ then we can conclude that $\mathrm{HamPath}$ is NPC.

It is indeed possible to show $\mathrm{SAT} \leq \mathrm{HamPath}$ but we omit the reduction as it is long and difficult.

So $\mathrm{HamPath}$ is NP-complete.

Intractability via NP-completeness

Proposition

Suppose $\mathrm{P} \neq \mathrm{NP}$. *If* D *is* NP*-hard then* $D \notin \mathrm{P}$.

Proof.

Assume $\mathrm{P} \neq \mathrm{NP}$ and D is NP-hard.

Suppose for a contradiction that $D \in \mathrm{P}$. We show that $\mathrm{NP} \subseteq \mathrm{P}$.

Take $D' \in \mathrm{NP}$. Since D is NP-hard, we have $D' \leq D$.

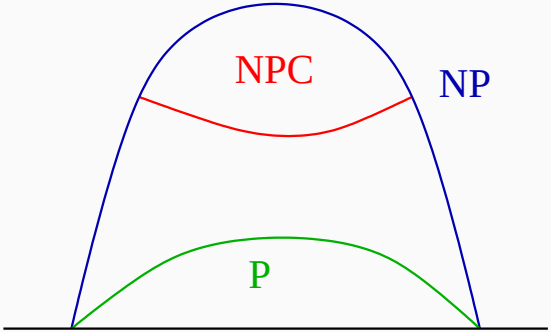
Hence $D' \in \mathrm{P}$.

We have shown that $\mathrm{NP} \subseteq \mathrm{P}$.

But we know $\mathrm{P} \subseteq \mathrm{NP}$. Hence $\mathrm{P} = \mathrm{NP}$ which contradicts our assumption. □

Thus if we can show that a problem is NPC, we know that it is intractable (assuming that $\mathrm{P} \neq \mathrm{NP}$, as is generally believed).

Complexity classes



Travelling Salesman Problem

Problem TSP

Given a (complete) weighted graph (G, W), find a tour of G of minimum weight which visits each node exactly once and returns to the start node.

We first define a decision version of TSP:

Problem $\mathrm{TSP(D)}$

Given a weighted graph (G, W) and a bound B,
is there a tour of G with total weight $\leq B$?

$\mathrm{TSP(D)}$ is NP-complete

We show that $\mathrm{TSP(D)}$ is NP-complete using the Method:

1. $\mathrm{TSP(D)} \in \mathrm{NP}$:
   If we guess a path π, we can check in p-time that π is a Ham circuit of G and that $W(\pi) \leq B$. Clearly $|\pi| \leq |G|$.

   More formally define $\mathrm{Ver\text{-}TSP(D)}((G, W), B, \pi)$ iff π is a Ham circuit of (G, W) and $W(\pi) \leq B$.
   Then $\mathrm{TSP(D)}((G, W), B)$ iff $\exists \pi . \mathrm{Ver\text{-}TSP(D)}((G, W), B, \pi)$.
   Also if $\mathrm{Ver\text{-}TSP(D)}((G, W), B, \pi)$ then $|\pi| \leq |G|$ under reasonable definitions of size.

2. $D' \leq \mathrm{TSP(D)}$ for some known NPC problem D':
   We choose $\mathrm{HamPath}$ as the known NPC problem
   and show $\mathrm{HamPath} \leq \mathrm{TSP(D)}$.

$\mathrm{HamPath} \leq \mathrm{TSP(D)}$

We need to define a p-time function f which transforms a graph G into a weighted graph (G', W) together with a bound B so that

$\mathrm{HamPath}(G)$ iff $\mathrm{TSP(D)}((G', W), B)$.

Given G we construct (G', W) as follows:

Set $\mathrm{nodes}(G') = \mathrm{nodes}(G)$.

Given any two distinct nodes x, y of G:

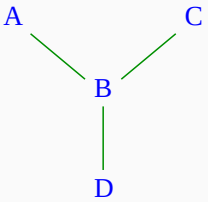
- If (x, y) is an arc of G then (x, y) is also an arc of G', with $W(x, y) = 1$.
- If (x, y) is not an arc of G then (x, y) is an arc of G', with $W(x, y) = 2$.

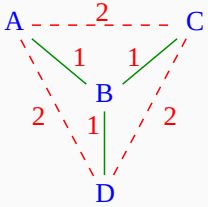
Thus we add in the missing arcs of G but with higher weight.

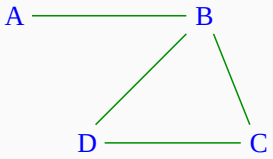
Finally we let $B = n + 1$ where G has n nodes.

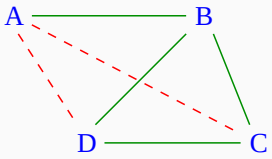
Not hard to see that $f(G) = ((G', W), B)$ is p-time
— easiest to see using the adjacency matrix representation.

Example graphs









f is a reduction

We now check $\mathrm{HamPath}(G)$ iff $\mathrm{TSP(D)}((G', W), B)$.

Suppose G has Ham path π with endpoints x and y.

The same path in G' has weight $n - 1$.

We get a TS tour by adding in arc (x, y) with $W(x, y) \leq 2$.

Thus we have a tour of weight $\leq n + 1 = B$.

Conversely, suppose (G', W) has a tour of weight $\leq B = n + 1$.

This has n arcs.

So at most one arc can have weight 2.

Suppose this arc has endpoints x, y.

Then omitting arc (x, y) gives us a Ham path in G.

We conclude that $\mathrm{TSP(D)}$ is NP-complete.

TSP is intractable

Finally we can show that TSP is intractable (assuming $\mathrm{P} \neq \mathrm{NP}$):

Suppose that TSP can be solved by a p-time algorithm.

We compute the optimal value *O* in p-time.

Then we can also solve $\mathrm{TSP(D)}$ in p-time — simply check whether $O \leq B$.

But this is impossible since $\mathrm{TSP(D)}$ is NP-complete and we assume $\mathrm{P} \neq \mathrm{NP}$.

Further examples

We look at further examples of NP-complete problems:

- Metric TSP MTSP
- Vehicle routing problem with capacities VRPC

Metric TSP

The metric TSP (MTSP) is the TSP restricted to graphs (G, W) satisfying the triangle inequality

$$W(x,z) \leq W(x,y) + W(y,z) \quad \text{for all } x,y,z \in \mathrm{nodes}(G)$$

A very natural condition, since if does not hold we would choose to travel from x to z via y rather than directly.

Decision version: $\mathrm{MTSP(D)}$ - can we find a tour not exceeding given bound B?

We want to show: $\mathrm{MTSP(D)}$ is NP-complete.

By Method:

(1) $\mathrm{MTSP(D)} \in \mathrm{NP}$: $\mathrm{Ver\text{-}MTSP(D)}((G,W),B,T)$ holds
if T is a TS tour of G with weight $\leq B$.

Easy to check in p-time.

Metric TSP

(2) We need D' such that $D' \leq \mathrm{MTSP(D)}$.

What about $\mathrm{TSP(D)} \leq \mathrm{MTSP(D)}$?

This is a bit difficult since $\mathrm{MTSP(D)}$ is a special case of $\mathrm{TSP(D)}$.

We could convert a graph (G, W) to a new graph (G, W') satisfying the triangle inequality (how?).

But this would not ensure

$$\mathrm{TSP(D)}((G, W), B) \iff \mathrm{MTSP(D)}((G, W'), B)$$

Metric TSP

Instead we look at the reduction $\mathrm{HamPath} \leq \mathrm{TSP(D)}$.

- If (x, y) is an arc of G then (x, y) is also an arc of G', with $W(x, y) = 1$.
- If (x, y) is not an arc of G then (x, y) is an arc of G', with $W(x, y) = 2$.

Clearly

$$W(x, z) \leq W(x, y) + W(y, z) \quad \text{for all } x, y, z \in \mathrm{nodes}(G')$$

since $1 \leq W(x, y) \leq 2$ for all $x \neq y$.

So we actually constructed an instance of $\mathrm{MTSP(D)}$.

Conclude $\mathrm{HamPath} \leq \mathrm{MTSP(D)}$.

Vehicle Routing Problem

Suppose we have a depot, some vehicles and some deliveries to be made to various addresses.

Want to find the smallest cost set of routes for the vehicles.

Can model the network as a (complete) graph:

- nodes are the depot plus customer addresses
- $W(x, y)$ is cost of shortest path from x to y.
- will satisfy triangle inequality

Give the vehicles a capacity and ensure that the total size of the packages transported on each trip is within the capacity.

Assume that each vehicle performs at most one trip.

Vehicle Routing Problem

Vehicle routing problem with capacities $\mathrm{VRPC(D)}$

Given

- a complete weighted graph (G, W) satisfying the triangle inequality,
- a distinguished node start (the depot),
- k vehicles with capacity C,
- a set of packages with sizes $s_1, \ldots, s_n$ to be delivered to nodes $x_1, \ldots, x_n$ respectively,
- and a budget B:

can the packages be delivered within total cost B, subject to total size of the packages on each vehicle being within the capacity C?

Thus we are looking for an assignment of packages to vehicles, plus an itinerary for each vehicle that takes it to each of its addresses.

Vehicle Routing Problem

We want to show: $\mathrm{VRPC(D)}$ is NP-complete.

By Method:

(1) $\mathrm{VRPC(D)} \in \mathrm{NP}$: Plainly given an assignment of packages to vehicles and itineraries for each vehicle we can check in p-time that:

- all packages are assigned to exactly one vehicle
- the total size of packages does not exceed *C* for each vehicle
- the itinerary for each vehicle takes it from the depot to each delivery address assigned to it and back to the depot
- the total cost does not exceed *B*

Vehicle Routing Problem

(2) We need D' such that $D' \leq \mathrm{VRPC(D)}$.

$\mathrm{TSP(D)}$ is the most obvious, but because of the triangle inequality we use $\mathrm{MTSP(D)}$ instead and show

$$\mathrm{MTSP(D)} \leq \mathrm{VRPC(D)}$$

Idea: convert an instance of $\mathrm{MTSP(D)}$ into a simple case of $\mathrm{VRPC(D)}$ via reduction f.

Given (G, W) with n nodes satisfying triangle inequality and B:

- keep (G, W) and B the same
- make one node into the start node (depot)
- assign one package with size 1 to each of the remaining nodes
- create just one vehicle with capacity $n - 1$

Clearly f is p-time.

Then a yes instance of $f((G, W), B)$ means that it is possible to visit all nodes at least once and return to the start within bound B.

Vehicle Routing Problem

We check

$$\mathrm{MTSP(D)}((G,W),B) \iff \mathrm{VRPC(D)}(f((G,W),B))$$

Suppose $\mathrm{MTSP(D)}((G,W),B)$: then we can start at start and visit each node exactly once returning to start all within cost B.

We can do the same in $f((G,W),B)$ since the graph is unaltered. Also we do not exceed the vehicle capacity.

Hence $\mathrm{VRPC(D)}(f((G,W),B))$.

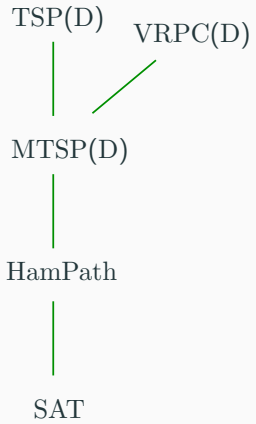
Conversely, assume $\mathrm{VRPC(D)}(f((G,W),B))$:
then we have a route starting at the depot which delivers packages to each node and returns to the depot within cost B.
This is a TS tour except that we might visit a node more than once.

But since graph satisfies triangle inequality we can remove
repeats by taking short cuts, without increasing the total cost.

Hence $\mathrm{MTSP(D)}((G,W),B)$.

Summary

We have seen that the following problems are NP-complete:



Therefore MTSP, TSP, VRPC are intractable (if $\mathrm{P} \neq \mathrm{NP}$).