Part 4

Algorithms for “hard” problems

- Backtracking and branch-and-bound
- Pseudo-polynomial-time algorithms
- Constant-factor approximation algorithms
- Polynomial-time approximation schemes
- Inapproximability results

Review of NP-completeness concepts

- The Class P
  – A *decision problem* Π is in P if and only if Π is solvable in *polynomial time* (i.e. $O(n^c)$ for input size n and constant c)

- The Class NP
  – A decision problem Π is in NP if and only if there is a *nondeterministic algorithm* A for Π such that, for any instance x of Π, x is a *yes-instance* if and only if some execution of A outputs yes in polynomial time
    - for every *yes-instance* there is a *certificate* that allows *polynomial-time verification*

- $P \subseteq NP$; the key question: is $P = NP$?
  – strong belief that the answer is no
  – but no proof of $P \neq NP$ is in sight

- The *theory of NP-completeness* addresses the question: **"If $P \neq NP$ then can we identify problems in NP but not in P?"**

- ***Polynomial-time reduction***

  For decision problems Π and Π' :

  - $\Pi \propto \Pi'$ if every instance of Π can be transformed, in polynomial time, to an instance of Π' with the same answer – we say that Π *is reducible to* Π'
  - if $\Pi \propto \Pi'$ and $\Pi' \in$ P then $\Pi \in$ P

- A problem Π in NP is *NP-complete* if
  - $\Pi' \propto \Pi$ for all problems Π' in NP

- If $\Pi \in$ P and Π is NP-complete then P = NP

- It follows that if $P \neq NP$ and Π is NP-complete then $\Pi \notin$ P

- If Π is an NP-complete problem, an algorithm that solves all instances of Π efficiently is unlikely to be found – so what next?

A few examples of NP-complete problems (1)

Satisfiability (SAT)

Instance: a boolean expression **B** in conjunctive normal form (CNF)
Question: is **B** satisfiable – i.e. can values be assigned to the variables to make **B** true?

- remains NP-complete if every clause in **B** has **3** literals – 3-SAT

Graph Colouring

Instance: a graph **G** and a positive integer **k** (the target number of colours)
Question: can one of **k** colours be assigned to each vertex of **G** so that adjacent vertices have different colours?

- remains NP-complete even if **k = 3**

Travelling Salesman Problem (TSP)

Instance: a complete weighted graph **G** and a target length **x**
Question: is there a cycle in **G** that visits every vertex (a 'tour' of the 'cities') and has total weight (length) $\leq$ **x**?

Hamiltonian Cycle (HC)

Instance: a graph **G**
Question: is there a cycle in **G** that visits every vertex exactly once?

A few examples of NP-complete problems (2)

Vertex Cover (VC)

Instance: a graph **G** and a positive integer **k** (the target size of the cover)
Question: is there a set **S** of $\leq$ **k** vertices such that every edge has at least one endpoint in **S**?

Clique

Instance: a graph **G** and a positive integer **k** (the target size of clique)
Question: is there a set **S** of $\geq$ **k** vertices such that every pair of vertices of **S** are adjacent?

Bin Packing

Instance: a set **S** of items each with positive integer size, a bin capacity **C** and a positive integer **k** (the target number of bins)
Question: can the items in **S** be distributed among $\leq$ **k** bins so that the total size of items in each bin is $\leq$ **C**?

Partition

Instance: a collection **S** of positive integers
Question: can **S** be partitioned into two sub-collections with equal sums?

A few examples of NP-complete problems (3)

Longest Common Subsequence

Instance: a set of strings and a positive integer k (the target length of subsequence)
Question: is there a string of length $\geq k$ that is a common subsequence of every string in the set?

Shortest Common Superstring

Instance: a set of strings and a positive integer k (the target size of the superstring)
Question: is there a string S of length $\leq k$ that is a common superstring of every string in the set (i.e. such that every string in the set is a substring of S)?

3-Dimensional Matching

Instance: 3 disjoint sets X, Y and Z of equal size and a set S of triples of the form (x,y,z) where $x \in X$, $y \in Y$, $z \in Z$
Question: does there exist a subset of S in which every element of X, Y and Z appears exactly once?

Maximum Cut

Instance: a weighted graph G and a positive integer k (the target value of the cut)
Question: can the vertices of G be split into two subsets X and Y such that the sum of the weights of the edges connecting X to Y is $\geq k$?

Coping with NP-completeness – some possibilities (1)

- **A vital practical question: What to do if faced with an NP-complete problem (or a related search / optimisation problem)?**
- **Perhaps only a *restricted* version is of interest – which may be in P**
  - **2-SAT is in P though 3-SAT is NP-complete**
  - **Graph Colouring for 2 colours is in P, though it's NP-complete for 3 colours**
  - **Vertex Cover restricted to bipartite graphs is in P**
- **Seek an algorithm that may be of exponential time complexity, but at least improves on naïve / exhaustive search**
  - ***backtracking***
  - ***branch-and-bound***
  - ***dynamic programming . . . . . .***

Coping with NP-completeness – some possibilities (2)

- **For an optimisation problem:**
  - settle for a (polynomial-time) *approximation algorithm* – with a provable approximation guarantee
  - use a *heuristic* – e.g. *local search*, *genetic algorithms*, *simulated annealing*, *neural networks*, *tabu search*, . . .
- **For a decision problem:**
  - settle for a *randomised* algorithm – one that gives the correct answer with (very) high probability
  - for example testing whether an integer is prime – a vital problem in public-key cryptography
    - now known to be in P (proved in 2002), but the polynomial-time algorithms are complex
    - relatively simple randomised algorithms are widely used

Backtracking

- Exhaustive search (or brute force) systematically generates and tests all possible solutions to a problem

- A backtracking algorithm builds *feasible* partial solutions incrementally
  - it stops and backtracks as soon as the current partial solution cannot lead to an overall solution to the problem
  - there are many variants, depending on
    - the order in which partial solutions are built
    - the way of checking when to backtrack
    - etc.

- Suppose a space of possible solutions to a problem consists of n-tuples $(a_1, . . ., a_n)$ where $a_i \in S$ for some finite set S

Generic backtracking algorithm

```
/** pseudocode for a recursive method to choose a
  * value for the i_th position in the n-tuple;
  * a is a suitable structure - say an arrayList */
public void choose(int i)
{ for (x : S)
  { a.add(i,x);    // add at next slot, position i
    if (a is feasible)  // suitable test
      if (i == n)
      // a is a solution - take appropriate action
      else
        choose(i+1);
    a.remove(i); // remove most recently added item
  }
}
```

Backtracking example – Graph Colouring

- **Assume adjacency matrix representation**

```
/** class representing a graph */

public class Graph {

  private int numVertices;
                // assume vertices indexed from 1
  private int [][] adj; // the adjacency matrix
  private int [] colour; // to store a colouring
  private boolean coloured;
                // indicates whether the most recent
                // colouring attempt was successful
```

```
/** recursive backtracking for graph colouring */
private void choose(int i, int k)
{ int c = 1;
  while (!coloured && c <= Math.min(k, i))**
  { colour[i]=c;  // try colour c on vertex i
    if (isOkay(i))
      if (i == numVertices)
        coloured = true;  // colouring complete
      else
        choose(i+1,k);
    c++;                  // move on to next colour
  }
}
```

** Note that colours higher than i need not be tried for vertex i

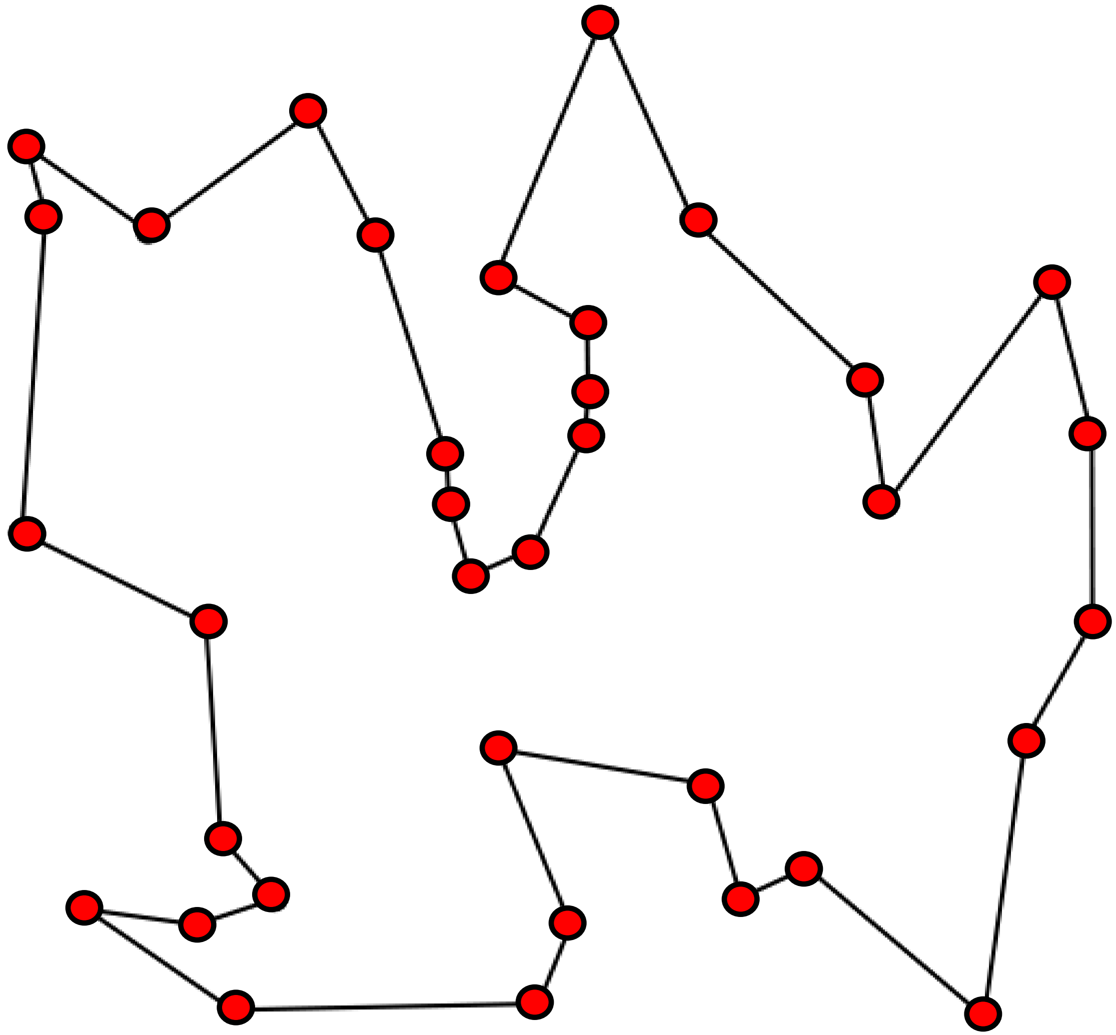
```java
/** checks whether the vertex just coloured is
  * compatible with vertices already coloured */
private boolean isOkay(int i)
{ for (int j = 1; j < i; j++)
    if (adj[j][i] == 1 && colour[j] == colour[i])
      return false;
  return true;
}

/** returns true if the graph is colourable with
  * k colours and returns false otherwise */
public boolean colourable(int k)
{ coloured = false;
  choose(1,k);
  return coloured;
}
```

Travelling Salesman Problem (TSP)

- **Input:** a set of n cities and a distance $d(c_i, c_j)$ between each pair of cities c_i, c_j
- **Output:** a ***travelling salesman tour*** (i.e., a permutation of the cities) of minimum total distance

Example TSP instance

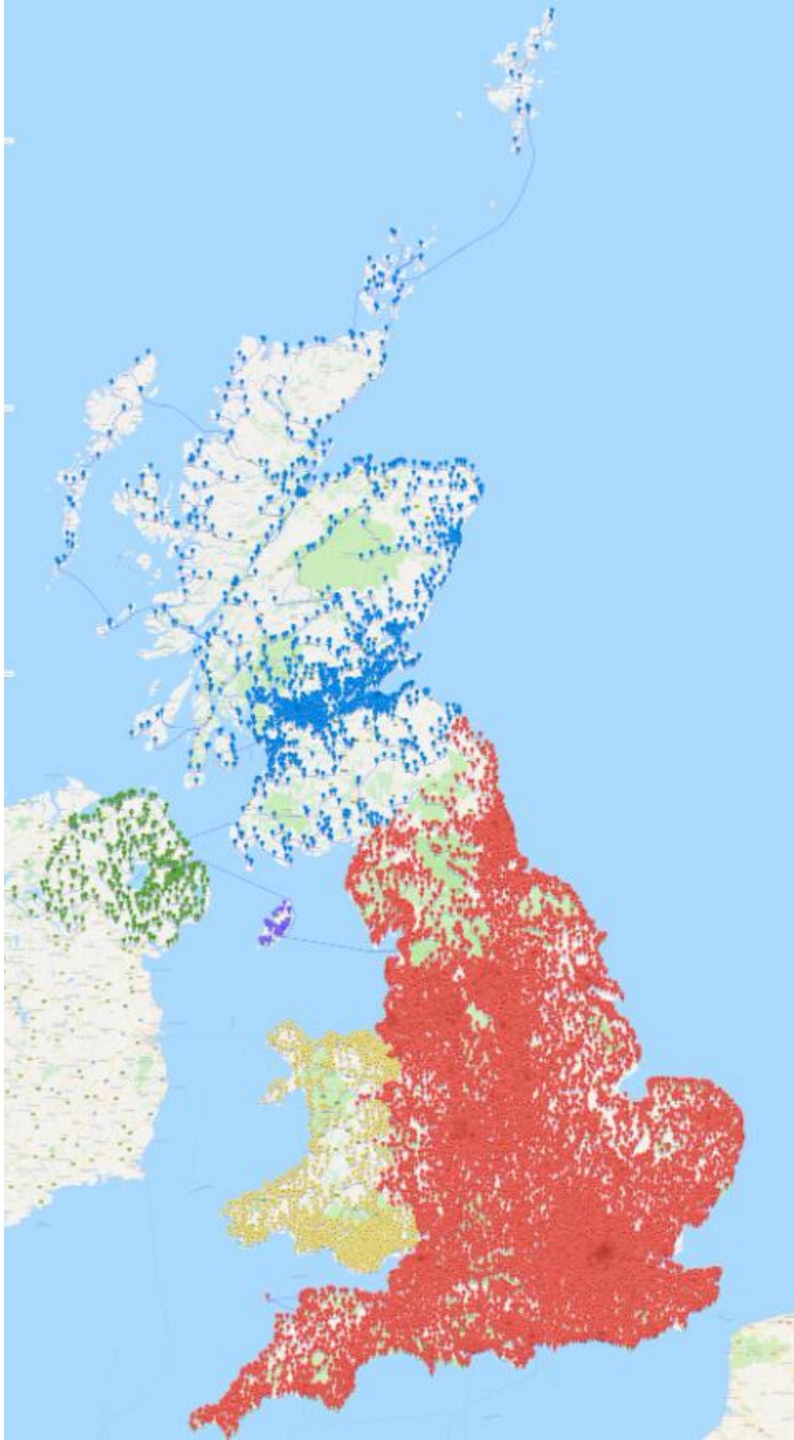


Travelling Salesman Problem (TSP)

- **Input:** a set of n cities and a distance $d(c_i, c_j)$ between each pair of cities c_i, c_j
- **Output:** a ***travelling salesman tour*** (i.e., a permutation of the cities) of minimum total distance
- TSP is an ***optimisation problem***
  - involves minimising or maximising some value over a set of candidate solutions
  - it is **NP-hard** (its decision version is **NP-complete**)
  - (NB: many decision problems are decision versions of optimisation problems)
- Wide range of practical applications
  - Consider Amazon deliveries, for example!

Another important application

- **Find the shortest route to visit 49,687 pubs listed in Pubs Galore - The UK Pub Guide**
- **Undertaken by a team of researchers led by Bill Cook, Waterloo**
  - **https://www.math.uwaterloo.ca/tsp/uk**
- **Distances provided by Google Maps**
- **Solved to optimality!**
  - **Total distance: 63,739,687 metres**
  - **Total computation time: 250 years**
  - **In reality, run on 288 cores**
  - **Total time taken to solve the problem: 14 months**



Branch-and-bound

- **A development of backtracking for optimisation problems**
  - for each partial solution generated, calculate (somehow) a *bound* on best possible overall solution it can lead to
  - if this is no better than the best seen so far, then backtrack

Example: TSP (Optimisation version)

- The possible solutions are *permutations* of {1, 2, . . ., n} (the 'cities')
- Generate partial permutations using a *list* of unvisited cities
- Each city on the list should be considered as the next city to visit
  - remove it from the list, and restore it when backtracking
- As a possible simple bound
  - let C_i be the distance from city i to its closest unvisited neighbour
  - sum the C_i over the current city and the unvisited cities and add this sum to the current partial tour length

- So a *lower bound* on the shortest tour obtainable by extending the current partial tour is

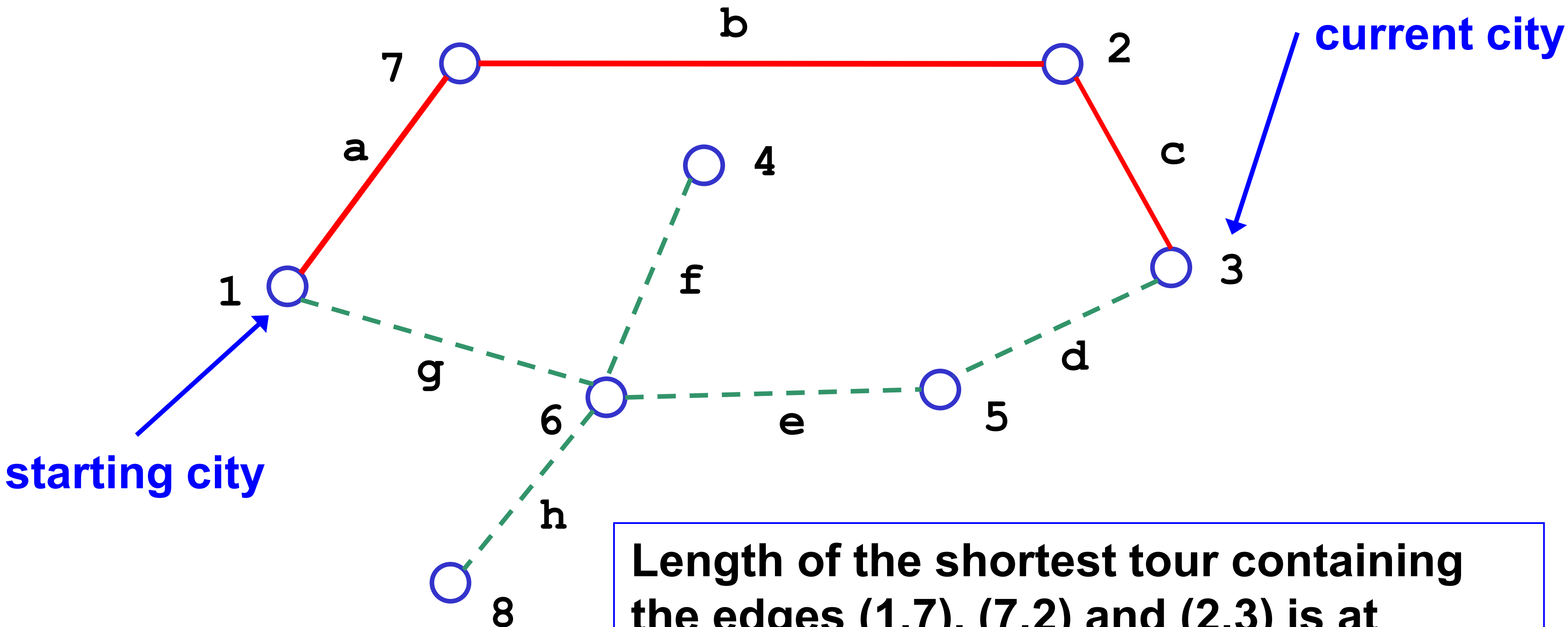
$$\text{CPT\_Length} + \sum C_i$$

where CPT_Length is the length of the current partial tour, and the sum is over the current city and all the *unvisited* cities

- A better bound: the weight of a *minimum weight spanning tree* on current city, starting city, and the unvisited cities
  - these cities must be linked by a path in an optimal TSP solution, and a path is a special kind of spanning tree
  - so a minimum weight spanning tree has weight ≤ length of shortest linking path – and it can be computed efficiently

- Generally – there is a trade-off between the *quality* of a bound and the *time* taken to calculate it

TSP Branch-and-bound – Illustration



Length of the shortest tour containing the edges (1,7), (7,2) and (2,3) is at least

$$\underbrace{(a + b + c)}_{\text{current partial tour length}} + \underbrace{(d + e + f + g + h)}_{\text{weight of minimum spanning tree}}$$

Current partial tour

Minimum weight spanning tree