4. Optimisation algorithms

CLRS, Chs. 34 & 35
Summary

- We will define optimisation problems and we will look at four important algorithmic techniques for tackling these problems
  - Greedy algorithms
  - Dynamic algorithms
  - Approximation algorithms
  - Gradient-based search algorithms
“Normal” problems

- Many problems that we face are absolute
  - there is a right answer, and there are a lot of wrong answers
- e.g. when we are given a list of $n$ distinct items to sort, there are $n!$ permutations of the items, only one of which is sorted
  - The other $n!-1$ permutations are simply wrong
- Sometimes (e.g. with longest common subsequence), there is a set of (equally) right answers, and a lot of wrong answers
- In both cases, the distinction between “right” and “wrong” is absolute
Optimisation problems

- But a lot of problems are more complicated
- e.g. consider a piece of machinery whose speed of operation is controlled by a dial that can be set to any real number between 1 and 1,000
- How do we set the dial?
  - We could set it to 1,000 to go as fast as possible
    - But then the machine might wear out quickly
  - We could set it to 1 to extend the machine’s life
    - But then it makes product very slowly
  - We could try to find some compromise setting in the middle of the range
    - Probably the usual approach
- The point for now is that no solution is “right”, and no solution is “wrong”
  - Whatever we set the dial to, the machine runs
  - Just some settings work “better”, and others work “worse”
  - The distinction is a matter of degree
- These problems are commonly called optimisation problems
  - We are trying to find a solution that performs well wrt some criterion (or criteria)
Another example

- Imagine you have to find a path from Point A to Point B in a complicated road network
- There will probably be many different paths that will get you from A to B
  - So all of them are “solutions”
  - But one (or a few) of them will get you there quickest, or safest, or using the least fuel, or …
  - i.e. some of them will be “better” wrt whatever criteria are important

- Note that many of these problems are often expressed in two different ways:
  - Find the best path
  - Find a path, and make it as good as possible
- The former phrasing obviously has a right answer in the absolute sense
  - The second does not
  - Hence only the second is an optimisation problem
Complications

- There are two general ways in which optimisation problems might be complicated
  - The criteria against which solutions are measured may be ill-defined
    - or they may change over time
  - The number of possible answers (the search space) may be very large
    - e.g. imagine if our machine was a nuclear reactor with 248 dials whose settings combined to determine its performance
- We will largely ignore the former issue: we will assume that our problems are always well-defined
  - But the large search space often means that we cannot expect to find the optimal answer
  - Or even if we find the optimal answer, we may not know that it is optimal!
- Also if there are many “solutions”, but some work better than others, the question arises: how long do we look?
  - Stop as soon as we find something that works?
  - Stop only when we have the optimal answer?
  - Spend all of the time available to us?
  - Stop when we have a solution that is “adequate”?
  - Stop when improvement appears to cease?
  - etc.
Optimisation algorithms

- We discuss four approaches to optimisation problems
- **Greedy algorithms** build up a solution bit-by-bit
  - At each step they make the locally-optimal choice for the next “bit”
- **Dynamic programming** we have seen before
  - Define recursive rules for solving the problem, then optimise the algorithm by eliminating repeated work
- **Approximation algorithms** focus on returning good solutions, but not necessarily the best one
  - Often they can operate very quickly
- **Gradient-based search algorithms** take known solutions and try to improve them
  - While they can be slow, they can work well in difficult domains

- Often there will be a trade-off between the run-time of the algorithm and the quality of the solution returned
  - We will see this issue again later in CITS3001
  - It is also an important issue with so-called “computational intelligence” approaches, studied in CITS4404
An activity selection problem

- Given a set of tasks, each with an associated start time and finish time, select the largest subset of the tasks that can be performed without any incompatibilities
  - Two tasks are incompatible if they overlap in time

- e.g. for \{ (6,9), (1,10), (2,4), (1,7), (5,6), (8,11), (9,11) \}, the following schedules are all valid
  - \{ (1,10) \}
  - \{ (1,7), (8,11) \}
  - \{ (2,4), (5,6), (9,11) \}

- We will assume that the activity intervals are closed on the left and open on the right
  - A closed end of an interval includes its endpoint
  - \((a,b) = \{ x \in \mathbb{R} \mid a < x < b \}\)
  - \([a,b) = \{ x \in \mathbb{R} \mid a \leq x < b \}\)
  - \((a,b] = \{ x \in \mathbb{R} \mid a < x \leq b \}\)
  - \([a,b] = \{ x \in \mathbb{R} \mid a \leq x \leq b \}\)
The natural approach to the activity-selection problem is to choose the tasks one at a time
- Clearly each choice restricts subsequent choices
- e.g. given \{[6,9),[1,10),[2,4),[1,7),[5,6),[8,11),[9,11)\}
  - If we (arbitrarily) choose [1,7), subsequent choices are restricted to \{[8,11), [9,11]\}
- This is just a smaller instance of the same problem
  - Effectively the activity-selection problem can be reduced to the question of (repeatedly) choosing one task from a set
  - Which suggests a greedy approach: try to make the locally-optimal choice at each stage

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
[6,9) & [1,7) & [2,4) \\
\hline
1 & 2 & 5 \\
10 & 4 & 6 \\
\hline
\end{tabular}
\end{center}
Greedy rules for activity-selection

- What greedy rules might be reasonable?
- Any of the following rules might be a candidate
  - Choose the shortest task
  - Choose the task that overlaps with fewest others
  - Choose the task that starts earliest
  - Choose the task that finishes earliest
  - Choose the task that starts latest
  - Choose the task that finishes latest
  - Or other possibilities, some much more complicated
- The greedy approach is a very local procedure
  - Make the choice that seems best now, without knowing how it affects future choices
- Sometimes the greedy approach works well – sometimes it even produces optimal answers – but frequently it doesn’t
An optimal choice

- In this case, it turns out there is a provably-optimal greedy rule
  - Always choose the task that finishes earliest
- e.g. given \{[6,9),[1,10),[2,4),[1,7),[5,6),[8,11),[9,11)\}
  - Choose [2,4), leaving \{[6,9),[5,6),[8,11),[9,11)\}
  - Choose [5,6), leaving \{[6,9),[8,11),[9,11)\}
  - Choose [6,9), leaving \{[9,11)\}
  - Choose [9,11)

- The formal proof of optimality is by contradiction (CLRS Ps. 373–5), but intuitively:
  - Suppose there exists a solution \{t_1, t_2, \ldots, t_k\} that does not include [2,4)
  - Assume that \(t_1, \ldots, t_k\) are ordered by finish time
  - Clearly \(t_1\) does not intersect with any of \(t_2 \ldots t_k\)
  - Clearly [2,4) finishes no later than \(t_1\)
  - Therefore [2,4) does not intersect with any of \(t_2 \ldots t_k\)
Running time for activity-selection

- The running time for this algorithm is dominated by the time to sort the tasks initially, i.e. $O(n \log n)$
- Greedy algorithms are often very fast, because they are usually very simple
- So why don’t we always use the greedy approach?
  - Because it doesn’t always work! (obviously…)
The Vertex Cover problem

- A vertex cover for a graph $G$ is a set of vertices $V' \subseteq V(G)$ such that every edge in $G$ has at least one end in $V'$
  - We say that $V'$ covers all the edges in $G$
- The Vertex Cover problem is to find the smallest vertex cover of $G$
- e.g. this graph:

```plaintext
\[ \text{\includegraphics[width=0.5\textwidth]{vertex_cover_graph.png}} \]
```

has a cover of size 4:

```plaintext
\[ \text{\includegraphics[width=0.5\textwidth]{vertex_cover_graph_cover.png}} \]
```
A greedy algorithm?

- One obvious greedy rule would be
  - At each stage, choose the vertex that covers the most remaining uncovered edges
- But for this graph:

![Graph 1](image1)

- This greedy rule would choose the middle vertex, and would ultimately give:

![Graph 2](image2)

- Whereas there is a better solution:

![Graph 3](image3)
The problem is that the locally-optimal choice (covering as many uncovered edges as possible) reduces our options for later choices.

Unfortunately most problems are not amenable to the greedy approach.

There is no known algorithm for Vertex Cover which is essentially better than just enumerating all possible subsets of the vertices.

Vertex Cover is an example of an \textit{NP-hard} problem.
P vs. NP

- A computational problem $x$ is in the class $\textbf{P}$ if there is a deterministic algorithm that solves $x$ and that runs in polynomial time (i.e. $O(n^k)$ for some $k$)
  - These are polynomial-time problems, often called *feasible* or *tractable*

- A computational problem $x$ is in the class $\textbf{NP}$ if there is a non-deterministic algorithm that solves $x$ and that runs in polynomial time
  - These are non-deterministic polynomial-time problems, often called *infeasible* or *intractable*
  - But these algorithms require lucky guesses to work efficiently
Two more NP problems

- Travelling Salesman (TSP): given a finite set of cities $C$ and a distance function $d(c_i, c_j) \in \mathbb{R}^+$, find the shortest circular tour that visits each city exactly once.

- Dominating Set: given a graph $G$, what is the smallest set of vertices $V' \subseteq V(G)$ such that every vertex in $V(G)$ is adjacent to at least one vertex in $V'$.
How hard are these problems?

• Every problem $x$ in the NP-hard class has the following properties
  o There is no known polynomial-time algorithm for $x$
  o The only known algorithms take exponential time
  o If you could solve $x$ in polynomial-time, then you could solve them all in polynomial-time

• Vertex Cover, Travelling Salesman, and Dominating Set are all NP-hard

• The most important open problem in theoretical computer science is whether or not this class of problems can be solved in polynomial-time
The 0-1 Knapsack problem

- Given a set of items $X$, each with a weight and a value, and given a knapsack $K$ that can hold weight $w$, find $X' \subseteq X$ such that
  - the items in $X'$ fit into $K$ (weight-wise), and
  - the total value of the items in $X'$ is maximised
- e.g. $K$ might be your MP3 player with capacity $w$, and $X$ might be the set of songs that you have, for each of which the weight represents its file-size, and the value represents how much you like it
- Whilst the 0-1 Knapsack problem is known to be NP-hard, the seemingly more-complicated Fractional Knapsack problem is trivial to solve with a greedy algorithm
  - In Fractional Knapsack, you can pack any fraction of any item into $w$
- We will examine a dynamic programming algorithm for 0-1 Knapsack that gives reasonable performance

Credit of the picture: "Knapsack". Licensed under CC BY-SA 2.5 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Knapsack.svg#mediaviewer/File:Knapsack.svg
Recursion for the 0-1 Knapsack problem

- Remember the essential issues with dynamic programming are to
  - Express the problem as one or more recurrence relations
  - Organise the sub-problems so that repeated work is minimised or eliminated
- An instance of 0-1 Knapsack with $n$ items has the form

\[
(\{w_1, \ldots, w_n\}, \{v_1, \ldots, v_n\}, w)
\]

- Consider solving sub-problems of the form

\[
(\{w_1, \ldots, w_m\}, \{v_1, \ldots, v_m\}, w')
\]

Where $m \leq n$ and $w' \leq w$

- Let $V(m, w)$ be the value of the optimal solution where we choose from the first $m$ items with capacity $w$
- Either the $m^{th}$ item is packed or it isn’t, so

\[
V(m, w) = \max(v_m + V(m - 1, w - w_m), V(m - 1, w))
\]

- With the trivial base cases $V(0, w) = V(m, 0) = 0$, and incorporating checks for exceeding $w$, this gives a (very inefficient) recursive algorithm
Dynamic programming for the 0-1 Knapsack problem

- We will construct a table in similar fashion to the longest common subsequence problem.
- Consider the instance (\{1, 2, 3\}, \{2, 3, 4\}, 5).
- Each entry in the table gives \(V(m, w)\).
- The first row and column come from the base case.
- The other entries come from the recursive rule.
  - Applied row-by-row left-to-right, as previously.

\[
V(m, w) = \max(V(m - 1, w - w_m) + v_m, V(m - 1, w)) \quad \text{when } w - w_m \geq 0
\]
\[
= V(m - 1, w) \quad \text{otherwise}
\]

<table>
<thead>
<tr>
<th>(m)</th>
<th>(0)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(V(0, m) = 0\) \quad \text{and} \quad \(V(m, 0) = 0\).

\[
V(1, 1) = \max(V(0, 0) + 2, V(0, 1)) \quad \text{Option 1}
\]
\[
= \max(V(0, 1) + 2, V(0, 2))
\]
\[
V(2, 1) = \max(V(1, 1), V(1, 2))
\]

\[\text{Dynamic Programming}\]
The final table looks like this

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

This algorithm is \(O(nw)\), because that’s the size of the table.
- And clearly it gives the optimal solution in this case.
- But be clear that this is not a polynomial-time algorithm for an NP-hard problem!
  - Why not?
- We can easily extract the items that form the solution.
  - Working from the bottom-right, use the equation:

  \[ T(m, w) = \begin{cases} T(m-1, w), & \text{if } V(m, w) = V(m-1, w) \\ \{m\} \cup T(m-1, w-w_m), & \text{otherwise} \end{cases} \]
Linear programming problems

- Recall that Fractional Knapsack is the same as 0-1 Knapsack except that you can pack any fraction of any item

- This is an example of a linear programming problem, which is any problem of the form
  - Find real numbers $x_1, \ldots, x_m, x_i \geq 0$
  - That maximise $\sum_{i=1}^{n} c_i x_i$
  - Subject to the constraints $\sum_{i=1}^{n} a_{ij} x_i < b_j, 1 \leq j \leq m$

- A linear programming problem is characterised by
  - A cost vector $c_1, \ldots, c_n$
  - A bounds vector $b_1, \ldots, b_m$
  - An $n \times m$ array of constraint coefficients $a_{ij}$

- For Fractional Knapsack:
  - The values constitute the cost vector
  - The capacity is the only bound (i.e. $m = 1$)
  - The weights constitute the constraint coefficients

- All linear programming problems can be solved by the simplex algorithm, which has exponential complexity but is generally feasible in practice
  - Simplex is effectively a hill-climbing algorithm
Integer linear programming problems

- Adding the requirement that all solutions to a linear programming problem be integer values gives an integer linear programming problem
  - i.e. all of the $x_i$ are required to be integers
- 0-1 Knapsack can be written as an integer linear programming problem, as can Travelling Salesman
- Given that both of these problems are NP-hard, as previously stated, we should not expect to find a feasible algorithm for solving integer linear programming problems
  - But if we did, that’d be great!
Approximation algorithms

- An approximation algorithm is an algorithm that produces solutions to NP-hard problems
  - But with no guarantee that the solutions are optimal
- e.g. an approximation algorithm for Travelling Salesman would return some circular tour – hopefully a good tour, but not necessarily the best tour
- The performance of an approximation algorithm $A$ on a given problem instance $I$ is often described by the ratio $A(I)/OPT(I)$
  - $A(I)$ is the value returned by $A$
  - $OPT(I)$ is the optimum value (if known)
  - Sometimes a known bound is substituted for $OPT(I)$
- For many such problems standard benchmark data exists for comparing algorithms’ performance
  - e.g. TSPLIB for Travelling Salesman
The football pools problem

• In many European countries, a popular form of gambling is the *football pools* based on soccer
• Each player picks the results of *n* matches, specifying for each either a home win, an away win, or a draw
• We can model a player’s bet as a word of length *n* over the alphabet \{0, 1, 2\}, where
  - 0 = home win
  - 1 = away win
  - 2 = draw

• e.g. 020201 would represent home wins for Matches 1, 3, and 5, an away win for Match 6, and draws for the others
• For six matches there are $3^6 = 729$ possible outcomes
  - To guarantee first prize would require 729 bets
• Suppose however that there is a second prize for five correct choices
  - How many bets must we place to guarantee at least second prize?
A graph domination problem

- Define a graph $F_6$ which has
  - 729 vertices: the words of length six over \{0, 1, 2\}
  - An edge connecting two vertices iff the vertices’ words differ in only one position
- Then the football pools second prize question corresponds to finding the size of the minimum dominating set for $F_6$
  - If we place every bet in the dominating set, then wherever the actual result is, it is guaranteed to be at most one jump from at least one of our bets!
- We have *reduced* the second prize question to the Dominating Set problem
  - “Reduced” in the sense that solving the latter gives a solution for the former
- This tells us that we shouldn’t expect to find an exact solution to the second prize question for non-trivial $n$
The $F_n$ graphs are called the *football pools graphs*
  o They are extensively studied

<table>
<thead>
<tr>
<th>$n$</th>
<th>No. of vertices</th>
<th>Best known dom. set</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>81</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>243</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>729</td>
<td>$\leq 73$</td>
</tr>
<tr>
<td>7</td>
<td>2,187</td>
<td>$\leq 186$</td>
</tr>
<tr>
<td>8</td>
<td>6,561</td>
<td>$\leq 486$</td>
</tr>
</tbody>
</table>

  e.g. the minimum dominating set for $F_4$ is a *perfect code*
    o Each vertex in $V(F_4)$ is adjacent to exactly
    one vertex in the dominating set

  A natural (greedy) approximation algorithm for this
  problem is at each stage to
    o Choose the vertex $v$ of maximum degree
    o Delete all of the vertices adjacent to $v
Greedy approximation algorithms for the TSP

• Consider geometric instances of the TSP
  o The distance function is symmetric
    ▪ \( d(c_i, c_j) = d(c_j, c_i) \)
  o And it satisfies the triangle inequality
    ▪ \( d(c_i, c_k) \leq d(c_i, c_j) + d(c_j, c_k) \)

• It is easy to imagine non-geometric instances, but many approximation algorithms work well only for geometric

• One simple algorithm is Nearest Neighbour (NN)
  o Start at a randomly-chosen city
  o Always visit next the closest unvisited city

• NN is clearly \( O(n^2) \)
  o Unfortunately it is not very good!

• We could imagine a version of NN that just tries all possible starting cities
  o Obviously that would be \( O(n^3) \)
A geometric instance of the TSP with NN

- The best case tour:
• The worst case tour:

Starting at city 12. Tour length is 842.94
TSP theorems

- Theorem: for any constant \( k > 1 \) there are instances \( I \) of the TSP such that \( NN(I) \geq k \, OPT(I) \)

- Theorem: suppose \( A \) is a polynomial-time approximation algorithm for the TSP such that \( A(I) \leq k \, OPT(I) \) for some constant \( k \) and for all instances \( I \)
  - Then there exists a polynomial-time algorithm to solve the TSP
  - Thus \( P = NP \)!
  - (Which most people don’t believe…)

- Thus it seems hopeless to search for a decent approximation algorithm for the TSP…
  - But for geometric instances we can do better!
Minimum spanning trees for the TSP

• The following algorithm is guaranteed to find a TSP tour for geometric $I$ that is at most twice the optimal length
  o Find a minimum spanning tree for $I$
  o Do a depth-first search on the tree
  o Visit the vertices in order of discovery time

• Given a graph $G$, a spanning tree for $G$ is a subset of $E(G)$ that is a tree, and that connects all of $V(G)$
  o A minimum spanning tree for $G$ is a spanning tree for $G$ with the smallest possible weight
  o $G$’s MST can be found in $O(|E(G)|)$ time
Create and search the MST …
… and take shortcuts

- If we simply follow the depth-first search – including the backtracking – we would walk along each edge once in each direction
  - This would create a tour that has length twice the MST, but with duplicated vertices
- The simplest solution is to take “shortcuts”, following the ordering of the vertices
  - i.e. visit them in order of discovery
Performance of MST-TSP

- The algorithm is guaranteed to find a TSP tour for geometric $I$ that is at most twice the optimal length.

- Observe first that removing one edge from the optimal tour for $I$ gives a spanning tree for $I$

\[ OPT(I) - \langle \text{one edge} \rangle = SpT \]
\[ \therefore OPT(I) - \langle \text{one edge} \rangle \geq MSpT \]
\[ \therefore MSpT < OPT(I) \]
\[ \therefore 2 MSpT < 2 OPT(I) \]

\[ A(I) \leq 2 MSpT \]
\[ \therefore A(I) < 2 OPT(I) \]
Coalesced simple paths for the TSP

- **Coalesced simple paths** is a greedy algorithm where at each stage, the partial tour is a collection of simple paths
  - Sort the edges by increasing weight
  - At each stage, add the lowest-weight edge that doesn’t create a cycle, or a vertex of degree 3

- At each stage we apply an *insertion method* $M$ that inserts one vertex into a closed tour $C$
  - $M$ selects an unused vertex $x$, then it inserts $x$ into $C$ at its *best position*

- To determine the best position, we consider each edge $(u, v) \in C$, and we select the edge that minimises

\[ d(u, x) + d(x, v) - d(u, v) \]

- Then $(u, v)$ is deleted, and $(u, x)$ and $(x, v)$ are added, creating a tour with one extra city

- Three common insertion methods are:
  - *Nearest insertion*: choose the $x$ closest to $C$
  - *Farthest insertion*: choose the $x$ furthest from $C$
  - *Random insertion*: choose $x$ randomly
Tour found by nearest insertion

- Nearest insertion tours ranged from 631–701 on the above example.
Tour found by farthest insertion

- Farthest insertion tours ranged from 594–679 on the above example.
Tour found by random insertion

- Random insertion tours ranged from 607–667 on the above example.
Cheapest insertion

- With this method we search through all edges \((u, v) \in C\) and all vertices \(x \notin C\) to find the pair that minimises

\[
d(u, x) + d(x, v) - d(u, v)
\]

- The previous three methods work in \(O(n^2)\) time
  - Because they separate choosing a vertex from choosing an insertion point
  - Cheapest insertion seems to require at least an additional factor of \(\log n\)

- The nearest insertion and cheapest insertion methods can be shown to produce tours of length no greater than twice the optimum
  - They are related to MST algorithms
Tour found by cheapest insertion

- Unfortunately I don’t have the length of this one…
Iterative improvement

- One common feature of the tours produced by the greedy heuristics is that it is usually easy to see how they can be improved by changing a few edges here and there.

- This leads to the idea of **iterative improvement**
  - Create a feasible solution (quickly)
  - Modify it slightly (and repeatedly) to improve it

- An iterative improvement algorithm requires
  - A rule for changing one feasible solution to another
  - A schedule for deciding which changes to make
Improving TSP tours

• A basic move for improving TSP tours involves deleting two edges and replacing them with two non-edges

• Consider the tour $A \rightarrow D \rightarrow \ldots \rightarrow C \rightarrow B \rightarrow \ldots \rightarrow A$

• $AD$ and $CB$ can be replaced by $AC$ and $DB$

• It should be clear that the result of this is still a valid tour
  ○ And that $D \rightarrow \ldots \rightarrow C$ is reversed
2-OPT

- Consider an iterative improvement algorithm that, in every iteration, examines every pair of edges in a tour, and performs an exchange if it would improve the tour.
- This procedure must eventually terminate:
  - The resulting tour is called 2-optimal.

- More complicated schemes involve deleting three edges and reconnecting the tour:
  - Or in general $k$ edges.

- A tour that cannot be improved by a $k$-edge exchange is called $k$-optimal:
  - In practice it is unusual to go beyond 3-optimal, because of the expense and because of diminishing returns.
A state-space graph

- We can abstract this process to consider a heuristic search on a huge graph called the state-space graph or the search space of the problem.

- The state-space graph of an instance $I$ of the TSP is denoted by $S(I)$
  - The vertices of $S(I)$ comprise all feasible tours for $I$.
  - Two vertices in $S(I)$ are connected iff they can be obtained from each other by the edge-exchange procedure described previously.

- Each vertex $T$ of $S(I)$ has an associated cost $c(T)$ which is the length of the tour $T$.

- To completely solve the instance $I$ requires finding which of the $(n-1)!$ vertices of $S(I)$ has the lowest cost.
Gradient-based search

- Needless to say $S(I)$ is usually VAST!
- But note that
  - A greedy insertion method returns one vertex in $S(I)$, i.e. one tour
  - An iterative improvement heuristic allows us to “walk” through $S(I)$ by moving from one tour to its neighbours
- Conceptually we have a “current” tour $T$, and in each iteration
  - We generate a neighbour $T'$ of $T$, and
  - We decide whether or not to “move” to $T'$
- This is the basis of gradient-based search algorithms
Hill-climbing

- The simplest gradient-based search procedure is known as hill-climbing
- Systematically generate neighbours $T'$ of the current tour $T$ and move to the first one with lower cost than $T$
- The process terminates when it is at a $T$ that has no neighbours of lower cost
  - At this point $T$ is 2-optimal

- An obvious variant (there are always variants!) is to always choose the best move at each step
  - Greedy iterative improvement!
Local optima

• A hill-climb will always finish at a vertex which has a lower cost than its neighbours
  o Such a vertex is a local minimum or local optimum of the state-space
• Unfortunately the state-space graph has an enormous number of local optima, only one of which is the global optimum that we would like to reach
  o Or sometimes there are multiple global optima…
• If we picture the search space as a kind of landscape where the height of a vertex corresponds to its cost, then $S(I)$ is a savagely jagged landscape of enormously high dimension
  o Hill-climbing has no way to avoid the local optima in this landscape
  o Because all moves are local improvements

• Many techniques have been proposed which incorporate into gradient search some way of escaping local optima
  o It is an area of very active research
• We will introduce three techniques
  o Simulated annealing
  o Tabu search
  o Genetic algorithms
Simulated annealing

- Simulated annealing tries to avoid local optima by allowing the search process to take “backward” moves
- Conceptually, it is based on the physical process of annealing, by which some solids can form crystals during cooling, if they cool slowly enough
  - The solid “searches for” the molecular configuration with the lowest potential energy
- Each iteration takes the form
  - Randomly generate a neighbour $T'$ of the current $T$
  - If $c(T') \leq c(T)$, accept $T'$
  - If $c(T') > c(T)$, accept $T'$ with probability $p$
  - So now “backward moves” are allowed and we can escape local optima
- But of course we don’t want to make backward moves near the end of the process
  - Backward move are intended to move us to a new part of the space, so we can search there
  - So we dynamically alter $p$ to make backward moves less likely as time goes on
- We maintain a *temperature variable* $t$ which goes down as time advances, and we relate $p$ to $t$
- Simulated annealing has worked well with many optimisation problems, but the performance is highly problem-specific and dependent on the cooling schedule
Tabu search

- The word *tabu* (or *taboo*) means something that is prohibited or forbidden
- *Tabu search* tries to avoid two weaknesses
  - The inability of hill-climbing to escape local optima
  - The early randomness of simulated annealing
- The aim is to spend most of the time exploring (near) local optima, whilst retaining the ability to escape them
- The fundamental idea is that we maintain a *tabu list* detailing the last $h$ vertices that have been visited, and at each iteration
  - Select the best possible neighbour $T'$ of $T$
  - If $T'$ is not on the tabu list, move to $T'$ and update the list
- Tabu search is very aggressive – each attempted move is in the best possible direction
  - Cycling would be inevitable without the tabu list
- The tabu list prevents the search from spending too much time near one local optimum, forcing it to visit other parts of the search space
- Tabu search also has been very successful, e.g. with football pools
  - But it is expensive and tricky to implement
  - Again results and settings (e.g. $h$) are highly problem-specific
Genetic algorithms

- *Genetic algorithms* (GAs) try to avoid getting stuck in local optima by maintaining a *population* of solutions
- The expectation is that these solutions will usually be distributed across the search space
  - It is highly unlikely that one (or even a few) local optima can trap all of them
- At each *generation* (iteration), the population of size $n$ is used to create $n$ new solutions, and the best $n$ of these “survives” into the next generation
  - Again the conceptual inspiration is from the physical world, in this case the principle of evolution by natural selection
  - As well as being perturbed locally, solutions are combined using *crossover* in the hope of finding improved solutions
- GAs have been extremely successful on a wide range of problems, and they work well in difficult domains that often are beyond simpler methods
  - But again much fine-tuning is required, and often much computational power too!
Computational intelligence techniques

• GAs are a fairly heavyweight technology, and we are moving far away from where we started this lecture
  o GAs and related technologies are studied extensively in CITS4404 in Semester 2
• Techniques that fall into this category are commonly called *computational intelligence* techniques:
  o Genetic (or evolutionary) algorithms
  o Swarm optimisation
  o Ant colony optimisation
  o Learning classifiers
  o Neural networks
  o Artificial immune systems
  o *etc.*
• Many of these techniques are based on processes or behaviour found in nature
  o Hence the term *nature-inspired computation*
• All of these techniques are conceptually simple, but much skill and judgement is required to apply them well
• It is simply not possible to say which one “works best” in general