Algorithm Design

In this section we will consider some general algorithmic techniques for optimization problems — namely greedy algorithms, dynamic programming and Approximation Algorithms.

A greedy algorithm proceeds by making a single choice at each stage of the computation — at each stage the algorithm chooses the “best” move to make based on purely local information. Previously seen examples include Kruskal’s algorithm, Prim’s algorithm and Huffman Coding.

Greedy algorithms are usually extremely efficient, but they can only be applied to a small number of problems.

Greedy Algorithms

Consider the following simple computational problem.

**ACTIVITY SELECTION**

Instance: A set \( S = \{t_1, t_2, \ldots, t_n\} \) of “activities” where each activity \( t_i \) has an associated start time \( s_i \) and finish time \( f_i \).

Question: Select the largest possible number of tasks from \( S \) that can be completed without incompatibilities (two activities are incompatible if they overlap).

**Example** Consider the following set of activities

\[
\{(6, 9), (1, 10), (2, 4), (1, 7), (5, 6), (8, 11), (9, 11)\}
\]

The following schedules are all allowable

\[
(1, 10) \\
(1, 7), (8, 11) \\
(2, 4), (5, 6), (9, 11)
\]

Intervals

There is an obvious relationship between activities and intervals on the real line.

An interval of the real line consists of the real numbers lying between two reals called the endpoints of the interval.

\[
(a, b) = \{x \in \mathbb{R} \mid a < x < b\}
\]

If the interval includes its endpoints then it is said to be closed, otherwise open. It can also be open at one endpoint and closed at the other.

\[
(a, b) = \{x \in \mathbb{R} \mid a < x < b\}
\]

\[
[a, b) = \{x \in \mathbb{R} \mid a \leq x < b\}
\]

\[
(a, b] = \{x \in \mathbb{R} \mid a < x \leq b\}
\]

\[
[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}
\]

For definiteness we will henceforth make the assumption that all the activity intervals are closed on the left and open on the right.

\[
t_i = [s_i, f_i)
\]
Problem reduction

To solve this problem we must make some choice of the first interval, then the second interval and so on. Clearly the later choices depend on the earlier ones in that some time-slots are no longer available.

Suppose that we (arbitrarily) select the interval \([1, 7)\) from the collection

\[
\{[6, 9), [1, 10), [2, 4), [1, 7), [5, 6), [8, 11), [9, 11)\}
\]

Then all the intervals that overlap with this one can no longer be scheduled, leaving the set

\[
\{[8, 11), [9, 11)\}
\]

from which we must choose the largest possible set of pairwise disjoint intervals — in this case just one of the remaining intervals.

This is simply a smaller instance of the same problem **ACTIVITY SELECTION**. Therefore an algorithm for the problem can be expressed recursively simply by specifying a rule for choosing one interval.

Greedy approach

It is easy to see that choosing \([1,7)\) was a bad choice, which raises the question of what would be a good choice?

A greedy algorithm simply chooses what is locally the best option at every stage. There are various possible ways to be greedy, including

- Choose the shortest interval
- Choose the interval starting the first
- Choose the interval finishing the first
- Choose the interval that intersects with the fewest others

The greedy approach can be viewed as a very local procedure — making the best choice for the current moment without regard for any possible future consequences of that choice.

**Sometimes a greedy approach yields an optimal solution, but frequently it does not.**

Activity Selection

Consider the greedy approach of selecting the interval that finishes first from the collection

\[
\{[6, 9), [1, 10), [2, 4), [1, 7), [5, 6), [8, 11), [9, 11)\}
\]

Then we would choose \([2, 4)\) as the first interval, and after eliminating clashes we are left with the task of finding the largest set of mutually disjoint intervals from the set

\[
\{[6, 9), [5, 6), [8, 11), [9, 11)\}
\]

At this stage, we simply apply the algorithm recursively. Therefore being greedy in the same way we select \([5, 6)\) as the next interval, and after eliminating clashes (none in this case) we are left with.

\[
\{[6, 9), [8, 11), [9, 11)\}
\]

Continuing in this way gives the ultimate result that the largest possible collection of non-intersecting intervals is

\[
[2,4) \text{ then } [5,6) \text{ then } [6,9) \text{ then } [9,11).
\]

Algorithm

As a precondition the list of tasks must be sorted into ascending order of their finish times to ensure

\[
\text{finish}(t_1) \leq \text{finish}(t_2) \leq \text{finish}(t_3) \leq \ldots
\]

The pseudo-code will then process the sorted list of tasks \(t\):

```plaintext
procedure GREEDY-ACTIVITY-SEL(t)
    \(A \leftarrow \{t_1\}\)
    \(i \leftarrow 1\)
    for \(m \leftarrow 2\) to length(t) do
        if start(t_m) \(\geq\) finish(t_i) then
            \(A \leftarrow A \cup \{t_m\}\)
            \(i \leftarrow m\)
        end if
    end for
    return \(A\)
```

It returns \(A\), a subset of compatible activities.
Does it work?

The greedy algorithm gives us a solution to the activity scheduling problem — but is it actually the best solution, or could we do better by considering the global impact of our choices more carefully.

For the problem ACTIVITY SELECTION we can show that the greedy algorithm always finds an optimal solution.

We suppose first that the activities are ordered by finishing time - so that

\[ f_1 \leq f_2 \leq \cdots \leq f_n \]

Now consider some optimal solution for the problem consisting of \( k \) tasks

\[ t_{i_1}, t_{i_2}, \ldots, t_{i_k} \]

Then

\[ t_1, t_{i_2}, \ldots, t_{i_k} \]

is also an optimal solution since it will also consist of \( k \) tasks.

Intuitive Proof

The formal proof that we can use \( t_1 \) as the first task and be certain that we will not change the number of compatible tasks is rather involved and you are referred to the textbook (see CLRS, pages 373-375).

However the basic idea is a proof by contradiction. Assume using \( t_1 \) results in a sub-optimal solution and therefore we can find a compatible solution with \((k+1)\) tasks. This would only be possible if we can find two tasks \( t'_1 \) and \( t''_1 \) which occupy the same interval as \( t_1 \). But this would imply

\[ (s_1 \leq s'_1 < f'_1 \leq s''_1 < f''_1 \leq f_1) \]

and hence that \( f'_1 < f_1 \) but we know that the tasks are sorted in order of ascending finish times, so no task can have a finish time less that that of \( t_1 \), leading to a contradiction. Hence using \( t_1 \) as the first task must lead to an optimal solution with \( k \) tasks.

Running time

The running time for this algorithm is dominated by the time taken to sort the \( n \) inputs at the start of the algorithm.

Using quicksort this can be accomplished in an average time of \( O(n \log n) \).

As greedy algorithms are so simple, they always have low degree polynomial running times.

Because they are so quick, we might be tempted to ask why we should not always use greedy algorithms.

Unfortunately, greedy algorithms only work for a certain narrow range of problems — most problems cannot be solved by a greedy algorithm.

Vertex Cover

A vertex cover for a graph \( G \) is a set of vertices \( V' \subseteq V(G) \) such that every edge has at least one end in \( V' \) (the set of vertices covers all the edges.

The following graph

has a vertex cover of size 4.

The VERTEX COVER problem is to find the smallest vertex cover for a graph \( G \).
A greedy algorithm

One greedy algorithm is to cover as many edges as possible with each choice, by choosing the vertex of highest degree at each stage and then deleting the covered edges.

For this graph

the greedy algorithm gives

while the true solution is

Greed is not always good

The previous example shows that it does not always pay to be greedy. Although choosing the vertex of highest degree does cover the greatest number of edges, that choice makes our later choices worse.

In problems where the greedy algorithm works, the earlier choices do not interfere negatively with the later choices.

Unfortunately, most problems are not amenable to the greedy algorithm.

VERTEX COVER is actually a very hard problem, and there is no known algorithm that is essentially better than just enumerating all the possible subsets of vertices. (Technically speaking, it is an example of a problem that is known to be NP-hard.)

Non-deterministic polynomial time

A computational problem is in the class \(P\), (the polynomial time problems) if there is a deterministic algorithm that solves the problem and runs in time \(O(n^k)\) where \(k\) is some integer. These problems are generally considered feasible.

A computational problem is in the class \(NP\), (the non-deterministic polynomial time problems) if there is a non-deterministic algorithm that that can solve the problem in polynomial time.

That is, an \(NP\) algorithm requires lucky guesses to work efficiently (i.e. guessing what the optimal vertex cover is).

More NP-problems

Consider the following two problems:

TRAVELLING SALESMAN
Instance: A finite set \(C = \{c_1, c_2, \ldots, c_n\}\) of "cities", a "distance" \(d(c_i, c_j) \in \mathbb{R}^+\) between each pair of cities.
Question: What is the shortest circular tour visiting each city exactly once?

DOMINATING SET
Instance: A graph \(G\)
Question: What is the smallest dominating set for \(G\)?

(A dominating set of a graph is a set of vertices \(V' \subseteq V\) such that every vertex of \(G\) has distance at most 1 from some vertex in \(V'\).)
How hard are these problems?

There are no algorithms known for these problems whose time complexity is a polynomial function of the size of the input. This means that the only known algorithms take time that is exponential in the size of the input.

There is a large class of problems, known as NP-hard problems which have the following properties:

- There is no polynomial time algorithm known for the problem
- If you could solve one of these problems in polynomial time, then you could solve them all in polynomial time

Both TRAVELLING SALESMAN and DOMINATING SET are NP-hard.

The most important problem in theoretical computer science is whether or not this class of problems can be solved in polynomial time.

The 0-1 Knapsack Problem

Suppose we are given a knapsack of a given capacity, and a selection of items, each with a given weight and value. The 0-1 knapsack problem is to select the combination of items with the greatest value that will fit into the knapsack.

Formally, if \( W \) is the size of the knapsack and \( \{1, ..., n\} \) is a set of items where the weight of \( i \) is \( w_i \) and the value of \( i \) is \( v_i \), then the problem is to:

Select \( T \subseteq \{1, ..., n\} \) that maximizes \( \sum_{i \in T} v_i \), given \( \sum_{i \in T} w_i < W \).

For example \( W \) might be the amount of memory on an MP3 player, \( w_i \) may be the size of the song \( i \), and \( v_i \) may reflect how much you like song \( i \).

A dynamic programming solution

The structure of a dynamic programming algorithm is to:

1. define the solution to the problem in terms of solutions to sub-problems;
2. recursively solve the smaller sub-problems, recording the solutions in a table;
3. construct the solution to the original problem from the table.

While there is no known “feasible” solution for the 0-1 knapsack problem we will examine a dynamic programming solution that can give reasonable performance.

The Fractional Knapsack Problem

The fractional knapsack problem is similar, except that rather than choosing which items to take, you are able to choose how much of each item you will take. That is the problem is to find a function \( T : \{1, ..., n\} \to [0, 1] \) that maximizes \( \sum_{i \in T} T(i)v_i \), given \( \sum_{i \in T} w_i < W \).

It is easy to see that the fractional knapsack problem can be solved by a greedy algorithm. However the 0-1 knapsack problem is much harder, and has been shown to be NP-complete.

While there is no known “feasible” solution for the 0-1 knapsack problem we will examine a dynamic programming solution that can give reasonable performance.
A recursive solution

Given the 0-1 knapsack problem specified by the pair \( (\{w_1, ..., w_n\}, \{v_1, ..., v_n\}, W) \), we will consider the solution to the sub-problems specified by the pairs \( (\{w_1, ..., w_m\}, \{v_1, ..., v_m\}, w) \) where \( m < n \) and \( w < W \).

Let \( V(m, w) \) be the value of the optimal solution to this subproblem. Then for any \( m \) and any \( w \), we can see

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

Since \( V(0, w) = 0 \) for all \( w \) this allows us to define a (very inefficient) recursive algorithm.

A dynamic programming solution

Often inefficient recursive algorithms can be made more efficient by using dynamic programming. The structure of a dynamic programming algorithm is to:

1. define as recursive solution to the problem in terms of solutions to sub-problems;
2. recursively solve the smaller sub-problems, recording the solutions in a table;
3. construct the solution to the original problem from the table.

For the 0-1 knapsack problem we will construct a table where the entries are \( V(i, j) \) for \( i = 0, ..., n \) and \( j = 0, ..., W \).

Example

Suppose \( W = 5 \) and we are given three items where

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_i )</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>( w_i )</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

The table initially looks like

<table>
<thead>
<tr>
<th>( i )</th>
<th>( w )</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></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></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Pseudo-code

```plaintext
Knapsack(\{w_1, ..., w_n\}, \{v_1, ..., v_n\}, W)
for w from 1 to n do
    V(0, w) ← 0
for i from 1 to n do
    for w from 1 to n do
        if V(i-1, w) > v_i + V(i-1, w-w_i) do
            V(i, w) ← V(i-1, w)
        else
            V(i, w) ← v_i + V(i-1, w-w_i)
    return V(n, W)
```

It is clear that the complexity of this algorithm is \( O(nW) \). Note that this is not a polynomial solution to an NP-complete problem. Why not
Example

\[
\begin{array}{c|ccccc}
  i \backslash w & 0 & 1 & 2 & 3 & 4 & 5 \\
  \hline
  0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 0 & 2 & 2 & 2 & 2 & 2 \\
  2 & 0 & 2 & 3 & 4 & 4 & 4 \\
  3 & 0 & 2 & 3 & 4 & 6 & 7 \\
\end{array}
\]

Note that the actual items contributing to the solution (that is, items 2 and 3) can be found by examination of the table. If \( T(i, w) \) are the items that produce the solution \( V(i, w) \), then

\[
T(i, w) = T(i - 1, w) \text{ if } V(i, w) = V(i - 1, w) \\
= \{i\} \cup T(i - 1, w - w_i) \text{ otherwise.}
\]

Linear Programming

The fractional knapsack problem is an example of a linear programming problem. A linear programme is an optimization problem of the form:

Find real numbers: \( x_1, \ldots, x_n \)

that maximizes \( \sum_{i=1}^{n} c_i x_i \)

subject to \( \sum_{j=1}^{m} a_{ij} x_i \leq b_j \) for \( j = 1, \ldots, m \)

and \( x_i \geq 0 \) for \( j = 1, \ldots, n \).

Therefore a linear programme is paramaterized by the the cost vector, \( (c_1, \ldots, c_n) \), an \( n \times m \) array of constraint coefficients, \( a_{ij} \), and a bounds vector \( (b_1, \ldots, b_m) \).

It is clear the fractional knapsack problem can be presented as a linear programme.

Applications of linear programming

Many natural optimization problems can be expressed as a linear programme.

For example, given a weighted, directed graph, \( G = (V, E) \), the length of shortest path from \( s \) to \( t \) can be described using a linear programme. Using the distance array from the Bellman-Ford algorithm, we have the programme:

Maximize \( d[t] \)

subject to \( d[v] - d[u] \leq w(u, v) \) for \( j = 1, \ldots, m \)

and \( d[s] = 0 \).

Maximum flow problems can also be easily converted into linear programmes.

Solving linear programmes

All linear programmes can be solved by the simplex algorithm, which requires exponential time, but is generally feasible in practise.

The simplex algorithm is effectively a hill-climbing algorithm that moves incrementally improves the solution until no further improvements can be made.

There are also polynomial interior point methods to solve linear programmes.

We won’t examine these algorithms. Rather we will simply consider the technique of converting problems into linear programmes.
Example

Maximize $x + y$, where $x + 2y \leq 4$
$y - x \leq 1$
$x - y \leq 1$
$x, y \geq 0$

Integer Linear Programming

Adding the constraint that all solutions to a linear programme be integer values, gives an integer linear programme.

The 0-1 knapsack problem can be written as an integer linear programme, as can the travelling salesmen problem.

Therefore we should not expect to find a feasible algorithm to solve the integer linear programming problem.

Approximation Algorithms

An approximation algorithm is an algorithm that produces some feasible solution but with no guarantee that the solution is optimal.

Therefore an approximation algorithm for the travelling salesman problem would produce some valid circular tour, but it may not be the shortest tour.

An approximation algorithm for the minimum dominating set problem would produce some dominating set for $G$, but it may not be the smallest possible dominating set.

The performance of an approximation algorithm on a given instance $I$ is measured by the ratio

$$A(I)/OPT(I)$$

where $A(I)$ is the value given by the approximation algorithm and $OPT(I)$ is the true optimum value.

Standard Instances

Both TRAVELLING SALESMAN and DOMINATING SET have been fairly extensively studied, and a number of algorithms for their solution have been proposed.

In each case there are some standard instances for would-be solvers to test their code on. A package called TSPLIB provides a variety of standard travelling salesman problems. Some of them have known optimal solutions, while others are currently unsolved and TSPLIB just records the best known solution.

There are problems with around 2000 cities for which the best solution is not known, but this problem has been very heavily studied by brilliant groups of researchers using massive computer power and very sophisticated techniques.
The football pool problem

In many European countries a popular form of lottery is the “football pools”, which are based on the results of soccer matches. Each player picks the results of \( n \) matches, where the result can be either a Home Win, Away Win or Draw.

By assigning three values as follows

- 0 for Home Win
- 1 for Away Win
- 2 for Draw

we can think of this choice as a word of length \( n \) with entries from the alphabet \( \{0, 1, 2\} \).

For example

\[ 020201 \]

would mean that the player had picked Home Wins for matches 1, 3 and 5, Away Win for match 6 and Draws for matches 2 and 4.

Winning 2nd prize

Now there are a total of 729 possible outcomes for the 6 matches. To guarantee winning the first prize we would need to make 729 different entries to cover every possible outcome.

Suppose however that getting all but one of the predictions correct results in winning second prize. So for example if our entry was 020201 and the actual outcome was 010201 then we would have 5 out of 6 correct and would win second prize.

In trying to generate pools “systems” we want to be able to answer the question

“How many entries do we need to make in order to guarantee winning at least second prize?”

A graph domination problem

We can define a graph \( F_6 \) as follows:

The vertices of \( F_6 \) are the 729 words of length 6 over \( \{0, 1, 2\} \).
Two vertices are adjacent if the corresponding words differ in only one coordinate position.

Then we are seeking a minimum dominating set for the graph \( F_6 \).

More generally, we can define a series of graphs \( F_n \) where the vertices are the \( 3^n \) words of length \( n \) with entries from \( \{0, 1, 2\} \) with the same rule for determining adjacency.

This collection of graphs is called the football pool graphs and has been quite extensively studied with regard to the size of the minimum dominating set.

Known records

The following are the best known values for a minimum dominating set for \( F_n \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>Number 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>2187</td>
<td>( \leq 186 )</td>
</tr>
<tr>
<td>8</td>
<td>6561</td>
<td>( \leq 486 )</td>
</tr>
</tbody>
</table>

Notice that the minimum dominating set for \( F_4 \) is perfect — each vertex is adjacent to 8 others, so that each vertex of the dominating set dominates 9 vertices. As there are 81 vertices in \( F_4 \) this means every vertex is dominated by exactly one vertex in the dominating set.

This is usually called a perfect code.
A greedy approximation algorithm

There is a natural greedy approximation algorithm for the minimum dominating set problem.

Start by selecting a vertex of maximum degree (so it dominates the greatest number of vertices). Then mark or delete all of the dominated vertices, and select the next vertex that dominates the greatest number of currently undominated vertices. Repeat until all vertices are dominated.

The graph $P_5$ (a path with 5 vertices) shows that this algorithm does not always find the optimal solution.

Types of Travelling Salesman Instance

Consider a travelling salesman problem defined in the following way. The "cities" are $n$ randomly chosen points $c_i = (x_i, y_i)$ on the Euclidean plane, and the "distances" are defined by the normal Euclidean distance

$$d(c_i, c_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

or the Manhattan distance

$$d(c_i, c_j) = |x_i - x_j| + |y_i - y_j|$$

Instance of the travelling salesman problem that arise in this fashion are called geometric travelling salesman problems. Here the "distance" between the cities is actually the geometric distance between the corresponding points under some metric.

Properties of geometric instances

All geometric instances have the properties that they are symmetric and satisfy the triangle inequality.

If

$$d(c_i, c_j) = d(c_j, c_i)$$

for all pairs of cities in an instance of TRAVELLING SALESMAN then we say that the instance is symmetric.

If

$$d(c_i, c_k) \leq d(c_i, c_j) + d(c_j, c_k)$$

for all triples of cities in an instance of TRAVELLING SALESMAN then we say that the instance satisfies the triangle inequality.

Non-geometric instances

Of course it is easy to define instances that are not geometric.

Let $X = \{A, B, C, D, E, F\}$

Let $d$ be given by

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>2</td>
<td>∞</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Many approximation algorithms only work for geometric instances because it is such an important special case, but remember that it is only a special case!
Nearest Neighbour

One example of an approximation algorithm is the following greedy algorithm known as Nearest Neighbour (NN).

- Start at a randomly chosen vertex
- At each stage visit the closest currently unvisited city

For an $n$-city instance of TRAVELLING SALESMAN this algorithm takes time $O(n^2)$.

For any instance $I$, let $NN(I)$ be the length of the tour found by $NN$ and let $OPT(I)$ be the length of the optimal tour. Then $NN(I)/OPT(I)$ is a measure of how good this algorithm is on a given instance.

Unfortunately this is not very good.

A geometric instance of NN

The best case gave a tour of length 636.28

A geometric instance of NN

The worst case gave a tour of length 842.94

Approximation algorithms

**Theorem** For any constant $k > 1$ there are instances of TRAVELLING SALESMAN such that $NN(I) \geq k \cdot OPT(I)$.

Even more seriously this is not just because $NN$ is not sufficiently sophisticated — we cannot expect good behaviour from any polynomial time heuristic.

**Theorem** Suppose $A$ is a polynomial time approximation algorithm for TRAVELLING SALESMAN such that $A(I) \leq k \cdot OPT(I)$ for some constant $k$. Then there is a polynomial time algorithm to solve TRAVELLING SALESMAN.

Therefore it seems hopeless to try to find decent approximation algorithms for TRAVELLING SALESMAN.
Minimum spanning tree

Suppose that we have an instance $I$ of TRAVELLING SALESMAN that is symmetric and satisfies the triangle inequality. Then the following algorithm called $MST$ is guaranteed to find a tour that is at most twice the optimal length.

- Find a minimum spanning tree for $I$
- Do a depth-first search on the tree
- Visit the vertices in order of discovery time

Then

$$MST(I) \leq 2 \text{OPT}(I).$$

In order to see why this works, we first observe that removing one edge from the optimal tour yields a spanning tree for $I$, and therefore the weight of the minimum spanning tree is less than the length of the shortest tour.

... and take shortcuts

If we were to simply follow the path of the depth-first search algorithm — including the backtracking — we would walk along each edge exactly once in each direction, creating a tour that has length exactly twice the weight of the minimum spanning tree, but is illegal because it visits some vertices twice.

The simple solution is to just take “shortcuts” according to the ordering of the vertices.

Search the tree ...

Perform a depth first search on the minimum spanning tree.

Coalesced simple paths

The method of coalesced simple paths uses a greedy method to build up a tour edge by edge. At every stage the “partial tour” is a collection of simple paths.

- Sort the edges into increasing weight
- At each stage add the lowest weight edge possible without creating a cycle or a vertex of degree 3.
- Join the ends of the path to form a cycle

This algorithm proceeds very much like Kruskal’s algorithm, but the added simplicity means that the complicated union-find data structure is unnecessary.
Insertion methods

There is a large class of methods called insertion methods which maintain a closed cycle as a partial tour and at each stage of the procedure insert an extra vertex into the partial tour.

Suppose that we are intending to insert the new vertex $x$ into the partial tour $C$ (called $C$ because it is a cycle).

In turn we consider each edge $\{u, v\}$ of the partial tour $C$, and we find the edge such that

$$d(u, x) + d(x, v) - d(u, v)$$

is a minimum.

Then the edge $\{u, v\}$ is deleted, and edges $\{u, x\}$ and $\{x, y\}$ added, hence creating a tour with one additional edge.

Three insertion techniques

Random insertion

At each stage the next vertex $x$ is chosen randomly from the untouched vertices.

Nearest insertion

At each stage the vertex $x$ is chosen to be the one closest to $C$.

Farthest insertion

At each stage the vertex $x$ is chosen to be the one farthest from $C$.

(In all three insertion methods the vertex $x$ is chosen first and then it is inserted in the best position.)

Tour found by nearest insertion

Nearest insertion tours ranged from 631 to 701 on the above example.

Example of a tour found by nearest insertion

Tour found by farthest insertion

Farthest insertion tours ranged from 594 to 679 on the above example.

Example of a tour found by farthest insertion
Tour found by random insertion

Random insertion tours ranged from 607 to 667 on the above example.

Example of a tour found by random insertion

A fourth insertion technique

Cheapest insertion

This method is a bit more expensive than the other methods in that we search through all the edges \( \{u, v\} \) in \( C \) and all the vertices \( x \notin C \) trying to find the vertex and edge which minimizes

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

The other three methods can all be programmed in time \( O(n^2) \) whereas this method seems to require at least an additional factor of \( \log n \).

Nearest insertion and cheapest insertion can be shown to produce tours of length no greater than twice the optimal tour length by their close relationship to minimum spanning tree algorithms.

Iterative Improvement

One common feature of the tours produced by the greedy heuristics that we have seen is that it is immediately easy to see how they can be improved, just by changing a few edges here and there.

The procedure of \textit{iterative improvement} refers to the process of starting with a feasible solution to a problem and changing it slightly in order to improve it.

An iterative improvement algorithm involves two things

- A rule for changing one feasible solution to another
- A \textit{schedule} for deciding which moves to make

Improving TRAVELLING SALESMAN tours

A basic move for TRAVELLING SALESMAN problems involves deleting two edges in the tour, and replacing them with two non-edges, as follows.

Suppose the tour runs \( AD, D \sim C, CB, B \sim A \). Then deleting \( AD \) and \( CB \), we replace them with \( AC \) and \( DB \).
Consider now an iterative improvement algorithm that proceeds by examining every pair of edges, and performing an exchange if the tour can be improved.

This procedure must eventually terminate, and the resulting tour is called 2-optimal.

There are more complicated “moves” that involve deleting 3 edges and reconnecting the tour, and in general deleting $k$ edges and then reconnecting the tour.

A tour that cannot be improved by a $k$ edge exchange is called $k$-optimal. In practice it is rare to compute anything beyond 2-optimal or 3-optimal tours.

**2-OPT**

A state space graph

We can view this process in a more abstract sense as a heuristic search on a huge graph called the state space graph.

We define the state space graph $S(I)$ for an instance of TRAVELLING SALESMAN as follows.

The vertices of $S(I)$ consist of all the feasible tours for the instance $I$. Two feasible tours $T_1$ and $T_2$ are neighbours if they can be obtained from each other by the edge exchange process above.

Each vertex $T$ has a cost $c(T)$ associated with it, being the length of the tour $T$.

To completely solve TRAVELLING SALESMAN requires finding which of the $(n-1)!$ vertices of $S(I)$ has the lowest cost.

Searching the state space graph

In general $S(I)$ is so vast that it is totally impossible to write down the entire graph.

The greedy insertion methods all provide us with a single vertex in $S(I)$ (a single tour), and the iterative improvement heuristics all involve doing a walk in $S(I)$ moving along edges from tour to neighbouring tour attempting to find the lowest cost vertex.

In this type of state space searching we have the concept of a “current” tour $T$ and at each stage of the search we generate a neighbour $T'$ of $T$ and decide whether the search should proceed to $T'$ or not.

Hill-climbing

The simplest heuristic state-space search is known as hill-climbing.

The rule for proceeding from one state to another is very easy

- Systematically generate neighbours $T'$ of $T$ and move to the first neighbour of lower cost than $T$.

This procedure will terminate when $T$ has no neighbours of lower cost — in this case $T$ is a 2-optimal tour.

An obvious variant of this is to always choose the best move at each step.
A local optimum

A hill-climb will always finish on a vertex of lower cost than all its neighbours — such a vertex is a local minimum.

Unfortunately the state space graph has an enormous number of local minima, each of them possibly tremendously different from the global minimum.

If we mentally picture the state space graph as a kind of “landscape” where costs are represented by heights, then $S(I)$ is a savagely jagged landscape of enormously high dimension.

Hill climbing merely moves directly into the nearest local optimum and cannot proceed from there.

State-space for DOMINATING SET

We can apply similar methods to the graph domination problem provided that we define the state-space graph carefully.

Suppose that we are trying to see whether a graph $G$ has a dominating set of size $k$. Then the “states” in the state space graph are all the possible subsets of $V(G)$ of size $k$. The “cost” of each can be taken to be the number of vertices not dominated by the corresponding $k$-subset. The solution that we are seeking is then a state of cost 0.

Now we must define some concept of “neighbouring states”. In this situation a natural way to define a neighbouring state is the state that results from moving one of the $k$ vertices to a different position.

Heuristic search for graph domination

We can now apply the hill-climbing procedure to this state space graph.

In this fashion the search “wanders” around the state-space graph, but again it will inevitably end up in a local minimum from which there is not escape.

Hill climbing is unsatisfactory because it has no mechanism for escaping locally optimum solutions. Ideally we want a heuristic search technique that tries to improve the current solution but has some method for escaping local optima.

Two techniques that have been proposed and extensively investigated in the last decade or so are

Simulated Annealing
Tabu Search

Annealing

Annealing is a physical process used in forming crystalline solids.

At a high temperature the solid is molten, and the molecules are moving fast and randomly. If the mixture is very gradually cooled, then as the temperature drops the mixture becomes more ordered, with molecules beginning to align into a crystalline structure. If the cooling is sufficiently slow, then at freezing point the resulting solid has a perfect regular crystalline structure.

The crystalline structure has the lowest potential energy, so we can regard the process as trying to find the configuration of a group of molecules with a global minimum potential energy.

Annealing is successful because the slow cooling allows the physical system to escape from local minima.
Simulated annealing

Simulated annealing is an attempt to apply these same principles to problems of combinatorial optimization.

For TRAVELLING SALESMAN we regard the optimal tour as the “crystal” for which we are searching and the other tours, being less perfect, as the flawed semi-molten crystals, while for GRAPH DOMINATION we regard the states with cost 0 (that is, genuine dominating sets) as the “crystals”.

The overall structure of simulated annealing is:

• Randomly generate a neighbour $T'$ of the vertex $T$
• If $c(T') \leq c(T)$ then accept the move to $T'$
• If $c(T') > c(T)$ then with a certain probability $p$ accept the move to $T'$

The probability $p$ of accepting an uphill move is dynamically altered throughout the algorithm.

Uphill moves

Dynamically altering $p$ is usually done by maintaining a temperature variable $t$ which is gradually lowered throughout the operation of the algorithm, and applying the following rules.

Suppose that we are currently at a vertex $T$ with a cost $c(T)$. The randomly generated neighbour $T'$ of $T$ has cost $c(T')$ and so if the move is made then the difference will be

$$\Delta c = c(T') - c(T)$$

Then the probability of accepting the move is taken to be

$$p = \exp(-\Delta c/t)$$

If $\Delta c < 0$, then $p > 1$, so this corresponds to accepting all moves to a lower cost neighbour.

Otherwise, if $t$ is high, then $-\Delta c/t$ is very small and $p \approx 1$. If $t$ is small then $-\Delta c/t$ will be large and negative and $p \approx 0$.

Cooling schedule

Therefore at high temperatures, almost all moves are accepted, good or bad, whereas as the temperature reduces, fewer bad moves are accepted and the procedure settles down again. When $t \approx 0$ then the procedure reverts to a hill-climb.

The value of the the initial temperature and the way in which it is reduced is called a cooling schedule:

• Start with some initial temperature $t_0$
• Perform $N$ iterations at each temperature
• Reduce the temperature by a constant multiplicative factor $t \leftarrow Kt$

For example the values $t_0 = 1$, $N = 1000$, $K = 0.95$ might be suitable.

Performance of this algorithm is highly problem-specific and cooling schedule-specific.

How good is it?

Simulated annealing has had success in several areas of combinatorial optimization, particularly in problems with continuous variables.

In general it seems to work considerably better than hill-climbing, though it is not clear whether it works much better than multiple hill-climbs.

Each of these combinatorial optimization heuristics has their own adherents, and something akin to religious wars can erupt if anyone is rash enough to say “X is better than Y”.

Experimentation is fraught with problems also, in that an empirical comparison of techniques depends so heavily on the test problems that almost any desired result can be convincingly produced by careful enough choice.

Nonetheless the literature is liberally dotted with “An empirical comparison of ... and ...”.

Performance of this algorithm is highly problem-specific and cooling schedule-specific.
Tabu search

The word *tabu* (or taboo) means something prohibited or forbidden.

Tabu search is another combinatorial search heuristic that combines some of the features of hill-climbing and simulated annealing. However it can only be used in slightly more restricted circumstances.

Tabu search attempts to combat two obvious weaknesses of hill-climbing and simulated annealing — the inability of hill-climbing to escape from local minima, and the early waste of time in simulated annealing where the temperature is very high and the search is proceeding almost randomly with almost no pressure to improve the solution quality.

Tabu search attempts to spend almost all of its time close to local minima, while still having the facility to escape them.

The basic idea

The basic idea of a tabu search is that it always maintains a *tabu list* detailing the last $h$ vertices that it has visited.

- Select the best possible neighbour $T'$ of $T$.
- If $T'$ is not on the tabu list, then move to it and update the tabu list accordingly.

We notice that the tabu search is very aggressive — it always seeks to move in the best possible direction. Without a tabu list this process would always end in a cycle of length 2, with the algorithm flipping between a local minimum and its nearest neighbour.

The tabu list prevents the search from immediately returning to a recently visited tour and (hopefully) forces it to take a different track out of that local minimum.

Practical considerations

The main problem of tabu search is that at each iteration it requires complete enumeration of the neighbourhood of a vertex — this may be prohibitively expensive.

Similarly to choosing a cooling schedule for simulated annealing, a tabu schedule must be chosen for tabu search. It is important to choose the length of the tabu list very carefully — this is again very problem-specific.

On the positive side, tabu search manages to examine many more “close-to-optimum” solutions than simulated annealing.

Another positive feature of tabu search is that provided care is taken to prevent cycling, the search can be left running for as long as resources allow, while the length of a simulated annealing run is usually fixed in advance.

Tabu search for graph domination

The best dominating sets for the football pool graphs were largely constructed by tabu search techniques, together with a mathematical construction that reduces the search to smaller but denser graphs.

There are many practical considerations in implementing a tabu search — firstly it is necessary to be very efficient in evaluating the cost function on the neighbouring states.

There are also many variants on a tabu search — for example, only searching a portion of the neighbourhood of a given state, maybe by concentrating on the moves that are likely to result in an improvement rather than all possible moves.
Genetic algorithms

Genetic algorithms provide an entirely different approach to the problems of combinatorial optimization.

Like simulated annealing, genetic algorithms try to model a physical process that improves “quality” — in this case the physical process is evolution.

A genetic algorithm proceeds by maintaining a pool containing many feasible solutions, each with its associated fitness.

At each iteration, a new population of solutions is created by breeding and mutation, with the fitter solutions being more likely to procreate.

A glimpse of GAs

Each solution is encoded as a string.

Breeding two strings involves selecting a position at random, breaking the strings into a head and tail at that point, and swapping tails. This operation is referred to as cross-over:

Parents are chosen in direct proportion to their fitness so that the fitter strings breed more often.

Mutation involves arbitrarily altering one of the elements of the string.

As usual there are several parameters to fine-tune the algorithm such as population size, mutation frequency and so on.

GAs for combinatorial optimization?

Although GAs have their adherents it may not be easy to adapt them successfully to combinatorial optimization problems such as TRAVELLING SALESMAN and GRAPH DOMINATION.

The problem here seems to be that there is no way one can arbitrarily combine two tours to create a third tour — simply hacking two tours apart and joining the bits together will not work in general.

Similarly, it is hard to come up with a good representation for a candidate dominating set in such a way that arbitrary cross-over does not destroy all its good properties.

The crucial distinction seems to be that hill-climbing, simulated annealing and tabu search are all local search methods whereas a genetic algorithm is not.

Recommended reading: CLRS, Chapter 35

Summary

1. Greedy algorithms solve optimization problems by searching the best local direction. They are applied in the Activity selection problem, Huffman coding and some graph algorithms.

2. Vertex cover, travelling salesman and the 0-1 knapsack problem are all instances of NP-complete problems, (i.e. for which no feasible algorithm is known).

3. A dynamic programming solution exists for the 0-1 knapsack problem.

4. Linear programmes are problems of optimizing a linear cost function, subject to linear constraints. They can be applied in many optimization problems, and may be solved by the simplex algorithm.
Summary cont.

5. Heuristic algorithms can be applied to approximate optimal solutions to geometric instances of the travelling salesman problem.

6. Other heuristic methods include hill-climbing, simulated annealing, tabu search and genetic algorithms.