INTRODUCTION

This week we cover the following subjects

12.1 NP COMPLETE PROBLEMS

- A problem is said to belong to the set *P* if it can be solved in Polynomial time or better. The inverse to set *P* is the set *NP*.

- There is an interesting class of problems which appear to belong to *NP* but no one has been able to prove it. These problems are called *NP-Complete* or *NP-Hard*.

- What has been proved is if a Polynomial solution can be found for any of the problems that are *NP*-Complete, then that algorithm can be applied to every other *NP*-Complete problem. This is known as Cook's Theorem.

- This fact, combined with the considerable amount of effort that has been applied to proving or disproving *NP*-Complete = *P*, has led to the general suspicion that *NP*-Complete ≠ *P*.

12.1.1 The Travelling Salesman Problem

- Probably the most well known *NP*-Complete problem is the Travelling Salesman Problem.

- The situation: A salesman starts on a tour of cities. He visits every city once and only once and ends up at the starting city.

- The problem: Find the route that takes up the shortest distance.

- While this problem is simple to state there are no known solutions that take less than factorial time. It doesn't take many cities before no computer could ever solve the problem. Even more tantalising, if a route is given it can be proven in polynomial time if it is the shortest route.

- There are many real world problems that boil down to being a variant of the Travelling Salesman problem. For example, a telecommunication company wanting to install telecommunication lines between cities, or a power company doing the same thing with power lines. Designing a new high speed CPU where you want to reduce the amount of wiring between transistors. In any of these cases, efficient resolving of these issues can result in a saving of millions of dollars, or produce sales worth billions of dollars for the new CPU.

12.2 COMPUTABILITY

- If there are problems that are *NP* it leads to the question of whether there are problems that are not computable at all. Alan Turing proved this true in 1936.

12.2.1 Turing Machines

- As part of his proof Turing devised a highly simplified model of a computer. It is interesting to note this is before the development of modern day computers. These days his model is known as a *Turing Machine*. Every computer in the world is a sophisticated version

of a Turing Machine and, while they may be far more efficient, there is nothing in principle a modern day computer can do that a Turing Machine cannot.

- Despite a considerable amount of research, no one has been able to devise a computational devise more powerful than a Turing Machine. It is interesting to speculate whether the human brain is a Turing Machine or not.

- A Turing Machine comprises the following five components.

  1. An *alphabet* $\sum$ of input letters
  2. A *tape* divided into a sequence of numbered cells, each containing one character or a blank. The input data is presented to the machine one letter per cell beginning at the left-most cell. The rest of the cells are initially filled with blanks.
  3. A *tape head* that can, in one step, read the contents of a cell on the tape, replace it with some other character, and reposition itself to the next cell to the right or left of the one it has just read. The tape head can print any characters from $\sum$ or erase a cell and leave it blank.
  4. A finite set of *states* including exactly one START state from which we begin execution (and which we may re-enter during execution) and some (maybe none) HALT states that cause execution to terminate when we enter them. The other states have no functions, only names such as 1, 2, 3, …

5. A *program*, which is a set of rules that tell us, on the basis of the letter read, how to change states, what to print and where to move the tape head.

- This model of a computer is capable of all the things a modern computer can do. This includes storing information on the tape, running loops and doing selection. The capacity for loops leads us to *the Halting Problem*.

12.2.2 The Halting Problem

- The halting Problem states

  Given a Turing Machine *T* and some arbitrary starting sequence *w* on the tape, is there an algorithm to decide, in a finite time, whether *T* halts when given the input *w*?

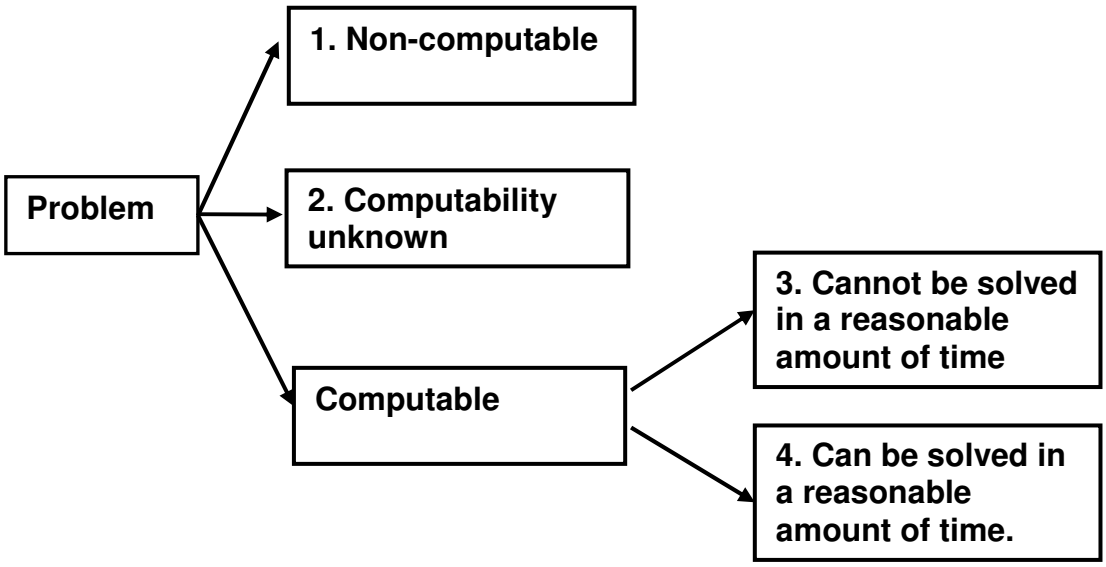
- We cannot just give *T* the input *w* and let it run, waiting for it to halt. It may run forever but we have to decide in a finite time if this is the case.

- Turing proved that there is no solution to this problem.

- Since then, many variations have been found of this problem, all unsolvable. Together they give a powerful idea of what is not possible with computers.

12.3 ALGORITHM ASSESSMENT

- When it comes to solving problems with computers we have 4 levels of possibility



- Big O notation allows us to gauge the relative merits of one algorithm to another.

12.4 OPTIMIZATION

- Since there are many real world problems that break down to being NP-Hard, not having a solution is a significant issue.

- If we are prepared to reduce our requirements to a solution that is very good, even if it isn't the absolute best, then there are many ways of getting such solutions in reasonable time. This is called *optimization* and there are many algorithms designed to find optimal solutions to difficult to solve problems

12.4.1 Greedy Algorithms

- Imagine you are standing on a large hilly area, it is night and you have only a candle to see by. You want to find the lowest area you can.

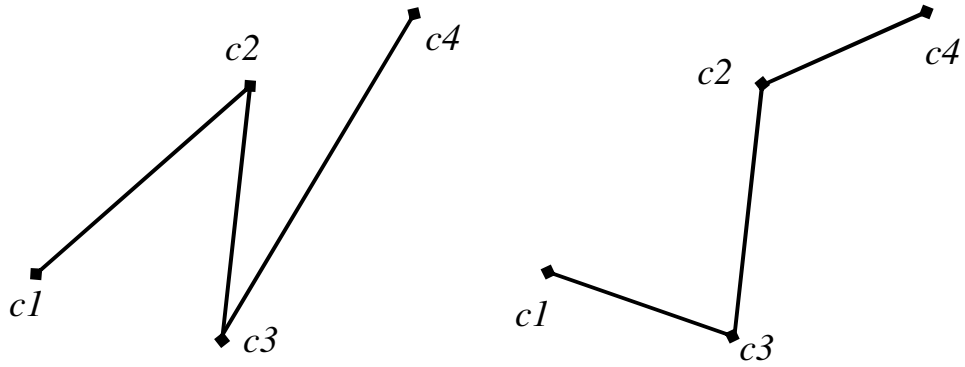
- What you do is examine the local area with your candle and determine which way the slope is going. You travel in the down direction and keep doing so until every direction where you are standing is up.

- You have employed what is called a *greedy* or *hill climbing* algorithm to find the lowest point. Many problems in optimization can be attacked using greedy algorithms.

- Because greedy algorithms always move in a favourable direction they are guaranteed to find a solution that is as good or better than the current one.

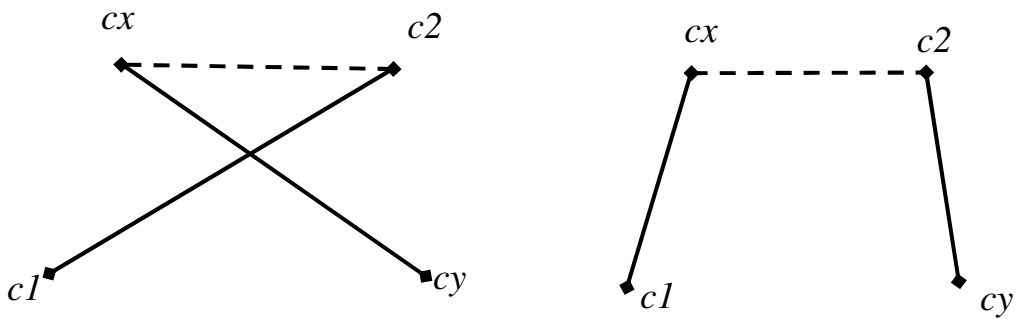
12.4.2 Travelling Salesman - Straightening Algorithm

- Consider the following situation



In the first route the salesman is going from cities *c1*→*c2*→*c3*→*c4*. In the second he is going from c1→*c3*→*c2*→c4. It is obvious that the second route is shorter.

- We can generalise this to the following situation



We have an entire section of cities, ranging from *c2* to *cx*, that are in the wrong order, thus lengthening the trip distance.

- The *straightening* algorithm consists of checking every consecutive group of 2 or more cities in the route to see if reversing the order will reduce the trip.

12.4.3 `travsale.cpp`

```
#include <iostream>
#include <vector>
#include <algorithm>

#include <math.h>

using namespace std;

const int NUMCITY = 1000;

/*************************************************\
    City struct
\*************************************************/
```

```
struct city
{
   double x, y;

   city() : x(0), y(0) {}
   city(double xx, double yy) : x(xx), y(yy) {}
   city(const city &c) : x(c.x), y(c.y) {}

   double dist(const city &c) const {
      double xdist = x - c.x;
      double ydist = y - c.y;
      return sqrt(xdist*xdist + ydist*ydist);
   }
};

/*********************************************\
    main program
\*********************************************/

void fillCities(vector<city> &cities);
double tripDist(const vector<city> &cities);
bool straightenTripDist(vector<city> &cities);

void fillCities(vector<city> &cities)
{
   // fill vector with cities
   // cities sit in box of dimension 1 each side

   srand(time(NULL));
   for (unsigned int i=0; i<cities.size(); i++) {
      cities[i].x = ((double) rand()) / RAND_MAX;
      cities[i].y = ((double) rand()) / RAND_MAX;
   }
}

double tripDist(const vector<city> &cities)
{
   double totalDist = 0;

   for (int i=0; i< NUMCITY; i++) {
      totalDist +=
         cities[i].dist(cities[(i+1) % NUMCITY]);
   }
   return totalDist;
```

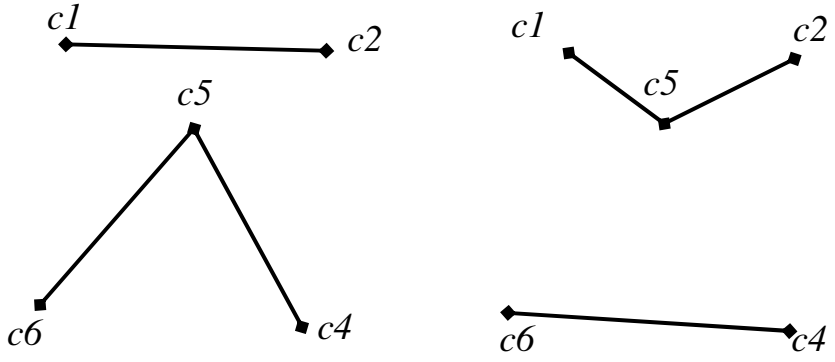
```
}

void reverseCities(vector<city> &cities,
                   int start, int end)
{
   // given a set of cities beginning at
   // start and finishing at end
   // reverse their order

   int gap, numcity = cities.size();

   // find the number of cites in the set
   gap = (end - start + numcity) % numcity + 1;

   // starting from each end,
   // move in and swap cities as we go
   while (gap >= 2) {
      swap(cities[start], cities[end]);
      start = (start + 1) % numcity;
      end = (end + numcity - 1) % numcity;
      gap -= 2;
   }
}

bool straightenTripDist(vector<city> &cities)
{
   // look for sets of cites in the entire set
   // that are reverse order
   // return true if we have re-ordered cities

   int swaps = 0;
   int i, j, x, m, n, numcity = cities.size();
   double dist1, dist2;

   for (i=0; i<numcity; i++) {
      // run through the cities
      // re-ordering to reduce trip distance
      j = (i+1) % numcity;

      for (x=1; x<numcity; x++) {
         m = (i+x) % numcity;
         n = (i+x+1) % numcity;
```

```
            // city i->j-> .. ->m->n
            dist1 = cities[i].dist(cities[j]) +
                    cities[m].dist(cities[n]);
            // city i->m-> .. ->j->n
            dist2 = cities[i].dist(cities[m]) +
                    cities[j].dist(cities[n]);

            if (dist2 < dist1) {
               reverseCities(cities, j, m);
               swaps++;
               break;
            }
         }
      }
      return (swaps > 0);
}

int main()
{
   vector<city> cities(NUMCITY);
   double dist1, dist2, percent;

   fillCities(cities);
   cout << cities.size() << " cities\n";
   dist1 = tripDist(cities);
   cout << "Starting trip distance = " <<
      dist1 << "\n";
   while (straightenTripDist(cities)) {
      dist2 = tripDist(cities);
      cout << "Reduced trip distance  = " <<
         dist2 << "\n";
   }
   percent = fabs(dist1 - dist2) / dist1 * 100;
   cout << "Trip reduced by " <<
           percent << "%%\n";

   return 0;
}
```

```
1000 cities
Starting trip distance = 517.754
Reduced trip distance  = 363.038
Reduced trip distance  = 256.522
Reduced trip distance  = 178.783
Reduced trip distance  = 122.591
Reduced trip distance  = 84.1273
Reduced trip distance  = 59.0545
Reduced trip distance  = 45.9546
Reduced trip distance  = 35.8859
Reduced trip distance  = 33.1093
Reduced trip distance  = 30.6161
Reduced trip distance  = 27.6552
Reduced trip distance  = 27.1043
Reduced trip distance  = 26.7275
Reduced trip distance  = 26.6342
Reduced trip distance  = 26.0083
Reduced trip distance  = 25.9135
Reduced trip distance  = 25.9006
Reduced trip distance  = 25.8372
Reduced trip distance  = 25.8317
Trip reduced by 95.0108%%
```

12.4.4 Travelling Salesman - Repositioning Algorithm

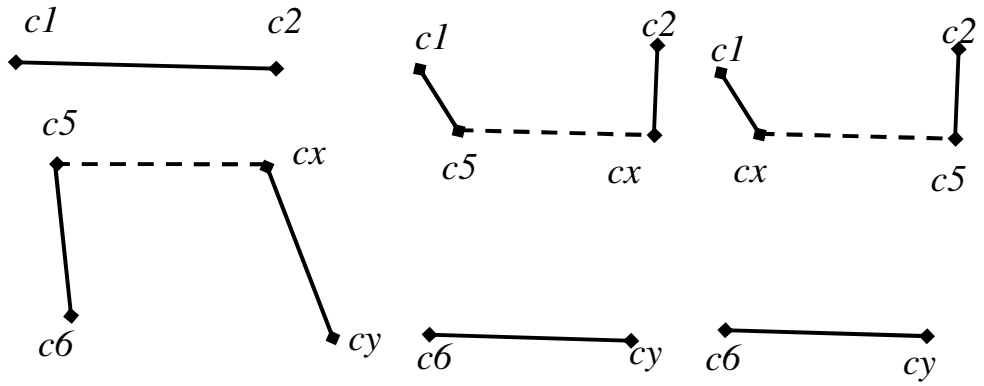
- Consider the following situation while trying to reduce the trip length along the route of cities.



In the first situation the salesman goes from *c1→c2* and later on *c4→c5→c6*. In the second situation he

goes from *c1→c5→c2* and later *c4→c6*. It is obvious the second situation is better so we can move *c5* and insert it between *c1→c26* to shorten the route.

- This can be generalised to the following



We looks for groups of 1 or more cities that fit better in a different location. Note that we may have to reverse the order of the cities being moved in order to get a better fit.

12.4.5 `RepositionTripDist`

```
void moveSection(vector<city> &cities,
                 int i, int j, int a, int b)
{
   int s, w, swaps, section;
   int numcity = cities.size();

   // move section a-b along enough
   // jumps to reposition it between i-j

   swaps = (i - a + numcity) % numcity;
   section = (b-a + numcity) % numcity + 1;
```

```
   for (s=0; s<section; s++)  {
      for (w=0; w<swaps; w++) {
         swap(cities[(a+w) % numcity],
              cities[(a+w+1) % numcity]);
      }
   }
}

bool repositionTripDist(vector<city> &cities)
{
   int i, j, a, b, c, d, count,
   int numcity = cities.size();
   // limit of the size of section we check for

   int limit = (int) log(numcity);
   double dist1, dist2;
   bool moved;
   int swaps = 0;

   for (i=0; i<numcity; i++) {
      // start off with two adjacent cities i-j
      j = (i+1) % numcity;
      moved = false;
      count = 0;

      for (a=0; a<numcity && !moved;
           a++, count++) {
         // a-b => c-d
         // b and c start off equal,
         // the section is one city
         c = b = (a+1) % numcity;
         d = (a+2) % numcity;
         count = 0;

         while (c != i && c != j &&
                count < limit) {
            // look at and find the shorter dist
            // i-j    a-b => c-d
            // i-b => c-j    a-d
            // i-c => b-j    a-d
             dist1 = cities[i].dist(cities[j]) +
                     cities[a].dist(cities[b]) +
                     cities[d].dist(cities[c]);
```

```
            dist2 = cities[i].dist(cities[b]) +
                    cities[j].dist(cities[c]) +
                    cities[a].dist(cities[d]);

            if (dist2 > dist1) {
               // check if a reversed section
               // reduces trip
               dist2 = cities[i].dist(cities[c])
                     + cities[j].dist(cities[b])
                     + cities[a].dist(cities[d]);

               if (dist2 < dist1)
                  reverseCities(cities, b, c);
            }

            if (dist2 < dist1) {
               moveSection(cities, i, j, b, c);
               swaps++;
               moved = true;
               break;
            }

            count++;
            c = d;
            d = (d+1) % numcity;
         }
      }
   }

   return swaps > 0;
}
```

- Running the program with the repositioning algorithm instead of the straightening algorithm gives the following output

```
1000 cities
Starting trip distance = 510.445
Reduced trip distance  = 325.794
Reduced trip distance  = 207.478
Reduced trip distance  = 133.686
Reduced trip distance  = 87.7981
```

```
Reduced trip distance  = 55.7713
Reduced trip distance  = 38.3008
Reduced trip distance  = 31.42
Reduced trip distance  = 28.514
Reduced trip distance  = 27.4497
Reduced trip distance  = 26.5831
Reduced trip distance  = 26.3014
Reduced trip distance  = 26.1574
Reduced trip distance  = 26.1135
Reduced trip distance  = 26.0658
Reduced trip distance  = 26.0611
Trip reduced by 94.8944%%
```

The algorithm gives similar results as the straightening algorithm.

- We can combine the two algorithms in the following way

```
int main()
{
   vector<city> cities(NUMCITY);
   double dist1, dist2, percent;
   int changed;

   fillCities(cities);
   cout << cities.size() << " cities\n";
   dist1 = tripDist(cities);
   cout << "Starting trip distance = "
            << dist1 << "\n";
   do {
      changed = 0;
      while (straightenTripDist(cities))
          changed++;
      dist2 = tripDist(cities);
      cout << "Straightened trip distance  = "
           << dist2 << "\n";
     if (changed > 0)
     {
         while (repositionTripDist(cities))
            changed++;
```

```
        }
        dist2 = tripDist(cities);
        cout << "Repositioned trip distance = "
             << dist2 << "\n";
    } while (changed > 0);
    percent = fabs(dist1 - dist2) / dist1 * 100;
    cout << "Trip reduced by " <<
            percent << "%%\n";

    return 0;
}
```

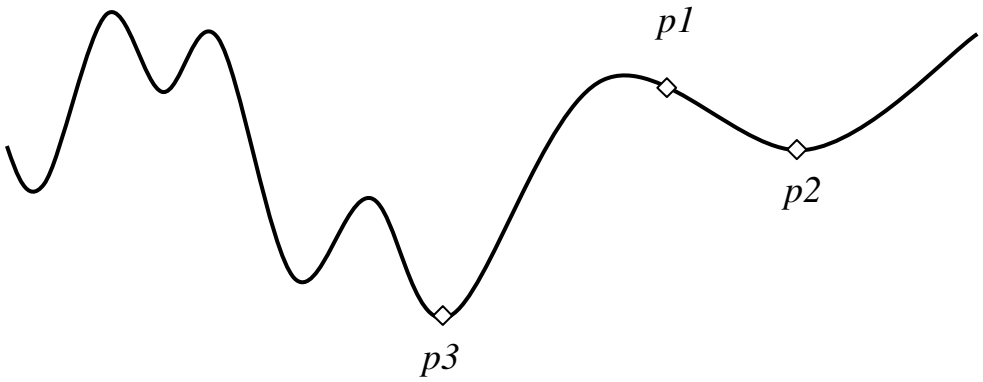
with the following results

```
1000 cities
Starting trip distance = 514.155
Straightened trip distance  = 25.2562
Repositioned trip distance  = 23.9757
Straightened trip distance  = 23.9757
Repositioned trip distance  = 23.9757
Trip reduced by 95.3369%
```

- The combined algorithms give an improved result.

12.4.6 Stochastic Algorithms

- While greedy algorithms will find a minima it cannot guarantee it will find the *global minimum*, just the *local minimum*. The following diagram illustrates on a 2 dimensional plane. If we start at point *p1* then, moving downward in the direction of the local slope, we will end up at point p2. However, the lowest point is *p3*.



- It's useful at this point to see if we can make an estimate of what an optimal solution to the travelling salesman problem would be.
  - If we are filling a square box of size *s* with *n* cities, then the average distance between cities will be approximately $s/\sqrt{n}$.
  - With *n* cities that would make a route distance of $ns/\sqrt{n} = s\sqrt{n}$. If we set *s* = 1 then this becomes $\sqrt{n}$.
  - However, given we are trying to find a minimum route, many of the trips between cities will be less than the average distance. Generally, the optimal route will be approximately $\frac{2}{3}\sqrt{n}$ given *s* = 1.

- Therefore, given 1000 cities and *s* = 1 we would expect an optimal route to be approximately 22.2 in length. Given that we have a current minimum length of 23.98 we would appear to be in a local minima that is fairly close to optimal.

- *Stochastic* algorithms use randomising processes to help bounce us out of local minima in order to find a better local minima.

- There are many types of stochastic algorithms designed to find optimal solutions in difficult to solve problems. Examples would be
  - Simulated Annealing
  - Genetic Algorithms
  - Neural Networks

12.4.7 Conclusions

- Optimization is an important area in computing science in which there are many important real world problems that cannot be solved explicitly in a reasonable amount of time.

- However, many optimization algorithms have been developed that give very good solutions in reasonable amounts of time.

- The previous examples applied to the Travelling Salesman problem give a very introductory idea to what is possible.