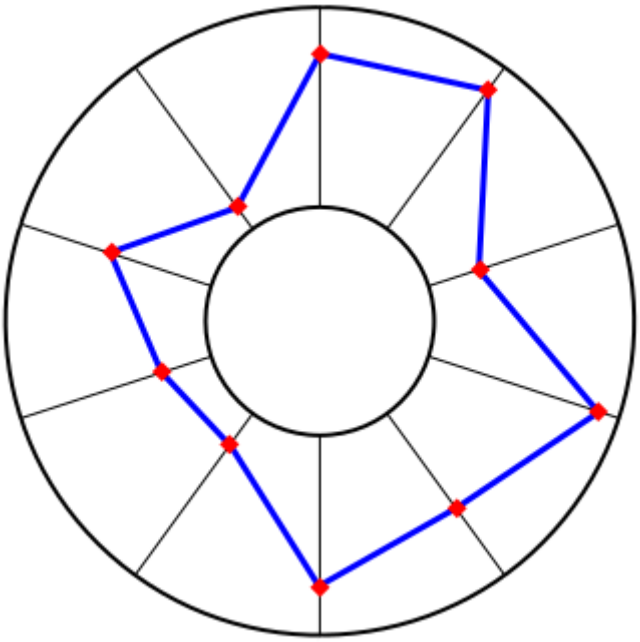
# Investigating the use of a genetic algorithm to obtain numerical solutions to the *Moving Sofa Problem*

Brian Tyrrell

*July 2015*



# Contents


**1 Abstract** **3**

**2 Introduction** **4**
2.1 Problem Overview . . . . . 4
2.2 Process & Terminology . . . . . 4

**3 The Unconstrained Genetic Algorithm** **7**
3.1 Making the initial population . . . . . 7
3.2 Selection . . . . . 10
3.2.1 Roulette wheel selection . . . . . 10
3.2.2 Tournament selection . . . . . 12
3.2.3 Chi Squared selection . . . . . 14
3.3 Crossover . . . . . 16
3.3.1 Two point crossover . . . . . 16
3.3.2 One point crossover . . . . . 20
3.3.3 Uniform crossover . . . . . 20
3.3.4 Cut-and-splice crossover . . . . . 20
3.3.5 Three parent crossover . . . . . 20
3.4 Mutation . . . . . 21
3.4.1 Reshaping Mutation . . . . . 21
3.4.2 Addition mutation . . . . . 23
3.4.3 Shift mutation . . . . . 25
3.4.4 Comparison of 3 mutation operators . . . . . 27

**4 The Path Finding Algorithm** **28**
4.1 Building a path in $S$ using a random walk . . . . . 28
4.1.1 Routines & functions for checking if a sofa is inside the hallway 30
4.1.2 Routines & functions to make a random walk & build the path 31
4.1.3 Testing the code . . . . . 37
4.2 Brute force method . . . . . 39
4.2.1 Calculation of points in $S$ . . . . . 39
4.2.2 Dead-end filling . . . . . 41
4.2.3 Wall following . . . . . 45

**5 The Constrained Genetic Algorithm** **47**
5.1 A change to the way area is calculated . . . . . 48
5.2 The nullifying constraint . . . . . 51
5.3 The DAM constraint . . . . . 52
5.4 Seeding the initial population . . . . . 54
5.4.1 Changes made to original code from Section 3 . . . . . 54
5.4.2 Seeding the population with the Circle sofa . . . . . 55
5.4.3 Seeding the population with the Hammersley sofa . . . . . 57

**6 Problems encountered & future modifications** **58**

**7 Thanks & Acknowledgement** **60**

**8 References** **61**

**9 Select code used in project** **62**
9.1 The Unconstrained Genetic Algorithm . . . . . 62
9.2 The Path Finding Algorithm . . . . . 72
9.3 The Constrained Genetic Algorithm . . . . . 81

# 1 Abstract

My objective in researching this topic was to:
a) investigate the terms used in evolutionary programming
b) using this knowledge, construct a genetic algorithm and
c) apply this algorithm to the *Moving Sofa Problem* to obtain results in agreement with the published literature.

In order to do this, I split the project into two parts - the first part being a program to generate sofa shapes, and the second part being a program to determine if a given sofa fits around the hallway. I wrote several algorithms in C++ for both parts of the project and determined the two which worked best; the first was a genetic algorithm used to 'breed' sofa shapes, and the second a path finding algorithm which used random walks to navigate a sofa around the hallway. To check all the codes were functioning correctly I analysed their outputs at various stages, compared and contrasted the results obtained from each algorithm to find a program best suited to this problem.

My final result (given in Section 5.4.3) approximates the current lower bound sofa (the *Gerver sofa*) though more research and time is needed to obtain more accurate results.

# 2 Introduction

## 2.1 Problem Overview

The *Moving Sofa Problem* first appeared in the *SIAM Review*, Vol. 8, No. 3 (Jul. 1966) and was posited by Leo Moser;

**What is the largest area region which can be moved through a 'hallway' of width one?**

Where we take the region to be rigid and the hallway, now referred to as the *Moser Hallway*, is an L shaped hallway of constant width one. In 1970 James Sebastian put forward an argument[5] that indicated the upper bound on the maximum area of a sofa is $2\sqrt{2}$ (this area corresponds to the *Sebastian Sofa*) whilst Joeseph Gerver in 1992 proved two theorems[3]; the first gives a "strong condition which must hold for at least one region of maximal area" and the second gives a region which satisfies this condition. The *Gerver Sofa* has an area $\approx 2.2195$ and is bounded by 18 analytic pieces.

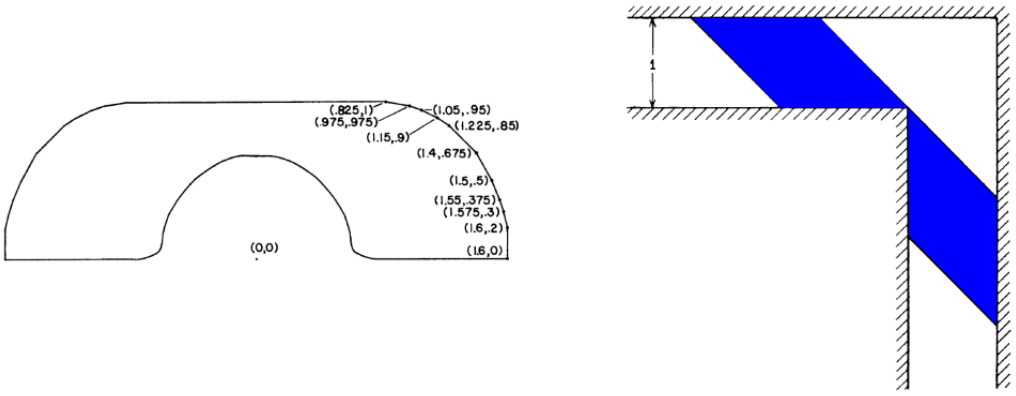



Figure 1: *The Gerver sofa (LHS)[6] and the Sebastian sofa (RHS)[4]*

The area of the Gerver Sofa is the known lower bound, the area of the Sebastian Sofa the known upper bound, and the 'Moving Sofa Problem' is these two figures don't agree.

## 2.2 Process & Terminology

*Genetic Algorithms* are a type of evolutionary algorithm, which, as their name suggests, imitate evolutionary biology in order to heuristically search a solution space for an optimum. Evolution is driven by natural selection - a process whereby organisms better adapted to their environment tend to survive and produce more offspring - and this process is key to how a genetic algorithm generates solutions.

I will be using a genetic algorithm to explore the solution space of all sofas which fit around the Moser Hallway, denoted by $X$. The terminology is as follows;

1. **Population/farm:** a subset of $X$.

2. **Individual/animal:** an element of the population.

3. **Organism string:** how an individual is represented using its alleles.

4. **Genes/Alleles:** each individual is composed of genes or alleles which uniquely describe the individual in the solution space using the organism string.

5. **Allelic value:** the value of a gene in an organism string (usually either 0 or 1) is known as the allelic value of the gene.

6. **Parent:** one of two sofas which contribute to form a child.

7. **Child:** a sofa produced by two parents which, as a collective, form the current generation.

8. **Allowed sofa:** a sofa that fits around the hallway.

9. **Disallowed sofa:** a sofa that doesn't fit inside the hallway or cannot be maneuvered around the hallway.

For this problem, I used a grid[1] with 1,000,000 evenly spaced points on the square $\{(x, y) : -2 \leq x, y \leq 2\}$, numbered the points 0 to 999,999 (with 0 being the bottom left, 999,999 being the top right) and created a 1,000,000 bit long organism string using the rule:
0 - there is not a corner in this position
1 - there is a corner in this position.

A genetic algorithm uses 3 main operators grouped under the term Reproduction; *Selection, Crossover,* and *Mutation.*

**1. Selection**
Each member of the population is evaluated by a fitness function and assigned a value (here, the area of the sofa). A new population is selected from the old population where the probability of being selected is directly proportional to the individuals fitness. In particular, an individual can be selected more than once for the breeding population and I've had to take precautions to prevent both a sofa breeding with itself and a sofa being too common in the breeding population[2].

**2. Crossover**
The individuals are randomly paired and each pair is recombined in a particular fashion - for *one-point crossover*, on each parent's organism string a mutual crossover point is randomly selected. Two children are generated from the parents by swapping the alleles of both parents after the crossover point. This method has variations such as *two-point crossover*, *uniform crossover* and the *cut-and-splice*

[1] From here on, this is referred to as *the grid*
[2] See Sections 3.2 & 7

*method* which are explained in more detail later[3]. Also, some research[2] in this area suggests that using more than two parents to generate children can lead to "higher quality alleles".

**3. Mutation**

The mutation operator acts on an organism string to switch the value of one (or more) of it's alleles; the biological equivalent being certain genes being switched on and off. I've tested (amongst others)[4] a *bit-string mutation* which randomly selects an allele on an organism string and inverts it.

I found the mutation operator the trickiest of the three to perfect - too low of a mutation rate leads to *genetic drift* and problems converging on a solution[7]. Too high of a mutation rate may lead to erratic changes from generation to generation and loss of good solutions. However a suitable mutation operator can be used to escape local optima to reach the global optimum.

[3]See Section 3.3

[4]See Section 3.4

# 3 The Unconstrained Genetic Algorithm

Using C++ I created a class *Sofa* which contains all the information required for each sofa shape:

```
typedef class Sofa
{
    public:
        int c;
        vector <int> orgstring;
        double corn[1000][2], in[1000][2];
        double area;

        void modc(int);
        void modarea(double);
        void modorgstring(int);
        void modcorn(int, double, double);
        void modin(int, double, double);
        void clearorgstring();
        void eraseorgstring(int);
        void sortorgstring();

        int outnumc();
        double outarea();
        int outorgstring(int);
        double outcorn_x(int);
        double outcorn_y(int);
        double outin_x(int);
        double outin_y(int);
} Sofa;
```

Where;

- $c$ is the number of corners (of a sofa)
- *orgstring* is the organism string
- the arrays *corn* and *initial* hold the $x$ and $y$ coordinates of the corners
- *area* is the area

## 3.1 Making the initial population

The initial population consists of 50 simple polygons of 10 sides contained in a circle of radius 0.5. They are generated by removing an inner circle of radius $r \in (0, 0.25)$, splitting the remaining annulus using 10 equally spaced lines, randomly choosing a point on each line and finally joining all the points together *(see figure 2):*

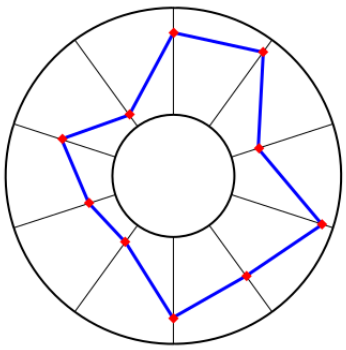


Figure 2: *An example of how a sofa shape is initially generated*

```
void makepop(Sofa a[50])
{
    double pos, r;
    for(int i = 0; i < 50; ++i)
    {
        a[i].modc(10);
        r = 0.25*ran();
        while(r == 0)
        {   r = 0.25*ran();
        }

        for(int j = 0; j < 10; ++j)
        {
            pos = ((0.5 - r)*ran() + r);
            a[i].modcorn(j, search(pos*cos((double) (2*pi/10)*j)),
                search(pos*sin((double) (2*pi/10)*j)));
        }
    }
}
```

This uses the function *search* which finds the nearest point on *the grid* and chooses that point as the corner:

```
double point[1000];

double search(double a)
{
    double min = 10;
    int cnt;
    for(int i = 0; i < 1000; ++i)
    {
        if(abs(a - point[i]) < min)
        {
            cnt = i;
            min = abs(a - point[i]);
        }
    }
```

```
    return point[cnt];
}
```

Once I have a collection of corners I use the routine *makecorners* to sort them anticlockwise (which is needed for computing the area using the *Shoelace algorithm*):

```
void makecorners(Sofa& p)
{
    int p1, q1 = 0;
    double cornx1[p.outnumc()], corny1[p.outnumc()];
    double angle1 = 0;
    vector < pair < double, int > > vec1;

    for(int j1 = 0; j1 < p.outnumc(); ++j1)
    {
        cornx1[j1] = p.outcorn_x(j1);
        corny1[j1] = p.outcorn_y(j1);
        angle1 = atan(corny1[j1]/cornx1[j1]);

        if(cornx1[j1] < 0 && corny1[j1] > 0)
        {   angle1 += pi;
        }

        if(cornx1[j1] < 0 && corny1[j1] <= 0)
        {   angle1 += pi;
        }

        if(cornx1[j1] >= 0 && corny1[j1] < 0)
        {   angle1 += 2*pi;
        }

        vec1.push_back(make_pair(angle1, j1));
    }

    sort(vec1.begin(), vec1.end());

    for(vector < pair < double, int > > :: iterator it1 = vec1.begin();
        it1 != vec1.end(); ++it1)
    {
        p1 = (*it1).second;
        p.modcorn(q1, cornx1[p1], corny1[p1]);
        ++q1;
    }

    double area = 0;
    for(int n = 0; n < p.outnumc()-1; ++n)
    {
        area += abs(p.outcorn_x(n)*p.outcorn_y(n+1)
            - p.outcorn_x(n+1)*p.outcorn_y(n));
    }
```

```
    p.modarea(0.5*area);
}
```

Finally, I use the corner information (which doesn't need to be sorted for this step, only for the area calculation) to form the organism string. For the organism sting, I need to keep track of the *position* of *only* the 1's, which is what the routine *makeorgstring* does:

```
void makeorgstring(Sofa& p)
{
    int temp;
    for(int k = -500; k < 500; ++k)
    {
        for(int l = -500; l < 500; ++l)
        {
            for(int m = 0; m < p.outnumc(); ++m)
            {
                if(p.outcorn_x(m) == (double) 4*k/1000 &&
                    p.outcorn_y(m) == (double) 4*l/1000)
                {
                    p.modorgstring(1000*(l+500)+(k+500));
                    break;
                }
            }
        }
    }
}
```

## 3.2 Selection

I want to select a breeding population where the probability of an individual being selected for breeding is directly proportional to that individuals fitness. For this project I considered three common methods of selecting a breeding population:

1. *Roulette wheel* selection
2. *Tournament* selection
3. *Chi Squared* selection

### 3.2.1 Roulette wheel selection

I sort the population in descending order according to their area, and define a vector of pairs *sel*, where the $i^{th}$ element is

$$\textit{(sum of the first i areas, } i^{th} \textit{ sofa).}$$

I randomly select a double $con \in (0,$ total area) and find which sofa *con* corresponds to. This sofa becomes part of my breeding population (snew) and using *concounter* I store which sofa has been selected for statistics later. The C++ code is:

```
int con2;
int concounter[5000];
double sum = 0;
vector < pair < double, int > > areas;

for(int i = 0; i < 5000; ++i)
{   areas.push_back(make_pair(s[i].outarea(), i));
}

sort(areas.rbegin(), areas.rend());

vector < pair < double, int > > sel;
for(vector < pair < double, int > > :: iterator it = areas.begin();
    it != areas.end(); ++it)
{
    sum += (*it).first;
    sel.push_back(make_pair(sum, (*it).second));
}

for(int p = 0; p < 5000; ++p)
{
    double con = sel.back().first*ran();
    con2 = 0;

    for(vector < pair < double, int > > :: iterator itt = sel.begin();
        itt != sel.end(); ++itt)
    {
        if(con < (*itt).first)
        {
            con2 = (*itt).second;
            break;
        }
        else
        {   con2 = sel.back().second;
        }
    }

    concounter[p] = con2;
    equatesofas(s[con2], snew[p]);
    makecorners(snew[p]);
}
```

To test the code I have a population of 1000 sofas, running 5 times to ensure adequate data spread *(figure 3)*. There is a 'hump' at about 0.4 to indicate the sofas of this area are chosen most often, however the data doesn't seem very spread out. This is because the small differences in area mean the selection isn't as well defined as it would be if there were large differences in area - if all the areas are very similar to each other, the selection will be less precise.

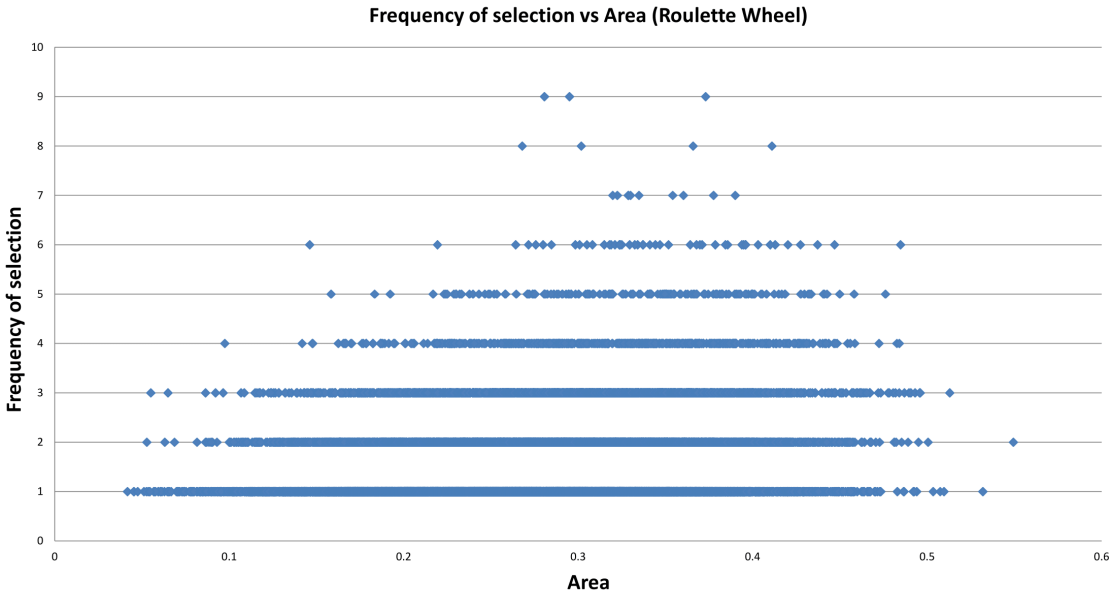



Figure 3: *Roulette Wheel selection*

### 3.2.2 Tournament selection

For tournament selection, I randomly choose 100 sofas from the initial population of 1000, and the fittest sofa from those 100 is added to the breeding population. I found this method to be almost too successful; the largest sofas would appear too often[5] in the breeding population, leading to problems like inbreeding[6] and slow convergence on a solution. To fix this, I used an array *choices* to keep track on how often a sofa was used in the breeding population, and disallowed any sofa appearing more than 10 times:

```
vector < pair < double, int > > sel;
int choices[50] = {0};
int choose1, choose2;
for(int j = 0; j < 1000; ++j)
{
    bool finishedsel = false;
    while(!finishedsel)
    {
        for(int i = 0; i < 100; ++i)
        {
            choose1 = 1000*ran();
            sel.push_back(make_pair(s[choose1].outarea(), choose1));
        }

        sort(sel.begin(), sel.end());
        choose2 = sel.back().second;
```

[5]For example, from a population of 50, 7 sofas occupied 31 spaces in the breeding population
[6]See Section 7

```
            if(choices[choose2] < 10)
            {
                ++choices[choose2];
                finishedsel = true;
                sel.clear();
            }
            sel.clear();
        }

        equatesofas(s[choose2], snew[j]);
        makecorners(snew[j]);
}
```

To test the code I have a population of 1000 sofas, running 5 times to ensure adequate data spread *(figure 4)*. It can be seen that the sofas with the largest areas are selected the most often and the sofas with smaller area are selected least often.

However some sofas of area $\approx 0.37$ have been selected 10 times - this seems to be because of the cap of 10; once the largest sofas have been selected 10 times they are effectively removed from the initial population, meaning some smaller sofas get relatively larger and will thus be selected more often.

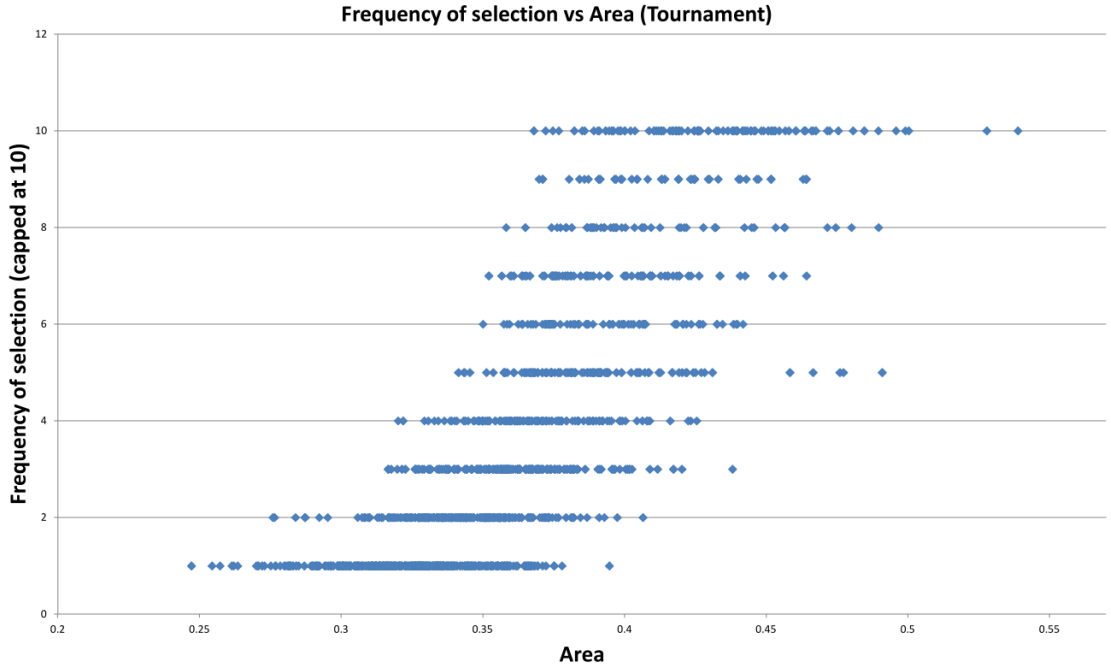



Figure 4: *Tournament selection*

### 3.2.3 Chi Squared selection

For this method of selection I used a Chi-Squared distribution[7] with $k = 4$; the sofas are sorted descendingly into a vector according to their area and an integer *con* is selected using the chi squared distribution built into the header <random>. The $con^{th}$ element in the vector is then selected to be part of the breeding population *snew*:

```
int con2, sel;
double sum = 0;
vector < pair < double, int > > areas;

for(int i = 0; i < 100; ++i)
{   areas.push_back(make_pair(s[i].outarea(), i));
}

sort(areas.rbegin(), areas.rend());

default_random_engine generator;
chi_squared_distribution <double> distribution(4.0);

for(int p = 0; p < 100; ++p)
{
    con2 = (int) distribution(generator);
    while(con2 < 0 || con2 > 99)
    {   con2 = (int) distribution(generator);
    }

    sel = areas[con2].second;
    equatesofas(s[sel], snew[p]);
}
```

To test the code I have a population of 1000 sofas, running 5 times to ensure adequate data spread *(figure 5)*. Again it can be seen that the smaller sofas (still quite large at area 0.43) are selected least often, and there is a definite peak in area selection at 0.49. The data is again more cluttered, but this is because of two reasons:

1. The difference in area between the smallest and largest sofas selected is small
2. Using this selection I get quite a small number of independent sofas in the breeding population[8]. With *Tournament selection* I could cap the number of times a sofa was selected, but I didn't do that here. A future modification to make to my code would then be some sort of limit on the number of times a sofa could be selected for breeding.

[7] *See figure 6 - image credit: Wikipedia, Chi-squared distribution*

[8] For example, in a population of 1000 I could get 10 independent sofas in the breeding population

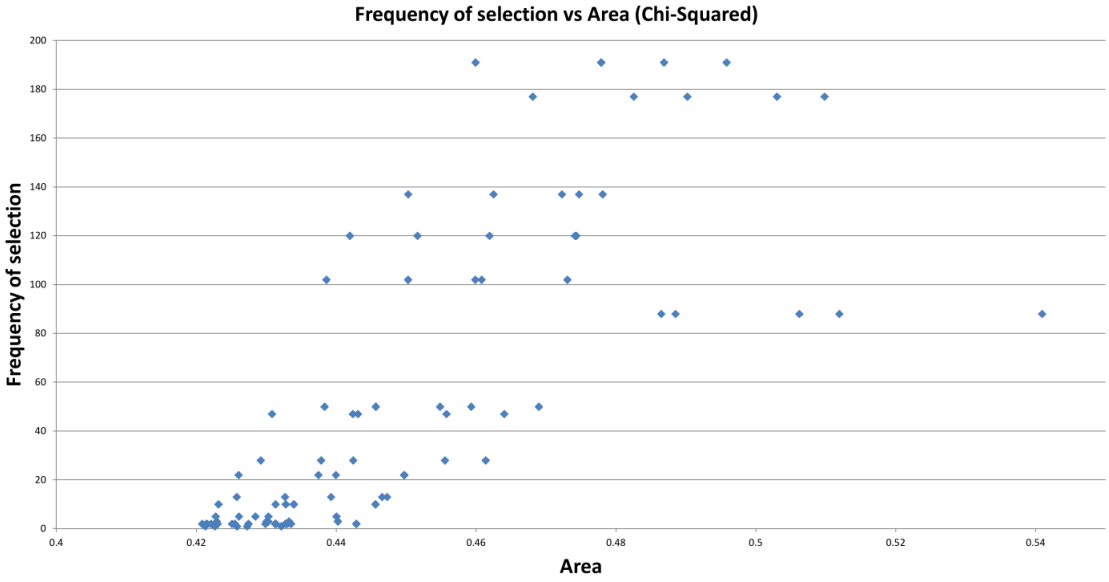



Figure 5: *Chi-squared selection*

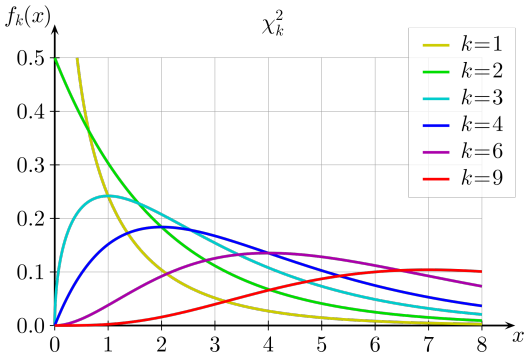



Figure 6: *Chi-squared distribution*

## 3.3 Crossover

### 3.3.1 Two point crossover

Two parents are randomly selected (and chosen that they're not *literally* the same sofa[9]) The organism strings of the two children are created like *figure 7*:

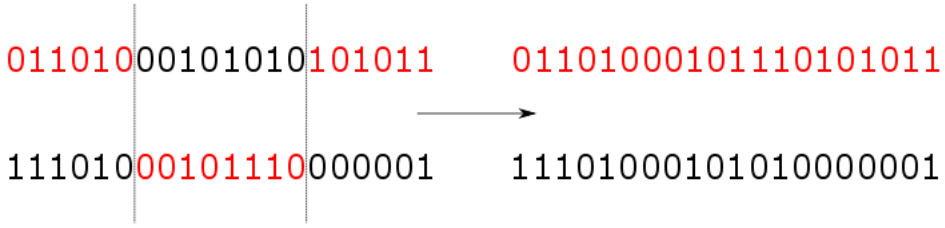



Figure 7: *Two point crossover*

The C++ code[10] is as follows:

```
int aa = 0;
while(aa < 100)
{
    int cho1 = 100*ran();
    int cho2 = 100*ran();

    while(cho1 == cho2)
    {   cho2 = 100*ran();
    }

    snew[cho1].sortorgstring();
    snew[cho2].sortorgstring();

    int cross1 = 1000000*ran();
    while(cross1 == 0 || cross1 > 999980)
    {   cross1 = 1000000*ran();
    }

    int cross2 = (1000000-cross1)*ran() + cross1;
    while(cross2 == cross1 || cross2 > 999995)
    {   cross2 = (1000000-cross1)*ran() + cross1;
    }

    int count1 = 0, count2 = 0;

    for(int i = 0; i < snew[cho1].outnumc(); ++i)
    {
        if(snew[cho1].outorgstring(i) <= cross1)
```

[9] A variation on this would be to ensure two sofas with identical organism strings are not chosen, i.e. the two parents being genetically the same

[10] For a population of 100

```
        {
            schild[aa].modorgstring(snew[cho1].outorgstring(i));
            ++count1;
        }
    }

    for(int j = 0; j < snew[cho2].outnumc(); ++j)
    {
        if(snew[cho2].outorgstring(j) > cross1 &&
            snew[cho2].outorgstring(j) <= cross2)
        {
            schild[aa].modorgstring(snew[cho2].outorgstring(j));
            ++count1;
        }
    }

    for(int k = 0; k < snew[cho1].outnumc(); ++k)
    {
        if(snew[cho1].outorgstring(k) > cross2)
        {
            schild[aa].modorgstring(snew[cho1].outorgstring(k));
            ++count1;
        }
    }

    for(int ii = 0; ii < snew[cho2].outnumc(); ++ii)
    {
        if(snew[cho2].outorgstring(ii) <= cross1)
        {
            schild[aa+1].modorgstring(snew[cho2].outorgstring(ii));
            ++count2;
        }
    }

    for(int jj = 0; jj < snew[cho1].outnumc(); ++jj)
    {
        if(snew[cho1].outorgstring(jj) > cross1 &&
            snew[cho1].outorgstring(jj) <= cross2)
        {
            schild[aa+1].modorgstring(snew[cho1].outorgstring(jj));
            ++count2;
        }
    }

    for(int kk = 0; kk < snew[cho2].outnumc(); ++kk)
    {
        if(snew[cho2].outorgstring(kk) > cross2)
        {
            schild[aa+1].modorgstring(snew[cho2].outorgstring(kk));
            ++count2;
        }
    }

    schild[aa].modc(count1);
```

```
    schild[aa+1].modc(count2);

    schild[aa].sortorgstring();
    schild[aa+1].sortorgstring();

    makecorners(schild[aa]);
    makecorners(schild[aa+1]);

    aa = aa+2;
}
```

See Section 3.1 for more information on the routine *makecorners*.

To test this code, I used a population of 10 sofas, and compared the children with the parents after crossover had taken place (and selected three examples for display in *figure 8*).

There are many reasons that a child would have only one parent:

1. In the breeding population, there are usually copies of the same sofa. It's possible for two different sofas with identical organism strings to breed and produce a child which *appears* to have one parent.

2. Since there are only 10 corners, in the organism string there are 10 1's and 999990 0's - it's possible for the crossover points to miss the corner alleles for one sofa; thus only one parent contributes genetic information, whereas there are in fact two parents.

| CHILDREN | | | PARENT(S) OF CHILDREN | | | |
|---|---|---|---|---|---|---|
| | | | | | | |
| **0.432** | **0** | | **0.432** | **0** | | |
| **0.356** | **0.256** | | **0.356** | **0.256** | | |
| **0.108** | **0.332** | | **0.108** | **0.332** | | |
| **-0.136** | **0.42** | | **-0.136** | **0.42** | | |
| **-0.356** | **0.26** | | **-0.356** | **0.26** | | |
| **-0.348** | **0** | | **-0.348** | **0** | | |
| **-0.34** | **-0.248** | | **-0.34** | **-0.248** | | |
| **-0.12** | **-0.368** | | **-0.12** | **-0.368** | | |
| **0.128** | **-0.396** | | **0.128** | **-0.396** | | |
| **0.232** | **-0.168** | | **0.232** | **-0.168** | | |
| | | | | | | |
| **0.388** | **0** | | **0.388** | **0** | 0.488 | 0 |
| **0.132** | **0.404** | | 0.392 | 0.284 | 0.124 | 0.092 |
| **-0.144** | **0.444** | | 0.112 | 0.348 | **0.132** | **0.404** |
| **-0.256** | **0.184** | | -0.096 | 0.292 | **-0.144** | **0.444** |
| **-0.34** | **0** | | **-0.256** | **0.184** | -0.32 | 0.232 |
| **-0.392** | **-0.284** | | **-0.34** | **0** | -0.428 | 0 |
| **-0.096** | **-0.3** | | **-0.392** | **-0.284** | -0.36 | -0.26 |
| **0.096** | **-0.296** | | **-0.096** | **-0.3** | -0.116 | -0.356 |
| **0.276** | **-0.2** | | **0.096** | **-0.296** | 0.152 | -0.464 |
| | | | **0.276** | **-0.2** | 0.084 | -0.06 |
| | | | | | | |
| **0.488** | **0** | | **0.488** | **0** | 0.388 | 0 |
| **0.392** | **0.284** | | **0.124** | **0.092** | **0.392** | **0.284** |
| **0.124** | **0.092** | | 0.132 | 0.404 | **0.112** | **0.348** |
| **0.112** | **0.348** | | -0.144 | 0.444 | **-0.096** | **0.292** |
| **-0.096** | **0.292** | | **-0.32** | **0.232** | -0.256 | 0.184 |
| **-0.32** | **0.232** | | **-0.428** | **0** | -0.34 | 0 |
| **-0.428** | **0** | | **-0.36** | **-0.26** | -0.392 | -0.284 |
| **-0.36** | **-0.26** | | **-0.116** | **-0.356** | -0.096 | -0.3 |
| **-0.116** | **-0.356** | | **0.152** | **-0.464** | 0.096 | -0.296 |
| **0.152** | **-0.464** | | **0.084** | **-0.06** | 0.276 | -0.2 |
| **0.084** | **-0.06** | | | | | |

Figure 8: *Children and parents of the children, with their genetic contribution highlighted*

*Due to time constraints I was unable to test the following crossover methods in the genetic algorithm code:*

### 3.3.2 One point crossover

One point crossover is identical to *two point crossover*, except there is only one crossover point:

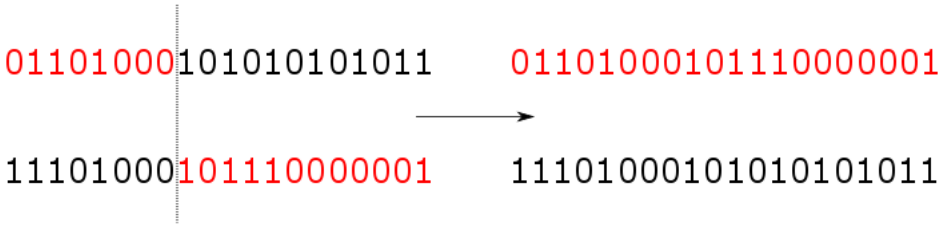



Figure 9: *One point crossover*

### 3.3.3 Uniform crossover

The parents are selected in the same way as *two point crossover*[11] However a mixing ratio $p : 0 < p < 1$ is introduced, such that for any allele in a child, it has a probability $p$ of coming from the first parent, and a probability *1-p* of coming from the second parent, as shown in figure 10:

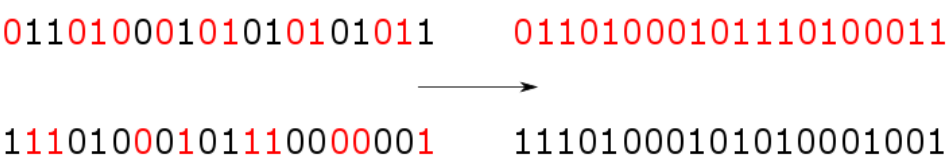



Figure 10: *Uniform crossover*

### 3.3.4 Cut-and-splice crossover

Two parents are selected in the usual way, and a single crossover point *on each parent* is chosen. The alleles after the crossover point on each parent are swapped (which could result in organism strings of different lengths).

### 3.3.5 Three parent crossover

This method is the same as *one point crossover*, except genetic information from three parents is chosen (another variation of this would be three parent crossover done in the style of *two point crossover*).

[11] See section 3.3.1

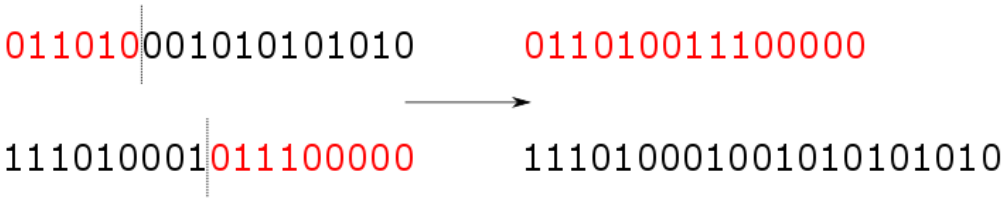



Figure 11: *Cut-and-splice crossover*

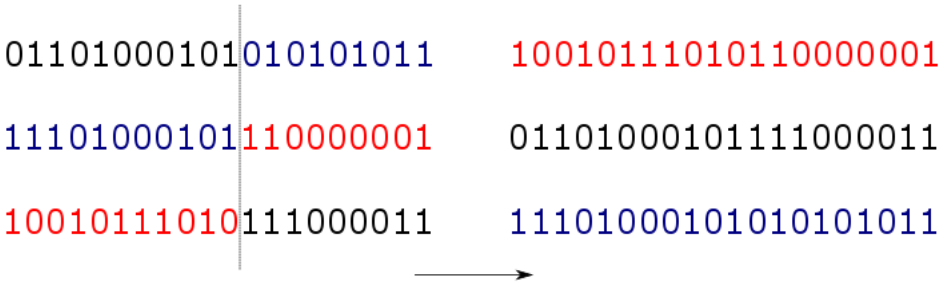



Figure 12: *Three parent crossover*

## 3.4 Mutation

I tested and compared three different mutation operators for this genetic algorithm; in this case, without constraint, the optimum solution would be a sofa which takes up all of *the grid* (an area of 16). To test any of the operators, I compared the sofa I input and the sofa output and I noted the difference(s). The three different styles I tested were:

1. Adding and subtracting a corner randomly (to be known as *reshaping mutation*).
2. Adding in a corner randomly every generation (to be known as *addition mutation*), every $10^{\text{th}}$ generation and every $100^{\text{th}}$ generation.
3. Shifting a corner slightly up, down, left or right (to be known as *shift mutation*).

### 3.4.1 Reshaping Mutation

I created a routine *mutation* to add and subtract a corner at random in each sofa;

```
void mutation(Sofa &p)
{
    p.sortorgstring();

    int flip1 = 1000000*ran();
    bool search1 = false;
    int track1, remove1, count = p.outnumc();

    for(int i = 0; i < p.outnumc(); ++i)
```

```
    {
        if(p.outorgstring(i) == flip1)
        {
            search1 = true;
            track1 = i;
            break;
        }
    }

    remove1 = p.outnumc()*ran();

    if(!search1)
    {
        p.modorgstring(flip1);
        p.eraseorgstring(remove1);
        p.sortorgstring();
    }
    else
    {
        int flip2 = 1000000*ran();
        while(flip2 == flip1)
        {   flip2 = 1000000*ran();
        }

        bool search2 = false;
        int track2;
        for(int i = 0; i < p.outnumc(); ++i)
        {
            if(p.outorgstring(i) == flip2)
            {
                search2 = true;
                track2 = i;
                break;
            }
        }

        if(search2)
        {
            p.eraseorgstring(track2);
            p.modc(count-1);
            p.sortorgstring();
        }
        else
        {
            p.eraseorgstring(track1);
            p.modorgstring(flip2);
            p.sortorgstring();
        }
    }
}
```

The organism string is sorted (as a precaution) and a random integer *flip1* $\in [0, 999999]$ is selected. The organism string is then searched to see if there is a corner at *flip1*. In

the unlikely occurrence there is, *track1* notes the corner number and I select a random corner *remove1* to remove. If there is no corner at *flip1*, I make a corner there and remove corner *remove1*. If there is a corner at *flip1*, I choose a different integer *(flip2)* and search the organism string to see if it's there. In the extremely unlikely event both *flip1* and *flip2* note corner locations, I simply remove corner *track2*. In the more likely event of *flip2* **not** noting a corner location, I remove the corner *track1* and add in the corner at *flip2*. Finally, in each case I sort the organism string again.

To indicate this is working, *figure 13* shows an example sofa before and after mutation:

| **Sect 3.4.1** | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| Original | | | Mutated | | |
| 1.464 | 0 | | 1.464 | 0 | |
| 0.964 | 0.7 | | **0.624** | **0.192** | (added) |
| 0.3 | 0.928 | | 0.964 | 0.7 | |
| -0.26 | 0.808 | | 0.3 | 0.928 | |
| -1.196 | 0.868 | | -0.26 | 0.808 | |
| -1.08 | 0 | | -1.196 | 0.868 | |
| -0.772 | -0.564 | | -1.08 | 0 | |
| -0.288 | -0.884 | | -0.772 | -0.564 | |
| 0.232 | -0.716 | | -0.288 | -0.884 | |
| 0.32 | -0.984 | | 0.232 | -0.716 | |
| **0.66** | **-0.48** | (subtracted) | 0.32 | -0.984 | |
| 1.464 | 0 | | 1.464 | 0 | |

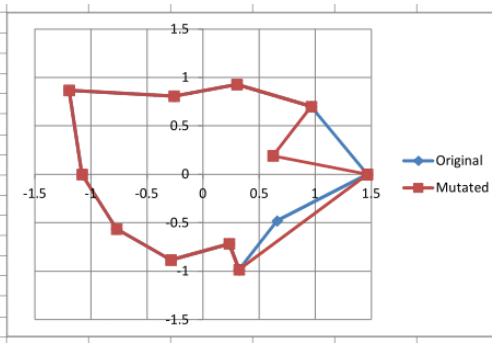



Figure 13: *Adding and subtracting a corner*

### 3.4.2 Addition mutation

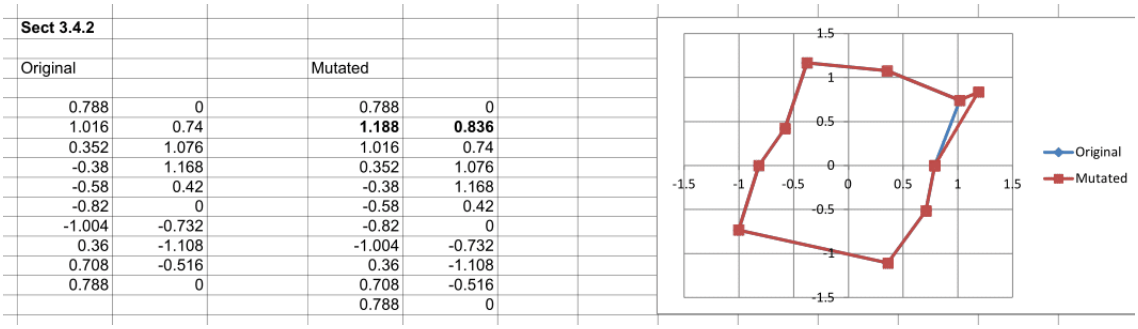
Adding in a corner randomly is a small modification to the above code, namely not removing any corners. I could also choose how often to call the routine, so I've compared every generation, every $10^{\text{th}}$ generation and every $100^{\text{th}}$ generation;

```
void mutation(Sofa &p)
{
    p.sortorgstring();

    int flip = 1000000*ran();

    bool search = false;
    int track, count = p.outnumc();
    for(int i = 0; i < p.outnumc(); ++i)
    {
        if(p.outorgstring(i) == flip)
        {
            search = true;
            track = i;
            break;
        }
    }

    if(search)
    {
```

```
            p.eraseorgstring(track);
            p.modc(count-1);
        }
        else
        {
            p.modorgstring(flip);
            p.modc(count+1);
            p.sortorgstring();
        }
}
```

One problem with this method, however, is how quickly it adds (sometimes unnecessary) corners to a sofa, and storage space for the corner coordinates soon becomes an issue.

To indicate this is working, *figure 14* shows an example sofa:



| Sect 3.4.2 | | | |
|---|---|---|---|
| Original | | Mutated | |
| 0.788 | 0 | 0.788 | 0 |
| 1.016 | 0.74 | **1.188** | **0.836** |
| 0.352 | 1.076 | 1.016 | 0.74 |
| -0.38 | 1.168 | 0.352 | 1.076 |
| -0.58 | 0.42 | -0.38 | 1.168 |
| -0.82 | 0 | -0.58 | 0.42 |
| -1.004 | -0.732 | -0.82 | 0 |
| 0.36 | -1.108 | -1.004 | -0.732 |
| 0.708 | -0.516 | 0.36 | -1.108 |
| 0.788 | 0 | 0.708 | -0.516 |
| | | 0.788 | 0 |



Figure 14: *Adding in a corner*

To compare how the frequency of this mutation operator affected the average area per generation, I ran **4** versions of the above code for 1000 generations. The result is displayed in *figure 15.*

The effects can be seen as follows:

1. With every generation, the plateau is reached quicker, however the area obtained ($\sim 14.66$) is still far from the optimum of 16.
2. With every $10^{\text{th}}$ generation, the plateau is reached slower, however the area obtained ($\sim 14.9$) is greater than the area obtained with the mutation operator in use every generation.
3. With every $100^{\text{th}}$ generation, the use of the operator has become too sparse and its effects no longer play as strong a role as previous, for the solution to converge to an area of 16. The graph (in green) indicates the plateau will be about 13.
4. With no mutation operator, the sofas fail to converge to area 16 and plateau (near immediately) at area 4.87.

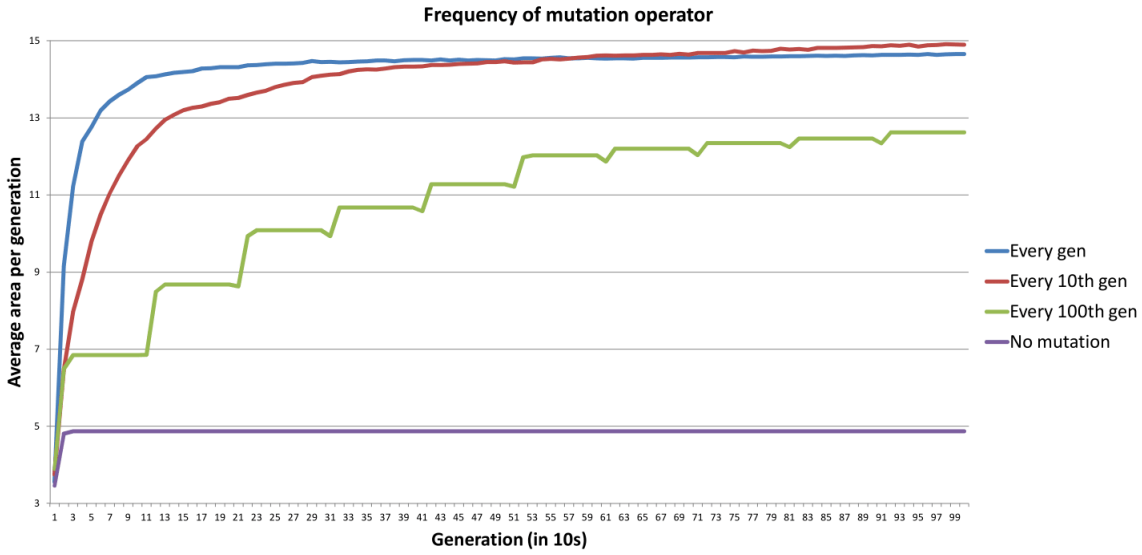



Figure 15: *Using the mutation operator every generation, every $10^{th}$ generation, every $100^{th}$ generation and not at all*

### 3.4.3 Shift mutation

In order to shift a corner slightly, I randomly select a corner to shift and a direction to shift it in (up, down, left or right), and erase that corner from the organism string. I first try shift the corner in the direction *dir*, based on whether the grid has space to shift a corner to the new position. If there is no space, I cycle through the remaining directions. The organism string is sorted both before and afterwards any shifting takes place. The C++ code is;

```
void mutation(Sofa &p)
{
    p.sortorgstring();

    int shift = p.outnumc()*ran();
    int flip = p.outorgstring(shift);
    int dir = 4*ran(), count = 0;

    p.eraseorgstring(shift);

    while(count < 4)
    {
        if(dir == 0)
        {
            if(flip < 998999)
            {
                p.modorgstring(flip+1000);
                count = 5;
            }
            else
            {
```

```
                dir = 1;
                ++count;
            }
        }

        if(dir == 1)
        {
            if(flip > 1)
            {
                p.modorgstring(flip -1);
                count = 5;
            }
            else
            {
                dir = 2;
                ++count;
            }
        }

        if(dir == 2)
        {
            if(flip > 1001)
            {
                p.modorgstring(flip -1000);
                count = 5;
            }
            else
            {
                dir = 3;
                ++count;
            }
        }

        if(dir == 3)
        {
            if(flip < 999999)
            {
                p.modorgstring(flip +1);
                count = 5;
            }
            else
            {
                dir = 0;
                ++count;
            }
        }
    }

    p.sortorgstring();
}
```

To indicate this is working, *figure 16* shows an example sofa:

| Sect 3.4.3 | | | |
|---|---|---|---|
| Original | | Mutated | |
| 1.48 | 0 | 1.48 | 0 |
| 0.368 | 0.268 | 0.368 | 0.268 |
| 0.22 | 0.676 | **0.224** | **0.676** |
| -0.364 | 1.124 | -0.364 | 1.124 |
| -0.9 | 0.652 | -0.9 | 0.652 |
| -1.124 | 0 | -1.124 | 0 |
| -0.824 | -0.6 | -0.824 | -0.6 |
| -0.364 | -1.116 | -0.364 | -1.116 |
| 0.284 | -0.88 | 0.284 | -0.88 |
| 0.848 | -0.616 | 0.848 | -0.616 |
| 1.176 | -0.852 | 1.176 | -0.852 |
| 1.48 | 0 | 1.48 | 0 |

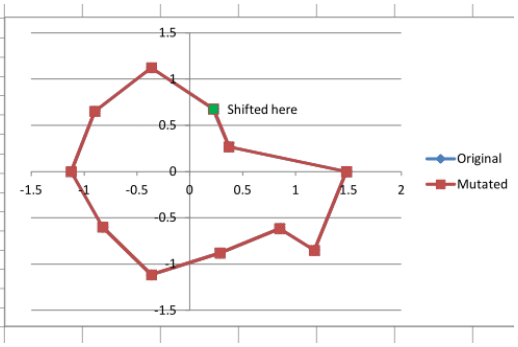



Figure 16: *Shifting a corner*

### 3.4.4 Comparison of 3 mutation operators

*Figure 17* compares how the 3 different mutation operators converge to a maximum area[12]:

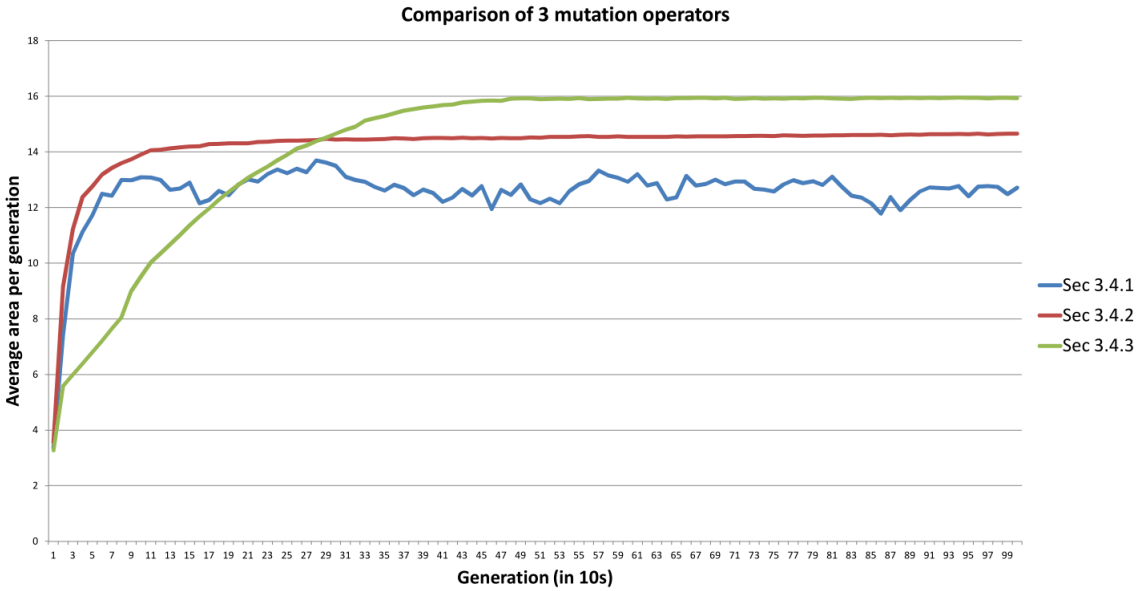



Figure 17: *Comparing Sections 3.4.1, 3.4.2, 3.4.3*

The results indicate the best mutation operator to use is Sect 3.4.3; shifting a corner slightly, which plateaus at an area of $\sim 15.94$[13]. The worst operator is Sect 3.4.1; randomly adding and subtracting a corner - it doesn't plateau off and fluctuates about the $\sim 12.5$ mark.

[12]Sect 3.4.2 here uses the mutation operator *every* generation

[13]In order to reach this area, the program needed to run for 10000 generations, rather than the usual 1000

# 4 The Path Finding Algorithm

The other half of this project was to write code that could determine whether a sofa fits around the hallway or not. To do this, I wrote several algorithms to navigate through (what can be viewed as) a subset $S$ of the space $\Re$ x $\Re$ x $[0, 2\pi]$. Define $S$ to be the box

$$S = \{(x, y, \theta) : 0 \leq x \leq 2,\ 0 \leq y \leq 2,\ 0 \leq \theta \leq 2\pi\}$$

where each sofa can be described with the coordinates $(x, y, \theta)$, which correspond to the $x$ coordinate of the centre point, the $y$ coordinate of the centre point, and the angle through which the sofa has been rotated clockwise from its original position.
I tried 3 different algorithms with varying degrees of success:

1. **Building a path using a random walk**

   This method worked and it worked efficiently, so I used this in the final version[14] of the genetic algorithm with constraint. This algorithm would allow a sofa to randomly wander through $S$ - whenever it hit a 'wall' the path would return a step, and randomly move forward again.

2. **Brute force evaluation of points in *S*, then using dead-end filling to solve**

   This was the most time consuming algorithm, and problems with speed and accuracy meant it was not useful. However once I had collected all the information on points in $S$, I could use a 3 dimensional version of a maze solving algorithm known as *dead-end filling* to find the path quickly (if it existed).

3. **Wall-following**

   Applying another maze solving algorithm in 3 dimensions, I also created a *wall-following* algorithm that would creep along the 'left wall' of the path in $S$; however I never got the chance to test it fully.

## 4.1 Building a path in *S* using a random walk

I created a class *Path* which contains all the information required for each path:

```
typedef class Path
{
    public:
        int num;
        vector <double> initial;
        vector <int> orgstring;

        void modnum(int);
        void modorgstring(int);
        void eraseorgstring();
        void modinitial(double, double, double);
```

[14] See Section 5

```
        void clearpath();

        int outnum();
        int outorgstring(int);
        double outinitial(int);
} Path;

void Path :: modnum(int a)
{   num = a;
}

void Path :: modorgstring(int a)
{   orgstring.push_back(a);
}

void Path :: eraseorgstring()
{   orgstring.pop_back();
}

void Path :: modinitial(double a, double b, double c)
{
    initial.push_back(a);
    initial.push_back(b);
    initial.push_back(c);
}

void Path :: clearpath()
{
    num = 0;
    initial.clear();
    orgstring.clear();
}

int Path :: outnum()
{   return num;
}

int Path :: outorgstring(int a)
{   return orgstring[a];
}

double Path :: outinitial(int a)
{   return initial[a];
}
```

Where;

- *num* is the number of points/length of walk
- *initial* is a vector which contains the starting point of the walk
- *orgstring* is a list of directions which construct the walk[15]

[15]Note: even though this isn't a genetic algorithm, I'm referring to this vector as an organism

### 4.1.1 Routines & functions for checking if a sofa is inside the hallway

I wrote various pieces of code to check if the sofa[16] is still inside the hallway at any particular point in *S*. *hallway* checks if corner $i$ from sofa $s$ lies inside the boundary of the hallway, and returns true if it does, false if it doesn't;

```
bool hallway(Sofa s, int i)
{
    if(s.outcorn_x(i) > 2.01 || s.outcorn_y(i) > 2.01)
    {   return false;
    }

    if(s.outcorn_x(i) < 0.99 && s.outcorn_y(i) < 0.99)
    {   return false;
    }

    return true;
}
```

(Accuracy concerns mean I modified the width of the hallway to be 1.02 units, to allow the path finding algorithm to complete in a reasonable amount of time). This is used in conjunction with *checks* which first moves the sofa centre to a point *(a1, a2, a3)* in *S*, then checks if each of the corners lies inside the hallway at that point;

```
bool checks(double a1, double a2, double a3, Sofa s)
{
    rot(s, a3);
    double temp1, temp2;
    for(int i = 0; i < s.outnumc(); ++i)
    {
        temp1 = s.outcorn_x(i);
        temp2 = s.outcorn_y(i);
        s.modcorn(i, (temp1 + a1), (temp2 + a2));
    }

    for(int j = 0; j < s.outnumc(); ++j)
    {
        if(!hallway(s, j))
        {   return false;
        }
    }
    return true;
}
```

This in turn calls *rot* which rotates all the sofa coordinates by an angle *a3*, and before each calling of *checks*, the sofa should be reinitialised to sit at *(0, 0, 0)* in *S*;

---

string

[16]See Section 3 for any details on the *Sofa* class

```
void rein(Sofa &s)
{
    for(int i = 0; i < s.outnumc(); ++i)
    {   s.modcorn(i, s.outin_x(i), s.outin_y(i));
    }

    makecorners(s);
}
```

### 4.1.2 Routines & functions to make a random walk & build the path

The routine *makeneigh* takes in a point *(a[0], b[0], c[0])* in $S$ and creates 8 different directions for the path to travel in, as illustrated by *figure 18:*

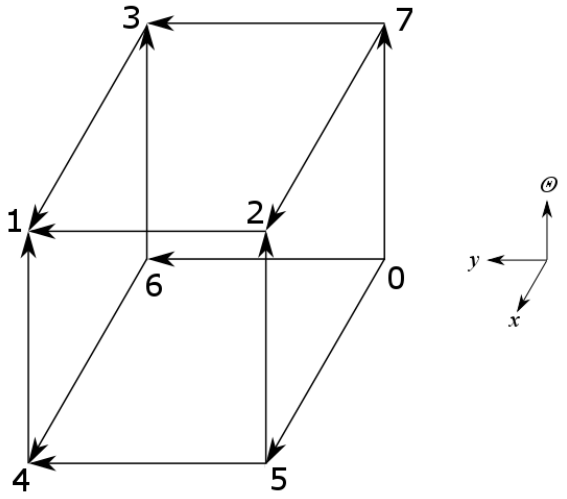



Figure 18: *Neighbours of point 0*

The C++ code is;

```
void makeneigh(double a[], double b[], double c[])
{
    a[1] = a[0] + 0.001;
    b[1] = b[0] - 0.001;
    c[1] = c[0] + 0.001;

    a[2] = a[0] + 0.001;
    b[2] = b[0];
    c[2] = c[0] + 0.001;

    a[3] = a[0];
    b[3] = b[0] - 0.001;
    c[3] = c[0] + 0.001;
```

```
    a[4] = a[0] + 0.001;
    b[4] = b[0] - 0.001;
    c[4] = c[0];

    a[5] = a[0] + 0.001;
    b[5] = b[0];
    c[5] = c[0];

    a[6] = a[0];
    b[6] = b[0] - 0.001;
    c[6] = c[0];

    a[7] = a[0];
    b[7] = b[0];
    c[7] = c[0] + 0.001;
}
```

Before the path is made, *fits* determines where the path should start;

```
bool fits(double nx[], double ny[], double nz[], Sofa s)
{
    nx[0] = 0;
    ny[0] = 1;
    nz[0] = 0;

        rein(s);
        if (!checks(nx[0], ny[0], nz[0], s))
        {
                rein(s);
                while (!checks(nx[0], ny[0], nz[0], s))
                {
                        if (ny[0] > 2)
                        {
                                rein(s);
                                return false;
                        }

                        if (nz[0] > 2*pi)
                        {
                                ny[0] += 0.1;
                                nz[0] = 0;
                                rein(s);
                        }
                        else
                        {
                            nz[0] += 0.1;
                            rein(s);
                        }
                }
                return true;
        }
        else
```

```
        {       return true;
        }
}
```

This function will also return *false* if the sofa doesn't fit in the hallway in any orientation at the entrance. At the cost of time, this can be made more accurate by only allowing *ny[0]* to increase in steps smaller than 0.1.

Using the routines and functions mentioned in Section 4.1.1, *makewalk*, constructs a path for the sofa;

```
bool makewalk(Path &p, int r, double nx[], double ny[], double nz[],
                Sofa s, bool first)
{
    bool notfinished = true;
    int dir, count, godir = 5, countdir = 0;

    if(first)
    {
        nx[0] = p.outinitial(0);
        ny[0] = p.outinitial(1);
        nz[0] = p.outinitial(2);
        count = 0;
        notdir[r] = 0;
    }
    else
    {
        nx[0] = modwalk(p, nx, ny, nz)[0];
        ny[0] = modwalk(p, nx, ny, nz)[1];
        nz[0] = modwalk(p, nx, ny, nz)[2];
        count = modwalk(p, nx, ny, nz)[3];
    }

    while(notfinished)
    {
        makeneigh(nx, ny, nz);
        rein(s);
        notfinished = checks(nx[godir], ny[godir], nz[godir], s);
        if(notfinished)
        {
            nx[0] = nx[godir];
            ny[0] = ny[godir];
            nz[0] = nz[godir];
            p.modorgstring(godir);
            ++count;

            if(ny[0] < -0.25)
            {
                p.modnum(count);
                return true;
            }
        }
        else
```

```
        {
            dir = 8*ran();
            while(dir == 0 || dir == notdir[r] || dir == godir)
            {   dir = 8*ran();
            }

            makeneigh(nx, ny, nz);
            rein(s);
            notfinished = checks(nx[dir], ny[dir], nz[dir], s);
            if(notfinished)
            {
                nx[0] = nx[dir];
                ny[0] = ny[dir];
                nz[0] = nz[dir];
                p.modorgstring(dir);

                if(dir == 7)
                {
                    if(countdir < 10)
                    {
                        godir = dir;
                        ++countdir;
                    }
                    else
                    {   godir = 5;
                    }
                }
                else
                {   godir = dir;
                }

                notdir[r] = 0;
                ++count;

                if(ny[0] < -0.25)
                {
                    p.modnum(count);
                    return true;
                }
            }
            else
            {   notdir[r] = dir;
            }
        }
    }

    p.modnum(count);
    return false;
}
```

This function first determines whether this is the first time it has been called. If so, it loads an initial starting point from the vector *initial* in the class *Path*. If not, it

calls *modwalk* which returns the last point the path was at such that the sofa fit.

After this, *makeneigh* makes the neighbours of said point and *checks* checks if the sofas fits in the hallway after travelling in the direction *godir*[17]. If the sofa fits in the hallway here, *godir* is added to the organism string for path *p*.

If the sofa doesn't fit in the hallway at this point, a new direction to travel in is chosen at random[18] and the sofa tries to move in direction *dir*. If this works, *dir* becomes *godir*[19] and *dir* gets added to path *p*'s organism string. If this does not work, *notdir* becomes *dir* and the function finishes (not having moved anywhere)[20].

There are two reasons this function would end without the sofa moving anywhere:

1. The function has chosen two unsuccessful directions to travel in: In this case, calling the function multiple times ensures I will eventually find a successful direction to travel in[21], unless;

2. There are no more successful directions, i.e. the sofa has hit a dead end.

In the case that a dead end is reached, the function *atdeadend* is set to be called every $10^{\text{th}}$ generation[21] to assess if the sofa is at a dead end;

```
bool atdeadend(Path p, int r, double nx[], double ny[], double nz[],
                  Sofa s)
{
    nx[0] = modwalk(p, nx, ny, nz)[0];
    ny[0] = modwalk(p, nx, ny, nz)[1];
    nz[0] = modwalk(p, nx, ny, nz)[2];
    makeneigh(nx, ny, nz);
    int count = 0;

    rein(s);
    if(!checks(nx[1], ny[1], nz[1], s))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(s);
```

[17]This is the direction in which the sofa traveled in last that worked

[18]This direction is prevented from being *godir* and *notdir*, which is a direction the algorithm tried previously

[19]As long as *dir* isn't 7 - if it is, the sofa could just spin in circles continuously instead. The routine counts how many times dir has been 7 in a row, and disallows *godir* becoming *dir* when *dir* is 7 ten times in a row

[20]This is noted as the end of a 'generation'; a generation in this case being when *makewalk* is called

[21]Which is why in the full code, this routine is allowed run as many *generations* as it wants until the sofa hits a dead end

```
    if(!checks(nx[2], ny[2], nz[2], s))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(s);
    if(!checks(nx[3], ny[3], nz[3], s))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(s);
    if(!checks(nx[4], ny[4], nz[4], s))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(s);
    if(!checks(nx[5], ny[5], nz[5], s))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(s);
    if(!checks(nx[6], ny[6], nz[6], s))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(s);
    if(!checks(nx[7], ny[7], nz[7], s))
    {   ++count;
    }
    else
    {   return false;
    }

    if(count == 7)
    {   return true;
    }
}
```

*atdeadend* calls *modwalk* which returns the point in $S$ at which the sofa fit in the

hallway last. The program then checks every direction to see if the sofa can move. If the sofa cannot, *atdeadend* returns true, if it can, false. If the sofa can indeed move the program continues, if it cannot the program ends with an error message saying the sofa is stuck.

### 4.1.3 Testing the code

To test the code I have the coordinates of 3 example sofas I know fit around the hallway - the Square[22], the Circle[23] and the Hammersley sofa[24].

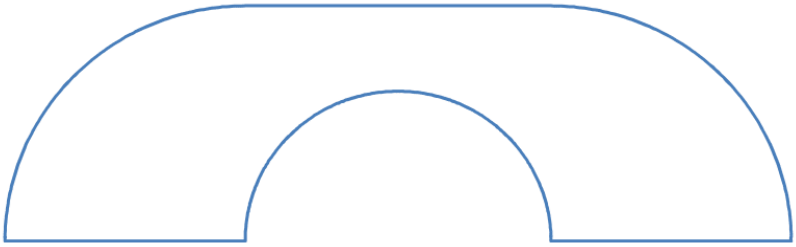


Figure 19: *The Hammersley sofa*

For the Square sofa, the paths it can take through $S$ and the path chosen are displayed in *figure 20.*

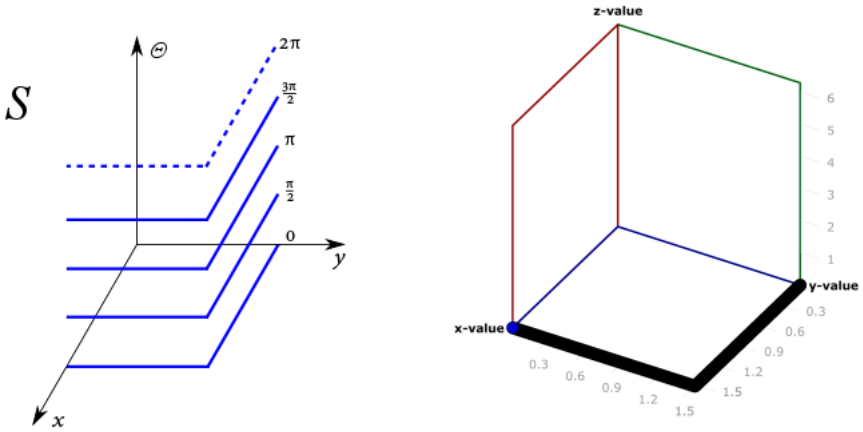



Figure 20: *Possible paths through $S$ and path computed for Square sofa*

For the Circle sofa, the paths it can take through $S$ and the path chosen are displayed in *figure 21.*

For the Hammersley sofa, the paths it can take through $S$ and the path chosen are displayed in *figure 22:*

[22] A 1 x 1 (unit) square
[23] A circe of radius 0.5
[24] Figure 19

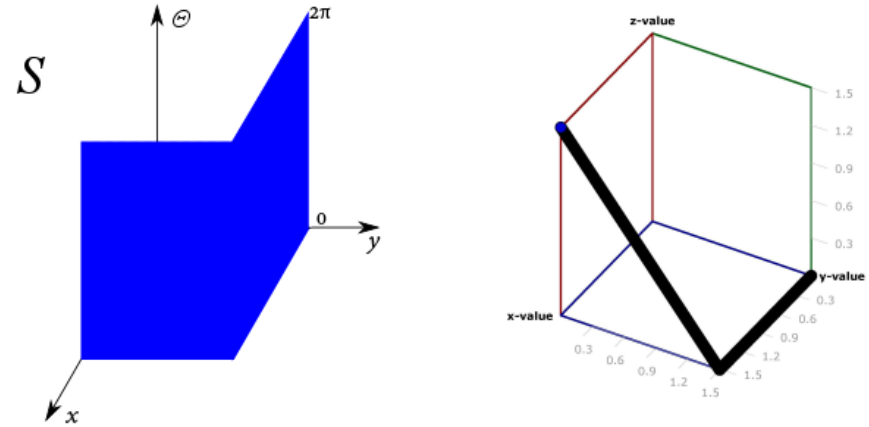



Figure 21: *Possible paths through S and path computed for Circle sofa*

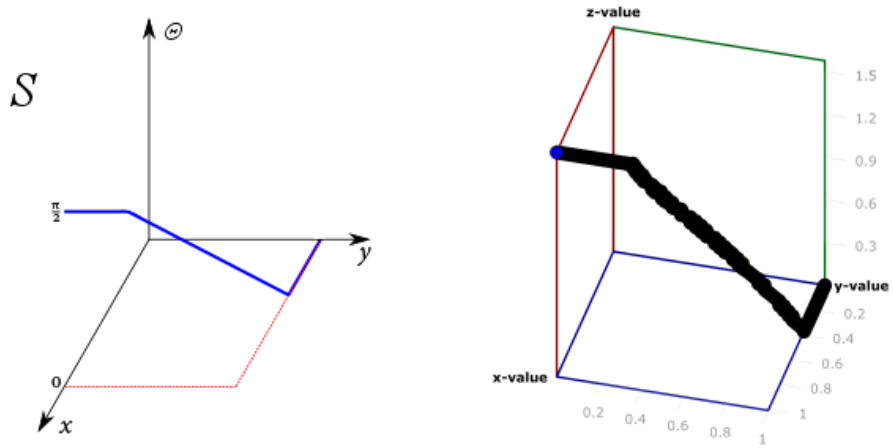



Figure 22: *Possible paths through S and path computed for Hammersley sofa*

Running this code I get the following information in *figure 23* (with the $5^{\text{th}}$ run being the data plotted above).

Note that the $4^{\text{th}}$ run for the Circle sofa took only 13.97 seconds, quite faster than the other run times of $\sim 35$ seconds. I believe this is because the $4^{\text{th}}$ run resulted in a path for the Circle sofa similar to the path computed in figure 20, which would be faster to compute than the path computed in figure 21, which I believe takes $\sim 35$ seconds (because the length of the path is longer).

| Sofa type: | Square | | Circle | | Hammersley | |
|---|---|---|---|---|---|---|
| | Generation | Time (sec) | Generation | Time (sec) | Generation | Time (sec) |
| | 0 | 11.12 | 0 | 34.87 | 80 | 119.31 |
| | 0 | 11.14 | 0 | 35.04 | 97 | 147.95 |
| | 0 | 11.13 | 0 | 34.9 | 99 | 159.02 |
| | 0 | 11.07 | 0 | 13.97 | 92 | 135.69 |
| | 0 | 11.19 | 0 | 35.16 | 89 | 117.38 |
| Average: | 0 | 11.13 | 0 | 30.788 | 91.4 | 135.87 |

Figure 23: *Generations and run times for different sofa types*

## 4.2 Brute force method

The brute force method evaluates (using *checks*) as many points as possible in $S$. It stores the allowed points, from which algorithms such as *dead-end filling* and *wall-following* can find the desired path.

### 4.2.1 Calculation of points in $S$

The C++ code has a similar structure to that of the code from Section 4.1[25], and uses the same routines and functions from Section 4.1.1. The main difference is this code includes a linked list structure[26]. A linked list is a container to store data, organised such that the first piece of data 'points' to the second piece, the second piece 'points' to the third, and so on. I used this structure because of the large amount of information I need to store in this program, and a linked list occupies very little memory for this sort of task. In comparison, a vector, array or deque would need to remember each piece of data separately, which, to gain any sort of level of accuracy, leads to immediate memory problems.

The linked list code I use is;

```
class LinkedList
{
    struct Node
    {
        double x;
        Node *next;
    };

    public:
        LinkedList()
```

[25]There is a major difference; in all of the following code I didn't implement a *Sofa* class structure, meaning the coordinates of the corners and the *number* of corners are two different arguments in every routine and function

[26]Unless otherwise stated, all code relating to linked lists originates at http://pastebin.com/yGh8hjnx (7th August 2015)

```
        {   head = NULL;
        }

        void addValue(double val)
        {
            Node *n = new Node();
            n->x = val;
            n->next = head;
            head = n;
        }

        double popValue()
        {
            Node *n = head;
            double ret = n->x;

            head = head->next;
            return ret;
        }

    private:
        Node *head;
};
```

Where *popValue* returns the value of the first (most recently added) element in the linked list. Note that there is no destructor here - the purpose of this program was to create a single linked list which contains all the path data for a particular sofa.

In theory, plotting all the points in the completed linked list should give graphs like *figures 20, 21, 22* with respect to the specific sofas, however it takes a large amount of time to collect this data[27]. To collect the data, I use the following code;

```
LinkedList xlist, ylist, zlist;

for(int i = 0; i < 200; ++i)
{
    for(int j = 0; j < 200; ++j)
    {
        cent[0] = (double) i/100;
        cent[1] = (double) j/100;
        for(int k = 0; k < 628; ++k)
        {
            t = (double) k/100;
            rot(s, c, t);

            if(checks(cent, s, c) == 0)
            {
                xlist.addValue(cent[0]);
                ylist.addValue(cent[1]);
```

[27]It would take approx. 9 days to run this program with 1,000,000 points per unit cube in $S$ which for some sofa shapes (like the Hammersley sofa) isn't accurate enough

```
                    zlist.addValue(t);

                    output<<fixed<<setprecision(10)<<xlist.popValue()<<" "<<
                            ylist.popValue()<<" "<<zlist.popValue()<<endl;
                }

                rein(s, in, c);
            }
        }
}
```

Here I use 3 linked lists to store the $x$, $y$ and $z$ coordinates of any point that allows the sofa to fit inside the hallway, centred at that point in $S$. For use in Sections 4.2.2 and 4.2.3, I output all the points *(x, y, z)* into an exterior file.

### 4.2.2 Dead-end filling

*Dead-end filling*[9] is a maze solving algorithm used when all the information about the maze is known - when a full view of the maze is available (as it will be, using data from Section 4.2.1). This algorithm identifies all dead ends in the maze, removes them, then repeats until there are no more dead ends - where a dead end is defined as having 0 or 1 neighbouring points (see *figure 24*). This should just leave a single path which solves the maze - this is the path the sofa must take in $S$ to manoeuvre around the hallway (if it exists).

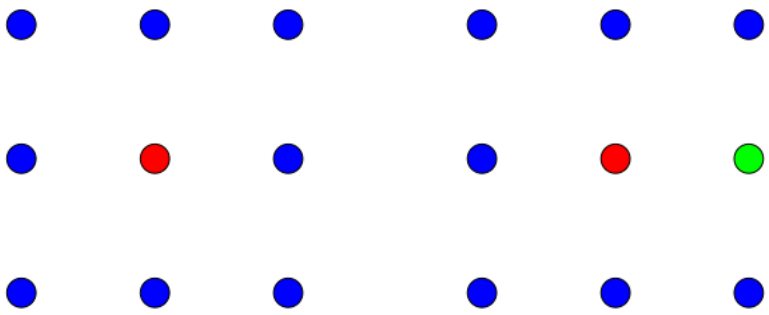


Figure 24: *In the two dimensional case, an example of having 0 (LHS) and 1 (RHS) neighbour(s) (where the green dot is a neighbour to the red, and the blue dots are background)*

In order to use this method, I need to have a clear idea on what the start and endpoints of the path should be (so I don't accidentally count them as dead ends). To do this, I wrote the routine *startfinish* to identify the start and endpoints;

```
double aaa;
void startfinish(double a[], double strtpnt[], double endpnt[],
                 double s[][2], double in[][2], int n)
{
```

```
    a[0] = 0;
    a[1] = 1;

    bool finished = false;
    while(!finished)
    {   finished = fits(a, s, in, n, 1);
    }

    strtpnt[0] = a[0];
    strtpnt[1] = a[1];
    strtpnt[2] = aaa;

    a[0] = 1;
    a[1] = 0;
    aaa = 0;

    bool finished2 = false;
    while(!finished2)
    {   finished2 = fits(a, s, in, n, 0);
    }

    endpnt[0] = a[0];
    endpnt[1] = a[1];
    endpnt[2] = aaa;
}
```

This requires a modification to *fits*; it now takes an additional argument telling it what direction it should be shifting in (0 for the $x$ direction, 1 for the $y$ direction) once the sofa has done a full 360°rotation of being disallowed:

```
int cnt = 0;
bool fits(double a[], double s[][2], double in[][2], int n, int g)
{
    rein(s, in, n);
    aaa = (2*pi/10000)*cnt;
    if (!checks(a, s, n))
    {
        rein(s, in, n);
        while (!checks(a, s, n))
        {
            if (a[g] > 2)
            {   cout<<"DOESN'T FIT"<<endl;
            }

            if (cnt > 10000)
            {
                rein(s, in, n);
                a[g] += 0.01;
                cnt = 0;
                return false;
            }
            else
```

```
            {
                rein(s, in, n);
                rot(s, n, ((2*pi/10000)*cnt));
                ++cnt;
                return false;
            }
        }
        return true;
    }
    else
    {   return true;
    }
}
```

With this framework, the code that determines where the dead end points are and deletes them is[28];

```
double a[7], b[7], c[7], rem[cnt][3];
int neigh, p = 0;
bool val, finished = false;

while(!finished)
{
    val = true;
    xxx = 0;
    xlist.rewind();
    ylist.rewind();
    zlist.rewind();

    while(xxx < cnt)
    {
        neigh = 0;
        a[0] = xlist.getValue();
        b[0] = ylist.getValue();
        c[0] = zlist.getValue();

        makeneigh(a, b, c);
        if((a[0] == strtpnt[0] && b[0] == strtpnt[1] &&
            c[0] == strtpnt[2]) || (a[0] == endpnt[0] &&
            b[0] == endpnt[1] && c[0] == endpnt[2]))
        {   neigh = 7;
        }
        else
        {
            for(int i = 1; i < 26; ++i)
            {
                xlist.rewind();
                ylist.rewind();
                zlist.rewind();
                while(xlist.hasValue())
                {
```

[28] *cnt* is the size of the linked lists *xlist, ylist* and *zlist*

```
                    if(a[i] == xlist.getValue() && b[i] ==
                        ylist.getValue() && c[i] == zlist.getValue())
                    {
                        ++neigh;
                        xlist.next();
                        ylist.next();
                        zlist.next();
                    }
                    else
                    {
                        xlist.next();
                        ylist.next();
                        zlist.next();
                    }
                }
            }
        }

        xlist.rewind();
        ylist.rewind();
        zlist.rewind();

        if(neigh == 0 || neigh == 1)
        {
            rem[p][0] = a[0];
            rem[p][1] = b[0];
            rem[p][2] = c[0];
            ++p;
        }

        ++xxx;

        for(int j = 0; j < xxx; ++j)
        {
            xlist.next();
            ylist.next();
            zlist.next();
        }

    }

    for(int j = 0; j < p; ++j)
    {
        xlist.remove(rem[j][0]);
        ylist.remove(rem[j][1]);
        zlist.remove(rem[j][2]);
        val = false;
    }

    if(val)
    {   finished = true;
    }
}
```

Note that a routine *makeneigh* is called - this is **not** the same *makeneigh* that appeared in Section 4.1. This *makeneigh* makes 26 neighbours in a cube in $S$ surrounding a point *(a[0], b[0], c[0])*.

This code is untested due to the large amount of time it would take to collect the necessary data required as input, but theoretically it works.

### 4.2.3 Wall following

Another maze solving algorithm which has applications in this problem is *wall-following*[9] - assuming the maze is simply connected, it is topologically equivalent to a (closed) loop with an entrance and an exit. By using the rule 'move forward and keep one hand on the wall to your left' the maze can be successfully navigated.

Using this idea, I wrote a program that would determine where next to move based on the sofas current position in $S$. I used the idea of neighbouring points from Section 4.2.2; I labeled the points 1-26 and based on whether certain points were disallowed[29] the program would determine where to move. In two dimensions, *figure 25* provides a visual representation to some of the cases that need to be dealt with.

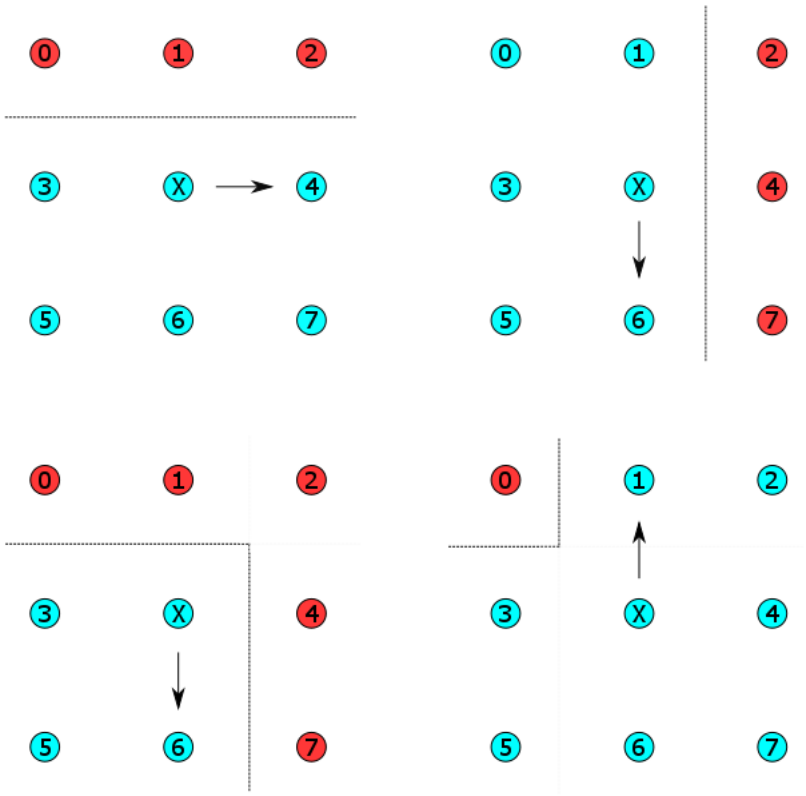



Figure 25: *In two dimensions, 4 examples of allowed points (in blue) and disallowed points (in red) and where the program determines where to move based on these points*

[29]See figure 25

The dotted lines represent the 'wall' of the maze - in reality, this is the boundary of the path the sofa can take through $S$.

Since this was never fully tested, I don't have results to display, however due to the complexity of the code (and the number of different cases I would have to consider) the method of building a path from a random walk (Section 4.1) is far superior[30].

[30]Note that I don't need to use the brute force calculation mentioned in Section 4.3.1 in order for this code to function; at any point I just need to examine the 26 neighbours of the point in order for the code to make a decision on where to move. However, I *can* use Section 4.3.1 as a database of allowed points instead of preforming 26 calculations at every step in the path.

# 5 The Constrained Genetic Algorithm

In this section I combine code from Section 3 and Section 4.1 to create a genetic algorithm subject to the constraint of fitting inside the hallway at all times; that is, each sofa generated must have a path through $S$. This is the code which I use to attempt the *Moving Sofa* problem. The reproduction operators I use are tournament selection, two-point crossover and shift mutation.

There are many ways to implement this constraint in the genetic algorithm;

1. **Setting the area of a sofa which doesn't fit to 0**

   This effectively[31] removes the sofa from the possible breeding population. The disadvantage to this is the population becomes smaller each generation and thus the members of the breeding population become less and less genetically diverse. This method will be known as the *nullifying constraint*.

2. **Halving & doubling the area**

   I implement the constraint of halving the area of any sofa which doesn't fit through the hallway, and doubling the area of any sofa that does. This has an advantage over (1) in that no sofas are removed from the population - they just become unlikely to be selected. This has the disadvantage however, of a disallowed[32] sofa being selected over an allowed sofa if the area of the disallowed sofa is particularly large[33]. Note that this method falls victim to the same problem as (1) - there is a small possibility of a disallowed sofa making it into the breeding population if the group chosen in tournament selection are all disallowed sofas. However I believe this method has an advantage over (1) in that the population isn't getting smaller each generation. Essentially this method transforms the *constrained* genetic algorithm into an *unconstrained* genetic algorithm - not only is the algorithm learning which sofas have a large area, it also learns which sofas are best at fitting around the hallway. This method will be known as the *determined area modification (DAM) constraint*.

3. **Seeding the initial population**

   Some research[1] has shown that, since the space the genetic algorithm must search for a global optimum[34] is so large, the algorithm finds an optimum faster when placed in a region of the space that is known to be near the optimum - that is, if the initial population is seeded with sofas close to the optimum, the genetic algorithm can use these to converge to the optimum faster and more efficiently.

[31] Using tournament selection, it is possible for a sofa of area 0 to be selected for breeding, however this will only happen when all the members of the group chosen for tournament have area 0 - an event which tends to occur only when a large portion of the population has area 0 (see Section 3.2.2) - in this case, this would mean the population is 'dying out' or becoming too small to sustain itself

[32] In the sense that the sofa doesn't fit around the hallway

[33] However this is unlikely to happen, as the area of the disallowed sofa would have to be at least 4 times larger than the area of the allowed sofa

[34] In this case the space is the set of all simple polygons which fit around the hallway

[1] evaluates the usefulness of seeding in the *Travelling Salesman Problem* and the *Job-Shop Scheduling Problem*, but notes seeding works well with the former and poorly with the later. With this in mind, I test the usefulness of seeding for the Moving Sofa Problem.

## 5.1 A change to the way area is calculated

A major flaw in the routine *makecorners*[35] is it requires the sofa to be a *star shaped domain* in order to accurately calculate the area[36]. However the Hammersley (and Gerver) sofa(s) are not star shaped domains and thus the genetic algorithm (as it stands) would not realise this shape is close to the global optimum! There doesn't seem to be a universal method in calculating the area of an irregular simple polygon, given an unordered list of its coordinates, so to fix this problem I instead calculate the area of the **convex hull**[37] of the sofa. The advantages of this method include;

- This method can always be done for any sofa shape I encounter.
- This method allows points to 'shrink' back into the sofa with no negative effect in the area; for example, the Hammersley sofa (using this method) has the same area as a sofa with the centre filled in - however the former will fit around the hallway whilst the later will not.
- This method broadens the set of possible sofas to test (both a good and bad thing; more sofas mean a greater chance of finding the optimum, however more sofas mean a longer computation time).
- In the case where a set of coordinates is ambiguous as to what sofa shape it defines, this method would assign all possible sofa shapes the same area and the only criterion to passing into the next generation would be whether it fits around the hallway or not.

This method does have its disadvantages however;

- This method doesn't actually calculate the area of a sofa (unless that sofa is convex) and thus the selection operator will select those with the largest convex hull, rather than largest area (which can be a good and bad thing; generally the larger the sofa the larger the convex hull, however two sofas with points not in the convex hull which look and act different have the same probability of being selected).
- Only changing the corners in the convex hull has an effect on the area of the sofa - thus changes to points which are not in the convex hull appears to have no effect on the sofa's selection probability, so points not in the convex hull mightn't be altered often.

[35] See Section 3.1

[36] Using the shoelace algorithm the points all need to be ordered clockwise or anticlockwise

[37] The convex hull of a shape is the smallest convex set containing that shape

The new *makecorners* uses the *gift-wrapping algorithm*[8] to determine the convex hull of a sofa, and then the shoelace algorithm to determine the area of the (now ordered) convex hull. The C++ code is;

```
void rot(Sofa &s, double k);
void createin(Sofa &s);
void rein(Sofa &s);

double dist(Sofa p, int i, int j)
{   return ((p.outcorn_y(i) - p.outcorn_y(j))*(p.outcorn_y(i) -
           p.outcorn_y(j)) + (p.outcorn_x(i) - p.outcorn_x(j))*
           (p.outcorn_x(i) - p.outcorn_x(j)));
}

void makecorners(Sofa& p)
{
    int w = 0;
    int temp1, temp2;
    p.sortorgstring();

    for(int i = 0; i < p.outnumc(); ++i)
    {
        temp1 = ((p.outorgstring(i)) % 1000) - 500;
        temp2 = (p.outorgstring(i))/1000 - 500;
        p.modcorn(w, (double) temp1/250, (double) temp2/250);
        ++w;
    }

    createin(p);

    double miny = 100;
    int county = 0;

    for(int i = 0; i < p.outnumc(); ++i)     //(1)
    {
        if(p.outcorn_y(i) <= miny)
        {
            if(p.outcorn_y(i) == miny)
            {
                if(p.outcorn_x(i) < p.outcorn_x(county))
                {   county = i;
                }
            }
            else
            {
                miny = p.outcorn_y(i);
                county = i;
            }
        }
    }

    vector <int> cvexhull;
    int temp2corn, tempcorn = county;
```

```
double angle, totangle = 0, minangle;
double cornx, corny;
bool finished = false;

while(!finished)
{
    cvexhull.push_back(tempcorn);
    minangle = 7;

    for(int j = 0; j < p.outnumc(); ++j)
    {
        if(j != tempcorn)
        {
            cornx = p.outcorn_x(j) - p.outcorn_x(tempcorn);
            corny = p.outcorn_y(j) - p.outcorn_y(tempcorn);

            if(cornx == 0)
            {
                if(corny > 0)
                {   angle = pi/2;
                }
                if(corny < 0)
                {   angle = 3*pi/2;
                }
            }
            else
            {   angle = atan(corny/cornx);
            }

            if(cornx < 0)
            {   angle += pi;
            }

            if(cornx > 0)
            {
                if(corny < 0)
                {   angle += 2*pi;
                }
                if(corny == 0)
                {   angle = 0;
                }
            }

            if(angle <= minangle)
            {
                if(angle < minangle)
                {
                    minangle = angle;
                    temp2corn = j;
                }
                else                    //(2)
                {
                    if(dist(p, tempcorn, temp2corn) <
                        dist(p, tempcorn, j))
```

```
                        {   temp2corn = j;
                        }
                    }
                }

            }
        }

        tempcorn = temp2corn;

        rein(p);
        totangle += minangle;
        rot(p, totangle);

        if(tempcorn == county)
        {   finished = true;
        }
    }

    rein(p);

    double area = 0;
    for(int n = 0; n < cvexhull.size()-1; ++n)
    {   area += abs(p.outcorn_x(cvexhull[n])*p.outcorn_y(cvexhull[n+1])
                - p.outcorn_x(cvexhull[n+1])*p.outcorn_y(cvexhull[n]));
    }

    p.modarea(0.5*area);
    cvexhull.clear();
}
```

The routines *rot, createin* and *rein* respectively rotate the sofa by $k$ radians, create an initial sofa template before any modifications are made and reinitialise the sofa to its template. Also note the central section marked by (1) which determines the point with the minimum $y$ coordinate (if this is not unique then the point with the minimal $x$ and $y$ coordinate), which forms the starting point for the convex hull. Finally, if three or more points are co linear whilst being chosen for the complex hull, the code identifies this case (2) and chooses the further away point.

## 5.2 The nullifying constraint

This method isolates any sofa which doesn't fit around the hallway (that is, 50 different random walks independently reach dead ends) and sets its area to 0. At the end of each generation the program outputs every sofas area and the coordinates of every sofa (separately).

Taking the average of every generations area I get the following graphs *(figures 26 and 27)* where *figure 26* uses the original method to exactly calculate the area, and *figure 27* uses the gift wrapping algorithm to calculate the area of the convex hull.

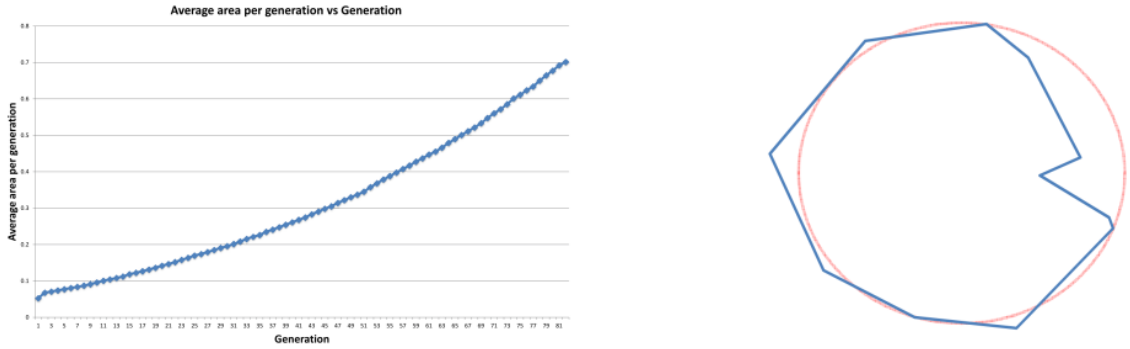



Figure 26: *On the LHS is the graph of average area per generation vs generation. On the RHS is a sofa from the final generation with area 0.676992 and the Circle sofa in the background*

Although the graphs in *figures 26* and *27* show the average area is increasing per generation, this is actually the average area of *allowed* sofas and in reality after generation 80 large numbers of sofas are being disallowed for reaching dead ends in $S$. One possible way to combat this would be to have a larger initial population[38] or using a less totalitarian constraint.

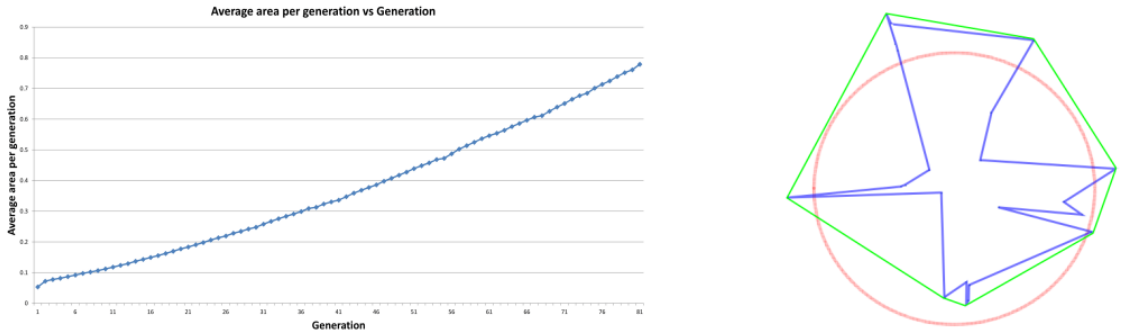



Figure 27: *On the LHS is the graph of average area per generation vs generation. On the RHS is a sofa from the final generation with (denoted in green) the convex hull area 0.76636 and the Circle sofa in the background*

The difference in the two sofa shapes is apparent - as I predicted, the convex hull method pays little attention to points not in the convex hull and thus they get 'left behind' as the sofa grows larger. In this scenario I believe the 'old' method of calculating the exact area works best.

## 5.3 The DAM constraint

This method isolates any disallowed sofa and halves (the number representing) the area, whilst doubling (the number representing) the area of every allowed sofa. I tried

[38]The initial population here in both cases was 50 randomly generated (see Section 3.1) sofas

this version of the program with the convex hull method of calculating a sofas area and found it didn't work very well, so the data obtained here has used the previous version of *makecorners* which calculates the exact area of a sofa. Like previous cases, as the number of generations increase, the average area of a sofa also increases *(figure 28):*

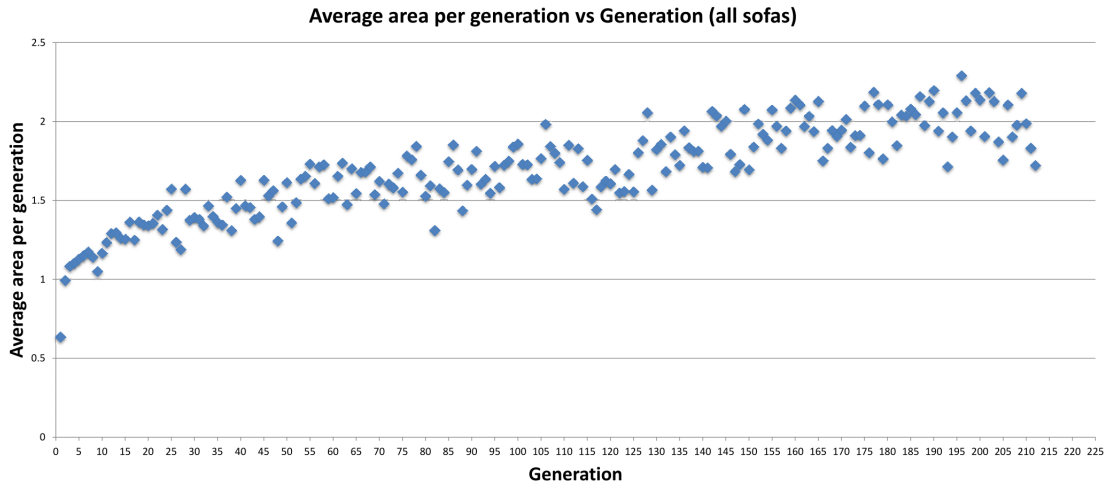



Figure 28: *Average area of all sofas (allowed and disallowed) per generation vs Generation*

The trend is perhaps better seen when I discard all disallowed sofas and take the average area of just the allowed sofas *(figure 29):*

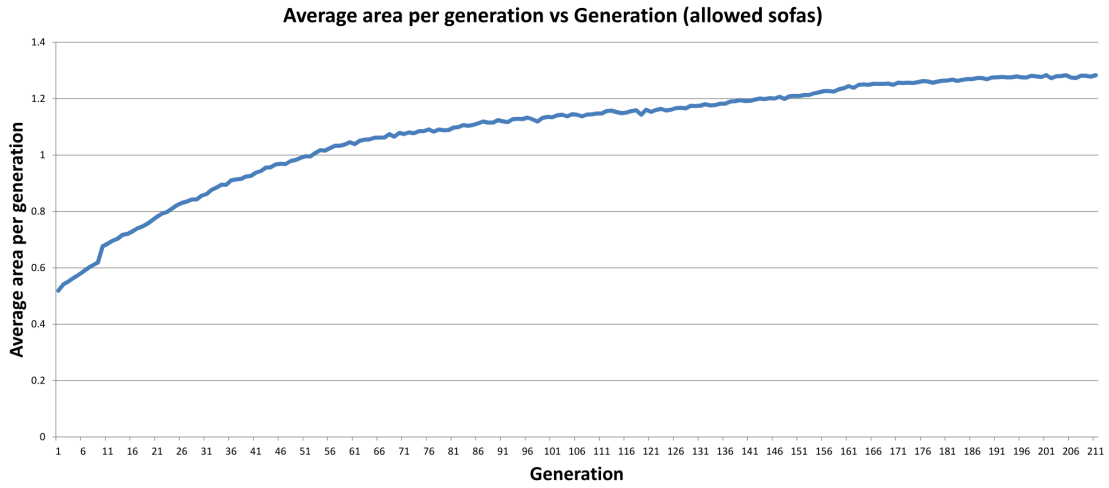



Figure 29: *Average area of (allowed) sofas per generation vs Generation*

Finally, the shape produced by the largest (allowed) sofas approximates (interestingly) a rhombus, with an inradius of 0.5 *(figure 30);*

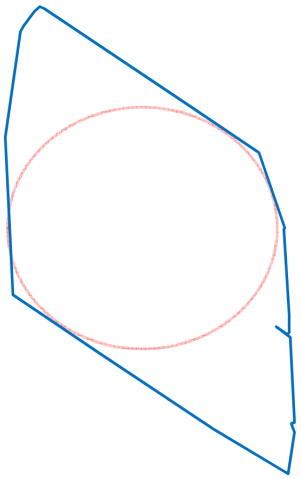


Figure 30: *The largest allowed sofa, calculated with the DAM constraint, with the incircle shown in red*

## 5.4 Seeding the initial population

I ran the genetic algorithm twice with two different initial populations; the first population composed of Circle sofas and the second composed of Hammersley sofas. With both populations I use the convex hull method of calculating area[39] and also I use the DAM constraint from Section 5.3.

### 5.4.1 Changes made to original code from Section 3

Instead of using the routine *makepop* to randomly generate my population, I read in data (the coordinates of the corners) from an external file then used *search* to *pixelize*[40] the sofas. From there I used the routines *makeorgstring* and *makecorners* as usual. I made no modifications to the reproduction operators and path finding algorithm[41].

The code to read in the data and pixelize the sofas is;

```
int numberofcorn;
double xxx, yyy;

cin>>numberofcorn;
for(int l = 0; l < 10; ++l)
{
    s[l].modc(numberofcorn);
    s[l].modcont(1.0);
    s[l].modarea(0);
}
```

[39]With Hammersley sofas this is a necessity

[40]Modify the corners of a sofa so they fit on *the grid*

[41]To keep things fast I tried this method both times with a population of 10 sofas.

```
double tempsearch1, tempsearch2;
for(int i = 0; i < numberofcorn; ++i)
{
    cin>>xxx>>yyy;
    tempsearch1 = search(xxx);
    tempsearch2 = search(yyy);

    for(int k = 0; k < 10; ++k)
    {   s[k].modcorn(i, tempsearch1, tempsearch2);
    }
}
```

Where:

- *numberofcorn* is the number of corners of the sofa (number of points of data to be read in).
- *search* is the same search as is used in Section 3.1.

One change I did make to *search*, however, was for the Hammersley sofa initial population; I found when the sofa was pixelized it could no longer turn around the corner in the hallway - I believe the reason behind this is its edges are no longer smooth enough to slide around the corner. To fix this, I made *the grid* 100 times more accurate - that is, I made a 10,000 x 10,000 point grid in the same style as the 1,000 x 1,000 point *grid.*

### 5.4.2 Seeding the population with the Circle sofa

The resulting sofa I obtain is displayed in figure 31.

Due to the number of points (relatively) flat and close to the original Circle sofa at the top, bottom, left and right sides of the sofa, I believe the genetic algorithm was breeding a Square sofa from a Circle sofa. However more time and generations are needed to explore this suspicion, as it could also be that the 2% increase in width in the hallway from Section 4.1.1 is allowing a slightly bigger circle through.

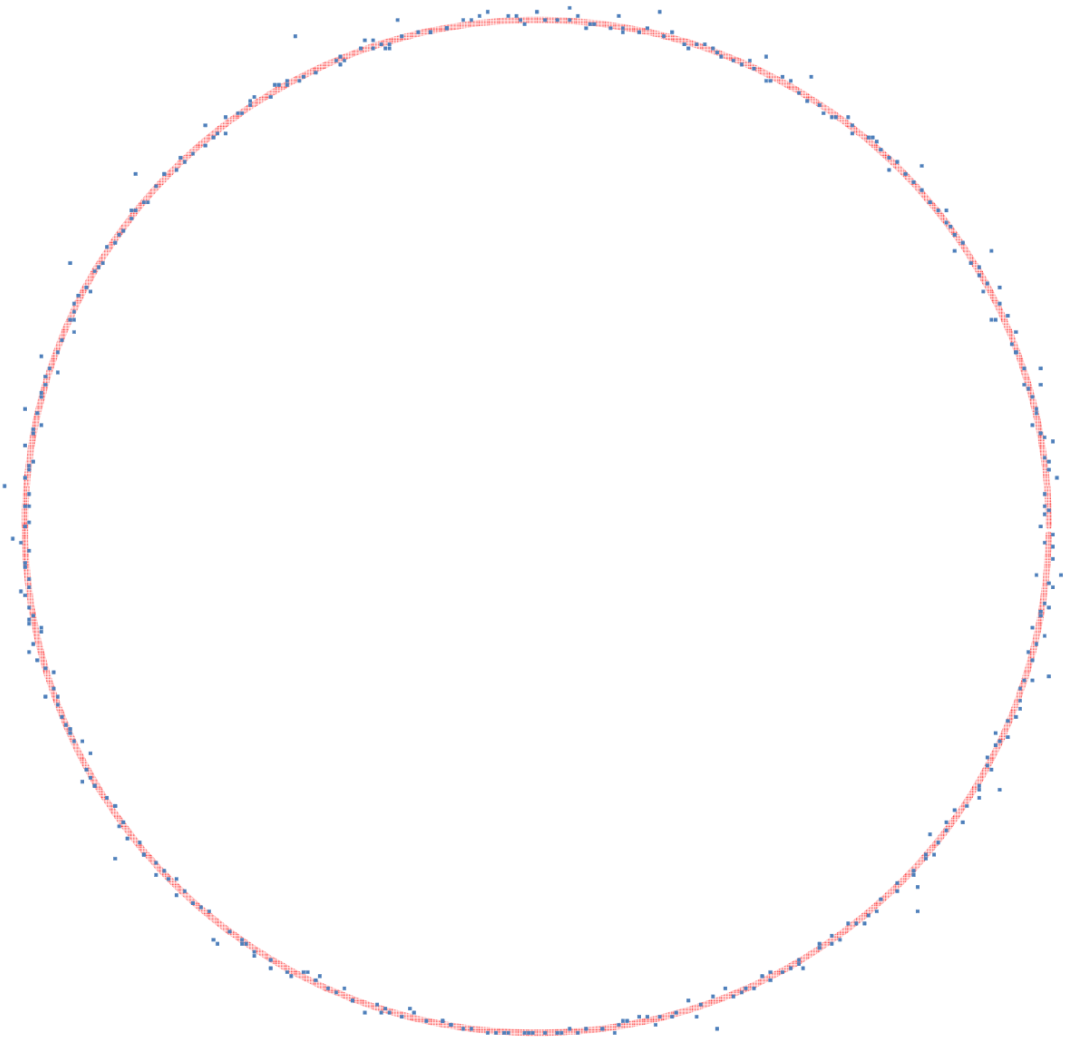


Figure 31: *The sofa (with corner points in blue) obtained by seeding the initial population with Circle sofas (original circle sofa shown in red).*

### 5.4.3 Seeding the population with the Hammersley sofa

The resulting sofa I obtained is displayed in *figure 32 (TOP)*:

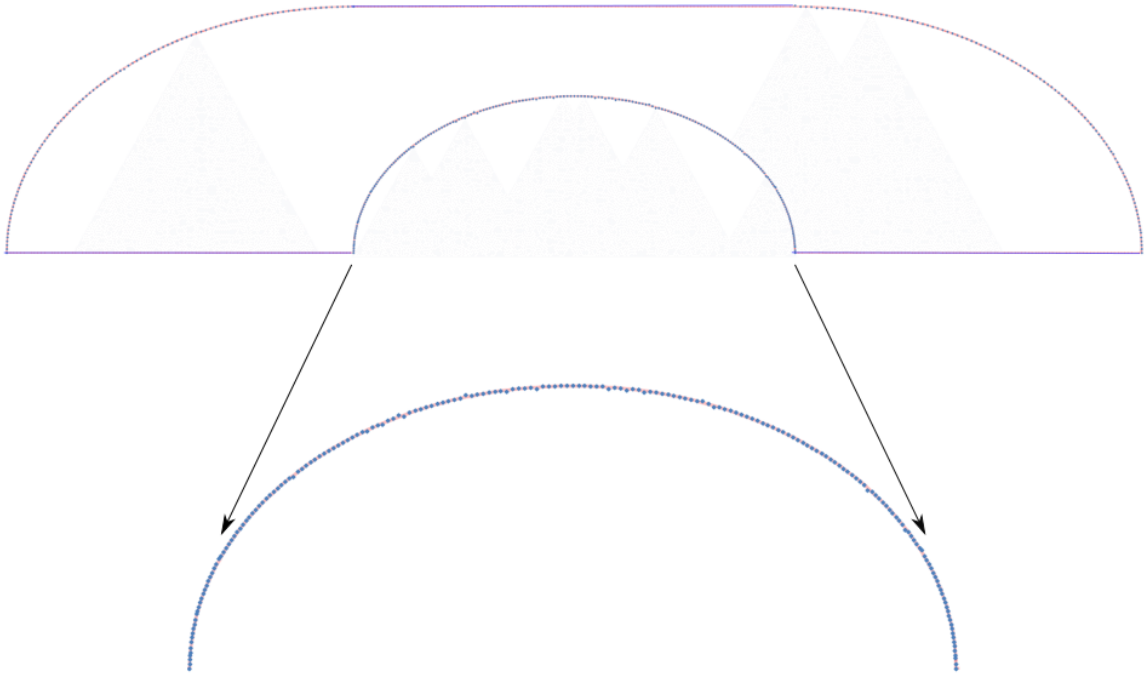


Figure 32: *(TOP) The sofa (with corner points in blue) obtained by seeding the initial population with Hammersley sofas (original Hammersley sofa shown in red - neither to scale). (BOTTOM) Close-up of the semicircular underside.*

This is very close to the original Hammersley sofa (in red), mainly because the program didn't get to run for very long - running the program for a week it reached 60 generations[42] - however some minor differences, mainly to the semicircular underside *(BOTTOM)*, have been made. The area of the convex hull of the original Hammersley sofa inputted was 2.81812, whereas the area of the new genetically altered sofa is 2.83929 - this sofa exists with an area slightly greater than the Hammersley sofa but smaller than the Gerver sofa.

More time for the program to run is clearly needed but I believe this program has made a step in the right direction - with enough time this constrained genetic algorithm will improve upon the Hammersley sofa and should approximate the Gerver sofa, which is currently believed[3] to be the optimum and solution to the Moving Sofa Problem.

[42] As opposed to the $\sim 500$ generations the program ran with the Circle sofa in Section 5.4.2

# 6 Problems encountered & future modifications

Various problems that I encountered along the way are;

1. **Inbreeding**

   For the 3 types of selection I considered in Section 3.2, each time I would run into the problem of inbreeding - that is, two parents having an identical organism string. This results from the fact that a sofa can be selected more than once (i.e. it can breed more than once) and thus there is a probability that a sofa can be chosen to breed with itself during *crossover*. This itself doesn't raise a problem, however the two children produced are genetically identical to the parent. Thus, without changing (so far) the parent sofa has been replicated twice into the next generation[43]. This will lead to a decrease in diversity and thus convergence on a solution will occur slower. This problem is also similar to *genetic drift* which in turn could lead to *fixation* - meaning some (not necessarily the fittest) solutions would dominate the population.

   To try combat this, in the various selection algorithms I considered I've included a counter that prevents any sofa from being in the breeding population more than a fixed number of times. Since this can still lead to inbreeding (although less often) in the crossover algorithms steps can be taken to prevent two parents of 'similar' genetic structure from breeding[44].

2. **A better path finding algorithm**

   The current path finding algorithm I use (from Section 4.1) was the best one I could create, but there could be a better path finding algorithms that are faster or more accurate.

3. **The constrained genetic algorithm**

   As mentioned in Section 5, applying a constraint to the genetic algorithm to get it to function correctly was a tricky process with many (figurative and literal) dead ends. It's possible there is a different system of constraint that works better with this problem and leads to a more accurate approximation of the global optimum.

Modifications that I could make to the code in the future are;

1. **Speed**

   Of course the modification that can always be made is speed. I would hope that I could increase the code's efficiency at points to make the overall generation time lower (see (2) as well).

[43] The mutation operator will change the organism strings of both children uniquely (probably) however not by a large amount

[44] However this could lead to problems - the whole point of a genetic algorithm is for solutions to converge to a single optimum

2. **Parallel computing**

   Certain points in the code (such as *crossover, mutation, testing a sofa to find a path, transferring genetic information from the old children to the new parents*) can be run in parallel, thus reducing the running time of the code.

3. **Accuracy**

   The other modification that can always be made in algorithms like these is accuracy. With more computing power, I could have a finer *grid*, a smaller step size in *Section 4.1.2,* more accurate hallway dimensions, and of course a better calculated optimum area.

4. **Better and more fitting reproduction operators**

   With more time to work on this project I could determine which of the various operators mentioned in Section 3 work best together to solve this problem, and explore more of the *crossover* operators from Sections 3.3.2 - 3.3.5.

5. **Variations on the *Moving Sofa Problem***

   All the work and results obtained are specifically for the Moser (or L-shaped) Hallway, but there are many variations[4] on this - the U-shaped Hallway, the S-shaped Hallway, the double L-shaped Hallway, etc. Of course this problem can be extended into the more practical 3 dimensions, but twisting and other rotations will have to be accounted for.

# 7 Thanks & Acknowledgement

Thanks to the TCD School of Mathematics for providing the resources and funding to allow me to do this project. Thanks to cplusplus.com, cppreference.com and stackoverflow.com for answering my questions on the details of coding. Thanks to Dr. Mike Peardon for being my supervisor for the duration of this project and giving me literature, sources of information and guidance, and thanks to Sunny for making the days off the best days of summer.

# 8 References


[1] Stephen Oman & Pádraig Cunningham. *Using Case Retrieval to Seed Genetic Algorithms*. Dept. of Computer Science, Trinity College, Dublin 2, Ireland.

[2] A. E. Eiben. *Genetic algorithms with multi-parent recombination*. PPSN III: Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature: 78–87, 1994.

[3] Joeseph L. Gerver. *On moving a sofa around a corner*. Geometriae Dedicata, Vol. 42, No. 3, pp 267-283, 1992.

[4] Kiyoshi Maruyama. *An approximation method for solving the sofa problem*. 1971.

[5] James D. Sebastian. *Problem 66-11*. SIAM Review, Vol. 12, No. 4, pp, 582-586, 1970.

[6] Neal R. Wagner. *The sofa problem*. The American Mathematical Monthly, Vol. 83, No. 3, pp. 188-189, Mar. 1976.

[7] Wikipedia. *Genetic Algorithms*.

[8] Wikipedia. *Gift wrapping algorithm*.

[9] Wikipedia. *Maze solving algorithm*.

# 9 Select code used in project

## 9.1 The Unconstrained Genetic Algorithm

For the unconstrained genetic algorithm, *sofagen9.cpp* approximates *the grid:*

```
#include <sys/time.h>
#include <algorithm>
#include <iostream>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <utility>
#include <random>
#include <vector>
#include <cmath>
#include <ctime>
using namespace std;
static bool seeded = false;

#define pi 3.14159265358979323846264338327950288419716939937510
ofstream output ("RESULTS3");
ofstream output2 ("RESULTS4");

typedef class Sofa
{
    public:
        int c;
        //number of corners
        vector <int> orgstring;
        //organism string
        double corn[200][2];
        //coordinates of corners
        double area;
        //area

        void modc(int);
        //modifies c
        void modarea(double);
        //modifies area
        void modorgstring(int);
        //modifies allele with int
        void modcorn(int, double, double);
        //modifies corner locations
        void clearorgstring();
        //clears organism string
        void eraseorgstring(int);
        //erases position int from orgstring
        void sortorgstring();
        //sorts orgstring

        int outnumc();
        //outputs number of corners
```

```
        double outarea();
        //outputs area
        int outorgstring(int);
        //outputs allele int
        double outcorn_x(int);
        //outputs x coord corner int
        double outcorn_y(int);
        //outputs y coord corner int
} Sofa;

void Sofa :: modc(int a)
{   c = a;
}

void Sofa :: modarea(double a)
{   area = a;
}

void Sofa :: modorgstring(int a)
{   orgstring.push_back(a);
}

void Sofa :: modcorn(int a, double b, double d)
{
    corn[a][0] = b;
    corn[a][1] = d;
}

void Sofa :: clearorgstring()
{   orgstring.clear();
}

void Sofa :: eraseorgstring(int a)
{   orgstring.erase(orgstring.begin()+a);
}

void Sofa :: sortorgstring()
{   sort(orgstring.begin(), orgstring.end());
}

int Sofa :: outnumc()
{   return c;
}

double Sofa :: outarea()
{   return area;
}

int Sofa :: outorgstring(int a)
{   return orgstring[a];
}

double Sofa :: outcorn_x(int a)
{   return corn[a][0];
```

```
}

double Sofa :: outcorn_y(int a)
{   return corn[a][1];
}

static void seed()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    srand(tv.tv_usec);
}

double ran()
{
    if(!seeded)
    {
        seed();
        seeded = true;
    }

    return ((double) rand() / (RAND_MAX + 1.0));
}

double point[1000];

double search(double a)
{
    double min = 10;
    int cnt;
    for(int i = 0; i < 1000; ++i)
    {
        if(abs(a - point[i]) < min)
        {
            cnt = i;
            min = abs(a - point[i]);
        }
    }

    return point[cnt];
}

void makepop(Sofa a[50])
{
    double pos, r2;
    for(int i = 0; i < 50; ++i)
    {
        a[i].modc(10);
        r2 = 0.25*ran();
        while(r2 == 0)
        {   r2 = 0.25*ran();
        }

        for(int j = 0; j < 10; ++j)
```

```
        {
            pos = ((0.5 - r2)*ran() + r2);
            a[i].modcorn(j, search(pos*cos((double) (2*pi/10)*j)),
                            search(pos*sin((double) (2*pi/10)*j)));
        }
    }
}

void makeorgstring(Sofa& p)
{
    int temp;
    for(int k = -500; k < 500; ++k)
    {
        for(int l = -500; l < 500; ++l)
        {
            for(int m = 0; m < p.outnumc(); ++m)
            {
                if(p.outcorn_x(m) == (double) 4*k/1000 && p.outcorn_y(m)
                    == (double) 4*l/1000)
                {
                    p.modorgstring(1000*(l+500)+(k+500));
                    break;
                }
            }
        }
    }
}

void makecorners(Sofa& p)
{
    int i = 0, w = 0;
    int temp1, temp2;

    for(int i = 0; i < p.outnumc(); ++i)
    {
        temp1 = ((p.outorgstring(i)) % 1000) - 500;
        temp2 = (p.outorgstring(i))/1000 - 500;
        p.modcorn(w, (double) temp1/250, (double) temp2/250);
        ++w;
    }

    int p1, q1 = 0;
    double cornx1[p.outnumc()], corny1[p.outnumc()];
    double angle1 = 0;
    vector < pair < double, int > > vec1;

    for(int j1 = 0; j1 < p.outnumc(); ++j1)
    {
        cornx1[j1] = p.outcorn_x(j1);
        corny1[j1] = p.outcorn_y(j1);
        angle1 = atan(corny1[j1]/cornx1[j1]);

        if(cornx1[j1] < 0 && corny1[j1] > 0)
        {   angle1 += pi;
```

```
        }

        if(cornx1[j1] < 0 && corny1[j1] <= 0)
        {   angle1 += pi;
        }

        if(cornx1[j1] >= 0 && corny1[j1] < 0)
        {   angle1 += 2*pi;
        }

        vec1.push_back(make_pair(angle1, j1));
    }

    sort(vec1.begin(), vec1.end());

    for(vector < pair < double, int > > :: iterator it1 = vec1.begin();
        it1 != vec1.end(); ++it1)
    {
        p1 = (*it1).second;
        p.modcorn(q1, cornx1[p1], corny1[p1]);
        ++q1;
    }

    double area = 0;
    for(int n = 0; n < p.outnumc()-1; ++n)
    {   area += abs(p.outcorn_x(n)*p.outcorn_y(n+1) - p.outcorn_x(n+1)
                    *p.outcorn_y(n));
    }

    p.modarea(0.5*area);
}

void equatesofas(Sofa a, Sofa &b)
{
    int count = 0;
    for(int i = 0; i < a.outnumc(); ++i)
    {
        b.modorgstring(a.outorgstring(i));
        ++count;
    }
    b.modc(count);
}

void mutation(Sofa &p)
{
    p.sortorgstring();

    int shift = p.outnumc()*ran();
    int flip = p.outorgstring(shift);
    int dir = 4*ran(), count = 0;

    p.eraseorgstring(shift);

    while(count < 4)
```

```
{
    if(dir == 0)
    {
        if(flip < 998999)
        {
            p.modorgstring(flip+1000);
            count = 5;
        }
        else
        {
            dir = 1;
            ++count;
        }
    }

    if(dir == 1)
    {
        if(flip > 1)
        {
            p.modorgstring(flip-1);
            count = 5;
        }
        else
        {
            dir = 2;
            ++count;
        }
    }

    if(dir == 2)
    {
        if(flip > 1001)
        {
            p.modorgstring(flip-1000);
            count = 5;
        }
        else
        {
            dir = 3;
            ++count;
        }
    }

    if(dir == 3)
    {
        if(flip < 999999)
        {
            p.modorgstring(flip+1);
            count = 5;
        }
        else
        {
            dir = 0;
            ++count;
```

```
            }
        }
    }

    p.sortorgstring();
}

void emptysofa(Sofa &p)
{   p.clearorgstring();
}

int main()
{
    clock_t start = clock();
    Sofa s[50], snew[50], schild[50];

    for(int i = -500; i < 500; ++i)
    {   point[i+500] = (double) 4*i/1000;
    }

    makepop(s);
    for(int j = 0; j < 50; ++j)
    {
        makeorgstring(s[j]);
        makecorners(s[j]);
    }

    cout<<"Finished making initial population."<<endl;

    int generation = 0;
    while(generation < 10000)
    {
        //**Selection**
        vector < pair < double, int > > sel;
        int choices[50] = {0};
        int choose1, choose2;
        for(int j = 0; j < 50; ++j)
        {
            bool finishedsel = false;
            while(!finishedsel)
            {
                for(int i = 0; i < 5; ++i)
                {
                    choose1 = 50*ran();
                    sel.push_back(make_pair(s[choose1].outarea(),
                                  choose1));
                }

                sort(sel.begin(), sel.end());
                choose2 = sel.back().second;
                if(choices[choose2] < 3)
                {
                    ++choices[choose2];
                    finishedsel = true;
```

```
                sel.clear();
            }
            sel.clear();
        }

        equatesofas(s[choose2], snew[j]);
        makecorners(snew[j]);
    }

    //**Crossover**
    int aa = 0;
    while(aa < 50)
    {
        int cho1 = 50*ran();
        int cho2 = 50*ran();

        while(snew[cho1].outarea() == snew[cho2].outarea())
        {
            cho1 = 50*ran();
            cho2 = 50*ran();
        }

        snew[cho1].sortorgstring();
        snew[cho2].sortorgstring();

        int cross1 = 1000000*ran();
        while(cross1 == 0 || cross1 > 999980)
        {   cross1 = 1000000*ran();
        }

        int cross2 = (1000000-cross1)*ran() + cross1;
        while(cross2 == cross1 || cross2 > 999995)
        {   cross2 = (1000000-cross1)*ran() + cross1;
        }

        int count1 = 0, count2 = 0;

        for(int i = 0; i < snew[cho1].outnumc(); ++i)
        {
            if(snew[cho1].outorgstring(i) <= cross1)
            {
                schild[aa].modorgstring(snew[cho1].outorgstring(i));
                ++count1;
            }
        }

        for(int j = 0; j < snew[cho2].outnumc(); ++j)
        {
            if(snew[cho2].outorgstring(j) > cross1 &&
                snew[cho2].outorgstring(j) <= cross2)
            {
                schild[aa].modorgstring(snew[cho2].outorgstring(j));
                ++count1;
            }
```

```
        }

        for(int k = 0; k < snew[cho1].outnumc(); ++k)
        {
            if(snew[cho1].outorgstring(k) > cross2)
            {
                schild[aa].modorgstring(snew[cho1].outorgstring(k));
                ++count1;
            }
        }

        for(int ii = 0; ii < snew[cho2].outnumc(); ++ii)
        {
            if(snew[cho2].outorgstring(ii) <= cross1)
            {
            schild[aa+1].modorgstring(snew[cho2].outorgstring(ii));
            ++count2;
            }
        }

        for(int jj = 0; jj < snew[cho1].outnumc(); ++jj)
        {
            if(snew[cho1].outorgstring(jj) > cross1 &&
                snew[cho1].outorgstring(jj) <= cross2)
            {
            schild[aa+1].modorgstring(snew[cho1].outorgstring(jj));
            ++count2;
            }
        }

        for(int kk = 0; kk < snew[cho2].outnumc(); ++kk)
        {
            if(snew[cho2].outorgstring(kk) > cross2)
            {
            schild[aa+1].modorgstring(snew[cho2].outorgstring(kk));
            ++count2;
            }
        }

        schild[aa].modc(count1);
        schild[aa+1].modc(count2);

        schild[aa].sortorgstring();
        schild[aa+1].sortorgstring();

        makecorners(schild[aa]);
        makecorners(schild[aa+1]);

        aa = aa+2;
    }

    //**Mutation**
    for(int hh = 0; hh < 50; ++hh)
    {
```

```
            mutation(schild[hh]);
            makecorners(schild[hh]);
        }

        //**New generation**
        for(int qq = 0; qq < 50; ++qq)
        {   emptysofa(s[qq]);
        }

        for(int ss = 0; ss < 50; ++ss)
        {   equatesofas(schild[ss], s[ss]);
        }

        for(int tt = 0; tt < 50; ++tt)
        {   makecorners(s[tt]);
        }

        for(int uu = 0; uu < 50; ++uu)
        {   emptysofa(snew[uu]);
        }

        for(int vv = 0; vv < 50; ++vv)
        {   emptysofa(schild[vv]);
        }

        if(generation % 10 == 0)
        {
            cout<<"End of generation "<<generation<<" time for
                    generation is "<<(double) (clock() - start)
                    /CLOCKS_PER_SEC<<" seconds"<<endl;

            for(int iiii = 0; iiii < 50; ++iiii)
            {  output2<<fixed<<setprecision(10)<<s[iiii].outarea()<<" ";
            }
            output2<<endl;

            for(int jjjj = 0; jjjj < 50; ++jjjj)
            {
                for(int kkkk = 0; kkkk < s[jjjj].outnumc(); ++kkkk)
                {   output<<s[jjjj].outcorn_x(kkkk)<<" "<<
                            s[jjjj].outcorn_y(kkkk)<<endl;
                }
                output<<endl;
            }
        }
        ++generation;
    }

    cout<<"Total program time: "<<(double) (clock() -
            start)/CLOCKS_PER_SEC<<" seconds"<<endl;
    return 0;
}
```

## 9.2 The Path Finding Algorithm

Building a path using a random walk; *sofagenpath4.cpp:*

```
#include <sys/time.h>
#include <functional>
#include <algorithm>
#include <iostream>
#include <iterator>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <utility>
#include <vector>
#include <cmath>
#include <ctime>
using namespace std;
static bool seeded = false;

#define pi 3.14159265358979323846264338327950288419716939937510
ofstream outputx ("RESULTS_Z");

typedef class Path
{
    public:
        int num;
        //length and number of points in path
        vector <double> initial;
        //starting point of path
        vector <int> orgstring;
        //organism string of path

        void modnum(int);
        //mod num
        void modorgstring(int);
        //adds int to orgstring
        void eraseorgstring();
        //erase element int from orgstring
        void modinitial(double, double, double);
        //modify inital point of path

        int outnum();
        //outputs num
        int outorgstring(int);
        //outputs allele int
        double outinitial(int);
        //outputs coordinate int of initial point of path
} Path;

void Path :: modnum(int a)
{   num = a;
}
```

```
void Path :: modorgstring(int a)
{   orgstring.push_back(a);
}

void Path :: eraseorgstring()
{   orgstring.pop_back();
}

void Path :: modinitial(double a, double b, double c)
{
    initial.push_back(a);
    initial.push_back(b);
    initial.push_back(c);
}

int Path :: outnum()
{   return num;
}

int Path :: outorgstring(int a)
{   return orgstring[a];
}

double Path :: outinitial(int a)
{   return initial[a];
}

static void seed()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    srand(tv.tv_usec);
}

double ran()
{
    if(!seeded)
    {
        seed();
        seeded = true;
    }

    return ((double) rand() / (RAND_MAX + 1.0));
}

bool hallway(double a[][2], int i)
{
    if(a[i][0] > 2.01 || a[i][1] > 2.01)
    {   return false;
    }

    if(a[i][0] < 0.99 && a[i][1] < 0.99)
    {   return false;
    }
```

```
    return true;
}

void rot(double b[][2], int n, double k)
{
    double j = 0;
    while(j < k)
    {
        for(int i = 0; i < n; ++i)
        {
            b[i][0] = (cos(2*pi/10000)*b[i][0] +
                       sin(2*pi/10000)*b[i][1]);
            b[i][1] = (cos(2*pi/10000)*b[i][1] -
                       sin(2*pi/10000)*b[i][0]);
        }
        j += 2*pi/10000;
    }
}

bool checks(double a1, double a2, double a3, double b[][2], int n)
{
    rot(b, n, a3);
    for(int i = 0; i < n; ++i)
    {
        b[i][0] = a1 + b[i][0];
        b[i][1] = a2 + b[i][1];
    }

    for(int j = 0; j < n; ++j)
    {
        if(!hallway(b, j))
        {   return false;
        }
    }
    return true;
}

void rein(double a[][2], double c[][2], int n)
{
    for(int i = 0; i<n; ++i)
    {
        a[i][0] = c[i][0];
        a[i][1] = c[i][1];
    }
}

void makeneigh(double a[], double b[], double c[])
{
    a[1] = a[0] + 0.001;
    b[1] = b[0] - 0.001;
    c[1] = c[0] + 0.001;

    a[2] = a[0] + 0.001;
```

```
    b[2] = b[0];
    c[2] = c[0] + 0.001;

    a[3] = a[0];
    b[3] = b[0] - 0.001;
    c[3] = c[0] + 0.001;

    a[4] = a[0] + 0.001;
    b[4] = b[0] - 0.001;
    c[4] = c[0];

    a[5] = a[0] + 0.001;
    b[5] = b[0];
    c[5] = c[0];

    a[6] = a[0];
    b[6] = b[0] - 0.001;
    c[6] = c[0];

    a[7] = a[0];
    b[7] = b[0];
    c[7] = c[0] + 0.001;
}

bool fits(double nx[], double ny[], double nz[], double b[][2],
            double in[][2], int n)
{
    nx[0] = 0;
    ny[0] = 1;
    nz[0] = 0;

    makeneigh(nx, ny, nz);
        rein(b, in, n);

        if (!checks(nx[0], ny[0], nz[0], b, n))
        {
                rein(b, in, n);
                while (!checks(nx[0], ny[0], nz[0], b, n))
                {
                        if (ny[0] > 2)
                        {
                                cout << "DOESN'T FIT" << endl;
                                rein(b, in, n);
                                return false;
                        }

                        if (nz[0] > 2*pi)
                        {
                                ny[0] += 0.1;
                                nz[0] = 0;
                                rein(b, in, n);
                        }
                        else
                        {
```

```
                            nz[0] += 0.1;
                            rein(b, in, n);
                        }
                }
                return true;
        }
        else
        {       return true;
        }
}

vector <double> modwalk(Path &p, double nx[], double ny[], double nz[])
{
    nx[0] = p.outinitial(0);
    ny[0] = p.outinitial(1);
    nz[0] = p.outinitial(2);
    int temp;
    vector <double> a;
    a.clear();

    for(int i = 0; i < p.outnum(); ++i)
    {
        temp = (p.outorgstring(i));
        makeneigh(nx, ny, nz);
        nx[0] = nx[temp];
        ny[0] = ny[temp];
        nz[0] = nz[temp];
    }

    a.push_back(nx[0]);
    a.push_back(ny[0]);
    a.push_back(nz[0]);
    a.push_back(p.outnum());

    return a;
}

int notdir[20];

bool makewalk(Path &p, int r, double nx[], double ny[], double nz[],
                double b[][2], double in[][2], int n, bool first)
{
    bool notfinished = true;
    int dir, count, godir = 5, countdir = 0;

    if(first)
    {
        nx[0] = p.outinitial(0);
        ny[0] = p.outinitial(1);
        nz[0] = p.outinitial(2);
        count = 0;
        notdir[r] = 0;
    }
    else
```

```
{
    nx[0] = modwalk(p, nx, ny, nz)[0];
    ny[0] = modwalk(p, nx, ny, nz)[1];
    nz[0] = modwalk(p, nx, ny, nz)[2];
    count = modwalk(p, nx, ny, nz)[3];
}

while(notfinished)
{
    makeneigh(nx, ny, nz);
    rein(b, in, n);
    notfinished = checks(nx[godir], ny[godir], nz[godir], b, n);
    if(notfinished)
    {
        nx[0] = nx[godir];
        ny[0] = ny[godir];
        nz[0] = nz[godir];
        p.modorgstring(godir);
        ++count;

        if(ny[0] < -0.25)
        {
            p.modnum(count);
            cout<<"DONE"<<endl;
            return true;
        }
    }
    else
    {
        dir = 8*ran();
        while(dir == 0 || dir == notdir[r] || dir == godir)
        {   dir = 8*ran();
        }

        makeneigh(nx, ny, nz);
        rein(b, in, n);
        notfinished = checks(nx[dir], ny[dir], nz[dir], b, n);
        if(notfinished)
        {
            nx[0] = nx[dir];
            ny[0] = ny[dir];
            nz[0] = nz[dir];
            p.modorgstring(dir);

            if(dir == 7)
            {
                if(countdir < 100)
                {
                    godir = dir;
                    ++countdir;
                }
                else
                {   godir = 5;
                }
```

```
                }
                else
                {   godir = dir;
                }

                notdir[r] = 0;
                ++count;

                if(ny[0] < -0.25)
                {
                    p.modnum(count);
                    cout<<"DONE"<<endl;
                    return true;
                }
            }
            else
            {   notdir[r] = dir;
            }
        }
    }

    p.modnum(count);
    return false;
}

void printpath(Path p, double nx[], double ny[], double nz[])
{
    int temp;
    nx[0] = p.outinitial(0);
    ny[0] = p.outinitial(1);
    nz[0] = p.outinitial(2);

    for(int j = 0; j < p.outnum(); ++j)
    {
        makeneigh(nx, ny, nz);
        temp = p.outorgstring(j);
        outputx<<fixed<<setprecision(10)<<nx[temp]<<" "<<ny[temp]
               <<" "<<nz[temp]<<endl;
        nx[0] = nx[temp];
        ny[0] = ny[temp];
        nz[0] = nz[temp];
    }
    outputx<<endl;
}

bool atdeadend(Path p, int r, double nx[], double ny[], double nz[],
               double b[][2], double in[][2], int n)
{
    nx[0] = modwalk(p, nx, ny, nz)[0];
    ny[0] = modwalk(p, nx, ny, nz)[1];
    nz[0] = modwalk(p, nx, ny, nz)[2];
    makeneigh(nx, ny, nz);
    int count = 0;
```

```
    rein(b, in, n);
    if(!checks(nx[1], ny[1], nz[1], b, n))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(b, in, n);
    if(!checks(nx[2], ny[2], nz[2], b, n))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(b, in, n);
    if(!checks(nx[3], ny[3], nz[3], b, n))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(b, in, n);
    if(!checks(nx[4], ny[4], nz[4], b, n))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(b, in, n);
    if(!checks(nx[5], ny[5], nz[5], b, n))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(b, in, n);
    if(!checks(nx[6], ny[6], nz[6], b, n))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(b, in, n);
    if(!checks(nx[7], ny[7], nz[7], b, n))
    {   ++count;
    }
    else
    {   return false;
```

```
    }

    if(count == 7)
    {   return true;
    }
}

int main()
{
    int c, generation = 0;
    cin>>c;
    double s[c][2], in[c][2];

    for(int iii = 0; iii < c; ++iii)
    {
        cin>>s[iii][0]>>s[iii][1];
        in[iii][0] = s[iii][0];
        in[iii][1] = s[iii][1];
    }

    double neighx[8], neighy[8], neighz[8], tneighx[8], tneighy[8],
           tneighz[8];
    Path p[50];
    bool start = true;
    bool finished = false;

    clock_t startt = clock();

    if(!fits(tneighx, tneighy, tneighz, s, in, c))
    {
        cout<<"Sofa doesn't fit at start of hallway - exiting
              program"<<endl;
        return -1;
    }

    for(int i = 0; i < 50; ++i)
    {   p[i].modinitial(tneighx[0], tneighy[0], tneighz[0]);
    }

    while(!finished)
    {
        for(int r = 0; r < 50; ++r)
        {
            if(makewalk(p[r], r, neighx, neighy, neighz, s, in, c,
                start))
            {
                finished = true;
                printpath(p[r], neighx, neighy, neighz);
                cout<<"Finished on generation "<<generation<<" with time
                    "<<(double) (clock() - startt)/CLOCKS_PER_SEC<<"
                    seconds"<<endl;
                break;
            }
        }
```

```
        start = false;

        if(!finished)
        {
            if(generation % 10 == 0)
            {   cout<<"GENERATION "<<generation<<endl;
            }

            if(generation % 10 == 0)
            {
                int dedend = 0;
                for(int t = 0; t < 50; ++t)
                {
                    if(atdeadend(p[t], t, neighx, neighy, neighz, s, in,
                        c))
                    {   ++dedend;
                    }
                }

                if(dedend == 50)
                {
                    cout<<"DEAD END"<<endl;
                    for(int i = 0; i < 50; ++i)
                    {   printpath(p[i], neighx, neighy, neighz);
                    }
                    finished = true;
                }
            }

            ++generation;
        }
    }

    return 0;
}
```

## 9.3 The Constrained Genetic Algorithm

Here I seed the initial population and use the DAM constraint; *(sofapath8.cpp)*

```
#include <sys/time.h>
#include <functional>
#include <algorithm>
#include <iostream>
#include <iterator>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <utility>
#include <random>
#include <vector>
#include <cmath>
```

```
#include <ctime>
using namespace std;
static bool seeded = false;

#define pi 3.14159265358979323846264338327950288419716939937510
ofstream output ("RESULTS");
ofstream output2 ("RESULTS2");

typedef class Sofa
{
    public:
        int c;
        //number of corners
        vector <int> orgstring;
        //organism string
        double corn[500][2], in[500][2];
        //coordinates of corners & initial coordinates
        double area, cont;
        //area, cont

        void modc(int);
        //modifies c
        void modcont(double);
        //modifies cont
        void modarea(double);
        //modifies area
        void modorgstring(int);
        //modifies allele with int
        void modcorn(int, double, double);
        //modifies corner locations
        void modin(int, double, double);
        //modifies initial corner locations
        void clearorgstring();
        //clears organism string
        void eraseorgstring(int);
        //erases position int from orgstring
        void sortorgstring();
        //sorts orgstring

        int outnumc();
        //outputs number of corners
        double outcont();
        //outputs cont
        double outarea();
        //outputs area
        int outorgstring(int);
        //outputs allele int
        double outcorn_x(int);
        //outputs x coord corner int
        double outcorn_y(int);
        //outputs y coord corner int
        double outin_x(int);
        //outputs initial x coord int
        double outin_y(int);
```

```
        //outputs initial y coord int
} Sofa;

void Sofa :: modc(int a)
{   c = a;
}

void Sofa :: modcont(double a)
{   cont = a;
}

void Sofa :: modarea(double a)
{   area = a;
}

void Sofa :: modorgstring(int a)
{   orgstring.push_back(a);
}

void Sofa :: modcorn(int a, double b, double d)
{
    corn[a][0] = b;
    corn[a][1] = d;
}

void Sofa :: modin(int a, double b, double d)
{
    in[a][0] = b;
    in[a][1] = d;
}

void Sofa :: clearorgstring()
{   orgstring.clear();
}

void Sofa :: eraseorgstring(int a)
{   orgstring.erase(orgstring.begin()+a);
}

void Sofa :: sortorgstring()
{   sort(orgstring.begin(), orgstring.end());
}

int Sofa :: outnumc()
{   return c;
}

double Sofa :: outcont()
{   return cont;
}

double Sofa :: outarea()
{   return area;
}
```

```
int Sofa :: outorgstring(int a)
{   return orgstring[a];
}

double Sofa :: outcorn_x(int a)
{   return corn[a][0];
}

double Sofa :: outcorn_y(int a)
{   return corn[a][1];
}

double Sofa :: outin_x(int a)
{   return in[a][0];
}

double Sofa :: outin_y(int a)
{   return in[a][1];
}

typedef class Path
{
    public:
        int num;
        //length and number of points in path
        vector <double> initial;
        //starting point of path
        vector <int> orgstring;
        //organism string of path

        void modnum(int);
        //mod num
        void modorgstring(int);
        //adds int to orgstring
        void eraseorgstring();
        //erase element int from orgstring
        void modinitial(double, double, double);
        //modify inital point of path
        void clearpath();
        //clears path information

        int outnum();
        //outputs num
        int outorgstring(int);
        //outputs allele int
        double outinitial(int);
        //outputs coordinate int of initial point of path
} Path;

void Path :: modnum(int a)
{   num = a;
}
```

```
void Path :: modorgstring(int a)
{   orgstring.push_back(a);
}

void Path :: eraseorgstring()
{   orgstring.pop_back();
}

void Path :: modinitial(double a, double b, double c)
{
    initial.push_back(a);
    initial.push_back(b);
    initial.push_back(c);
}

void Path :: clearpath()
{
    num = 0;
    initial.clear();
    orgstring.clear();
}

int Path :: outnum()
{   return num;
}

int Path :: outorgstring(int a)
{   return orgstring[a];
}

double Path :: outinitial(int a)
{   return initial[a];
}

static void seed()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    srand(tv.tv_usec);
}

double ran()
{
    if(!seeded)
    {
        seed();
        seeded = true;
    }

    return ((double) rand() / (RAND_MAX + 1.0));
}

double point[10000];
```

```
double search(double a)
{
    double min = 10;
    int cnt;
    for(int i = 0; i < 10000; ++i)
    {
        if(abs(a - point[i]) < min)
        {
            cnt = i;
            min = abs(a - point[i]);
        }
    }

    return point[cnt];
}

void makeorgstring(Sofa& p)
{
    int temp;
    for(int k = -5000; k < 5000; ++k)
    {
        for(int l = -5000; l < 5000; ++l)
        {
            for(int m = 0; m < p.outnumc(); ++m)
            {
                if(p.outcorn_x(m) == (double) 4*k/10000 &&
                    p.outcorn_y(m) == (double) 4*l/10000)
                {
                    p.modorgstring(10000*(l+5000)+(k+5000));
                    break;
                }
            }
        }
    }
}

void rot(Sofa &s, double k);
void createin(Sofa &s);
void rein(Sofa &s);

double dist(Sofa p, int i, int j)
{   return ((p.outcorn_y(i) - p.outcorn_y(j))*(p.outcorn_y(i) -
            p.outcorn_y(j)) + (p.outcorn_x(i) - p.outcorn_x(j))
            *(p.outcorn_x(i) - p.outcorn_x(j)));
}

void makecorners(Sofa& p)
{
    int w = 0;
    int temp1, temp2;
    p.sortorgstring();

    for(int i = 0; i < p.outnumc(); ++i)
    {
```

```
        temp1 = ((p.outorgstring(i)) % 10000) - 5000;
        temp2 = (p.outorgstring(i))/10000 - 5000;
        p.modcorn(w, (double) temp1/2500, (double) temp2/2500);
        ++w;
    }

    createin(p);

    double miny = 100;
    int county = 0;

    for(int i = 0; i < p.outnumc(); ++i)
    {
        if(p.outcorn_y(i) <= miny)
        {
            if(p.outcorn_y(i) == miny)
            {
                if(p.outcorn_x(i) < p.outcorn_x(county))
                {   county = i;
                }
            }
            else
            {
                miny = p.outcorn_y(i);
                county = i;
            }
        }
    }

    vector <int> convexhull;
    int temp2corn, tempcorn = county;
    double angle, totangle = 0, minangle;
    double cornx, corny;
    bool finished = false;

    while(!finished)
    {
        convexhull.push_back(tempcorn);
        minangle = 7;

        for(int j = 0; j < p.outnumc(); ++j)
        {
            if(j != tempcorn)
            {
                cornx = p.outcorn_x(j) - p.outcorn_x(tempcorn);
                corny = p.outcorn_y(j) - p.outcorn_y(tempcorn);

                if(cornx == 0)
                {
                    if(corny > 0)
                    {   angle = pi/2;
                    }
                    if(corny < 0)
                    {   angle = 3*pi/2;
```

```
                }
            }
            else
            {   angle = atan(corny/cornx);
            }

            if(cornx < 0)
            {   angle += pi;
            }

            if(cornx > 0)
            {
                if(corny < 0)
                {   angle += 2*pi;
                }
                if(corny == 0)
                {   angle = 0;
                }
            }

            if(angle <= minangle)
            {
                if(angle < minangle)
                {
                    minangle = angle;
                    temp2corn = j;
                }
                else
                {
                    if(dist(p, tempcorn, temp2corn) <
                        dist(p, tempcorn, j))
                    {   temp2corn = j;
                    }
                }
            }

        }
    }

    tempcorn = temp2corn;

    rein(p);
    totangle += minangle;
    rot(p, totangle);

    if(tempcorn == county)
    {   finished = true;
    }
}

rein(p);

double area = 0;
for(int n = 0; n < convexhull.size()-1; ++n)
```

```
    {   area += abs(p.outcorn_x(convexhull[n])*
        p.outcorn_y(convexhull[n+1]) - p.outcorn_x(convexhull[n+1])
        *p.outcorn_y(convexhull[n]));
    }

    double tempcont = p.outcont();
    p.modarea(0.5*area*tempcont);
    convexhull.clear();
}

void equatesofas(Sofa a, Sofa &b)
{
    int count = 0;
    for(int i = 0; i < a.outnumc(); ++i)
    {
        b.modorgstring(a.outorgstring(i));
        ++count;
    }
    b.modc(count);

    double conta = a.outcont();
    b.modcont(conta);

    double area = a.outarea();
    b.modarea(area);
}

void mutation(Sofa &p)
{
    p.sortorgstring();

    int shift = p.outnumc()*ran();
    int flip = p.outorgstring(shift);
    int dir = 4*ran(), count = 0;

    p.eraseorgstring(shift);

    while(count < 4)
    {
        if(dir == 0)
        {
            if(flip < 99998999)
            {
                p.modorgstring(flip+100000);
                count = 5;
            }
            else
            {
                dir = 1;
                ++count;
            }
        }

        if(dir == 1)
```

```
        {
            if(flip > 1)
            {
                p.modorgstring(flip -1);
                count = 5;
            }
            else
            {
                dir = 2;
                ++count;
            }
        }

        if(dir == 2)
        {
            if(flip > 100001)
            {
                p.modorgstring(flip -100000);
                count = 5;
            }
            else
            {
                dir = 3;
                ++count;
            }
        }

        if(dir == 3)
        {
            if(flip < 99999999)
            {
                p.modorgstring(flip +1);
                count = 5;
            }
            else
            {
                dir = 0;
                ++count;
            }
        }
    }

    p.sortorgstring();
}

void emptysofa(Sofa &p)
{   p.clearorgstring();
}

void createin(Sofa &s)
{
    for(int i = 0; i < s.outnumc(); ++i)
    {   s.modin(i, s.outcorn_x(i), s.outcorn_y(i));
    }
```

```
}

bool hallway(Sofa s, int i)
{
    if(s.outcorn_x(i) > 2.01 || s.outcorn_y(i) > 2.01)
    {   return false;
    }

    if(s.outcorn_x(i) < 0.99 && s.outcorn_y(i) < 0.99)
    {   return false;
    }

    return true;
}

void rot(Sofa &s, double k)
{
    double temp1, temp2, j = 0;

    while(j < k)
    {
        for(int i = 0; i < s.outnumc(); ++i)
        {
            temp1 = s.outcorn_x(i);
            temp2 = s.outcorn_y(i);
            s.modcorn(i,(cos(2*pi/10000)*temp1 + sin(2*pi/10000)*temp2),
                    (cos(2*pi/10000)*temp2 - sin(2*pi/10000)*temp1));
        }
        j += 2*pi/10000;
    }
}

bool checks(double a1, double a2, double a3, Sofa s)
{
    rot(s, a3);

    double temp1, temp2;
    for(int i = 0; i < s.outnumc(); ++i)
    {
        temp1 = s.outcorn_x(i);
        temp2 = s.outcorn_y(i);
        s.modcorn(i, (temp1 + a1), (temp2 + a2));
    }

    for(int j = 0; j < s.outnumc(); ++j)
    {
        if(!hallway(s, j))
        {   return false;
        }
    }
    return true;
}

void rein(Sofa &s)
```

```
{
    for(int i = 0; i < s.outnumc(); ++i)
    {   s.modcorn(i, s.outin_x(i), s.outin_y(i));
    }
}

void makeneigh(double a[], double b[], double c[])
{
    a[1] = a[0] + 0.001;
    b[1] = b[0] - 0.001;
    c[1] = c[0] + 0.001;

    a[2] = a[0] + 0.001;
    b[2] = b[0];
    c[2] = c[0] + 0.001;

    a[3] = a[0];
    b[3] = b[0] - 0.001;
    c[3] = c[0] + 0.001;

    a[4] = a[0] + 0.001;
    b[4] = b[0] - 0.001;
    c[4] = c[0];

    a[5] = a[0] + 0.001;
    b[5] = b[0];
    c[5] = c[0];

    a[6] = a[0];
    b[6] = b[0] - 0.001;
    c[6] = c[0];

    a[7] = a[0];
    b[7] = b[0];
    c[7] = c[0] + 0.001;
}

bool fits(double nx[], double ny[], double nz[], Sofa s)
{
    nx[0] = 0;
    ny[0] = 1;
    nz[0] = 0;

        rein(s);
        if (!checks(nx[0], ny[0], nz[0], s))
        {
                rein(s);
                while (!checks(nx[0], ny[0], nz[0], s))
                {
                        if (ny[0] > 2)
                        {
                                rein(s);
                                return false;
                        }
```

```
                        if (nz[0] > 2*pi)
                        {
                                ny[0] += 0.1;
                                nz[0] = 0;
                                rein(s);
                        }
                        else
                        {
                            nz[0] += 0.01;
                            rein(s);
                        }
                }
                return true;
        }
        else
        {       return true;
        }
}

vector <double> modwalk(Path &p, double nx[], double ny[], double nz[])
{
    nx[0] = p.outinitial(0);
    ny[0] = p.outinitial(1);
    nz[0] = p.outinitial(2);
    int temp;
    vector <double> a;
    a.clear();

    for(int i = 0; i < p.outnum(); ++i)
    {
        temp = (p.outorgstring(i));
        makeneigh(nx, ny, nz);
        nx[0] = nx[temp];
        ny[0] = ny[temp];
        nz[0] = nz[temp];
    }

    a.push_back(nx[0]);
    a.push_back(ny[0]);
    a.push_back(nz[0]);
    a.push_back(p.outnum());

    return a;
}

int notdir[50];

bool makewalk(Path &p, int r, double nx[], double ny[], double nz[],
                Sofa s, bool first)
{
    bool notfinished = true;
    int dir, count, godir = 5, countdir = 0;
```

```
if(first)
{
    nx[0] = p.outinitial(0);
    ny[0] = p.outinitial(1);
    nz[0] = p.outinitial(2);
    count = 0;
    notdir[r] = 0;
}
else
{
    nx[0] = modwalk(p, nx, ny, nz)[0];
    ny[0] = modwalk(p, nx, ny, nz)[1];
    nz[0] = modwalk(p, nx, ny, nz)[2];
    count = modwalk(p, nx, ny, nz)[3];
}

while(notfinished)
{
    makeneigh(nx, ny, nz);
    rein(s);
    notfinished = checks(nx[godir], ny[godir], nz[godir], s);
    if(notfinished)
    {
        nx[0] = nx[godir];
        ny[0] = ny[godir];
        nz[0] = nz[godir];
        p.modorgstring(godir);
        ++count;

        if(ny[0] < -0.25)
        {
            p.modnum(count);
            return true;
        }
    }
    else
    {
        dir = 8*ran();
        while(dir == 0 || dir == notdir[r] || dir == godir)
        {   dir = 8*ran();
        }

        makeneigh(nx, ny, nz);
        rein(s);
        notfinished = checks(nx[dir], ny[dir], nz[dir], s);
        if(notfinished)
        {
            nx[0] = nx[dir];
            ny[0] = ny[dir];
            nz[0] = nz[dir];
            p.modorgstring(dir);

            if(dir == 7)
            {
```

```
                    if(countdir < 100)
                    {
                        godir = dir;
                        ++countdir;
                    }
                    else
                    {   godir = 5;
                    }
                }
                else
                {   godir = dir;
                }

                notdir[r] = 0;
                ++count;

                if(ny[0] < -0.25)
                {
                    p.modnum(count);
                    return true;
                }
            }
            else
            {   notdir[r] = dir;
            }
        }
    }

    p.modnum(count);
    return false;
}

bool atdeadend(Path p, int r, double nx[], double ny[], double nz[],
               Sofa s)
{
    nx[0] = modwalk(p, nx, ny, nz)[0];
    ny[0] = modwalk(p, nx, ny, nz)[1];
    nz[0] = modwalk(p, nx, ny, nz)[2];
    makeneigh(nx, ny, nz);
    int count = 0;

    rein(s);
    if(!checks(nx[1], ny[1], nz[1], s))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(s);
    if(!checks(nx[2], ny[2], nz[2], s))
    {   ++count;
    }
    else
```

```
    {   return false;
    }

    rein(s);
    if(!checks(nx[3], ny[3], nz[3], s))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(s);
    if(!checks(nx[4], ny[4], nz[4], s))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(s);
    if(!checks(nx[5], ny[5], nz[5], s))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(s);
    if(!checks(nx[6], ny[6], nz[6], s))
    {   ++count;
    }
    else
    {   return false;
    }

    rein(s);
    if(!checks(nx[7], ny[7], nz[7], s))
    {   ++count;
    }
    else
    {   return false;
    }

    if(count == 7)
    {   return true;
    }
}

int main()
{
    clock_t start = clock();

    Sofa s[10], snew[10], schild[10];
    double neighx[8], neighy[8], neighz[8], tneighx[8], tneighy[8],
```

```
        tneighz[8];
Path p[50];
bool starter = true;
bool finished = false;

for(int i = -5000; i < 5000; ++i)
{   point[i+5000] = (double) 4*i/10000;
}

int numberofcorn;
double xxx, yyy;

cin>>numberofcorn;
for(int l = 0; l < 10; ++l)
{
    s[l].modc(numberofcorn);
    s[l].modcont(1.0);
    s[l].modarea(0);
}

double tempsearch1, tempsearch2;
for(int i = 0; i < numberofcorn; ++i)
{
    cin>>xxx>>yyy;
    tempsearch1 = search(xxx);
    tempsearch2 = search(yyy);

    for(int k = 0; k < 10; ++k)
    {   s[k].modcorn(i, tempsearch1, tempsearch2);
    }
}

makeorgstring(s[0]);
makecorners(s[0]);

for(int j = 1; j < 10; ++j)
{
    equatesofas(s[0], s[j]);
    makecorners(s[j]);
}

cout<<"Finished making initial population."<<endl;

int generation2, generation = 0;
while(generation < 10000)
{
    clock_t start2 = clock();

    //**Selection**
    vector < pair < double, int > > sel;
    int choices[10] = {0};
    int choose1, choose2;
    for(int j = 0; j < 10; ++j)
    {
```

```
        bool finishedsel = false;
        while(!finishedsel)
        {
            for(int i = 0; i < 3; ++i)
            {
                choose1 = 10*ran();
                sel.push_back(make_pair(s[choose1].outarea(),
                                        choose1));
            }

            sort(sel.begin(), sel.end());
            choose2 = sel.back().second;
            if(choices[choose2] < 2)
            {
                ++choices[choose2];
                finishedsel = true;
                sel.clear();
            }
            sel.clear();
        }

        equatesofas(s[choose2], snew[j]);
        makecorners(snew[j]);
    }

    //**Crossover**
    int aa = 0;
    while(aa < 10)
    {
        int cho1 = 10*ran();
        int cho2 = 10*ran();

        while(cho1 == cho2)
        {
            cho1 = 10*ran();
            cho2 = 10*ran();
        }

        snew[cho1].sortorgstring();
        snew[cho2].sortorgstring();

        int cross1 = 100000000*ran();
        while(cross1 == 0 || cross1 > 99999980)
        {   cross1 = 100000000*ran();
        }

        int cross2 = (100000000-cross1)*ran() + cross1;
        while(cross2 == cross1 || cross2 > 99999995)
        {   cross2 = (100000000-cross1)*ran() + cross1;
        }

        int count1 = 0, count2 = 0;

        for(int i = 0; i < snew[cho1].outnumc(); ++i)
```

```
{
    if(snew[cho1].outorgstring(i) <= cross1)
    {
        schild[aa].modorgstring(snew[cho1].outorgstring(i));
        ++count1;
    }
}

for(int j = 0; j < snew[cho2].outnumc(); ++j)
{
    if(snew[cho2].outorgstring(j) > cross1 &&
        snew[cho2].outorgstring(j) <= cross2)
    {
        schild[aa].modorgstring(snew[cho2].outorgstring(j));
        ++count1;
    }
}

for(int k = 0; k < snew[cho1].outnumc(); ++k)
{
    if(snew[cho1].outorgstring(k) > cross2)
    {
        schild[aa].modorgstring(snew[cho1].outorgstring(k));
        ++count1;
    }
}

for(int ii = 0; ii < snew[cho2].outnumc(); ++ii)
{
    if(snew[cho2].outorgstring(ii) <= cross1)
    {
    schild[aa+1].modorgstring(snew[cho2].outorgstring(ii));
    ++count2;
    }
}

for(int jj = 0; jj < snew[cho1].outnumc(); ++jj)
{
    if(snew[cho1].outorgstring(jj) > cross1 &&
        snew[cho1].outorgstring(jj) <= cross2)
    {
    schild[aa+1].modorgstring(snew[cho1].outorgstring(jj));
    ++count2;
    }
}

for(int kk = 0; kk < snew[cho2].outnumc(); ++kk)
{
    if(snew[cho2].outorgstring(kk) > cross2)
    {
    schild[aa+1].modorgstring(snew[cho2].outorgstring(kk));
    ++count2;
    }
}
```

```
        schild[aa].modc(count1);
        schild[aa+1].modc(count2);
        schild[aa].modcont(1);
        schild[aa+1].modcont(1);

        schild[aa].sortorgstring();
        schild[aa+1].sortorgstring();

        makecorners(schild[aa]);
        makecorners(schild[aa+1]);

        aa = aa+2;
    }

    //**Mutation**
    for(int hh = 0; hh < 10; ++hh)
    {
        mutation(schild[hh]);
        makecorners(schild[hh]);
    }

    //**Test a path**
    int dedend = 0;
    double tempcont;
    int numsofas = 10;
    for(int d = 0; d < 10; ++d)
    {
        createin(schild[d]);
        bool willitstart = true;
        cout<<d<<endl;

        if(!fits(tneighx, tneighy, tneighz, schild[d]))
        {
            tempcont = schild[d].outcont();
            schild[d].modcont(0.5*tempcont);
            willitstart = false;
            cout<<"Sofa "<<d<<" doesn't fit at start"<<endl;
        }

        if(willitstart)
        {
            for(int i = 0; i < 50; ++i)
            {   p[i].modinitial(tneighx[0], tneighy[0], tneighz[0]);
            }

            generation2 = 0;
            starter = true;
            finished = false;

            while(!finished)
            {
                for(int r = 0; r < 50; ++r)
                {
```

```
                    if(makewalk(p[r], r, neighx, neighy, neighz,
                        schild[d], starter))
                    {
                        finished = true;
                        break;
                    }
                }
                starter = false;

                if(!finished)
                {
                    if(generation2 % 10 == 0)
                    {
                        dedend = 0;
                        for(int t = 0; t < 50; ++t)
                        {
                            if(atdeadend(p[t], t, neighx, neighy,
                                neighz, schild[d]))
                            {   ++dedend;
                            }
                        }

                        if(dedend == 50)
                        {
                            tempcont = schild[d].outcont();
                            schild[d].modcont(0.5*tempcont);
                            --numsofas;
                            cout<<"Sofa "<<d<<" doesn't fit around
                                hallway"<<endl;
                            finished = true;
                        }
                    }
                    ++generation2;
                }
            }

            if(dedend != 50)
            {
                tempcont = schild[d].outcont();
                schild[d].modcont(2*tempcont);
            }

            for(int ii = 0; ii < 50; ++ii)
            {   p[ii].clearpath();
            }
        }
    }

    if(numsofas < 2)
    {
        cout<<"TOO FEW SOFAS - finished on generation
            "<<generation<<endl;
        return -1;
    }
```

```
        else
        {   cout<<numsofas<<endl;
        }

        //**New generation**
        for(int qq = 0; qq < 10; ++qq)
        {   emptysofa(s[qq]);
        }

        for(int ss = 0; ss < 10; ++ss)
        {   equatesofas(schild[ss], s[ss]);
        }

        for(int tt = 0; tt < 10; ++tt)
        {   makecorners(s[tt]);
        }

        for(int uu = 0; uu < 10; ++uu)
        {   emptysofa(snew[uu]);
        }

        for(int vv = 0; vv < 10; ++vv)
        {   emptysofa(schild[vv]);
        }

        cout<<"End of gen "<<generation<<" gen time is "<<(double)
            (clock() - start2)/CLOCKS_PER_SEC<<" secs, "<<(double)
            (clock() - start)/CLOCKS_PER_SEC<<" secs overall"<<endl;

        if(generation % 10 == 0)
        {
            for(int iiii = 0; iiii < 10; ++iiii)
            {
            output2<<fixed<<setprecision(10)<<s[iiii].outarea()<<" ";
            }
            output2<<endl;

            for(int jjjj = 0; jjjj < 10; ++jjjj)
            {
                for(int kkkk = 0; kkkk < s[jjjj].outnumc(); ++kkkk)
                {   output<<s[jjjj].outcorn_x(kkkk)<<" "<<
                          s[jjjj].outcorn_y(kkkk)<<endl;
                }
                output<<endl;
            }
        }

        ++generation;
    }

    return 0;
}
```