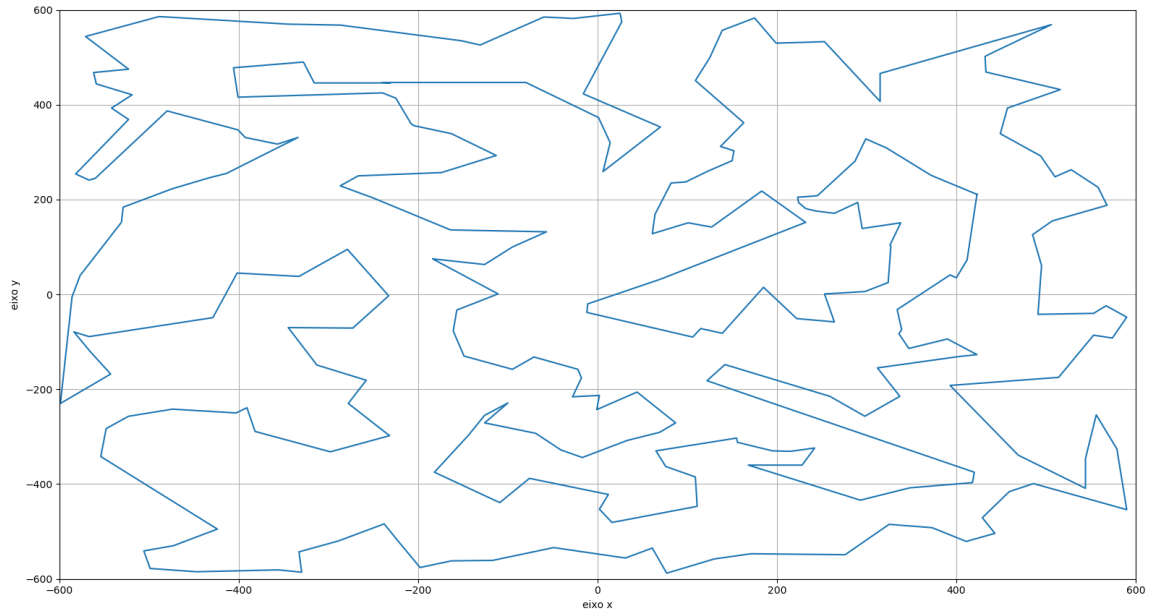
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

# Traveling Salesman Problem




## Inteligência Artificial (CC2006)

2020/2021

Duarte Alves - up201805437
Manuel Sá - up201805273

# Introdução

Problemas de cobrimento de nós têm aplicações práticas no dia a dia da sociedade. Um camião do lixo, por exemplo, precisa traçar um itinerário para apanhar os sacos de lixo sem
repetir a passagem pelo ponto já visitado e depois levar os resíduos para o depósito. Quanto menor o caminho traçado, menos desperdício de recursos e tempo haverá para se executar a tarefa.
O objetivo deste trabalho é implementar diferentes algoritmos e métodos para resolver este problema, bem como discutir os resultados obtidos através desses mesmos algoritmos e métodos. Os algoritmos implementados neste trabalho foram:

-Criação de uma permutação aleatória;
-Nearest neighbor;
-Hill Climbing ( com 4 métodos diferentes de aplicar o algoritmo);
-Simulated annealing.;
-Ant Colony.

# Estruturação do programa

O nosso programa é constituído pelos seguintes ficheiros:

-libraries.h - ficheiro com as bibliotecas necessárias como a math.h;

-Map.cpp e Map.h - Os ficheiros responsáveis por conter as classes necessárias como a de criação dos gráficos ou dos pontos;

-search.cpp e search.h - Os ficheiros que contém os algoritmos de pesquisa nearest neighbor, hill climbing e ant colony.

-util.cpp e util.h - Ficheiros que contém todas as funções auxiliares necessárias para alguns cálculos e prints. Alguns exemplos são: cálculo do cross product, verificação de interseções, cálculo do perímetro, entre outros.

-main.cpp-ficheiro principal, onde é recebido o input e selecionado os métodos a utilizar.

-myplot.py - script de python que nos permite visualizar graficamente as soluções criadas pelo programa;

# Classes de Ponto e Mapa

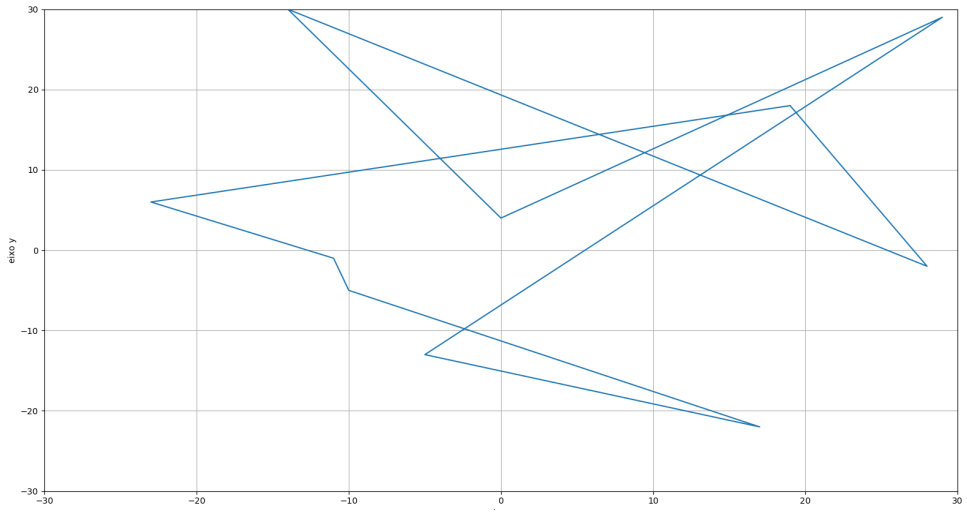
A classe Point representa o ponto gerado no gráfico. Cada ponto possui uma coordenada x, outra y e um booleano que verifica se aquele ponto já foi visitado. As funções associadas são o construtor, uma função que o printa e outra que verifica se o ponto em questão é o mesmo que outro.
Quanto à classe Map, possui uma variavel tamanho. Também contém três listas: uma que guarda todos os pontos, outra os pontos não visitados e outra que guarda a ordem pelo qual os pontos são percorridos. As funçõ̃es associadas printam os pontos dentro do mapa e verificam se um ponto existe.

```
class Point{
    public:
        int x;
        int y;

        bool visited;

        Point(int xcoord, int ycoord);
        void Point_Print();
        bool Compare(Point* p);
};

class Map{
    public:
        int MaxN; // l h
        vector<Point*> Points;
        vector<Point*> Openlist;
        vector<Point*> Path;


        //Point inicial

        Map (int m);
        void Print_Points();
        void Print_Vecs();
        void Print_Map();
        bool Contains( Point* p);

};
```

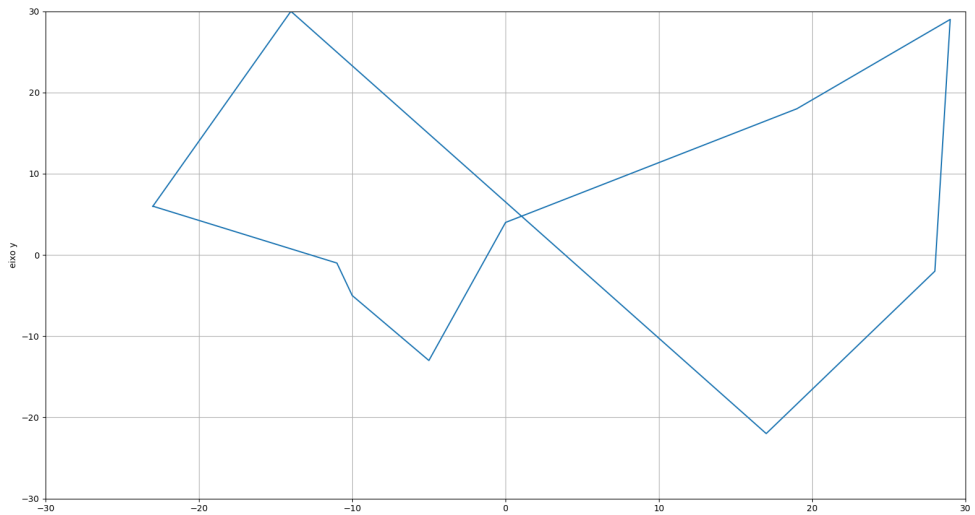
# Geradores de Mapas e Permutações

Através de um input dado, os nossos programas são capazes de criar um gráfico x y, com n pontos. Existem duas possibilidades para a criação de pontos, manualmente ou gerados aleatoriamente. A permutação inicial também pode ser gerada de duas formas diferentes. A primeira ordena os pontos pela ordem de criação. A segunda aplica o Nearest Neighbor, que é um algoritmo greedy. Esta última seleciona o ponto mais próximo do último ponto selecionado. O exemplo abaixo mostra uma permutação de 10 pontos num gráfico de xy de valor máximo 30 e mínimo -30 ( acima gerada pela ordem de criação e abaixo gerada por  nearest neighbor, com o primeiro nó escolhido aleatoriamente).

Permutação aleatória:
(-14,30)(28,-2)(19,18)(-23,6)(-11,-1)(-10,-5)(17,-22)(-5,-13)(29,29)(0,4)(-14,30)




Nearest neighbor:
(-23,6)(-11,-1)(-10,-5)(-5,-13)(0,4)(19,18)(29,29)(28,-2)(17,-22)(-14,30)(-23,6)



Nota: a geração de aleatórios é na verdade pseudo-aleatória, visto que não mudamos a seed. O motivo pelo qual não mudamos a seed tem a ver com a facilidade de manter os mesmos inputs, o que permite uma melhor análise de resultados e correção de erros.

# Nearest-Neighbor

```
void nearest_Neighbour(Map *m){

    int r =rand()%(m->Points.size()-1);

    Point *inicial_Point = m->Points[r];
    Point *final_point;

    m->Openlist.push_back(inicial_Point);
    m->Path.push_back(inicial_Point);

    bool goalFound=false;

    while(!goalFound){
        Point *current_Point = m->Openlist[0];
        m->Openlist.erase(m->Openlist.begin());

        current_Point->visited=true;

        double dist_min=sqrt(pow(m->MaxN*2,2)+pow(m->MaxN*2,2));//diagonal
        Point *next_Point;
        next_Point=current_Point;

        for(auto const& i :m->Points){

            if(!i->visited){

                int x = i->x - current_Point->x; //calculating number to square in next step
                int y = i->y - current_Point->y;
                double dist;
```

```
                dist = pow(x, 2) + pow(y, 2);//calculating Euclidean distance
                dist = sqrt(dist);

                if(dist<=dist_min){
                    dist_min=dist;
                    next_Point=i;
                }
            }
        }

        if(!next_Point->visited){
            m->Openlist.push_back(next_Point);
            m->Path.push_back(next_Point);
        }

        else{
            goalFound=true;
            final_point=next_Point;
        }

    }
    m->Path.push_back(inicial_Point);

}
```

A função calcula a partir de um ponto aleatório a distância a todos os não visitados e seleciona o mais próximo, marcando o primeiro como visitado. Quando todos os nós estiverem visitados, retorna o caminho todo.

# Função "2-exchange"

A função “2-exchange” tem como objetivo selecionar o melhor filho de um determinado nó, ou seja, escolher a melhor interseção para desfazer no momento. A forma como a remoção da interseção funciona consiste no seguinte:

Tendo em conta que uma linha tem um sentido, a função remove uma interseção ligando os pontos iniciais um ao outro e os respetivos finais também, invertendo todos os pontos entre eles.

Esta função é chamada nos algoritmos hill climbing e simulated annealing.

```
vector<vector<Point*>>* two_exchange(vector<Point*> p){
    vector<vector<Point*>>* Solution = new vector<vector<Point*>>;
    for(int i = 1; i<p.size()-2; i++){  //i-1 ->i
        for(int j = i+1; j<p.size()-1; j++){ //j->j+1
            if(p.at(i-1)!=p.at(j+1)){
                if(vectors_Intersect(p[i-1],p[i],p[j],p[j+1])) {
                    vector<Point*> tmp=p;
                    reverse(tmp.begin()+i+1,tmp.begin()+j);
                    Point *t = tmp[i];
                    tmp[i]=tmp[j];
                    tmp[j]=t;
                    Solution->push_back(tmp);
                }
            }
        }
    }
    return Solution;
}
```

# Hill climbing

O Hill Climbing (HC) é um algoritmo clássico para otimização, sendo bastante eficiente na tarefa de encontrar máximos ou mínimos locais. No caso do problema abordado, os máximos e mínimos não são locais mas sim totais, o que faz com que o HC retorne sempre a solução ótima em todos os casos. Foram implementadas 4 condições para determinar a seleção correta do próximo nó:

- candidato que reduz mais o perímetro;
-primeiro candidato da vizinhança;
-candidato com menos interseções;
-candidato aleatório;

```
vector<Point*> hill_climbing(char opt, vector<Point*> inicial ){

    vector<Point*> best = inicial;
    vector<vector<Point*>>* candidates = two_exchange(best);

    vector<Point*> neighbour = choose_opt(opt, candidates,best);

    while((neighbour.size()>0)){

        double best_per=perimeters(best);
        double neighbour_per=perimeters(neighbour);

        if(neighbour_per<best_per){
            best=neighbour;
            candidates=two_exchange(best);
            neighbour=choose_opt(opt,candidates,best);
        }
        else{
            break;
        }
    }

    return best;
}
```

O algoritmo funciona da seguinte forma: A partir do nó inicial da permutação dada através do nearest neighbor ou permutação aleatória, manda os candidatos possíveis do próximo nó e a condição que seleciona o melhor nó para a função choose_opt.

# Choose opt

Esta é a função que, ao receber um conjunto de candidatos do próximo nó e o fator que determina o mesmo a escolher, retorna a melhor opção possível de acordo com o input.

-Caso menor perímetro: cria uma variável menor perímetro, que é alterada sempre que um candidato de menor perímetro é detetado no ciclo. Retorna o candidato com o menor perímetro;

-Caso primeiro candidato: retorna o primeiro candidato da lista;

-Caso menos conflitos: retorna o candidato com o menor número de interseções, chamando uma função auxiliar(n_Intersections) com esse objetivo;

-Caso aleatório: retorna candidato na posição gerada aleatoriamente.

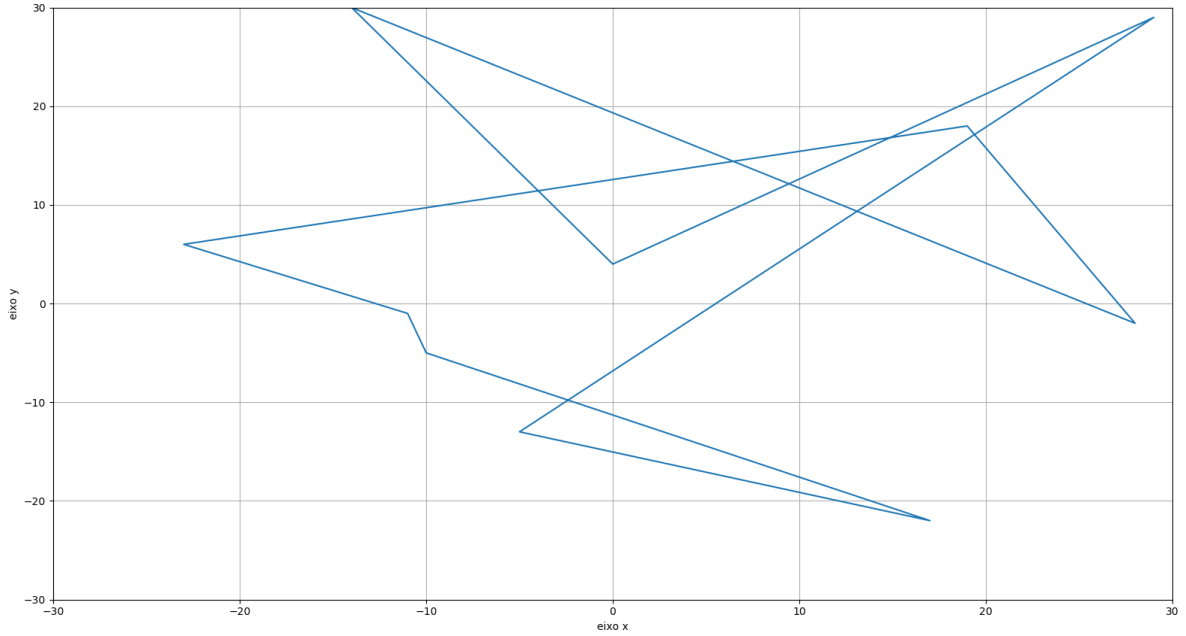
```
vector<Point*> choose_opt(char opt, vector<vector<Point*>>* candidates,vector<Point*> best){
    vector<Point*> n;
    double min_per = DBL_MAX;
    int min_intr = INT_MAX;
    switch(opt) {
        case 'a'://
            for(int i=0;i<candidates->size();i++){
                double perimeter=perimeters(candidates->at(i));
                if(perimeter<=min_per){
                    min_per=perimeter;
                    n=candidates->at(i);
                }
            }
            break;
        case 'b': // primeiro candidato
            if(candidates->empty()){
                return vector<Point*>();
            }
            n=candidates->front();
            break;
        case 'c':// menos conflitos
            if(candidates->empty()){
                return vector<Point*>();
            }
            for(int i=0;i<candidates->size();i++){
                int min=n_Intersections(candidates->at(i));
                if(min<min_intr){
                    min_intr=min;
                    n = candidates->at(i);
                }
            }
            if(n_Intersections(n)>n_Intersections(best)){
                n=best;
            }
            break;
        default:// random
            if(candidates->empty()){
                return vector<Point*>();
            }
            int r =rand()%(candidates->size());
            n=candidates->at(r);
    }
    return n;
}
```

**nota**: a melhor opção a tomar destas quatro, é sem sombra de dúvidas a primeira, visto que verdadeiramente melhora o resultado a cada iteração. Tanto o segundo como o quarto casos são pobres no que toca à seleção da melhor opção, visto que se baseiam nas posições onde os candidatos estão guardados na estrutura, o que não melhora o nosso caso, pois a ordem na estrutura não é relevante ao peso de cada candidato. No que toca ao terceiro caso, a regra possui uma falha que não permite retornar a solução ótima em todos os casos. O facto de ao remover uma interseção, existir a possibilidade de criar mais do que uma interseção, faz com que o método retorne por vezes **máximos locais**. Tendo em conta que os candidatos retornam sempre perímetros menores, se ignorássemos o facto do número de interseções do candidato ser maior que o do respetivo nó atual, o terceiro método seria também um bom partido para aplicar.

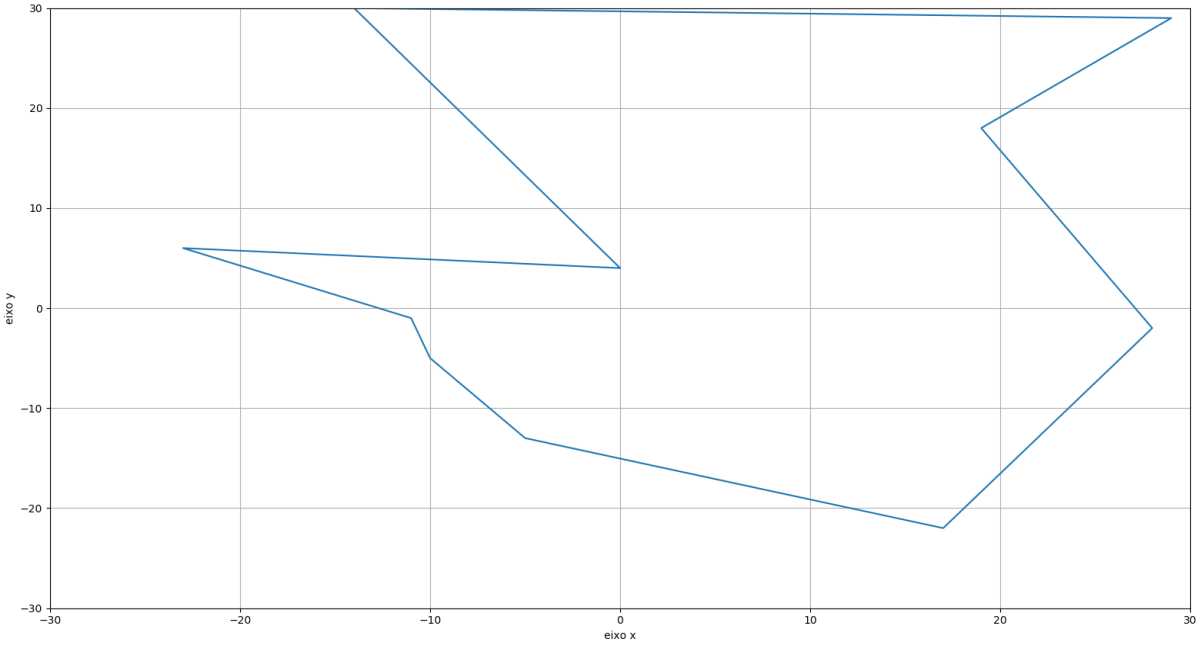
# Exemplo (Hill Climbing)

Foi gerada a seguinte solução inicial a partir da criação de 30 pontos aleatórios num gráfico de xy max 30, aplicando o nearest neighbor para a permutação inicial:

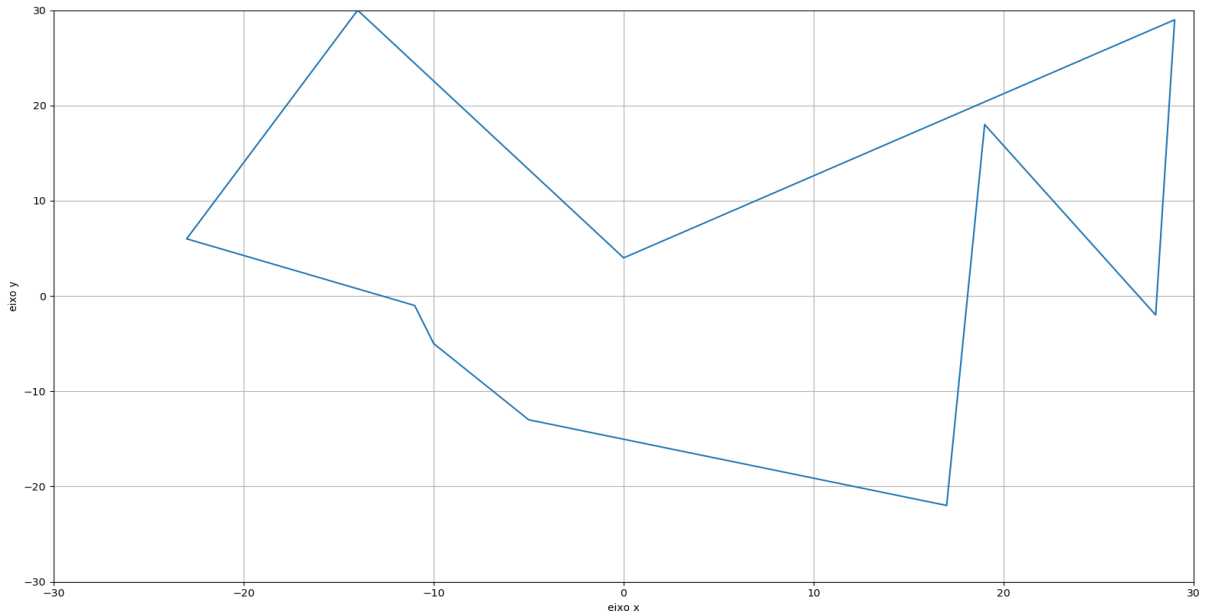
(-14,30)(28,-2)(19,18)(-23,6)(-11,-1)(-10,-5)(17,-22)(-5,-13)(29,29)(0,4)(-14,30)




Este é o resultado após aplicar o HC para os quatro casos:




a) menor perimetro 4 iterações, b) primeiro da lista 6 iterações e c) menor número de interseções 4 iterações;
(-14,30)(29,29)(19,18)(28,-2)(17,-22)(-5,-13)(-10,-5)(-11,-1)(-23,6)(0,4)(-14,30)




d) candidato aleatório 4  iterações
(-14,30)(-23,6)(-11,-1)(-10,-5)(-5,-13)(17,-22)(19,18)(28,-2)(29,29)(0,4)(-14,30)

# Simulated Annealing

É uma meta-heurística para otimização que consiste numa técnica de busca local probabilística, Assim, o algoritmo tem liberdade para percorrer casos que não melhorem a solução atual com o intuito de obter um melhor resultado final.

```
vector<Point*> simulated_annealing(vector<Point*> inicial){
    vector<Point*> best = inicial;

    vector<Point*> neighbour = choose_opt('c',two_exchange(inicial),best);
    vector<vector<Point*>>* candidates = two_exchange(best);

    double temp =  (double) n_Intersections(inicial);

    while(temp>0 && candidates->size() > 0){
        double best_per=perimeters(best);
        double neighbour_per=perimeters(neighbour);

        if(accept(best_per,neighbour_per,temp)){
            best=neighbour;
            candidates=two_exchange(best);
            neighbour=choose_opt('c',candidates,best);
        }
        temp*=0.95;
    }
    return best;
}
```

O nosso código funciona da seguinte forma:
Ao receber a permutação inicial, assume-se a mesma como a melhor solução no momento, e a temperatura inicial que vamos usar será  o número de interseções do inicial. Entramos depois num ciclo que termina quando a temperatura for 0 ou não existirem candidatos, o que significa que chegamos à solução. Dentro do ciclo  é chamada a função accepted, que caso verifique a condição,altera o melhor caso para o atual. no fim de cada iteração do ciclo a temperatura reduz 5%. No fim retorna o melhor caso.

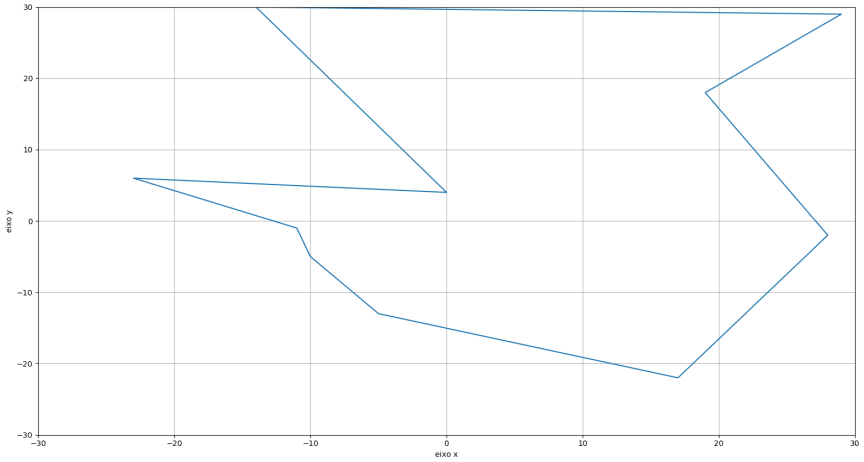
# Função accepted

```
bool accept(int actual_p,int neighbour_p,int temp){

    double dif = neighbour_p-actual_p;
    double e = -dif/temp;
    double x = ((double) rand() / (double) (RAND_MAX));

     if(actual_p>neighbour_p){
        return true;
    }

    else return e > x;
}
```

A função accepted  tem como objetivo aceitar soluções não ótimas

# Exemplo(Simulated Annealing)

Aplicando o algoritmo ao caso de 10 ponto no gráfico de xy 30, aplicando primeiramente a permutação aleatória, obtivemos o seguinte resultado:
4 iterações



# ANT COLONY

O algoritmo Ant Colony Optimization (ACO) é uma heurística baseada em probabilidade, criada para solução de problemas computacionais que envolvem procura de caminhos em grafos. Este algoritmo foi inspirado na observação do comportamento das formigas ao saírem de sua colônia para encontrar comida.

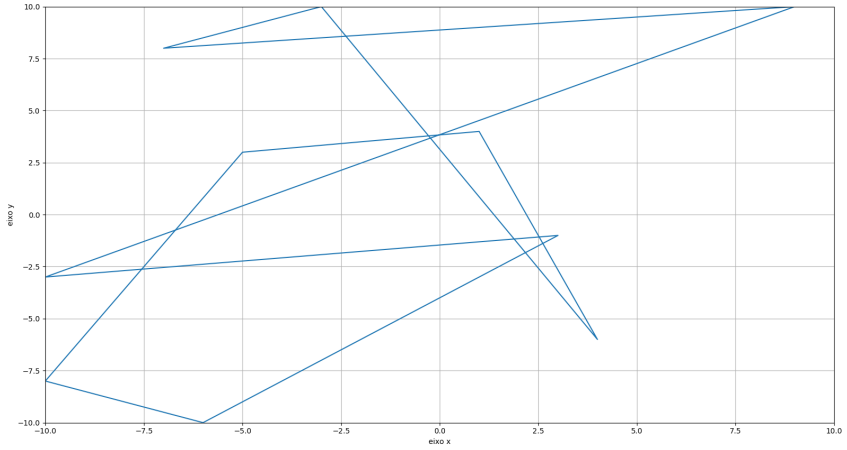
No caso deste trabalho as formigas não procuram comida mas sim resolver o problema TSP.

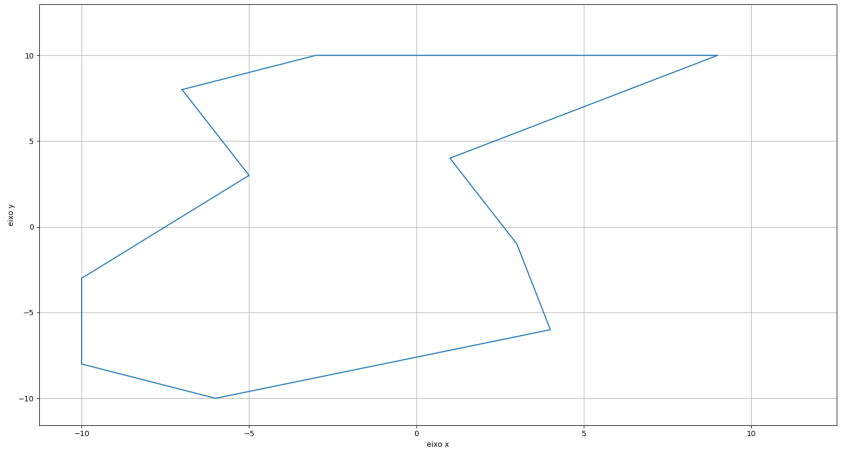
```
vector<Point*> ant_colony(vector<Point*> Points){
    vector<Point*> solution;
    vector<Point*> ant_path;

    double best_per=DBL_MAX;
    double actual_per=0.0;
    int size= Points.size();

    int n_iterations=size;
    //numero de iteracoes do ciclo aumentar faz sobrecarga no processador mas o resultado fica mais preciso
    int n_ants=1000*size;

    double ALPHA=0.0;//pheromone
    double BETA =0.750;//distance

    double mult=0.0000000000000000001;

    for(int n = 0; n<n_iterations; n++){
        for(int k = 0; k<n_ants; k++)   {
            for(auto const& i : Points){
                i->visited=false;
            }
            int p_inicial=k%size;
            Point* actual=Points[p_inicial];
            ant_path.push_back(actual);
            actual->visited=true;
            Point* nextPoint=ant_choose_point(actual,&Points,ALPHA,BETA);
            while(!nextPoint->visited){
                actual=nextPoint;
                ant_path.push_back(actual);
                for(auto const& i : ant_path){
                    i->visited=true;
                }
                nextPoint=ant_choose_point(actual,&Points,ALPHA,BETA);
            }
            Point* final=Points[p_inicial];
            ant_path.push_back(final);
            actual_per=perimeters(ant_path);

            if((best_per>actual_per) && (ant_path.size()==size+1)){
                best_per=actual_per;
                solution=ant_path;
            }
            ant_path.clear();
            for(int a=0;a<size;a++){
                for(int b=0;b<size;b++){
                    if(pheromones[a][b]>100){
                        pheromones[a][b]=100;
                    }
                    else
                        pheromones[a][b]=pheromones[a][b]*0.75;
                }
            }
        }
        mult=sqrt(mult);
        ALPHA=ALPHA+(mult);
        if(ALPHA>2){
            ALPHA=2.0;
        }
    }
    return solution;
}
```

Na implementação do nosso código, existem duas variáveis condicionantes para determinar a solução: o número de iterações feitas no gráfico e o número de formigas lançadas em cada iteração. Dependendo do peso de cada uma destas variáveis, obtemos resultados com precisão variada.
Ambas são variáveis cíclicas, ou seja, são iteradas pelos ciclos.
Existem dois ciclos: o primeiro solta ‘n_ants’ formigas numa iteração, onde cada uma vai percorrer no segundo ciclo possíveis soluções a partir de um ponto.A transição para o próximo ponto da solução, é determinada a partir de um cálculo de probabilidade, onde os valores de distância e feromona são fatores essenciais para o resultado.
Após cada iteração do segundo ciclo o índice de feromonas de cada ligação é enfraquecido.
Após cada iteração do primeiro ciclo, o valor que as formigas dão à feromona aumenta, ou seja, é incrementado o peso da feromona e é desvalorizado o valor da distância.
A função retorna o melhor caminho percorrido por uma formiga.

# Exemplo (Ant Colony)

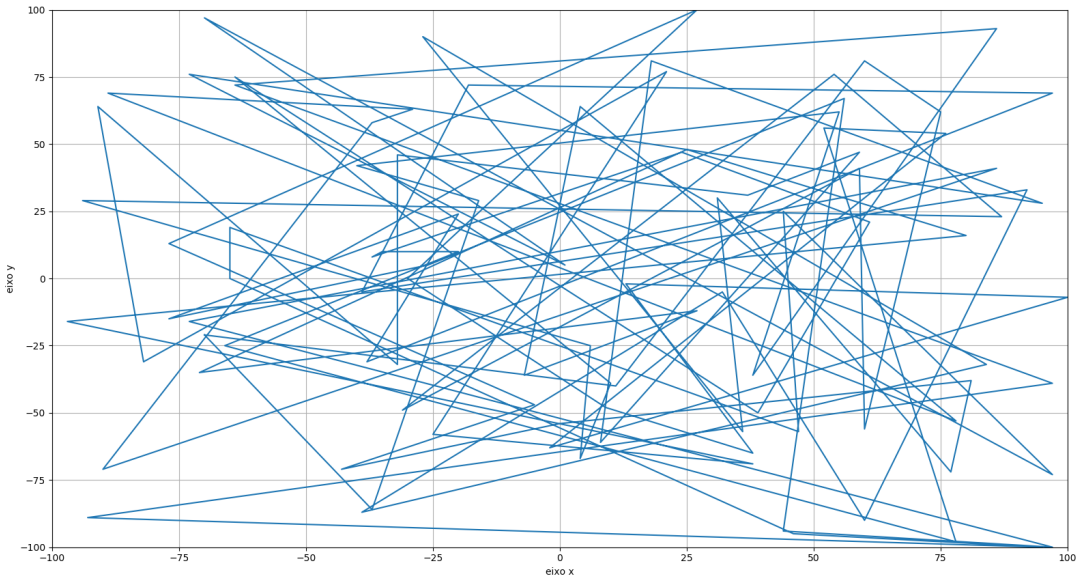
Exemplo criado com 10 pontos gerados aleatoriamente num gráfico xy de tamanho 10.
n de iterações: 100000



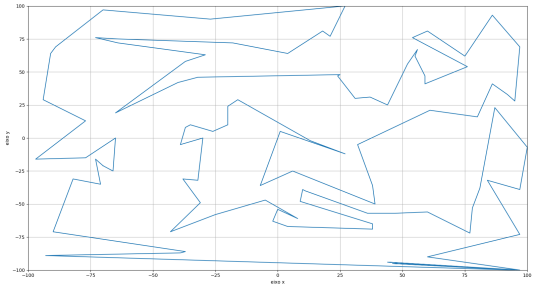
# Outros exemplos

Caso de 100 pontos num gráfico xy de tamanho 100, gerados aleatoriamente e aplicada a permutação aleatória ( o número a frente representa o n de iterações).

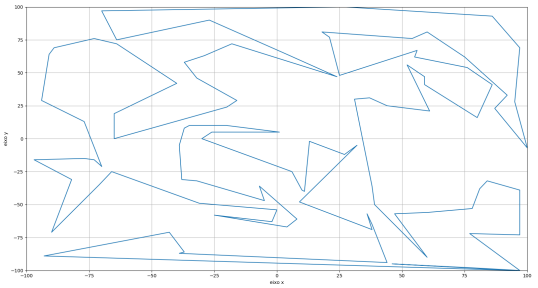



Aplicando os seguintes algoritmos:

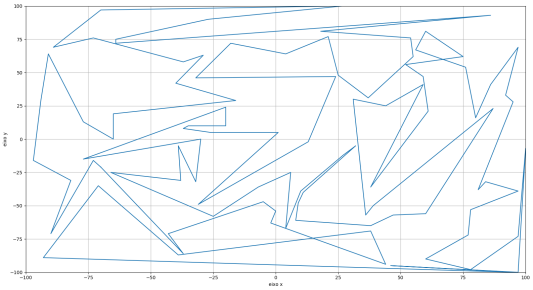
-Hill climbing a) 86



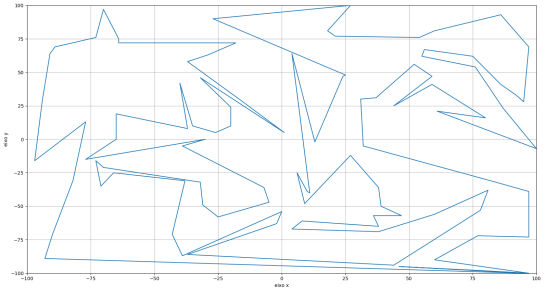
-Hill Climbing b) 199



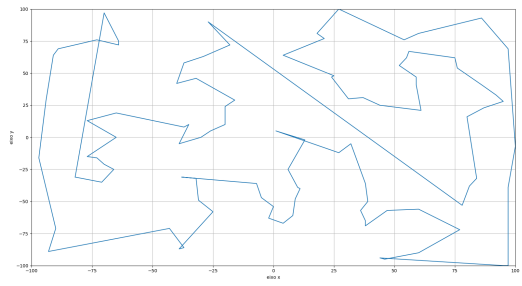
-Hill Climbing c) 77



-Hill climbing d) 171



-Simulated Annealing 13864



# Conclusões Finais

Resumindo, apesar de desafiante este trabalho a nível da quantidade de organização de dados, não terá sido uma tarefa impossível.
Algo que não foi um desafio foi a divisão de tarefas, que foram rapidamente distribuídas sem qualquer tipo de quezílias no grupo.
Durante a realização do nosso trabalho, o debate de ideias intergrupo foi essencial para a ultrapassagem de alguns obstáculos. Queremos agradecer, portanto, a relação de simbiose que nos permitiu progredir e avançar no projeto. Os respetivos grupos estão devidamente identificados no código.
O trabalho foi uma grande oportunidade, tanto a nível da aprendizagem do material de estudo abrangido neste projeto, como também a nível de matéria não lecionada (como por exemplo, o script de python).

# Referências


https://en.wikipedia.org/wiki/Travelling_salesman_problem

#matplotlib python
https://stackoverflow.com/questions/21519203/plotting-a-list-of-x-y-coordinates-in-python-matplotlib
https://www.kite.com/python/answers/how-to-plot-points-in-matplotlib-in-python
https://jakevdp.github.io/PythonDataScienceHandbook/04.01-simple-line-plots.html

#ant colony
https://www.youtube.com/watch?v=wfD5xlEcmuQ&t=470s
https://www.youtube.com/watch?v=783ZtAF4j5g