# Relatório MC458 - Laboratório 1

1. Introdução
    a. O problema do laboratório é: dado um conjunto de pontos, x1, x2, x3 … xn-1, xn, com as coordenadas x, y. Determinar o menor caminho, que passe por todos os pontos, iniciando em x1 e finalizando em xn. Calculando a respectiva distância e qual a sequência de pontos a ser percorrida
    b. Este problema está relacionado com o problema do caixeiro viajante, no qual o vendedor tem de passar por um conjunto de cidades e retornar ao início. Porém neste problema, nós temos o ponto inicial e final definidos. Portanto temos de fazer algumas alterações em nosso algoritmo para aplicarmos a solução do caixeiro viajante
2. Implementação
    a. A minha implementação se baseia na solução do caixeiro viajante, mas com uma pequena adaptação para que o algoritmo termine em xn
    b. O problema do caixeiro viajante tem a seguinte fórmula de recorrência:
        i. G(i, S) = min { Cik + G(k, S - {k}) } para todo k ∈ S
        ii. i é o ponto onde o viajante está
        iii. S é o conjunto de cidades que ele ainda não visitou
        iv. k é a próxima cidade a ser visitada que minimiza o caminho
    c. Para adaptar esta fórmula de recorrência ao nosso problema, eu adicionei um ponto “fake”, que possui distância 0 do ponto final e inicial, e distância infinita dos demais pontos.
    d. Assim, nossa solução sempre irá conter a seguinte sequência de pontos: final -> ponto “fake” -> inicial. Isto ocorre por uma razão simples, ele nunca será o melhor caminho entre outros 2 pontos, pois ele está a uma distância infinita deles, e ele também é o melhor caminho entre a origem e final.
    e. A distância entre os pontos foi memorizada em um grafo representado na forma de uma matriz de adjacência, os elementos da matriz representam a distância de i a j. Inicializada da seguinte forma
        i. `vector<vector<double>>`
    f. A estrutura de dados utilizada para a memorização no algoritmo A e B foi a mesma, a inicialização dela é dada da seguinte forma:
        i. `unordered_map<string, vector<tuple<double, int>>>`
    g. Este é um hash, com as seguintes propriedades:
        i. Chave: String que representa os índices dos pontos não visitados
        ii. Valor: Vector de Tuplas, cujo tamanho é o número de pontos. Ele possui 2 valores:
            1. Double: Menor distância partindo do ponto (indice do vetor) considerando os pontos não visitados (chave). Consiste no G(i, S)

da fórmula de recorrência. i é o ponto e S é o conjunto de pontos não visitados

2. Int: Próximo ponto a ser visitado que minimiza a distância. Consiste no k da fórmula de recorrência

h. Algoritmo Bottom Up
   i. O algoritmo inicia criando as combinações possíveis para se visitar os pontos. Consiste em criar as possibilidades para o S
   ii. Ordena o vetor de combinações pelo número de elementos, para que o preenchimento da nossa estrutura de memorização seja bottom Up. Primeiro resolvemos os casos menores, depois os casos maiores.
   iii. Iniciamos nosso loop
   iv. `for (string subset : subsets)`
      1. Percorre os conjuntos de pontos não visitados 1 a 1, preenchendo de baixo para cima
      2. Primeiro preenchemos os casos base
      3. `if (subset.size() == 1)`
         a. `get<0>(tuplePos) = points[i][splited[0]] + points[splited[0]][0];`
         b. `get<1>(tuplePos) = splited[0];`
      4. Depois preenchemos os valores de acordo com a fórmula de recorrência
      5. `double distance = points[i][nextPos] + get<0>(path.at(nextSequence)[nextPos]);`
   v. Algoritmo TopDown
      1. Este algoritmo é mais simples, ele é composto de 3 partes:
         a. Caso base: verifica se ja visitamos todos os pontos
            i. `if(notVisited == "")`
            ii. `return points[point][0];`
         b. Verifica se o ponto já possui sua distância calculada
            i. `if(get<0>(pointTuple) != MAX_DISTANCE)`
            ii. `return get<0>(pointTuple);`
         c. Recorrência
            i. `for(int i = 1; i < points.size(); i++)`
            ii. `string newNotVisited = removeString(notVisited, i);`
            iii. `double distance = points[point][i] + memoizationSolution(points, i, newNotVisited, path);`
   vi. Reconstrução do caminho e menor distância
      1. Ao final dos dois algoritmos temos a tabela preenchida, para descobrirmos a menor distância basta conferirmos o valor armazenado na chave (sequência onde nenhum ponto foi visitado), index (0). Isto significa responder a seguinte pergunta:

partindo do ponto 0 (inicial) com o conjunto S completo, qual a menor distância ?

2. Para reconstruir o caminho é simples, cada elemento da nossa estrutura de memorização contém o próximo ponto a ser visitado para minimizar a distância. Logo o que fazemos é o seguinte
    a. Dizemos que nosso próximo ponto é o que está armazenado
    b. `int nextPos = get<1>(path.at(sequence)[pos]);`
    c. Removemos o próximo ponto da string de não visitados
    d. `string nextSequence = removeString(sequence, nextPos);`
    e. Pulamos para esta posição em nossa estrutura
    f. `bestPath = recreatePath(path, nextSequence, nextPos);`

## 3. Resultados Computacionais

a. Configurações da máquina usada:
    i. Processador i7-8750H (2.2 GHz até 4.1 GHz, cache de 9MB, hexa-core)
    ii. RAM: 16GB 2666Mhz DDR4
    iii. Placa de Vídeo: GTX 1060-MaxQ (6GB de GDDR5)
    iv. Sistema Operacional: Ubuntu 18.04

| Entrada | Algoritmo | Tempo Real | Saída | Path | Tempo Limite |
|---|---|---|---|---|---|
| arq1 | A | 0,005s | Distance: 3 | 0 1 2 3 | 2s |
| arq2 | A | 0,006s | Distance: 7.82843 | 0 2 1 4 3 5 | 2s |
| arq3 | A | 0,008s | Distance: 6.11803 | 0 4 1 2 3 5 6 | 2s |
| arq4 | A | 0,006s | Distance: 5 | 0 1 3 4 2 5 | 2s |
| arq5 | A | 0,058s | Distance: 63.8257 | 0 2 4 6 5 7 3 1 8 | 2s |
| arq6 | B | 0,004s | Distance: 3 | 0 1 2 3 | 2s |
| arq7 | B | 0,006s | Distance: 7.82843 | 0 2 1 4 3 5 | 2s |
| arq8 | B | 0,009s | Distance: 6.11803 | 0 4 1 2 3 5 6 | 2s |
| arq9 | B | 0,002s | Distance: 5 | 0 1 3 4 2 5 | 2s |
| arq10 | B | 0,138s | Distance: 63.8257 | 0 2 4 6 5 7 3 1 8 | 2s |