Projeto de Concepção e Análise de Algoritmos

Tema 5 - BosHBus: transporte de trabalhadores

Relatório Final

24 de Maio de 2020

Turma 5, Grupo 6:

João Carlos Carreira Martins up201605373@fe.up.pt
Luís Miguel Guimas Marques ei11159@fe.up.pt
Pedro Miguel Afonso Teixeira up201505916@fe.up.pt

Descrição do tema

Neste trabalho, pretende-se implementar um sistema que permita a gestão do transporte de trabalhadores da BosH Global por parte de uma empresa especializada. Os veículos, que estão numa garagem, saem logo cedo pela manhã percorrendo um caminho com destino às instalações da BosH Global, recolhendo os trabalhadores em pontos de entrada e saída pré-identificados. No final do horário de trabalho, os veículos fazem o caminho inverso, das instalações da BosH Global até à garagem, deixando os trabalhadores no mesmo local de entrada.

O problema pode ser decomposto em três iterações:

- **1ª iteração: um veículo e uma empresa**

Inicialmente, é considerado apenas um veículo que irá transportar todos os trabalhadores de uma empresa, passando por todos os pontos de entrada e saída antes de chegar ao destino. O objetivo é percorrer a distância mínima entre a origem e o destino, passando por todos os pontos de entrada e saída identificados.

- **2ª iteração: vários veículos e uma empresa**

Cada veículo vai recolher um número de trabalhadores, menor ou igual à sua capacidade, de uma empresa e terá um percurso específico. O número de veículos usados varia conforme o número de trabalhadores a transportar. Todos os veículos partem da mesma origem e tem o mesmo destino e todos os pontos de entrada e saída tem que ser passados por pelo menos 1 veículo. O objetivo é, utilizando o mínimo número de veículos possível, percorrer a distância mínima entre a origem e o destino, passando por todos os pontos de entrada e saída identificados.

- **3ª iteração: vários veículos e várias empresas**

Este caso é semelhante à 2ª iteração mas o serviço de transporte é feito a mais do que uma empresa. Tal como na 2ª iteração, o número de autocarros usados para cada empresa é minimizado. Cada veículo transporta apenas trabalhadores da mesma empresa, efetuando o percurso desde a garagem até à respetiva empresa e vice-versa.

Identificação e formalização do problema

Dados de entrada

- Grafo dirigido, composto por:
    - Vértices que representam interseções.
    - Arestas, representando vias, com peso proporcional à distância entre os 2 vértices que a delimitam.
- Ponto de origem, sendo este ponto a garagem de onde partem os veículos.
- Sequência de veículos na frota (na 1ª iteração apenas existe 1 veículo), cada um caracterizado por:
    - Capacidade de um veículo.
- Sequência de empresas que contrataram o serviço (na 1ª e 2ª iteração apenas existe uma empresa). Cada uma caracterizada por:
    - Nome
    - A sua localização.
    - Pontos de entrada e saída, onde serão recolhidos e deixados os trabalhadores da empresa.

Dados de saída

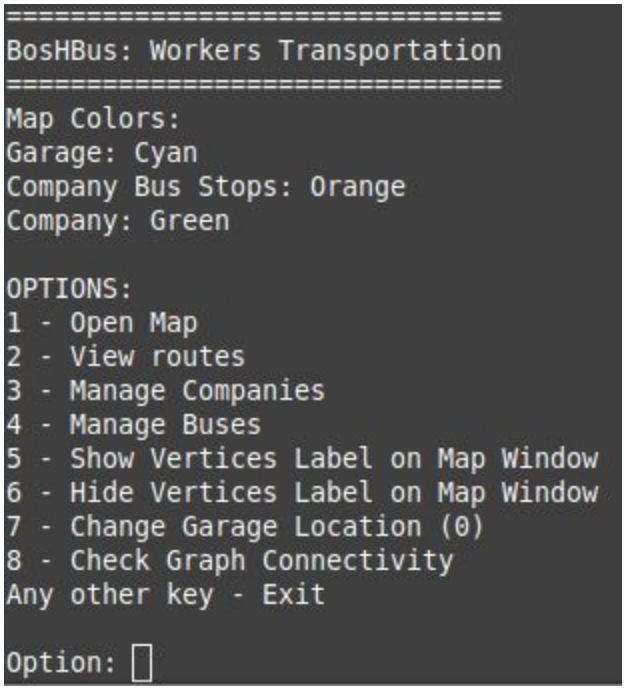
- Sequência de veículos usados, cada um com:
    - Capacidade do veículo.
    - Sequência ordenada dos vértices que correspondem ao trajeto percorrido.

Restrições

- Os pontos fornecidos pelo utilizador tem que existir no grafo.
- A capacidade de um veículo tem que ser maior que 0.
- Um ponto de entrada/saída tem que ter pelo menos 1 trabalhador.
- A garagem tem obrigatoriamente um vértice associado.
- Na ida o ponto inicial é a garagem e o ponto final a empresa em questão. Na volta é o oposto.

Principais casos de uso implementados

Quando o utilizador corre o nosso programa depara-se com um menu para escolher o mapa. Após escolher o mapa, a localização da garagem dos autocarros e a capacidade do primeiro autocarro da empresa, aparece o menu principal, onde é possível realizar as seguintes operações:



1 - Open Map: visualização do grafo através da GraphViewer API.
2 -View routes: mostra, se possível, os melhores percursos (ida e volta) obtidos pelo nosso algoritmo para todas as empresas. Caso o mapa esteja aberto, para além de mostrar o percurso no terminal em forma de texto, mostra também o percurso no mapa, colorindo os vértices percorridos a vermelho.
3 - Manage Companies: permite consultar e atualizar informação sobre as empresas que contrataram o serviço de transportes (nome, localização, pontos de entrada/saída de trabalhadores). Permite ainda adicionar ou remover uma empresa do serviço de transportes.
4 - Manage Buses: permite adicionar ou remover autocarros.
5/6 - Show/Hide Vertices Label on Map Window: mostra/esconde os ids dos vértices no mapa caso este esteja aberto. Esta funcionalidade pode ajudar a localizar vértices no mapa. Por padrão, os ids dos vértices são omitidos no mapa.
7 - Change Garage Location: atualiza a localização no grafo da garagem dos veículos. Entre parêntesis, mostra o id do vértice da garagem atual.
8 - Check Graph Connectivity: executando o algoritmo **Depth-First Search** mostra ao utilizador se o grafo está totalmente conectado, partindo da garagem.

Estruturas de dados utilizadas

Informação do grafo

Para guardar informação relativa ao grafo usamos as classes **Graph**, **Vertex** e **Edge**, contidas no ficheiro **Graph.h** disponibilizado pelos docentes de Concepção e Análise de Algoritmos.

Gestão da empresa

Para gerir a empresa prestadora do serviço de transporte criámos a classe **Manager**, contida no ficheiro **Manager.h**, que guarda o grafo do mapa em questão, a localização da garagem dos autocarros, a frota de autocarros e o conjunto de empresas às quais é prestado o serviço. Este ficheiro contém também as estruturas (struct):

- **Bus:** guarda o id único do autocarro, a sua capacidade e o percurso pelo qual é responsável.
- **Stop:** guarda a localização do ponto de entrada/saída e o número de trabalhadores nesse ponto.
- **Company:** guarda o nome da empresa, a sua localização e um vetor do tipo **Stop**, com os pontos de entrada/saída dessa empresa.

Interface

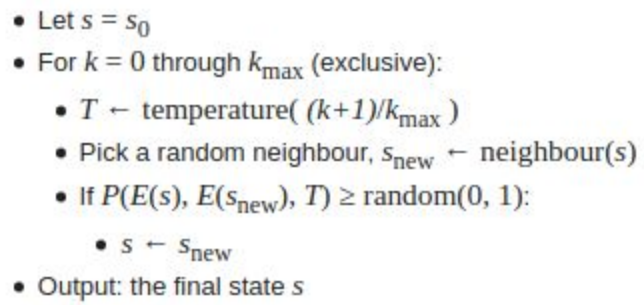
Para permitir a interação entre o utilizador e o nosso programa criamos ainda uma classe **Interface** contida no ficheiro **Interface.h** que guarda um apontador do tipo **Manager** e outro do tipo **GraphViewer** e permite ao utilizador efetuar as operações mencionadas nos **principais casos de uso implementados**.

Algoritmos implementados

- **Dijkstra:** Utilizado para obter o trajeto mais curto entre 2 pontos do grafo. Apresenta complexidade temporal **O(E + V * log(V))** e complexidade espacial **O(V^2)**, sendo **E** o número de arestas e **V** o número de vértices do grafo.
- **Depth-First Search:** Utilizado para verificar a conectividade do grafo a partir de um determinado ponto de origem. Todos os vértices do grafo (classe **Vertex**) contém uma variável *visited* do tipo bool que inicialmente tem o valor false. É feita uma pesquisa em profundidade por todos os vértices do grafo conectados

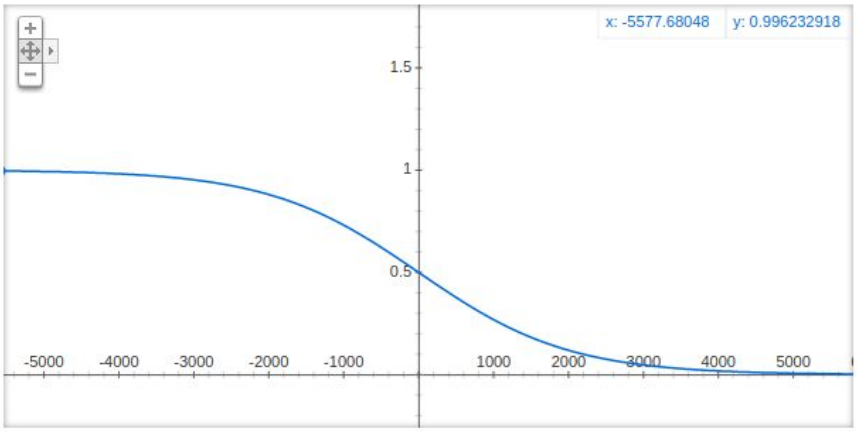
ao ponto de origem, sendo a variável *visited* modificada para true quando um determinado vértice é visitado. No fim, caso algum vértice tenha a variável *visited* com o valor false concluímos que o grafo não se encontra totalmente conectado a partir do ponto de origem escolhido. Apresenta complexidade temporal **O(V+E)** e complexidade espacial **O(V)**, sendo **E** o número de arestas e **V** o número de vértices do grafo.

- **Simulated Annealing:** Algoritmo escolhido para obter o melhor percurso entre 2 pontos do grafo, passando por pontos de entrada/saída de trabalhadores. O pseudocódigo do algoritmo pode ser visto na imagem abaixo (retirado da Wikipedia, ver referências):

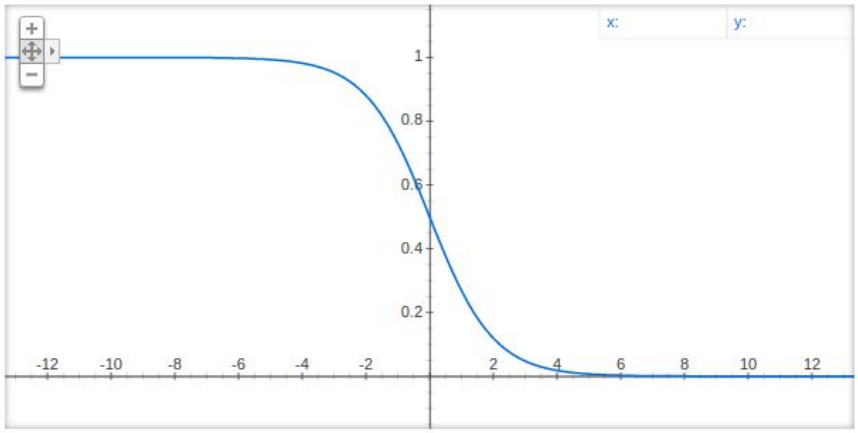


Parte-se de uma solução inicial, no nosso caso um vetor com os pontos de entrada e saída ordenados, por onde os autocarros têm que passar, e uma temperatura inicial (usamos 1000). O ciclo For executa enquanto a temperatura for maior que 0, ou seja, **kmax** (variável na imagem do pseudocódigo) é o número de iterações necessárias para que a temperatura chegue a 0. Em cada ciclo, a temperatura diminui a um ritmo constante e é selecionada uma nova solução vizinha da solução **s**. Uma solução vizinha é obtida trocando aleatoriamente a posição de 2 pontos de entrada/saída no vetor da solução **s**. Posteriormente, usando uma função de probabilidade **P** (no nosso caso com argumentos Δdistância e temperatura) e um valor obtido aleatoriamente entre 0 e 1 decide-se se a solução **s** é alterada ou não. A função de probabilidade **P** usada foi 1/(1+e^(Δdistance/temperature)). Os gráficos seguintes, retirados do Google, mostram o comportamento desta função para valores de temperatura 1000 e 1 respetivamente:

Graph for 1/(1+e^(x/1000))

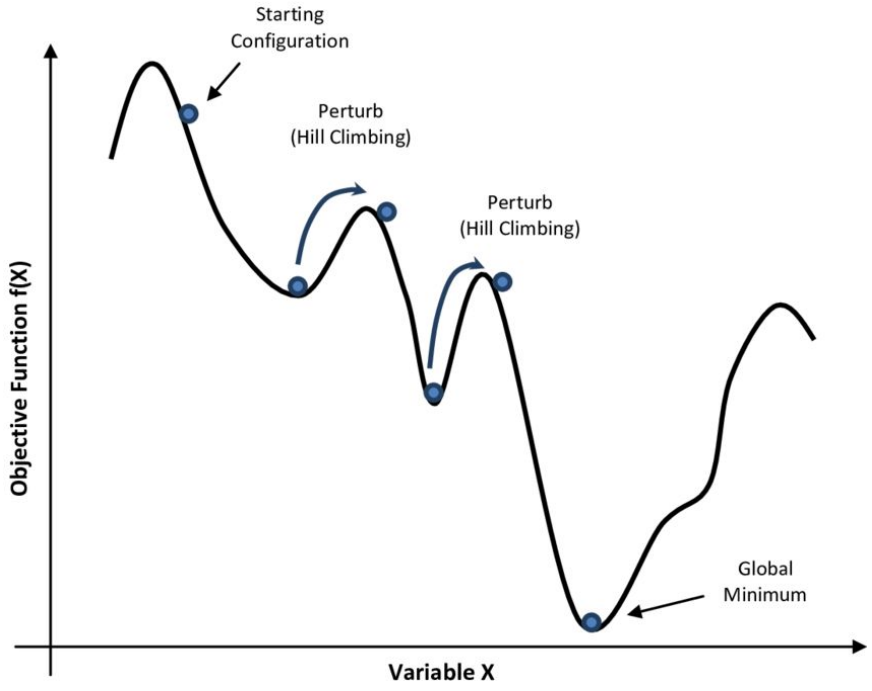


Graph for 1/(1+e^(x/1))



A variável **x** representa a diferença da distância percorrida entre a nova solução e a atual portanto um valor negativo de **x** significa que a nova solução tem uma menor distância percorrida do que a atual logo é melhor pois o nosso objetivo é minimizar a distância percorrida.
Soluções melhores tem sempre uma maior probabilidade de serem selecionadas comparativamente a soluções piores mas à medida que a temperatura vai diminuindo a probabilidade varia.
Quando a temperatura é alta (1000), soluções piores são aceites com alguma probabilidade. Por exemplo, uma solução com uma distância percorrida de 1000 unidades maior do que a distância percorrida na solução atual tem uma probabilidade de ser selecionada de cerca de 27%. Isto ajuda a evitar mínimos locais.
Quando a temperatura é baixa (1), a função de probabilidade praticamente aceita melhores soluções e rejeita piores soluções. Como vemos no gráfico, para uma

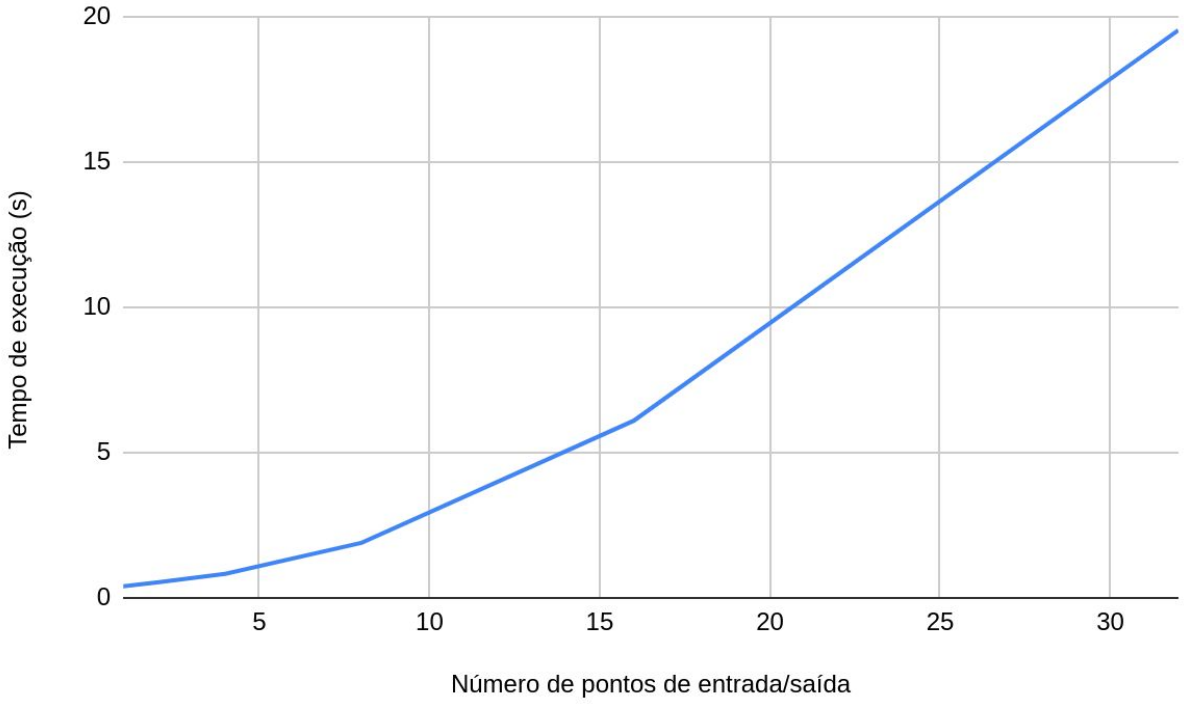
diferença de distâncias maior ou igual a 6 unidades a probabilidade da nova solução ser selecionada é praticamente 0 e para uma diferença de distâncias menor ou igual a -6 unidades a probabilidade da nova solução ser selecionada é praticamente 1.
Um algoritmo muito semelhante a este é o **Hill Climbing** que difere deste apenas no facto de que altera a solução atual sempre que encontra uma melhor, enquanto que o **Simulated Annealing** usa uma função de probabilidade. Esta função de probabilidade ajuda a evitar ficar preso em mínimos locais como acontece frequentemente com o **Hill Climbing**. Caso seja encontrada uma solução que corresponde a um mínimo local, ao contrário do **Hill Climbing** que fica lá preso pois não encontra uma melhor solução na vizinhança da solução atual, no **Simulated Annealing** há uma probabilidade de ser selecionada uma solução pior à atual que possibilitará encontrar o mínimo global, ou seja, a melhor solução possível. Isto pode ser visto na imagem abaixo:



Resumindo, a função de probabilidade usada permite no início quando a temperatura é maior aceitar piores soluções para escapar mínimos locais e à medida que a temperatura diminui a função passa a aceitar cada vez mais melhores soluções e a rejeitar as piores.

Também testamos o nosso algoritmo quanto à evolução do tempo de execução com o aumento do tamanho do input (número de paragens de autocarro).

Fizemos este teste no mapa **Porto (Strong)** (um dos últimos mapas disponibilizados pelos docentes desta cadeira) com uma empresa com paragens de autocarro selecionadas aleatoriamente. Os resultados obtidos encontram-se no gráfico abaixo. O tempo de execução no eixo vertical é o tempo que o algoritmo demorou a calcular o percurso entre a garagem e a empresa em questão.



Conectividade dos grafos utilizados

Consideramos que o grafo se encontra conectado a partir de um ponto quando, partindo desse ponto, se consegue chegar a todos os vértices do grafo. Foi implementada a função **isConnected** no ficheiro **Graph.h** que faz esta verificação chamando a função **dfs** (que implementa o algoritmo **Depth-First Search**) que percorre recursivamente um vértice e todos os seus vértices vizinhos. No fim, caso algum vértice não tinha sido visitado, conclui-se que o grafo não se encontra totalmente conectado.

Conclusão

Consideramos que alcançamos com sucesso todos os objetivos propostos neste projeto. O nosso programa para além de proporcionar todo o tipo de operações possíveis (especificadas na secção **Principais casos de uso implementados**), permite também usar qualquer número de autocarros, com um variado número de lugares, e qualquer número de empresas, cada uma com um variado número de paragens de autocarro, estas que por sua vez podem ter qualquer número de trabalhadores. Para além disto, usa um algoritmo que provou ser bastante eficiente na resolução deste problema, o **Simulated Annealing**.

Distribuição das tarefas

Todos os elementos do grupo contribuíram igualmente na realização deste projeto, sendo as tarefas divididas da seguinte forma:

João Martins:

- Implementação dos algoritmos **Dijkstra** e **Simulated Annealing**.
- Implementação da classe **Manager**.

Pedro Teixeira

- Implementação da interface.
- Implementação da verificação da conectividade do grafo (funções **isConnected** e **dfs**, que implementa o algoritmo **Depth-First Search**).

Luís Marques

- Extração de informação dos ficheiros.
- Redação dos relatórios.

Notas

Devido à má conectividade dos grafos fornecidos, o professor Francisco Rebello de Andrade deu-nos permissão para usar todos os mapas com as arestas em modo **UNDIRECTED**, o que significa que se uma aresta conecta 2 vértices, cada um dos vértices pode aceder ao outro (num contexto real significa que todas as estradas, representadas pelas arestas, são de duplo sentido).

Referências

Pseudocódigo do algoritmo Simulated Annealing retirado de https://en.wikipedia.org/wiki/Simulated_annealing#Pseudocode

Imagem usada para demonstrar como o algoritmo Simulated Annealing escapa de mínimos locais retirada de https://www.researchgate.net/figure/Simulated-Annealing-optimization-of-a-one-dimensional-objective-function_fig1_308786233