Técnicas de Busca e Ordenação 2023/1

Trabalho Prático T2

12 de maio de 2023

Leia atentamente **todo** esse documento de especificação. Certifique-se de que você entendeu tudo que está escrito aqui. Havendo dúvidas ou problemas, fale com o professor o quanto antes. As dúvidas podem ser sanadas usando o fórum de dúvidas do AVA.

1 Objetivo

O objetivo deste trabalho é aplicar o algoritmo de Dijkstra em um contexto relacionado a otimização de rotas para veículos autônomos. Para isso, vocês precisarão utilizar e estender a estrutura de dados *heap* vista em aula.

2 O problema de otimização de rotas

Carros autônomos são veículos que utilizam a inteligência artificial e outros recursos computacionais para navegar nas estradas sem a necessidade de um motorista humano. Para que os carros autônomos possam trafegar de maneira satisfatória, é necessário otimizar suas rotas de forma a minimizar o tempo de viagem. O algoritmo de Dijkstra é um algoritmo clássico para encontrar o caminho mais curto entre dois vértices em um grafo ponderado (com pesos não negativos), o que é essencial para a o problema de otimização de rotas.
De forma complementar, os aplicativos de mapa mais populares utilizam dados dos usuários para obter informações sobre o tráfego. Um exemplo é o uso de dados de localização de cada usuário ativo do sistema, que são agregados e analisados em tempo real para identificar a velocidade média nas vias e estimar tempos de chegada mais precisos. Sendo assim, os veículos autônomos podem utilizar dados de outros veículos, assim como de aplicativos de mapas, para identificar as rotas mais adequadas entre a localização atual do usuário e seu ponto de destino. Nos casos de rotas mais longas, por exemplo, é ideal que estes cálculos sejam realizados constantemente, de maneira que o veículo possa "desviar" de rotas nas quais tenham surgido problemas ou lentidão no decorrer da viagem.

3 Formalização e Exemplo

Para formalizar o problema, vamos assumir que um mapa pode ser representado como um grafo direcionado $G(V, E, \omega)$, onde:

- V é o conjunto de vértices e representa todas as interseções entre diferentes estradas, como cruzamentos, rotatórias, entroncamentos, bifurcações, etc;
- E é o conjunto de arestas direcionadas e representa as estradas, ruas e rodovias. Se $(a, b) \in E$, então é possível se deslocar do vértice a para o vértice b.
- ω é uma função que define os pesos das arestas. Mais especificamente, $\omega(a, b)$ indica o custo de se deslocar do vértice a até o vértice b. Vamos assumir que $\omega(a, b)$ é sempre um valor não negativo para quaisquer vértices a e b.

A Figura 1 mostra um exemplo do tipo de grafo que vamos considerar. O nó 1 representa uma das saídas de veículos da Ufes e o nó 7 uma das saídas para a Praia de Camburi. Os demais nós representam as famosas praças de Jardim da Penha e as arestas são as principais avenidas e ruas que as conectam. Os pesos definidos nas arestas indicam, nessa figura, a distância, em metros, entre os nós (e não representam fielmente a realidade).

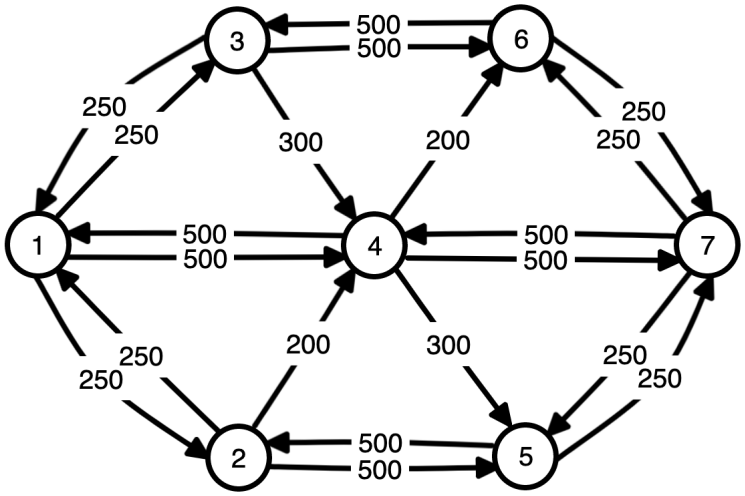

Figura 1: Exemplo de rede. Cada aresta representa uma rua/avenida em Jardim da Penha e o peso de cada aresta indica sua extensão em metros.

A partir do grafo, podemos identificar o melhor caminho entre a Ufes (1) e a Praia de Camburi (7), e se considerarmos a velocidade média dos carros em vias urbanas como 40km/h, também teremos o tempo médio da viagem.
Vamos supor que, a partir dos dados do grafo e considerando que a velocidade média em todas as pistas seja igual a 40km/h, o caminho definido pelo seu carro seja 1-2-4-6-7. Ao chegar ao nó 4, no entanto, quando o tempo decorrido de viagem é igual a 40.5 segundos (considerando o deslocamento de 450 metros a 40km/h), o sistema do seu carro identifica, a partir dos dados dos demais carros circulando pelo bairro, que a velocidade média na avenida que liga a nó 6 ao 7 é de 10km/h. Dessa forma, é necessário atualizar o peso da aresta e recalcular o restante da rota a partir das novas informações.

4 Sua tarefa

A sua tarefa será simular a viagem realizada pelo carro autônomo entre duas localidades no mapa, através do caminho **mais rápido**, registrando a rota, a distância percorrida e o tempo gasto.

1. Sua entrada será:

   o grafo $G(V, E, \omega)$, representando o mapa e os custos de deslocamento **em metros**;

   os nós de origem e destino;

   a velocidade média inicial das rodovias;

   as atualizações das velocidades médias das rodovias, **em km/h**, nos determinados instantes de tempo.

2. Utilize o algoritmo de Dijkstra para calcular o caminho mais rápido entre a origem e o destino;

3. Simule o deslocamento pela primeira aresta da melhor rota definida pelo algoritmo, incrementando o tempo de viagem de acordo seu peso;

4. Ao chegar no próximo nó, utilize-o como nó de origem, atualize os pesos das arestas de acordo com as últimas atualizações e recalcule a rota de maneira otimizada;

5. Repita os passos 3 e 4 até chegar ao destino;

6. Indique na saída do programa:

   a rota percorrida (a sequência de nós);

   a distância percorrida (em quilômetros);

   e o tempo total gasto (de acordo com o formato definido posteriormente).

5 Entrada e Saída

5.1 Entrada

Todas as informações serão apresentadas em um arquivo de texto com campos separados pelo delimitador `;`, organizado da seguinte maneira:

1. A primeira linha indica o número de vértices N seguido do número de arestas M do grafo G. Os vértices necessariamente serão numerados de 1 a N;
2. A segunda linha indica os nós de origem S e destino T;
3. A terceira linha indica a velocidade média inicial de todas as rodovias (em km/h). Esse campo será um número real positivo.
4. As próximas M linhas representam as arestas, indicando sequencialmente: o vértice de origem, o vértice de destino e a distância entre os nós conectados pela aresta, em metros. As arestas não estão ordenadas.
5. As próximas linhas representam as atualizações referentes ao tráfego ao longo da viagem. Cada atualização indica o instante de tempo (em segundos após o início da viagem), uma aresta no formato origem e destino e a nova velocidade média dos veículos naquela aresta (em km/h). As velocidades informadas nas atualizações serão números reais positivos. As atualizações seguirão uma ordem cronológica e nenhuma aresta será adicionada ou removida. O número de atualizações não é conhecido a princípio. Elas devem ser lidas enquando o fim do arquivo não for atingido.

Veja como seria a entrada referente ao exemplo da Figura 1, considerando uma única atualização:

```
7;20
1;7
40
1;2;250
2;1;250
1;3;250
3;1;250
1;4;500
4;1;500
2;4;200
3;4;300
4;5;300
4;6;200
2;5;500
5;2;500
3;6;500
6;3;500
6;7;250
7;6;250
5;7;250
7;5;250
4;7;500
7;4;500
40;6;7;10
```

Observação: Se o veículo estiver trafegando em uma rodovia e houver uma atualização de velocidade média para esta mesma rodovia, tal atualização não terá efeito na velocidade atual do veículo. Em outras palavras, ela deverá ser considerada para possíveis novos caminhos mínimos e cálculo do tempo de viagem apenas quando o veículo chegar ao próximo vértice.

5.2 Saída

A saída do trabalho deverá ser salva em um arquivo de texto, utilizando o delimitador `;` e contendo 3 linhas. Na primeira delas haverá uma sequência de vértices indicando o caminho percorrido pelo veículo. Na segunda, a distância percorrida em quilômetros. E na terceira, o tempo total gasto no percurso, no formato HH:MM:SS.fff, onde ".fff" representam os milisseegundos.
Para o exemplo utilizado anteriormente, a saída deverá ser como descrita abaixo:

```
1;2;4;7
0.95
00:01:25.5
```

Observação: utilizem o formato padrão de `double` para a impressão da distância, na segunda linha, e dos segundos na terceira. Erros de precisão nas últimas casas decimais serão desconsiderados.

Observação 2: notem que, ao imprimir o tempo do percurso, a hora pode ultrapassar o valor 23 e até mesmo conter mais que 2 dígitos. Minutos e segundos, no entanto, não podem ultrapassar o valor 60.

Observação 3: Desconsidere a possibilidade de múltiplos caminhos mais curtos. Os testes realizados pelo professor na avaliação não possuirão esses casos.

5.3 Execução do trabalho

Para testar seu trabalho, o professor executará comandos seguindo o seguinte padrão.

```
tar -xzvf <nome_arquivo>.tar.gz
make
./trab2 <nome_arquivo_entrada> <nome_arquivo_saida>
```

É extremamente importante que vocês sigam esse padrão. Seu programa não deve solicitar a entrada de nenhum valor e também não deve imprimir nada na tela.
Por exemplo, se o nome do arquivo recebido for `2004209608.tar.gz`, os dados de entrada estiverem em `entrada.txt` e o nome do arquivo de saída for `saida.txt`, o professor executará:

```
tar -xzvf 2004209608.tar.gz
make
./trab2 entrada.txt saida.txt
```

6 Sobre o algoritmo de Dijkstra

A parte central deste trabalho é a implementação do algoritmo de Dijkstra utilizando a estrutura *heap* para implementar uma fila de prioridades. Para tal, vocês deverão utilizar e estender o código visto em aula.
Uma boa explicação de como representar grafos em memória (utilizando uma lista de adjacências) e de como o algoritmo de Dijkstra funciona pode ser encontrada facilmente na Web. Em especial, recomenda-se o vídeo do Professor Sedgewick[1]. O algoritmo também está descrito em detalhes na Wikipedia[2].

7 Detalhes de implementação

A seguir, alguns detalhes, comentários e dicas sobre a implementação. Muita atenção aos usuários do Sistema Operacional Windows.

- o trabalho deve ser implementado em C. A versão do C a ser considerada é a presente no Sistema Operacional Ubuntu dos Computadores do LabGrad.

[1] `https://www.youtube.com/watch?v=uzHJXbToiIU&list=PLRdD1c6QbAqJn0606RlOR6T3yUqFWKwmX&index=75`

[2] `https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`

- o caractere de nova linha será o `\n`.
- Seu programa deve ser, obrigatoriamente, compilado com o utilitário `make`. Crie um arquivo `Makefile` que gera como executável para o seu programa um arquivo de nome `trab2`.
- Ao longo do desenvolvimento do trabalho, certifique-se que o seu código não está vazando memória testando-o com o `valgrind`. Não espere terminar o código para usar o `valgrind`, incorpore-o no seu ciclo de desenvolvimento. Ele é uma ferramenta excelente para se detectar erros sutis de acesso à memória que são muito comuns em C. Idealmente o seu programa deve sempre executar sem nenhum erro no `valgrind`.
- Não é necessária nenhuma estrutura de dados muito elaborada para o desenvolvimento deste trabalho. Todas as estruturas que você vai precisar foram discutidas em aula ou no laboratório. Veja os códigos disponibilizados pelo professor para ter ideias. Prefira estruturas simples a coisas muito complexas. Pense bem sobre as suas estruturas e algoritmos antes de implementá-los: quanto mais tempo projetando adequadamente, menos tempo depurando o código depois.

8 Regras para desenvolvimento e entrega do trabalho

- **Data da Entrega**: O trabalho deve ser entregue até as 23:59h do dia 11/06/2023. Não serão aceitos trabalhos após essa data.
- **Prazo para tirar dúvidas**: Para evitar atropelos de última hora, você deverá tirar todas as suas dúvidas sobre o trabalho até o dia 09/06/2023. A resposta para dúvidas que surgirem após essa data fica a critério do professor.
- **Grupo**: O trabalho pode ser feito em grupos de até três pessoas. Lembrando que duas pessoas que estiverem no mesmo grupo nesse trabalho não poderão estar no mesmo grupo nos próximos trabalhos. Além disso, pessoas do mesmo grupo no trabalho 1 não poderão estar no mesmo grupo neste trabalho.
- **Como entregar**: Pela atividade criada no AVA. Envie um arquivo compactado, no formato `.tar.gz`, com todo o seu trabalho. A sua submissão deve incluir todos os arquivos de código e um `Makefile`, como especificado anteriormente. **Somente uma pessoa do grupo deve enviar o trabalho no AVA. Coloque a matrícula de todos integrantes do grupo (separadas por vírgula) no nome do arquivo do trabalho.**
- **Recomendações**: Modularize o seu código adequadamente. Crie códigos claros e organizados. Utilize um estilo de programação consistente. Comente o seu código extensivamente. Não deixe para começar o trabalho na última hora.

9 Relatório de resultados

Esse trabalho não demandará relatório.

10 Avaliação

- Assim como especificado no plano de ensino, o trabalho vale 10 pontos.
- A parte de implementação será avaliada de acordo com a fração e tipos de casos de teste que seu trabalho for capaz de resolver de forma correta. Casos *pequenos* e *médios* (4 pontos) serão utilizados para aferir se seu trabalho está correto. Casos *grandes* (4 pontos) serão utilizados para testar a eficiência do seu trabalho. Casos *muito grandes* (2 pontos) serão utilizados para testar se seu trabalho foi desenvolvido com muito cuidado e tendo eficiência máxima como objetivo. Todos os casos de teste serão projetados para serem executados em poucos minutos (no máximo 5) em uma máquina com 16GB de RAM.
- Trabalhos com erros de compilação receberão nota zero.
- Trabalhos que gerem *segmentation fault* para algum dos casos de teste disponibilizados no AVA serão severamente penalizados na nota.

- Trabalhos com *memory leak* (vazamento de memória) sofrerão desconto na nota.
- O uso de variáveis globais ou da primitiva `goto` implica em nota zero.
- Organização do código e comentários valem nota. Trabalhos confusos e sem explicação sofrerão desconto na nota.
- Caso seja detectada **cópia** (entre alunos ou da Internet), todos os envolvidos receberão nota zero. Caso as pessoas envolvidas em suspeita de cópia discordem da nota, amplo direito de argumentação e defesa será concedido. Neste caso, as regras estabelecidas nas resoluções da UFES serão seguidas.
- A critério do professor, poderão ser realizadas entrevistas com os alunos, sobre o conteúdo do trabalho entregue. Caso algum aluno seja convocado para uma entrevista, a nota do trabalho será dependente do desempenho na entrevista. (Vide item sobre cópia, acima.)