Primeiro Trabalho Prático

Neste trabalho prático de Teoria dos Grafos, cada aluno (individualmente ou em duplas) deverá apresentar um programa completo de sua autoria para implementar a busca por caminhos mínimos em um grafo orientado, com pesos positivos nas arestas.

O Problema

Nós vimos em aula o funcionamento dos algoritmos de caminhos mínimos, como Dijkstra e Bellman-Ford, que computam caminhos mínimos em um grafo, a partir de um vértice de origem para todos os demais vértices do grafo.

Uma aplicação imediata e cotidiana deste algoritmo é no problema de roteamento de veículos, resolvido por dispositivos móveis com recepção de GPS. Numa situação ideal (onde não houvessem semáforos, lombadas e os motoristas se coordenassem para nunca precisar parar) estes algoritmos forneceriam a solução ótima com relativa velocidade. No mundo real, a qualidade da solução (proximidade do valor ótimo) depende basicamente da qualidade do mapa disponível (precisão das medidas, existência das vias e marcação correta das mãos de direção e restrições de conversão, entre outros) e da quantidade de informações que ele nos disponibiliza (existência de lombadas, elevação dos pontos, existência de semáforos e pontos de parada obrigatória e dados estatísticos de navegação, como velocidade média real da via e temporização e sincronização dos semáforos, que geralmente dependem do horário).

Muitas destas restrições podem ser traduzidas em ajustes de pesos nas arestas do grafo e, assim, podem ser resolvidas pelos mesmos algoritmos. Algumas, não são tão simples, como a proibição de circulação ou a inversão da mão de uma via num horário específico.

O objetivo do nosso trabalho será apenas encontrar a rota mais curta entre dois pontos usando mapas públicos de algumas áreas urbanas do Brasil, com valores aproximados de distância. O mapa será representado por um grafo de rotas possíveis e estará disponível como um arquivo texto em formato definido abaixo. Dados dois pontos (vértices do grafo) deve-se determinar a rota de custo mínimo. Isto significa que você deverá imprimir a lista dos vértices a serem percorridos desde a origem até o destino.

Para computar o caminho basta alterar o algoritmo de Dijkstra visto em aula para salvar em cada vértice visitado, o vértice de onde se veio. Ao atingir o destino, para-se o Dijkstra antecipadamente e computa-se o caminho. Veja o Algoritmo **??** para um exemplo, usando uma fila de prioridades.

Em grafos maiores e locais distantes, você observará que este algoritmo é um pouco lento. Duas variações podem ser implementadas para acelerá-lo. A primeira, Dijkstra alternante, consiste em executar o algoritmo a partir da origem e do destino alternadamente até que ambas árvores de caminhos mínimos coincidam em algum vértice. Esta técnica pode reduzir o espaço de busca (número de vértices visitados) em até um quarto, mas também pode ter pouco efeito. É necessário adaptar o algoritmo para obter o caminho a percorrer no final do processamento. Veja o Algoritmo **??** para um exemplo. Este algoritmo retorna dois vetores de predecessores, `prede` que vai em direção à origem e `predd` que vai em direção ao destino. Junto com o vértice de encontro, (`meio`) estes vetores lhe permitem reconstruir o caminho mínimo.

A segunda técnica, Dijkstra com limitantes, consiste em usar um limite inferior na distância entre pontos intermediários e o destino para identificar decisões que levam a rotas custosas e apenas visitar tais vértices quando identifica-se que as rotas mais diretas ultrapassaram o custo adicional da escolha. Esta técnica oferece uma eficiência melhor, se a aproximação pelo limite inferior for boa o suficiente. Uma boa aproximação levaria em consideração a topografia da região, a curvatura e formato do nosso planeta, bem como a sinuosidade mínima observada pelas vias de uma região. No nosso exemplo, podemos usar uma aproximação muito mais simples traçando uma linha reta entre os dois pontos e usando um valor mínimo para o raio do planeta. Isto exige que saibamos a latitude e longitude dos pontos de partida e chegada, que serão fornecidas no grafo. Pode ser calculada da seguinte forma:

$$\begin{aligned} x &= (long_2 - long_1)\cos\frac{lat_1 + lat_2}{2} \\ y &= (lat_2 - lat_1) \\ dist &= R\sqrt{x^2 + y^2} \end{aligned}$$

Use $R = 6356,75$, o raio mínimo da Terra em km e as latitudes e longitudes em radianos (converta-as pois no grafo estão em graus).

Algorithm 1 Algoritmo de Dijkstra

```
procedure DIJKSTRA(G, o, d)                          ▷ Encontra o caminho de o a d em G
    for all v ∈ V do                                 ▷ G(V, E)
        peso[v] ← ∞
        visitado[v] ← F
    end for
    peso[o] ← 0, 0
    Q ← {o}                                          ▷ Fila de prioridade dos vértices a serem visitados
    while não visitado[d] e Q ≠ ∅ do
        v ← RemoveMenor(Q)
        visitado[v] ← T
        for all (v, w) ∈ E do
            if não visitado[w] e peso[v] + custo(v, w) < peso[w] then
                peso[w] ← peso[v] + custo(v, w)
                pred[w] ← v
                if w ∈ Q then
                    Atualiza o peso de w em Q para peso[w]
                else
                    Insere w em Q com peso peso[w]
                end if
            end if
        end for
    end while
    return pred, visitado
end procedure
```

Veja o Algoritmo **??** para um exemplo. Note que o vetor `limitante` é desnecessário se você conseguir obter o valor armazenado na fila Q para o vértice w. O cálculo de distância mínima entre dois pontos é feito pela função `minimo`. Uma modificação interessante é o fato que vértices já visitados podem ser visitados novamente e reinseridos na fila Q. Isto ocorre porque não é mais possível garantir que, ao visitarmos w pela primeira vez, o faremos pelo melhor caminho entre o e w. Logo, sempre que visitarmos um vértice, verificamos se o caminho para seus vizinhos visitados pode ser melhorado e, neste caso, o enfileiramos para que seus vizinhos sejam reconsiderados.

Algorithm 2 Algoritmo de Dijkstra Alternado

```
procedure DijkstraAlternante(G, o, d)                    ▷ Encontra o caminho de o a d em G
    for all v ∈ V do                                     ▷ G(V, E)
        pesoe[v] ← pesod[v] ← ∞
        visitadoe[v] ← visitadod[v] ← F
    end for
    pesoe[o] ← pesod[d] ← 0, 0
    Qe ← {o}, Qd ← {d}                                   ▷ Inicializa filas de prioridade.
    meio ← 0
    while Qe ≠ ∅ e Qd ≠ ∅ do
        if topo(Qe) ≤ topo(Qd) then                      ▷ Remove da fila com menor topo
            v ← RemoveMenor(Qe)
            visitadoe[v] ← T
            if visitadod[v] then
                meio ← v; break                          ▷ Árvores encontraram-se
            end if
            for all (v, w) ∈ E do
                if não visitadoe[w] e pesoe[v] + custo(v, w) < pesoe[w] then
                    pesoe[w] ← pesoe[v] + custo(v, w)
                    prede[w] ← v
                    Insere (ou atualiza) w em Qe com peso pesoe[w]
                end if
            end for
        else
            v ← RemoveMenor(Qd)
            visitadod[v] ← T
            if visitadoe[v] then
                meio ← v; break                          ▷ Árvores encontraram-se
            end if
            for all (v, w) ∈ E do
                if não visitadod[w] e pesod[v] + custo(v, w) < pesod[w] then
                    pesod[w] ← pesod[v] + custo(v, w)
                    predd[w] ← v
                    Insere (ou atualiza) w em Qd com peso pesod[w]
                end if
            end for
        end if
    end while
    if meio ≠ 0 then                                     ▷ Procura caminhos melhores, se encontrou um
        pesomin ← pesoe[meio] + pesod[meio]
        for all (w, pesow) ∈ Qe do
            if pesow + pesod[w] < pesomin then
                meio ← w
            end if
        end for
        for all (w, pesow) ∈ Qd do
            if pesow + pesoe[w] < pesomin then
                meio ← w
            end if
        end for
    end if
    return meio, prede, predd
end procedure
```

Algorithm 3 Algoritmo de Dijkstra com Limitantes

```
procedure DijkstraLimitantes(G, o, d)                    ▷ Encontra o caminho de o a d em G
    for all v ∈ V do                                                              ▷ G(V, E)
        peso[v] ← ∞
        limitante[v] ← ∞          ▷ Mantém o peso acrescido da estimativa de distância ao destino
        visitado[v] ← F
    end for
    peso[o] ← 0, 0
    Q ← {o}                                          ▷ Fila de prioridade dos vértices a serem visitados
    while não visitado[d] e Q ≠ ∅ do
        v ← RemoveMenor(Q)
        visitado[v] ← T
        for all (v, w) ∈ E do
            if não visitado[w] then
                if peso[v] + custo(v, w) + minimo(G, w, d) < limitante[w] then
                    peso[w] ← peso[v] + custo(v, w)
                    limitante[w] ← peso[w] + minimo(G, w, d)
                    pred[w] ← v
                    if w ∈ Q then
                        Atualiza o peso de w em Q para limitante[w]
                    else
                        Insere w em Q com peso limitante[w]
                    end if
                end if
            else                                    ▷ Corrige o custo de vértices já visitados
                if peso[v] + custo(v, w) < custo[w] then
                    peso[w] ← peso[v] + custo(v, w)
                    limitante[w] ← peso[w] + minimo(G, w, d)
                    pred[w] ← v
                    if w ∈ Q then
                        Atualiza o peso de w em Q para limitante[w]
                    else
                        Insere w em Q com peso limitante[w]
                    end if
                end if
            end if
        end for
    end while
    return pred, visitado
end procedure
```

Solução

Você deve produzir um programa chamado *rotear.c*, escrito em linguagem C (nenhuma outra será aceita) e compilado com gcc em uma máquina Linux. Seu programa será executado da seguinte forma:

```
./rotear grafo algoritmo origem destino
```

Os parâmetros são os seguintes:

- grafo: Nome do arquivo que contém o grafo;
- algoritmo: Número identificando algoritmo desejado. 1 para Dijkstra simples, 2 para Dijkstra alternante (entre origem e destino) e 3 Dijkstra com limitantes;
- origem: número do vértice de origem;
- destino: número do vértice de destino.

Por exemplo, a seguinte linha de comando deseja que seu programa carregue o grafo do arquivo `cg.g` e procure o caminho mínimo partindo do vértice 1021 para o 3042, pelo algoritmo de Dijkstra simples.

```
./rotear cg.g 1 1021 3042
```

O grafo está representado no arquivo da seguinte forma:

```
G n m D
N 1 lat long viz n1 d1 n2 d2 ... ni di
N 2 lat long viz n1 d1 n2 d2 ... nj dj
...
N n lat long viz n1 d1 n2 d2 ... ni di
```

A primeira linha dá informações gerais para preparar sua estrutura de dados para o grafo, n é o número de vértices, m é o número total de arcos (arestas orientadas) e D é o grau máximo do grafo. Os vértices do grafo são numerados de 1 a n em sequência. Para cada vértice há uma linha "N"com o número do vértice, sua latitude `lat` e longitude `long` (números de ponto flutuante em graus), o número de vizinhos `viz` (vértices que você pode chegar a partir do vértice) e então, para cada vizinho, seu número e a distância para o mesmo em quilômetros (os cálculos foram truncados para precisão em metros, mas é provável que a precisão seja pior). Veja exemplos no Moodle.

Não faça suposições sobre o grafo. Em particular, não o considere conexo e não suponha que se você pode ir de a para b significa que você possa voltar de b para a. Contudo, você sabe que as distâncias não são negativas.

O resultado deve ser apresentado na saída padrão e consiste de uma lista de números em algumas linhas:

```
enc dist
v1 p1 v2 p2 v3 p3 ... vi pi
visitados
```

A primeira linha contém apenas um inteiro `enc` indicando se um caminho foi encontrado (1) ou não (0) e a distância do menor caminho. A segunda linha contém os vértices que formam o caminho e seus pesos (distâncias em relação ao vértice de origem, ou destino). Note que o v_1 deve ser o vértice de origem, v_i o de destino e $p_1 = 0,0$. Quando o algoritmo for o Dijkstra simples ou com limitantes, os pesos serão crescentes e $p_i = dist$. Quando o algoritmo for Dijkstra duplo os pesos crescem até um vértice intermediário e decrescem depois até o vértice de destino, tal que $p_i = 0,0$. Por fim, a terceira linha indica o número de vértices visitados (contando origem e destino). As distâncias (e pesos) devem ser impressos sempre com três casas decimais. Veja exemplos no Moodle.

Caso não exista caminho entre a origem e o destino, a saida deve ser apenas:

```
0 0.000
v1 0.000 v2 0.000
visitados
```

Sendo v_1 e v_2 os números dos vértices de origem e destino. Não esqueça que visitados deve listar o número de vértices consultados.

Observação para implementação de Dijkstra com limitantes: trunque o valor da distância calculada entre um ponto e o destino para múltiplos de $0,001$, para garantir que este valor seja menor que a distância real e evitar erros da heurística.

Nota: eu utilizei tipos `double` para armazenas e calcular as distâncias. Se você usar `float`, alguns caminhos podem ficar diferentes.

Entrega e Avaliação

O programa deve ser entregue pelo Moodle até às 23:55 do dia 21/07/2013. Seu código deve ser escrito em Linguagem C para compilar com o `gcc` e executar em Linux. Ele deverá se chamar `rotear.c` e será compilado pelo comando

```
gcc -o rotear rotear.c -std=c99 -pedantic -lm
```

A minha opinião sobre seu código pode ser influenciada pelo quantidade e gravidade dos avisos que o compilador emitir.

A avaliação considerará o código fonte e sua execução correta. Comente claramente o que cada trecho de código faz. Quanto mais claro e simples, melhor. A nota vai de 0,0 a 10,0 sendo que 3 pontos correspondem a avaliação do código fonte e 7 pontos a completude da tarefa, distribuídos da forma abaixo:

Algoritmo	Peso
Dijkstra	3,0
Dijkstra alternante	2,0
Dijkstra com limites	2,0