Caminhos Mínimos (Com Custo Positivo e Algoritmo de Dijkstra)

Aristiklever R. Sousa[1]

[1]Ciências Da Computação – Universidade Estadual Vale Do Acaraú (UVA)
Sobral – CE – Brasil

aristiklever_rodrigues@uvanet.com.br

Abstract. *This article clears a little bit on a subject in very important graphs theory for the understanding of logical reasoning: minimum paths. It starts giving an introduction to the subject stating what problems are presented and how they categorize, after this is explained a bit about the general problem and their applications. Moreover, the algorithm used as a source in this article is presented and explained, through the ways to structure it and the most important differences of implementation to the other. And to finish, we have the conclusion, highlighting the syntheses of knowledge presented by me in this article.*

Resumo. *O presente artigo descorre um pouco sobre um assunto na Teoria dos Grafos muito importante para o entendimento do raciocínio lógico: Caminhos Mínimos. Ele inicia dando uma introdução ao assunto informando de que forma os problemas são apresentados e como eles se categorizam, após isso é explicado um pouco sobre o problema geral e suas aplicações. Mais adiante, o algoritmo utilizado como fonte nesse artigo é apresentado e explicado, passando pelas formas de se estruturar ele e as diferenças mais importantes de uma implementação para a outra. E, para finalizar, temos a conclusão, destacando as sínteses do conhecimento apresentado por mim nesse artigo.*

1. Introdução

Os estudos de Caminhos Mínimos se fazem necessário quando há a necessidade de se obter uma eficiência maior em relação a percorrer o grafo de um ponto s a t. Sempre que estruturas de dados como os grafos são usados para representar algo real, a eficiência no rastreio de informações é altamente requerida e o estudo de Caminhos Mínimos, fazendo uso de suas estratégias, torna-se necessário.

Portanto, o estudo dos Caminhos Mínimos tem como objetivos estudar a melhor maneira de chegar em um ponto w partindo de outro ponto v, em outras palavras, analisar o caminho de menor comprimento, assumindo que há pelo menos um caminho entre esses pontos e que a distância entre dois pontos adjacentes tem como valor 1. Além disso devemos analisar os Caminhos de Menor Custo também, já que é nele que iremos determinar qual rota (caminho no grafo) realmente é mais eficiente que outra.

Para análise de Caminhos de Custos Mínimos usa-se como algoritmo o de Dijkstra, que possui implementações em várias linguagens e que tem o objetivo de resolver, computacionalmente, o problema. Ele será abordado com mais detalhes posteriormente neste artigo.

Este artigo abordará um pouco sobre a descrição geral do problema de Caminhos Mínimos e outros aspectos, mais internos, que devem ser colocados dentre os assuntos estudados pois são de extrema importância para a compreensão. Será desmonstrado exemplos de aplicações do problema, será explicado um pouco sobre um dos algoritmos mais usados neste problema e será demonstrado sua implementação, juntamente com alguns comentários sobre. No fim, haverá considerações finais a cerca do tema.

2. Problema Geral

Dado um grafo G e vértices v e w, cálcular o menor caminho em G que tenha origem em v e término em w.

3. Aplicações

Os problemas de Caminhos Mínimos possuem diversas aplicações e versões diferentes.

Descobrir o menor caminho entre dois quarteirões não adjacentes seria um bom exemplo de problema a ser resolvido com Caminhos Mínimos, já que devemos selecionar o menor trajeto entre estes quarteirões.

Outro exemplo seria o de roteamento de rede, onde cada roteador possui uma tabela de roteamento,a qual armazena informações sobre quais caminhos estão disponíveis naquele momento e quais não, havendo aqui um cálculo do melhor caminho, levando em consideração um métrica específica para cada ligação entre roteadores adjacentes. Nesse caso, iria se calcular o Caminho de Custo Mínimo.

O cálculo de melhor rota para se seguir em uma viagem também evidencia a ideia de busca pelo caminho menos custoso. Nesse caso, várias variáveis estão em jogo, pois o tempo que se leva para ir de uma cidade a outra pode ser modificado depedendo, por exemplo, do modo que você pretende ir (a pé, de bicicleta, de motocicleta, etc.), além do próprio status das pistas (se alguma possui congestionamentos, em quais pontos o sinal está fechado, etc).

Trazendo agora para dentro da informártica, a análise de melhor caminho entre as trilhas de placas-mãe também é outro exemplo dos Caminhos Mínimos, já que deverá ser estabelecido o melhor caminho possível para que haja eficiência na comunicação entre os componetes existentes na placa, como entre o processador e a(s) memória(s), ou entre os dispositivos I/O e o próprio processador, nesse último caso o cálculo de caminho menos custo é de extrema importância, já que de todos os componentes existentes na placa, os mais distantes do processador são eles (Input/Output).

4. Custos em caminhos

Antes de demonstrar o algoritmo que é a solução para este problema, deve-se estudar os custos arbitrários nos arcos, que é de suma importância para entendimento do que vai se seguir. Deve se atentar também que a definição “Caminho Mínimo” é dúbia e pode hora representar o comprimento do caminho, hora representar o custo do caminho (se o grafo permitir a atribuição de custos às arestas).

Um grafo com custos em seus arcos possui uma subdivisão na classificação desses custos, podendo ser de custos *positivos* ou *negativos*. Os problemas com custos positivos são aqueles onde um pedaço do caminho sempre será mais barato que o caminho todo

e são estes problemas que o algoritmo de Bellamn-Ford, o para Dags e o de Dijkstra solucionam.

Os problemas com custos negativos são bem mais complexos e escorregadios, pois nos casos onde há valores negativos para os arcos, um pedaço do caminho pode ser mais caro que o caminho inteiro. Essa ideia de ser mais caro se dá porque os arcos negativos representam ganho de eficiência, não gasto. Se em um caminho há arcos negativos e positivos, os subcaminhos que possuem valores positivos podem resultar em um valor maior que a do caminho inteiro, pois o caminho inteiro irá considerar as subtrações dos arcos negativos.

5. Dijkstra

Como já foi exposto, existem alguns algortimos que tratam de maneira eficiente o problema dos custos mínimos sob custos positivos, tais como o de Bellamn-Ford e o para Dags. Neste artigo, o foco será específico para o algortimo de Dijkstra, que aborda as Árvores de Caminhos Baratos (problema da CPT).

O algoritmo de Dijkstra, rebendo um grafo G de custos positivos e um vértice arbitrário s, constrói uma subárvore radicada T de G, englobando todos os vértices alcançáveis por s. Se, porventura, todos os vértices deo grafo G estiverem ao alcance de s, a árvore raizada T será também geradora.

O algoritmo é relativamente simples, mas aborda problemas de grande importância. Para que seja possível explicá-lo de meneira clara e abstrata deve-se primeiro entender o conceito de *franja*. Franja em uma subárvore radicada T de um grafo G é o conjunto de arcos que possuem como ponta inicial um vértice já pertencente a árvore geradora T e, como ponta final, um vértice fora de T.

Como o algoritmo é iterativo e faz uso de um vetor para armezenar as informações de distância ($dist[]$) entre o vértice raiz e os alcançáveis por ele, o conceito de franja passa a ser facilmente entendível, pois o fluxo se torna claro. Primeiramente, o único vértice da subárvore T é s e o único valor de dist[s] é 0. A partir daí ele vai, enquanto a franja não estiver vazia faça:

1. Escolher na franja um arco v-w cujo o resultado da soma $dist[v] + c_{vw}$ seja o menor dentre os outros;
2. Colocar o arco v-w e o vértice w a T, pois até esse momento o vértice w estava fora de T. Daí vem o porque de na franja haver conjuntos de arcos quais tem pontas finais fora de T;
3. Faça $dist[w] = dist[v] + c_{vw}$.

   O algoritmo utiliza o c_{vw} como a representação do valor de custo do arco v-w.

Note que a forma de implementação acima acaba por ser ineficiente, já que ela refaz o cáculo de toda a franja a cada iteração, o problema chega a ter tempo $V * A$, pois, no pior caso, todos os vértices seriam alcançáveis por v e todos os arcos seriam analisados. Para que haja uma melhora, deve-se adicionar em T apenas aqueles arcos que, aparentemente, são os melhores para aquele momento.

6. Dijkstra Eficiente

Uma forma mais interessante de se pensar nesse algoritmo seria fazer uso do conceito de vértice maduro, qual define que um vértice só está maduro se todo o seu leque de saída

estiver sido examinado.

Para as iterações agora, vamos utilizar um vetor pa[] que reprensetará a subárvore radicada T o vetor de custo (potencial) dist[] e um conjunto de vértices maduros. No ínico da primeira iteração, v é o único vértice de T, todos são imaturos, $dist[v]$ vale 0 e $dist[s]$ vale ∞ para todo $s \neq v$.

O funcionamento geral será, enquanto houver vértices imaturos:

1. Seja w um vértice imaturo de T que minimiza dist[];
2. Para cada arco w-x de G que está *tenso* faça $dist[x] = dist[w] + c$ e $pa[x] = w$, o;
3. Declare w maduro.

Desse modo, o desepenho aumenta sua eficiência para $V^2 + A$ no pior caso utilizando lista de adjacências (que só visita os arcos que realmente existem). Em outras palavras, se todos os vértices são alcançáveis pelo inicial e, por exemplo, tem-se um conjunto de 5 vértices no grafo, então haverá 5 iterações (para alcançar os imaturos) com 5 verificações em cada uma delas (para saber qual vértice imaturo possui o menor custo), totalizando $5 * 5$ ou V^2. Levando em consideração que a cada iteração ele vai percorrer todos os arcos do vértice atual, temos: $V^2 + A$.

Implementação em C dessa parte:

```
void GRAPHcptD1( Graph G, vertex s, vertex *pa, int *dist)
{
   bool mature[1000];
   for (vertex v = 0; v < G->V; ++v)
      pa[v] = -1, mature[v] = false, dist[v] = INT_MAX;
   pa[s] = s, dist[s] = 0;

   while (true) {
      // escolha de y:
      int min = INT_MAX;
      vertex y;
      for (vertex z = 0; z < G->V; ++z) {
         if (mature[z]) continue;
         if (dist[z] < min)
            min = dist[z], y = z;
      }
      if (min == INT_MAX) break;
      // atualizacao de dist[] e pa[]:
      for (link a = G->adj[y]; a != NULL; a = a->next) {
         if (mature[a->w]) continue;
         if (dist[y] + a->c < dist[a->w]) {
            dist[a->w] = dist[y] + a->c;
            pa[a->w] = y;
         }
      }
      mature[y] = true;
   }
}
```

6.1. Há como ganhar mais eficiência?

Bem, a resposta é sim. A implementação acima percorre todos os vértices a cada iteração para determinar qual será o que vai ser relaxado naquela iteração. Para minimizar os custos disso, pode-se usar uma fila de prioridades, onde, nesse caso, levaria em consideração o valor do custo de cada vértice, fazendo com que o com menor valor sempre ficasse mais no topo da fila.

Implementação dessa ideia em C:

```
void GRAPHcptD2( Graph G, vertex s, vertex *pa, int *dist)
{
   bool mature[1000];
   for (vertex v = 0; v < G->V; ++v)
      pa[v] = -1, mature[v] = false, dist[v] = INT_MAX;
   pa[s] = s, dist[s] = 0;
   PQinit( G->V);
   for (vertex v = 0; v < G->V; ++v)
      PQinsert( v, dist);

   while (!PQempty( )) {
      vertex y = PQdelmin( dist);
      if (dist[y] == INT_MAX) break;
      // atualizacao de dist[] e pa[]:
      for (link a = G->adj[y]; a != NULL; a = a->next) {
         if (mature[a->w]) continue;
         if (dist[y] + a->c < dist[a->w]) {
            dist[a->w] = dist[y] + a->c;
            PQdec( a->w, dist);
            pa[a->w] = y;
         }
      }
      mature[y] = true;
   }
   PQfree( );
}
```

Antes do ínicio da primeira iteração todos os valores de custos (que são padrões até aquele momento INT_MAX) de cada arco são inserido na fila de prioridades. Todas as vezes que houver uma alteração no vetor dist[], para alterar o custo de um arco, haverá uma reordenação da fila, para que ela sempre mantenha as prioridades corrretas. Nesse caso, essa fila foi implementada utilizando um Heap, então assumiremos que tudo relacionado a fila consumirá um tempo limitado a $\log N$.

Como no pior caso o vértice inicial é alcançável por todos os outros, temos um valor de iterações equivalentes a V resultando em $V \log N$, podendo atingir $A \log N$ no caso de $A \geq V - 1$ (na maioria das vezes).

7. Conclusão

Após as análises de cada parte do assunto de Caminhos Mínimos, seus comprimentos e custos, entende-se que o apredizado de grafos como um todo e, mais especifica-

mente, desses problemas torna-se indispensável, não só para aprefeiçoamento da lógica de programação, mas também como forma de entender como certos fluxos da vida real funcionam e de que maneira pode-se melhorá-los. O algoritmo de Dijkstra é um belo exemplo de que dependendo da forma de implementação, o algoritmo terá um bom desenpenho ou serátão ruim que durará vários segundos.

Conseguir determinar os melhores caminhos a se tomar e também as melhores formas para se calcular isso é o que faz tanto a teria de grafos, quando a Análise de Algoritmos fortes áreas dentro das Ciências da Computação.

[Feofiloff 2020]

References

Feofiloff, P. (2020). Algoritmos para grafos via sedgewick.