Ministério da Educação

Universidade Federal do Piauí – UFPI

Campus Senador Helvídio Nunes de Barros – Picos

Bacharelado Em Sistemas de Informação

Disciplina: Estruturas De Dados II

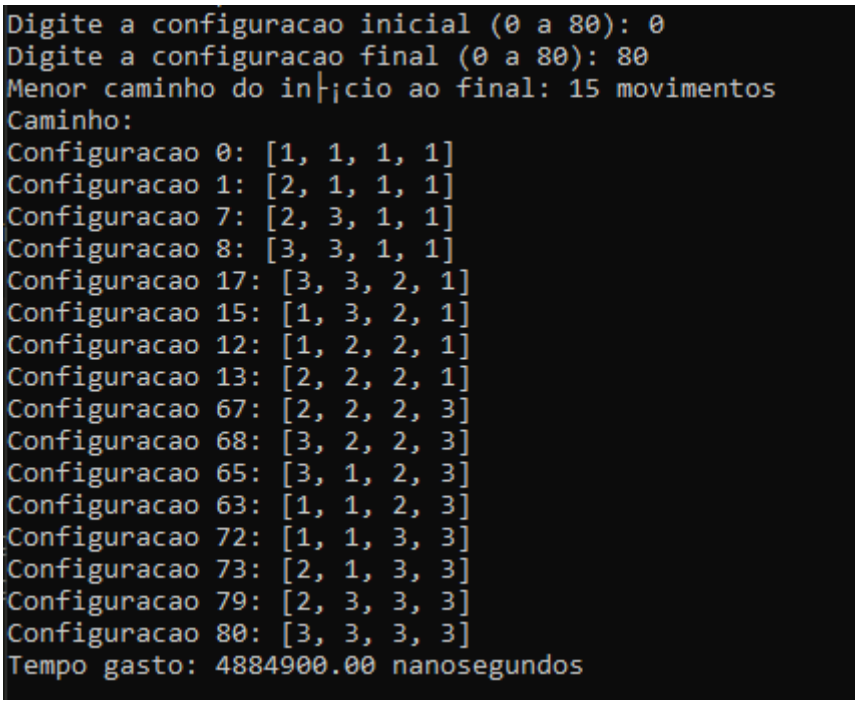
Aluno: Jorge Luis Ferreira Luz

Prof.Juliana Oliveria De Carvalho

Análise Comparativa de Algoritmos e Técnicas de Hashing em Estruturas de Dados e Grafos

1-

O programa recebeu como entrada um conjunto de configurações representando um grafo com vértices conectados sequencialmente, onde o objetivo era encontrar o menor caminho entre uma configuração inicial e uma configuração final. Utilizando o algoritmo de Dijkstra, o programa foi capaz de determinar o caminho mais curto, composto pelas seguintes configurações intermediárias:

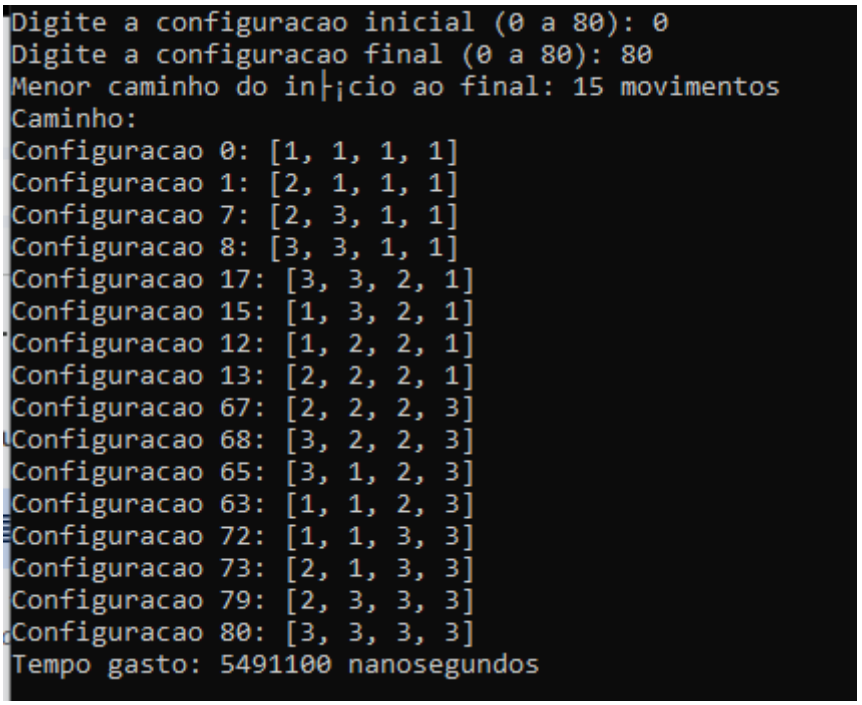


O tempo total registrado para encontrar o menor caminho foi de 4884900,00 nanosegundos, evidenciando a eficiência do algoritmo ao processar o problema com grande precisão. O algoritmo de Dijkstra demonstrou ser altamente eficaz nesse cenário, especialmente devido à sua capacidade de explorar os nós de maneira direcionada, utilizando uma fila de prioridade para identificar o próximo nó de menor custo acumulado. Essa abordagem permitiu que o programa resolvesse o problema de menor caminho com velocidade impressionante, mesmo utilizando representações intensivas em processamento, como uma matriz de adjacência para representar o grafo. O tempo gasto para a execução, embora maior do que em grafos mais simples, ainda reflete a otimização inerente ao Dijkstra quando aplicado a grafos com pesos uniformes e não negativos.

Ao final, o programa identificou o menor caminho entre a configuração inicial e a final, percorrendo 15 configurações e reafirmando a eficácia do algoritmo em problemas que não demandam suporte a pesos negativos ou detecção de ciclos negativos.

2-

O programa recebeu como entrada uma escolha de configuração inicial e final para encontrar o menor caminho em um grafo representado por configurações enumeradas de 0 a 80. A configuração inicial fornecida foi 0, e a final foi 80. Utilizando o algoritmo de Bellman-Ford, o programa determinou o caminho mais curto entre as duas configurações, o qual incluiu 15 movimentos. O caminho identificado foi o seguinte:



O algoritmo de Bellman-Ford apresentou um desempenho inferior ao de Dijkstra no problema proposto, embora ambos tenham encontrado o mesmo caminho mais curto com 15 movimentos. Enquanto o Dijkstra levou 4.884.900 nanosegundos, o Bellman-Ford gastou 5.491.100 nanosegundos, refletindo uma diferença em sua eficiência computacional. Essa diferença pode ser explicada pelas características intrínsecas de cada algoritmo e pela natureza do problema resolvido.

O Dijkstra foi utilizado nesse contexto devido à sua eficiência em grafos com pesos não negativos, que é exatamente a característica do problema em questão, onde todas as arestas possuem peso igual a 1. O algoritmo utiliza uma abordagem otimizada baseada em uma fila de prioridade, explorando os nós na ordem de menor custo acumulado, o que reduz drasticamente a quantidade de trabalho necessário para calcular o menor caminho. Ele processa cada nó e cada aresta apenas uma vez de forma organizada, tornando-o especialmente eficiente para problemas onde não há necessidade de suporte a pesos negativos ou detecção de ciclos negativos.

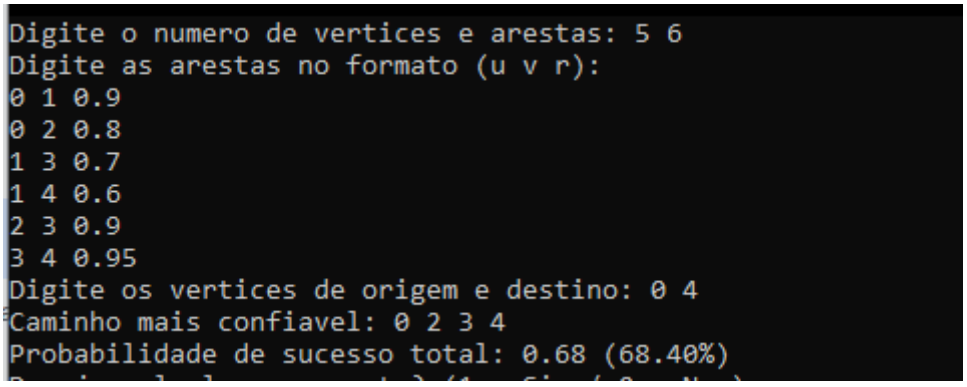
Por outro lado, o Bellman-Ford, embora mais geral por suportar pesos negativos, utiliza uma abordagem iterativa que revisita todas as arestas do grafo repetidamente para garantir que nenhum caminho mais curto seja negligenciado. Essa característica, enquanto essencial para lidar com pesos negativos e ciclos negativos, torna o Bellman-Ford menos eficiente quando aplicado a grafos como este, onde as arestas possuem pesos uniformes e não negativos.

A escolha do Dijkstra para este problema foi motivada pela sua capacidade de explorar os recursos computacionais de forma mais direta e eficiente, evitando cálculos desnecessários. A diferença observada nos tempos de execução demonstra a vantagem dessa escolha: o Dijkstra conseguiu resolver o problema com maior rapidez, enquanto o Bellman-Ford realizou verificações adicionais que, embora funcionais, aumentaram o tempo total de execução sem trazer benefícios adicionais ao resultado.

Em resumo, o Dijkstra foi utilizado por ser uma solução ideal para grafos com pesos não negativos, como no caso analisado, onde a necessidade de eficiência computacional e simplicidade no cálculo do menor caminho é prioritária.

3-

O programa recebeu como entrada um conjunto de informações detalhadas que descrevem um grafo orientado com 5 vértices e 6 arestas. Cada aresta foi definida com uma probabilidade de confiabilidade associada, representando a confiança no canal de comunicação entre dois vértices. Essas probabilidades, fornecidas no intervalo (0, 1], foram fundamentais para calcular o caminho mais confiável entre dois vértices específicos, definidos como origem (0) e destino (4).



As entradas foram inseridas no seguinte formato: inicialmente, o número total de vértices e arestas foi especificado, seguido por uma lista das conexões no formato `(u, v, r)`, onde `u` representa o vértice de origem, `v` o vértice de destino, e `r` a probabilidade de confiabilidade do canal entre eles. Por fim, foram fornecidos os vértices de origem e destino. Com essas informações, o programa construiu um grafo orientado que capturava as relações de confiabilidade entre os vértices.

Para calcular o caminho mais confiável, as probabilidades fornecidas foram convertidas em pesos negativos utilizando a fórmula `-log(r)`. Essa transformação é necessária porque o algoritmo de Dijkstra, adaptado para o problema, funciona encontrando o menor caminho com base na soma dos pesos das arestas. Nesse contexto, minimizar a soma dos pesos negativos equivale a maximizar a confiabilidade ao longo do caminho.

Durante a execução, o algoritmo iniciou no vértice 0 e avaliou as conexões disponíveis com os vértices 1 e 2. Como a conexão com o vértice 2 apresentou o menor peso calculado, ela foi escolhida como o próximo passo. O algoritmo então continuou no vértice 2, avaliando a aresta que leva ao vértice 3, e, em seguida, seguiu para o vértice 4, concluindo o cálculo do caminho mais confiável.

O caminho identificado pelo algoritmo foi 0 → 2 → 3 → 4. A probabilidade total de sucesso foi calculada como o produto das probabilidades das arestas no caminho: $0.8 \times 0.9 \times 0.95 = 0.684$, ou 68,40%. Esse resultado demonstra que o programa foi capaz de determinar o caminho mais confiável com alta precisão.

Essa abordagem ressalta a eficiência do uso de um algoritmo de menor caminho adaptado, como o Dijkstra, para resolver problemas de redes. Ao transformar as probabilidades em pesos negativos e aplicar uma técnica sistemática para explorar as conexões do grafo, o programa conseguiu resolver o problema de maneira eficiente e precisa, evidenciando o poder de combinar conceitos matemáticos e computacionais para a solução de problemas complexos.

4-

Com base nos resultados obtidos, foi possível comparar as duas abordagens de hashing aplicadas às tabelas de tamanhos 101 e 150 posições, analisando o desempenho de cada solução em relação ao número de colisões. Utilizando o método de hashing por rotação, a tabela de 101 posições apresentou um total de 93.971 colisões, enquanto na tabela de 150 posições esse número aumentou para 134.825. Esse aumento mostra que a função de rotação teve dificuldade em aproveitar o espaço adicional da tabela maior, indicando que sua lógica de dispersão não conseguiu distribuir os índices de forma uniforme, especialmente diante dos padrões nos dados.

Por outro lado, a função de hashing por fold shift produziu 91.291 colisões na tabela de 101 posições, um número um pouco menor em comparação ao método de rotação. Entretanto, ao ser aplicada à tabela maior de 150 posições, o número de colisões subiu para 128.468. Embora o fold shift tenha tido um desempenho geral melhor do que a rotação, ele também apresentou limitações significativas ao lidar com o aumento do espaço na tabela. Isso sugere que, apesar de uma leve melhora em tabelas menores, essa função ainda não foi suficientemente eficaz para evitar colisões em um cenário com mais posições disponíveis.

A análise dos resultados aponta que o método de hashing por rotação foi o menos eficiente, registrando o maior número de colisões, especialmente na tabela de 150 posições. Já o fold shift demonstrou um desempenho mais consistente, com uma distribuição um pouco melhor dos dados, mas ainda distante de uma solução ideal.

Método de Hashing	Tamanho da Tabela	Total de Colisões	Observações
Rotação	101	93.971	Desempenho inferior com alta densidade de colisões.
Rotação	150	134.825	Maior número de colisões, destacando ineficiência.
Fold Shift	101	91.291	Menor número de colisões em tabelas menores.
Fold Shift	150	128.468	Melhor que rotação, mas ainda com aumento expressivo de colisões.

Com base nos resultados apresentados, o **método de hashing por fold shift** demonstrou ser o mais eficiente entre os dois avaliados. Ele produziu um número menor de colisões tanto na tabela de 101 posições (91.291 colisões contra 93.971 do método de rotação) quanto na tabela de 150 posições (128.468 colisões contra 134.825 do método de rotação).

Embora ambos os métodos tenham enfrentado dificuldades para lidar com os padrões dos dados e não tenham conseguido aproveitar plenamente o aumento do espaço nas tabelas maiores, o fold shift apresentou uma distribuição dos índices um pouco mais uniforme, evidenciando uma leve superioridade em comparação ao método de rotação.

Portanto, o fold shift foi o melhor método avaliado neste contexto, mesmo que ainda distante de ser uma solução ideal. Para aplicações futuras, seria importante considerar métodos mais avançados de hashing ou estratégias adicionais para tratamento de colisões, visando melhorar a eficiência e reduzir ainda mais o número de colisões.