**Relatório**

Algoritmos e Estruturas de Dados I

**Universidade de São Paulo**

Campus São Carlos

Alunos: Renan Trofino Silva, Guilherme Pereira de Sá, Augusto Fernandes Ildefonso

Professor: Rudinei Goularte

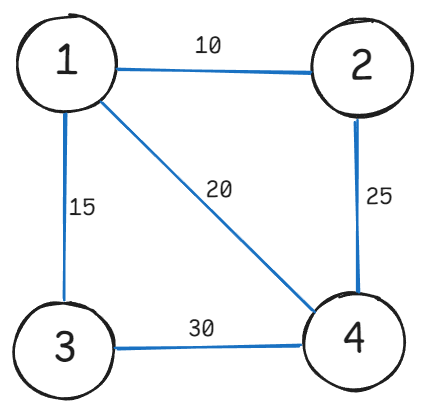
Sumário

[Introdução ao projeto.c 3](#_Toc179213899)

[Backtracking, um Brute Force mais eficiente 4](#_Toc179213900)

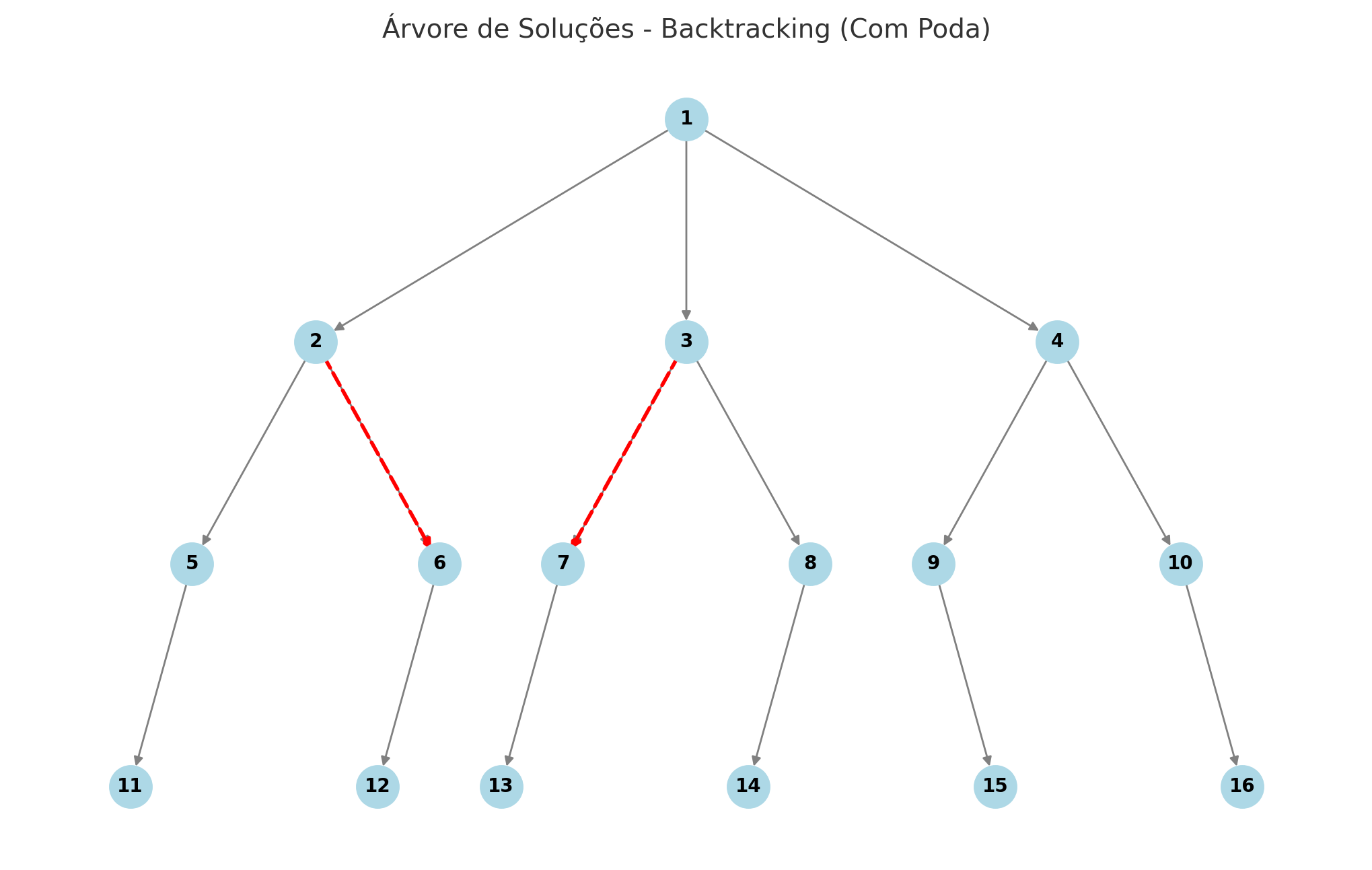
# Introdução ao projeto.c

Na pasta-pai entregada para a resolução deste problema, *O Problema do Caixeiro Viajante*, onde precisamos optar pelo menor custo de distâncias que o viajante tem de fazer ao visitar cada uma das cidades, saindo, obviamente, da cidade de origem. Leremos de um arquivo externo as informações número de cidades, cidade de origem, número de arestas que o grafo possuirá, logo, o número de possibilidades que teremos, por exemplo:

Este grafo ao lado não possui aresta de 2 para 3.

Portanto o número de arestas é 5.

Backtracking, um Brute Force mais eficiente

A ideia aqui é de "podar", já que tratamos de uma árvore de "processos", escolhas, a serem tomadas por cada cidade, e o algoritmo ao analisar um "ramo" da árvore, percebendo que é pior que o anterior, ou seja, fixamos, tomamos o anterior, ou que aparecer, no nosso caso, ter o menor custo, como a melhor opção e apenas comparamos com o atual caminho seguido. E, sem algum caso, ultrapassar esse menor custo, já podemos retirar esse "ramo", essa possibilidade de caminho, como a que garantiria o menor custo. Então seguimos com a ideia do Brute Force, de analisar todas as combinações possíveis, porém, agora, retirando aquelas rotas que já começam com um custo acima do requerido.

Ao lado vemos como é representado o backtracking. E, como já falado anteriormente o backtracking remove podas que já não são uma solução local ótima. Então aqui teríamos um exemplo onde o custo de 1 – 2 – 6 e 1 – 3 – 7 já são maiores que o menor custo, ultrapassam algum limite pré-definido.

O algoritmo começa na cidade de origem e marca-a como visitada. Ele recursivamente tenta visitar todas as cidades não visitadas, empilhando cada cidade visitada. Se uma rota completa (visitando todas as cidades) é formada, o algoritmo calcula o custo da rota incluindo o retorno à cidade de origem. Se o custo da rota atual for menor que o menor custo encontrado, essa rota se torna a nova melhor solução. O algoritmo desfaz o último movimento (backtrack), desempilhando a cidade e marcando-a como não visitada, para explorar outras rotas.

Abaixo se encontra um gráfico que permite analisar a eficiência do algoritmo de backtracking e do algoritmo de força bruta:

Gráfico, Gráfico de linhas

Descrição gerada automaticamente

# Modelagem da Solução

Agora que foi explicada a lógica para resolver o problema, vamos analisar as estruturas de dados. As estruturas de dados escolhidas para modelar o problema foram a pilha e o item.

Escolheu-se a pilha pois no algoritmo de backtracking deseja-se ter um caminho de volta para os casos em que a rota atual já é mais cara que a melhor rota. Então, como a pilha permite essa operação de desfazer, optou-se por utilizá-la na implementação, no formato de pilha sequencial.

As vantagens do uso da pilha é a intrínseca função de desfazer (obtida pela pilha desempilhar) e também a sua fácil implementação. Já a sua limitação é justamente a fixa estrutura que segue política FILO, mas que no caso do projeto não chega a atrapalhar como sempre removeremos último elemento.

Já o item foi utilizado pois com ele é criada uma camada de abstração que permite utilizar qualquer tipo de dados dentro das outras estruturas de dados. Porém para o objetivo do projeto não é necessário usar a informação do valor do item, apenas utilizando as chaves dele é possível resolver o problema.

Como vantagem do uso da lista pode-se mencionar justamente a sua abstração que permite utilizar qualquer tipo de dado. Mas a desvantagem do uso do item para esse problema é que para resolver o problema só é necessário a chave do item, então uma simples pilha de inteiro bastaria, desse modo o uso do item, que contribui para um pequeno custo no código, é a desvantagem.

# Cálculo de Complexidade

O algoritmo continua tendo o pior caso como o (n-1)!, porém, na prática, ele diminui significativamente os casos cedo, reduzindo número de caminhos que devem ser explorados. A eficiência do Backtracking se torna mais relevante quanto mais aumentarmos o número de entradas. Para provarmos nosso tempo, precisamos fazer uma análise empírica do backtracking comparando-a com o Brute Force.

// time-back-perm.c

# Referências

* **Wikipedia.** Backtracking. Disponível em: <https://pt.wikipedia.org/wiki/Backtracking>. Acesso em: 07/10/2024.
* **Wikipedia.** Problema do caixeiro-viajante. Disponível em: <https://pt.wikipedia.org/wiki/Problema_do_caixeiro-viajante>. Acesso em: 07/10/2024.
* **Livro.** Algorithm Design Manual. Disponível em: <https://www.google.com/search?q=Algorithm+Design+Manual>. Acesso em: 07/10/2024.
* **Livro.** Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, e Clifford Stein. Disponível em: <https://www.google.com/search?q=%22Introduction+to+Algorithms%22+-+Thomas+H.+Cormen%2C+Charles+E.+Leiserson%2C+Ronald+L.+Rivest%2C+e+Clifford+Stein>. Acesso em: 07/10/2024.
* **GeeksforGeeks.** Backtracking Algorithms. Disponível em: <https://www.geeksforgeeks.org/backtracking-algorithms/?ref=header_outind>. Acesso em: 07/10/2024.
* **GeeksforGeeks.** Travelling Salesman Problem Implementation Using Backtracking. Disponível em: <https://www.geeksforgeeks.org/travelling-salesman-problem-implementation-using-backtracking/?ref=header_outind>. Acesso em:
* 07/10/2024.