Problema do Caixeiro Viajante: Comparação entre um algoritmo exato e um algoritmo aproximado

Victor Deluca Almirante Gomes[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Roraima (UFRR) – Boa Vista, RR, Brazil.

victorlinkp@hotmail.com

Abstract. *This work presents the Traveling Salesman Problem, as well as two algorithms to solve it: One with an exact solution of exponential cost, and another with an approximated solution of polynomial cost. Then, both solutions are compared through several test cases, using execution time as a metric. Finally, it presents a problem that can be solved with the algorithms presented in this article, and an adaptation of the approximated algorithm to solve instances of this problem.*

Resumo. *Este trabalho apresenta o problema do Caixeiro Viajante, bem como dois algoritmos para resolvê-lo: Um de solução exata de custo exponencial, e outro de solução aproximada de custo polinomial. Ainda, são comparadas ambas as soluções através de vários testes utilizando como métrica o tempo de execução. Por fim, é apresentado um problema que pode ser resolvido utilizando os algoritmos apresentados neste artigo, e uma adaptação do algoritmo aproximado a instâncias deste problema.*

1. O problema do Caixeiro Viajante

O problema do Caixeiro Viajante é definido da seguinte forma: "Um caixeiro viajante deseja visitar um conjunto de cidades passando por cada cidade exatamente uma única vez, e então retornar à sua cidade de origem. Ajude-o a encontrar a rota de menor custo para sua viagem".

Esta definição pode ser formalizada como se segue: "Dado um conjunto de vértices $V = \{v_0, v_1, ..., v_n\}$, um vértice inicial V_0, e uma função de distância entre dois vértices *dist*, encontrar a permutação *P* do conjunto V que minimize $F(P,P_0)$, onde $F(P,P_0)$ é dado por $F(P,P_0) = dist(P_0,P_1) + dist(P_1,P_2) + ... + dist(P_{n-1},P_n) + dist(P_n,P_0)$, e $P_0 = V_0$".

O problema do Caixeiro Viajante possui múltiplas variantes, como casos em que as arestas são simétricas, ou seja, $dist(v_i,v_j) = dist(v_j,v_i)$, e casos em que há mais de um caixeiro viajante trabalhando para cobrir um conjunto de vértices. Neste trabalho em particular, trataremos da instância em que as arestas são assimétricas, ou seja, $dist(v_i,v_j)$ não possui relação com $dist(v_j,v_i)$, e apenas um "caixeiro" está envolvido.

2. Algoritmos exatos: Força bruta

A forma mais direta de resolver o problema do Caixeiro Viajante utiliza o método de força bruta. Como o enunciado pede uma permutação do conjunto de vértices V, a ideia

é simplesmente gerar todas as permutações possíveis e dentre elas escolher a que possui custo mínimo. A complexidade é a mesma de gerar todas as permutações possíveis de um conjunto V, ou seja, O(N*N!).

```
P <- P.first_special_permutation()
BEST <- P
while P.has_next_special_permutation():
  P <- P.next_special_permutation()
  if P.cost < BEST.cost:
    BEST <- P
```

Figura 1: Algoritmo de força bruta. É definido um tipo especial de permutação, tal que o vértice S*ource* esteja sempre na primeira e na última posição da permutação.

3. Analisando a solução por força bruta

Tendo em mãos a solução mais básica, a pergunta que surge é: "Existe uma forma de melhorar esta solução? ". Para responder a esta pergunta, vamos analisar a solução por força bruta.

Inicialmente, note que podemos dividir nossa solução em subproblemas. A ideia é simples: Para encontrar $min(F(P,P_0))$, procuramos a melhor solução dentre $F((P-P_0),i)$, para todo i que puder ser alcançado a partir de P_0. Temos assim uma *subestrutura ótima*.

Além disso, podemos obter o mesmo subproblema $F((P-P_0),i)$ a partir de múltiplos problemas diferentes. Por exemplo, suponha que visitamos os vértices V_0, V_1, V_2 e V_3, nessa ordem. Precisamos agora da solução para $F((P-(V_0,V_1,V_2,V_3),i)$, que também pode ser obtida visitando os vértices V_0, V_2, V_1 e V_3, nessa ordem. Temos então *repetições de subproblemas.*

Com essas duas propriedades em mãos, é visível que podemos reduzir o custo deste problema utilizando Programação Dinâmica. Contudo, ao montarmos a solução por programação dinâmica (Que não será descrita aqui, pois não é o foco deste trabalho), notamos que seu custo é $O(N^2*2^N)$, que é melhor que O(N*N!), porém ainda muito custoso. Além disso, o custo de espaço também passa a ser $O(N^2*2^N)$, o que é horrível comparado com a solução anterior.

Contudo, não existe uma solução que garanta resultado exato com complexidade melhor que a solução por programação dinâmica. Muitas buscam reduzir o custo do caso médio, contudo seu pior caso sempre possui custo exponencial ou pior. Surge assim uma nova pergunta: "É realmente necessário obter o *melhor* caminho possível? ".

E, em muitos casos, a resposta é "não". Embora formalmente o problema do Caixeiro Viajante demande a solução de custo mínimo, em muitos casos existem várias soluções de custo próximo do mínimo que podem ser consideradas aceitáveis. Podemos reescrever o enunciado do problema como: "Ajude o Caixeiro a encontrar uma rota de custo baixo para sua viagem", e com essa simples mudança nosso problema se torna muito mais simples.

Existem diversos algoritmos que geram soluções aproximadas para problemas NP-Completos, e para um problema clássico como o que estamos tratando, não é muito

difícil encontrar um: Colônia de formigas, aproximações utilizando árvores geradoras mínimas, algoritmos genéticos, as soluções são diversas. Só precisamos escolher um.

Vamos então olhar para nosso problema sob uma perspectiva diferente. Suponha que nós já possuímos uma solução temporária P_i, que pode ou não ser uma solução ótima. Podemos realizar transformações aleatórias nesta solução, que podem ou não melhorar o resultado atual. Define-se como "transformação" qualquer operação sobre uma solução que a transforme em outra solução. Por exemplo, uma transformação sobre *P* poderia consistir em inverter elementos em posições *Pi, Pj, i!=j* aleatórias em nossa permutação.

Para gerar uma solução *boa*, portanto, basta realizar várias transformações na solução atual, mantendo as transformações que a solução, e descartando as que piorem. Executamos um número limitado de transformações (Ou limitamos o número de transformações ruins consecutivas), e ao final teremos uma solução muito melhor que a inicial (Ou pelo menos tão boa quanto). Este algoritmo é chamado de *Subida de Encosta (Hill Climbing)*.

Esta técnica busca um máximo local, ou uma solução muito próxima disso. Um máximo local é uma solução tal que nenhuma transformação aplicada a ela gere uma solução melhor. O problema é que o máximo local obtido pode ser uma solução muito inferior à solução ótima, não se qualificando como solução aceitável. Precisamos de um método para eliminar, ou ao menos reduzir esse risco.

Uma melhoria para este algoritmo consiste em modificar o curso de ação tomado quando uma transformação gera uma solução ruim. Em vez de rejeitá-la imediatamente, o novo algoritmo dá a ela uma chance, por assim, dizer, e considera aceitá-la, com uma probabilidade baixa. Dessa forma, podemos realizar múltiplas Hill Climbings em uma única execução de nosso algoritmo e, como resultado, visitar diversos máximos locais, ou soluções próximas disso, aumentando as chances de uma solução próxima à ótima ser encontrada.

Resta agora descobrir uma boa forma de definir essa probabilidade de aceitação. Uma probabilidade muito baixa em pouco difere da Hill Climbing convencional, pois visita muito poucos máximos locais. Uma probabilidade muito alta levaria a uma busca exaustiva, sem garantia de que quaisquer máximos seriam encontrados. O algoritmo de *Recozimento Simulado (Simulated Annealing)*, que será estudado nesse trabalho, propõe uma forma de determinar essa probabilidade.

4. Algoritmos de aproximação: Recozimento Simulado

O algoritmo de recozimento simulado possui suas raízes na física, baseando-se numa técnica de nome "recozimento" que busca remover "defeitos" em um material sólido através de banhos de calor. A ideia é iniciar o banho de calor a uma temperatura suficientemente alta, e lentamente ir reduzindo a temperatura, até que o resultado esperado seja obtido [Kirkpatrick et al. 1983].

O interessante sobre essa técnica é o comportamento que o elemento apresenta durante o banho de calor. Inicialmente, as partículas estão completamente dispersas, devido à alta temperatura. Contudo, conforme a temperatura vai diminuindo, as

partículas começam a convergir para uma posição de equilíbrio. O resultado é um sólido maleável e sem falhas.

O algoritmo de Recozimento Simulado opera da mesma forma. Nosso sólido é o espaço de soluções possíveis, onde cada átomo é uma solução. Nossa temperatura é um parâmetro que será constantemente reduzido ao longo do algoritmo, e a cada passo nós exploramos os arredores do nosso sólido com o objetivo de encontrar uma solução boa. Inicialmente, várias soluções são possíveis, e com o tempo a busca passa a ser mais restritiva.

Aplicando este conceito ao algoritmo proposto na seção anterior, a probabilidade de aceitação de uma solução "ruim" passará a ser definida pela temperatura atual e pela diferença entre os custos da solução ruim e da solução anterior a ela. Assim, conforme a temperatura diminui, mais improvável é a aceitação de uma solução, tornando mais restritivo o espaço de soluções possíveis. Mantemos a melhor solução encontrada até o momento, e ao final do algoritmo espera-se que essa melhor solução possua custo próximo à solução ótima. O pseudocódigo para essa técnica é apresentado na figura 2.

```
S <- generateArbitrary()
BEST <- S
T <- MAX_TEMPERATURE
while T > MIN_TEMPERATURE:
  it <- MAX_ITERATIONS
  while it > 0:
    S' <- transform(S)
    if S'.cost < S.cost:
      S <- S'
      if S'.cost < BEST.cost
        BEST <- S
    else:
      if probability(S'.cost - S.cost, T):
        S <- S'
    it <- it - 1
  T <- T * COOLING_FACTOR
```

Figura 2: Algoritmo de Recozimento Simulado

Basta agora aplicar essa técnica ao problema do Caixeiro Viajante. Inicialmente, gera-se uma solução arbitrária, e em seguida esta solução é constantemente melhorada utilizando o método previamente citado. A probabilidade de aceitação de uma solução ruim é dada por $e^{(-(Cj - Ci) / T)}$, conforme a literatura sugere, onde Ci é o custo da solução antiga e Cj o custo da solução atual. Para melhor confiabilidade, executa-se o algoritmo múltiplas vezes, sendo mantida a melhor solução.

5. Análise estatística dos algoritmos apresentados

5.1 Descrição das implementações

Para a análise estatística, as soluções utilizando força bruta e a técnica de recozimento simulado foram implementadas em C++. Ambos os algoritmos foram testados com entradas de diferentes tamanhos, e gráficos foram gerados para uma melhor análise de

suas performances assintóticas. Os códigos-fonte estão disponíveis em tsp_brute.cpp, e tsp_annealing.cpp, referindo-se respectivamente à solução por força bruta e à solução por recozimento simulado.

Para a implementação da solução por recozimento simulado, a transformação utilizada foi “Reverse”, que inverte as posições de todos os elementos em um intervalo [Ci, Cj]. A temperatura inicial T foi 100ºC, que foi reduzida por um fator de 10% após um limite de *it* = 50 iterações. O algoritmo era encerrado quando T <= 0.1ºC, e o número de repetições foi R = 100. A cada iteração, várias operações O(N) são feitas, sendo a complexidade total O(T*it*R*N), onde T, *it* e R são parâmetros determinados pelo programador.

Em ambas as implementações, a entrada é dada na forma:

V E

S1 D1 C1

S2 D2 C2

...

SE DE CE

S

Onde V é o número de vértices, E é o número de arestas, e as próximas E linhas representam uma aresta cada, onde uma aresta é dada por Si, sua origem, Ei, o destino, e Ci, seu custo. Por fim, S é o ponto de partida para a rota do Caixeiro Viajante.

5.1. Dados estatísticos

Foram estudadas instâncias com 3, 6, 9 e 12 vértices. As figuras 3.1 e 3.2 mostram o tempo de execução associado a cada uma dessas instâncias, para ambas as soluções. A figura 3.1 mostra os dados da solução por força bruta. A figura 3.2 exibe os dados da solução por recozimento simulado.

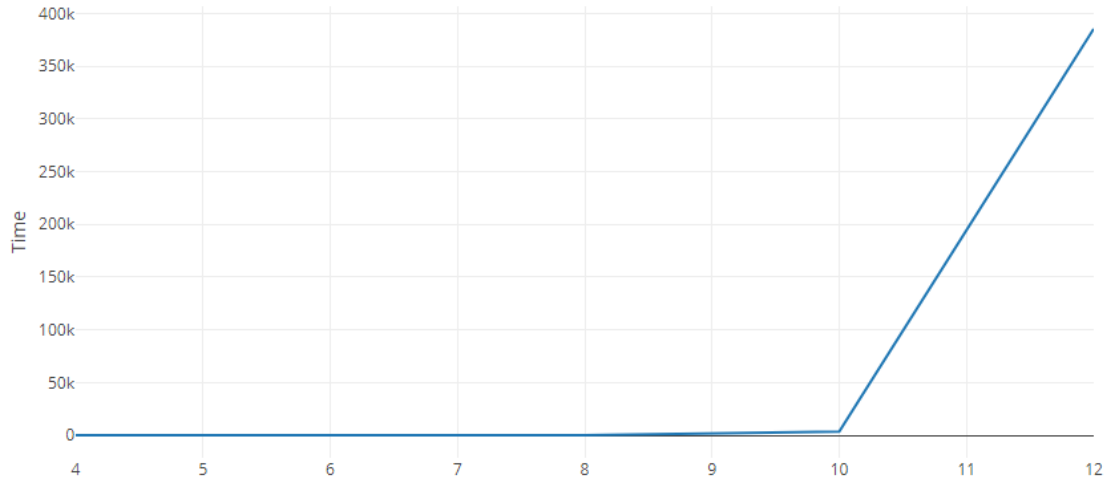

Figura 3.1: Custo de tempo relativo ao número de vértices dados, em milissegundos, para a solução por força bruta.

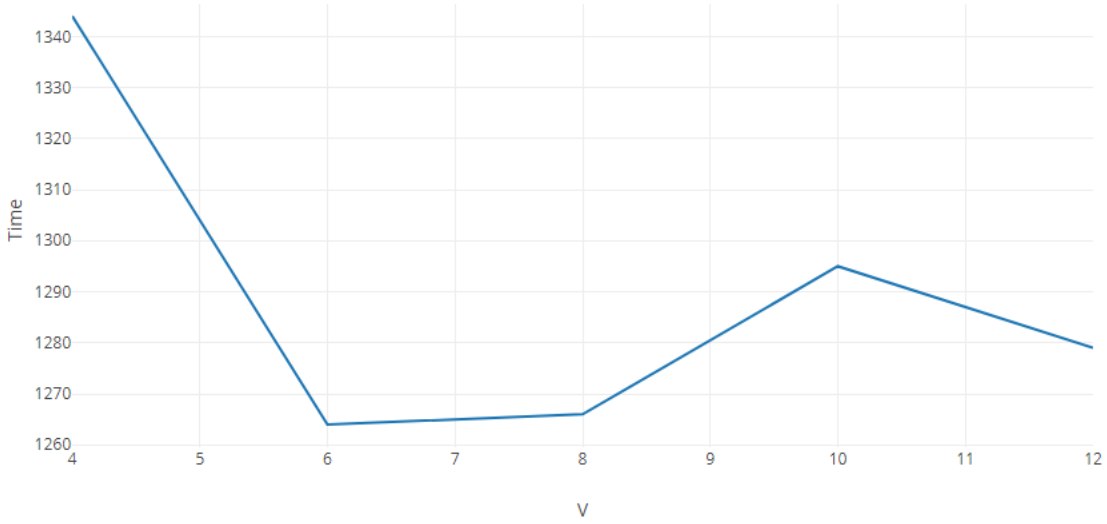

Figura 3.1: Custo de tempo relativo ao número de vértices dados, em milissegundos, para a solução por recozimento simulado.

Podemos notar que a solução por força bruta depende muito do número de vértices, uma vez que seu custo de tempo cresce de força exponencial conforme o número de vértices aumenta. Por outro lado, o número de vértices em pouco influencia a solução por recozimento simulado, onde parâmetros ditados pelo programador, como número de iterações e temperatura também devem ser levados em consideração. Além disso, o fator aleatório, e o próprio conteúdo da entrada também apresentam influência no tempo de execução.

De forma geral, contudo, podemos observar que o custo de tempo da solução por força bruta cresce muito rapidamente, enquanto que a solução por recozimento simulado mantém um custo de tempo quase que constante (Embora ele tenda a crescer quando o número de vértices é muito maior, afinal o algoritmo possui custo linear, não constante). Portanto, para entradas relativamente grandes (A partir de 10 vértices), o algoritmo de aproximação é uma escolha muito melhor.

6. Aplicação da solução proposta

Finalmente, tendo sido comprovada a eficiência da solução proposta, iremos aplica-la a um problema real, denominado Rota dos Carteiros: “Suponha que existam numa cidade N caixas de correio que precisam ser esvaziadas todos os dias dentro de um dado limite de tempo. A agência de correios possui K caminhões disponíveis, e deseja saber qual é o mínimo de caminhões que eles precisam despachar para esvaziar as caixas de correio no tempo determinado, ou se isso não é possível. Note que, ao final do limite de tempo, todos os caminhões devem estar de volta à agência”.

Inicialmente, note que nosso problema não está mais restrito a um único caixeiro viajante. Na verdade, precisamos determinar o número mínimo de caixeiros tais que o melhor caminho que todos eles podem cobrir custe no máximo o limite de tempo dado. Esta é uma variação do Multi-TSP (Versão do problema com múltiplos caixeiros), ou seja, precisamos primeiro adaptar nosso algoritmo a essa versão.

Nessa adaptação, é fixado um número de caixeiros, Ki, e o algoritmo busca descobrir se é possível obter uma solução de custo menor que o tempo limite com Ki caixeiros. Assim, geramos um vetor aleatório de caixeiros, cada um deles contendo (Ou não) alguns dos vértices do grafo, no caso, as caixas de correio. A partir desse vetor, são realizadas transformações que buscam melhorá-lo. Cada posição do vetor representa um caminhão, e ao final do algoritmo todos os elementos deste vetor são designados a este caminhão, na ordem em que eles foram dados.

Duas novas transformações são introduzidas: “Swap” troca elementos entre dois caixeiros diferentes, enquanto “Insert” remove um elemento de um caixeiro e o adiciona em outro. “Reverse”, a última transformação, é a mesma transformação introduzida no algoritmo original. As novas transformações garantem que estados com designações alternativas para cada caminhão serão visitados. Cada transformação possui 0.33% de probabilidade de ser escolhida.

O número de caminhões Ki é modificado iterativamente: Se a solução anterior não foi suficiente, talvez designar um novo caminhão seja o suficiente. Caso o número de caminhões e cidades seja muito grande (Superior a 1000, por exemplo), podemos alterar este procedimento para uma busca binária, ao custo de uma possível leve perda de precisão, uma vez que estamos lidando com algoritmos probabilísticos (Na pior das hipóteses, um caminhão desnecessário seria solicitado).

Os testes foram feitos utilizando instâncias pequenas da análise estatística da solução anterior. Modificações no limite de tempo e de caminhões foram feitas, com o objetivo de verificar a corretude da solução, que apresentou resultados adequados para todas as consultas. A implementação do algoritmo modificado pode ser encontrada no arquivo tsp_annealing_modified.cpp.

7. Conclusões

Neste trabalho, foi apresentado um algoritmo de aproximação para solucionar o clássico problema do Caixeiro Viajante, o qual foi comparado com a solução exata. Verificou-se que as soluções apresentadas não diferem significativamente, tornando a aproximação uma solução perfeitamente viável.

Naturalmente, em algumas instâncias pode ser necessário o resultado exato para cada consulta. Portanto, para analisar se uma solução aproximada é desejável ou não, basta perguntar-se se o custo a se pagar pelo erro na aproximação é aceitável quando comparado ao custo de tempo para se obter uma solução exata. Esta pergunta é muito mais fácil de se responder quando o erro na aproximação é mínimo, portanto trabalhos futuros podem focar no estudo de aproximações mais exatas.

References

Kirkpatrick, S., Gelatt Jr, C. D. and Vecchi, M. P., 1983, Optimization by simulated annealing. Science 220:671-680.