Reduzindo para complexidade linear a solução do TSP com datas de liberação em caminhos com depósito na extremidade

Thailsson Clementino[1], **Rosiane de Freitas**[1]

[1] Instituto de Computação - Universidade Federal do Amazonas

{thailsson.clementino, rosiane}@icomp.ufam.edu.br

Abstract. *This work explores the Traveling Salesman Problem with release dates (TSP-rd), for the special case modeled as a path in graphs, where the vertex representing the depot is one of the two ends. In TSP-rd, release dates indicate that the initial availability of goods for delivery is partial, with items becoming available over time. This paper presents improvements to a dynamic programming algorithm proposed by Reyes et al. 2018, which has quadratic complexity, reducing the time complexity to $O(n)$.*

Resumo. *Este trabalho explora o Problema do Caixeiro Viajante com datas de liberação (TSP-rd), para o caso especial modelado como um caminho em grafos, onde o vértice representando o depósito é uma das duas extremidades. No TSP-rd as datas de liberação indicam que a disponibilidade inicial de mercadorias para entrega é parcial, com os itens se tornando disponíveis ao longo do tempo. Neste trabalho são apresentadas melhorias em um algoritmo de programação dinâmica proposto por Reyes et al. 2018, de complexidade quadrática, reduzindo-se a complexidade de tempo para $O(n)$.*

1. O Problema do Caixeiro Viajante com datas de liberação

O Problema do Caixeiro Viajante com datas de liberação (TSP-rd) pode ser definido da seguinte forma: Dado um grafo simples e conectado $G = (V, E)$, onde o conjunto de vértices é a união de dois conjuntos, $V = \{0\} \cup N$. O vértice 0 denota o vértice inicial (depósito), enquanto o conjunto de vértices $N = \{1, \ldots, n\}$ representa os $n > 0$ vértices a serem visitados, denominados como clientes. Cada aresta $(i, j) \in E$ está associada a um tempo (distância) de viagem, denotado por d_{ij}. Além disso, uma data de liberação $r_i \geq 0$ está associada a cada vértice $i \in N$, indicando o momento mais cedo em que o item a ser entregue no vértice i pode partir do depósito.

Neste caso, uma *rota* é um passeio (*walk*) em G, sendo uma sequência alternada de vértices e arestas, podendo haver repetições, que devem ser percorridos pelo Caixeiro Viajante, alguns para realização de entrega e outros apenas para viabilizar isto. O início e o final de uma rota é sempre a extremidade que representa o depósito. Formalmente, $\mathcal{R} = \{(i_0, b_0), (i_1, b_1), \ldots, (i_s, b_s), (i_{s+1}, b_{s+1})\}$ com $i_0 = i_{s+1} = 0$ e $b_0 = b_{s+1} = 0$. O conjunto $S = \{i_1, \ldots, i_s\} \subseteq N$ e $(i_k, i_{k+1}) \in E$. Os vértices dentro de S são divididos em dois conjuntos, os vértices em que são realizadas entregas na rota são chamados de *vértices de entrega*, $S^d = \{i_k \in S \mid b_k = 1\}$. Os outros vértices são chamados de *vértices de travessia*, $S^t = S \setminus S^d$. A distância total percorrida em uma rota é determinada por $d_{\mathcal{R}} = \sum_{k=0}^{s} d_{i_k, i_{k+1}}$.

Uma *solução* para o TSP-rd consiste em um conjunto de rotas $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_x$ contendo o conjunto de vértices $S_1, S_2, \ldots, S_x$, essas rotas devem ser feitas consecutivamente pelo Caixeiro Viajante (um único veículo) em ordem de tempo de despacho. Ou seja, dado o tempo de despacho de uma rota $T_{\mathcal{R}} \geq \max_{v \in S^d}\{r_v\}$, então $T_{\mathcal{R}_1} < T_{\mathcal{R}_2} < \cdots < T_{\mathcal{R}_x}$. O caixeiro inicia a rota i, a partir do depósito, se ele tiver concluído a rota $i-1$, ou seja, $T_{\mathcal{R}_{i-1}} + d_{\mathcal{R}_{i-1}} \leq T_{\mathcal{R}_i}$. Uma solução para o TSP-rd é viável se todos os vértices do conjunto $S_i^d \subseteq S_i$ formarem uma partição de N.

A função objetivo que queremos minimizar é dada pelo tempo total necessário para se completar todas as rotas $T_{\mathcal{R}_x} + d_{\mathcal{R}_x}$, essa versão é conhecida como (**TSP-rd(tempo)**) [Archetti et al. 2015, Reyes et al. 2018].

O problema TSP-rd é NP-difícil no caso geral. Algumas soluções exatas e heurísticas foram propostas para abordar esse desafio [Archetti et al. 2018, Montero et al. 2023, Soares et al. 2023]. Se $r_i = 0$ para cada $i \in N$, então o TSP e o TSP-rd são considerados equivalentes, tornando o TSP um caso especial do TSP-rd. No entanto, pesquisas anteriores [Archetti et al. 2015, Reyes et al. 2018] demonstraram que certas estruturas de entrada, como semirretas, linhas e estrelas, permitem soluções polinomiais para o problema. Neste trabalho são apresentadas melhorias em um algoritmo de programação dinâmica proposto po Reyes et al. [2018], de complexidade quadrática, reduzindo-se a complexidade de tempo para $O(n)$.

2. TSP-rd em Caminhos com depósito na extremidade

Nesta seção, abordaremos o TSP-rd em caminhos, enfocando em um cenário especial em que o depósito está localizado em uma das extremidades do caminho. O objetivo é o de examinar uma solução proposta na literatura e sugerir melhorias para essa abordagem.

Para simplificar, denotamos a distância do vértice i até o vértice 0 como τ_i. Essa distância pode ser encontrada usando um algoritmo de busca simples ao longo do caminho. Para um caminho $P' \subseteq P$ com o vértice u em uma extremidade e o vértice 0 na outra, $\tau_u = \sum_{e \in E(P')} d_e$.

No trabalho de Reyes et al. 2018 é apresentada uma solução de complexidade de tempo $O(n^2)$ via programação dinâmica. Nele, é considerado que os clientes são ordenados de forma não-decrescente pela data de liberação ($r_i \leq r_{i+1}$), e de modo estritamente decrescente em relação ao tempo de viagem a partir depósito ($\tau_i > \tau_{i+1}$). Uma instância, então, se parece com o que é ilustrado na Figura 1.

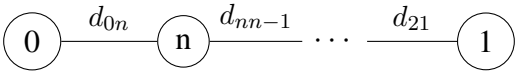

Figura 1. Instancia TSP-rd, caminho com depósito na extremidade.

O trabalho de Reyes et al. 2018 mostra ainda que existe uma solução ótima, contendo apenas rotas não-intercaladas. Uma solução $\mathcal{X}$ contendo rotas não-intercaladas pode ser caracterizada pelo conjunto de últimos clientes em cada rota, ou seja, $\mathcal{X} = \{i_1, i_2, \ldots, i_k, n\}$ com $1 \leq i_1 \leq i_2 \leq \cdots \leq i_k \leq n$, indicando que os clientes $S_1^d = \{1, \ldots, i_1\}$ são atendidos na primeira rota, os pedidos $S_2^d = \{i_{1+1}, \ldots, i_2\}$ são entregues na segunda rota, e assim por diante. Na recorrência proposta, $c(i)$ representa o tempo

mínimo de conclusão para atender os clientes $\{1, \ldots, i\}$ contendo somente rotas não-intercaladas:

$$c(i) = \begin{cases} 0, & \text{se } i = 0 \\ \min_{0 \leq j < i}\{\max\{c(j), r_i\} + 2\tau_{j+1}\}, & \text{caso contrário.} \end{cases} \quad (1)$$

Considere o processo recursivo para se calcular $c(i)$. A entrega ao cliente i pode ser incluída em dois tipos de rotas. O primeiro tipo é junto com outros clientes de entrega (vértices) $j+1, \ldots, i-1$, e é adicionado à solução parcial atendendo aos clientes $\{1, \ldots, j\}$ quando $j \leq i-2$. O segundo tipo de rota envolve a criação de uma nova rota contendo apenas i quando $j = i-1$. Em ambos os casos, o tempo mínimo de conclusão para a nova rota incluindo i é o tempo de despacho mais cedo possível para esta rota, $\max\{c(j), r_i\}$, adicionado ao tempo de viagem da rota que seria $2\tau_{j+1}$.

Sendo assim, não é difícil observar que essa relação de recorrência pode ser calculada em $O(n^2)$, uma vez que para cada $i \in N$ precisamos calcular os termos $\max\{c(j), r_i\} + 2\tau_{j+1}$ da minimização onde $0 \leq j < i$. Mostraremos como calcular essa equação em $O(n)$. O Lema 1 mostra que a função $c(i)$ é não-decrescente (a prova foi omitida por se tratar de um resumo estendido).

Lema 1. *A função $c(i)$ é não-decrescente.*

2.1. Proposta de Resolução em $O(n)$

Como $c(i)$ é uma função não-decrescente (Lema 1), pode-se dividir a Equação 1 em duas partes. O primeiro termo da soma em $\max\{c(j), r_i\} + 2\tau_{j+1}$ será r_i enquanto $c(j) \leq r_i$ for verdade. A partir do momento em que isso não for mais verdade, $c(j)$ será o primeiro termo da soma. Vamos definir k como o último valor de j tal que $c(j) \leq r_i$. Formalmente, $k = \max\{j \mid c(j) \leq r_i\}$. Assim, a Equação 1 pode ser escrita da seguinte maneira:

$$c(i) = \min_{0 \leq j < i} \begin{cases} r_i + 2\tau_{j+1}, & \text{se } j \leq k \\ c(j) + 2\tau_{j+1}, & \text{caso contrário} \end{cases} \quad (2)$$

Teremos dois conjuntos de rotas nas quais podemos adicionar o cliente i. Se $j \leq k$, então a rota montada anteriormente já foi concluída, e o tempo de despacho mais cedo possível é r_i. Caso contrário, a rota anterior será concluída após a liberação de i, e o tempo de despacho mais cedo possível é $c(j)$.

A Equação 2 pode ser reescrita da seguinte forma:

$$c(i) = \min\{\min_{0 \leq j \leq k}\{r_i + 2\tau_{j+1}\}, \min_{k < j < i}\{c(j) + 2\tau_{j+1}\}\} \quad (3)$$

O objetivo é demonstrar que $c(n)$ na Equação 3 pode ser calculado em $O(n)$. Para conseguir isso, devemos estabelecer que k, $\min_{0 \leq j \leq k}\{r_i + 2\tau_{j+1}\}$ e $\min_{k < j < i}\{c(j) + 2\tau_{j+1}\}$ podem ser determinados em tempo constante ($O(1)$).

Lema 2. *$k = max\{j \mid c(j) \leq r_i\}$ é calculado em $O(1)$ para cada $i \in [1 \ldots n]$.*

Demonstração. Vamos ilustrar o processo para computar k para cada valor de $i = 1, \cdots, n$. Para $i = 1$, iteramos j incrementalmente de 0 até que $c(j) \leq r_1$, armazenando este valor como k. Posteriormente, para cada $i \in [2 \ldots n]$, iteramos j começando

do último valor de k armazenado, continuando até que $c(j) \leq r_i$. Isso produz um novo valor para k a ser armazenado. Ao concluir estas operações para todo $i \in N$, a variável j terá variado de 0 a $n-1$ no pior caso. Então, todo o processo requer tempo $O(n)$ para ser executado completamente, onde cada escolha de k feita para $i = 1, \cdots, n$ sendo realizada em tempo constante, $O(1)$. □

Lema 3. $\min_{0 \leq j \leq k}\{r_i + 2\tau_{j+1}\}$ *é calculado em* $O(1)$ *para cada* $i \in [1 \dots n]$.

Demonstração. A propriedade $i < j \Rightarrow \tau_i \geq \tau_j$ garante que o *array* de distância d esteja ordenado de maneira não crescente. Portanto, $\min_{0 \leq j \leq k}\{r_i + 2\tau_{j+1}\} = r_i + 2\tau_{k+1}$. Isso é válido porque r_i é constante para cada $i \in [1 \dots n]$ e $\tau_k \leq \tau_j$ para cada $j \in [1 \dots k-1]$. Portanto, $\min_{0 \leq j \leq k}\{r_i + 2\tau_{j+1}\}$ pode ser substituído por $r_i + 2\tau_{k+1}$ e calculado em tempo $O(1)$ para algum $i \in [1 \dots n]$ e em tempo $O(n)$ para calcular para todos $i \in [1 \dots n]$. □

Como demonstrado no Lema 3, $\min_{0 \leq j \leq k}\{r_i + 2\tau_{j+1}\}$ simplifica para $r_i + 2\tau_{k+1}$. Isso ocorre porque todos os clientes j onde $0 \leq j \leq k$ têm tempos de conclusão menores que r_i, permitindo-nos inserir o cliente i em uma rota com clientes $j+1, \dots, i-1$. Das rotas disponíveis, é ótimo, em termos de tempo de conclusão, selecionar a rota que é mais curta, iniciando no depósito. Consequentemente, escolhemos a rota com clientes $k+1, \dots, i-1$ para incluir o cliente i na mesma rota. Podemos reescrever a Recorrência 3 como:

$$c(i) = \min\{r_i + 2\tau_{k+1}, \min_{k<j<i}\{c(j) + 2\tau_{j+1}\}\} \tag{4}$$

Lema 4. $\min_{k<j<i}\{c(j) + 2\tau_{j+1}\}$ *é calculado em* $O(1)$ *para cada* $i \in [1 \dots n]$.

Demonstração. Podemos calcular eficientemente $\min_{k<j<i}\{c(j) + 2\tau_{j+1}\}$ em $O(n)$ para todos $i \in [1 \dots n]$, o que implica em tempo constante para alguns $i \in [1 \dots n]$.

Primeiramente, definimos $v_j = c(j) + 2\tau_{j+1}$, e o objetivo é encontrar o menor valor v_j tal que $k < j < i$ com complexidade constante para cada $i \in N$ e $O(n)$ para todas as operações. Para isso, vamos utilizar o conceito de janela deslizante. Vamos manter os itens na janela $[k+1, i-1]$ em uma fila com dois fins. A ideia é manter na fila apenas os itens necessários para determinar o valor mínimo de v_j no intervalo (k, i).

Os itens são mantidos ordenados na fila. Durante a inserção de todos os v_j para $k < j < i$, o menor valor de v_j dentro deste intervalo será sempre o primeiro da fila. A cada iteração de i, inserimos o elemento v_{i-1} na fila e removemos todos os v_j para $j \in [k', k-1]$, onde k' representa o valor de k da iteração anterior. Assim, são realizadas n operações de inserção e no pior caso são removidos $n-1$ elementos, concluindo que calcular $\min_{k<j<i}\{c(j) + 2\tau_{j+1}\}$ para cada $i \in N$ é realizado em tempo constante $O(1)$ e $O(n)$ para calcular todas as janelas juntas. □

Teorema 1. *A complexidade para se calcular* $c(n)$ *através da Equação 4 é de* $O(n)$.

Demonstração. Os Lemas 2, 3 e 4 demonstram que os dois termos da relação de recorrência da Equação 4 pode ser computado em tempo constante ou $O(n)$. Como a operação de determinar o menor de dois valores possui custo constante, $c(n)$ pode ser calculado em $O(n)$. □

Referências

Archetti, C., Feillet, D., Mor, A., and Speranza, M. G. (2018). An iterated local search for the traveling salesman problem with release dates and completion time minimization. *Computers & Operations Research*, 98:24–37.

Archetti, C., Feillet, D., and Speranza, M. G. (2015). Complexity of routing problems with release dates. *European journal of operational research*, 247(3):797–803.

Montero, A., Méndez-Díaz, I., and Miranda-Bront, J. J. (2023). Solving the traveling salesman problem with release dates via branch and cut. *EURO Journal on Transportation and Logistics*, 12:100121.

Reyes, D., Erera, A. L., and Savelsbergh, M. W. (2018). Complexity of routing problems with release dates and deadlines. *European journal of operational research*, 266(1):29–34.

Soares, G., Bulhoes, T., and Bruck, B. (2023). Um algoritmo genético híbrido para o problema do caixeiro viajante com tempos de liberação. In *Anais do VIII Encontro de Teoria da Computação*, pages 160–164. SBC.