Pedro Maia de Sampaio Ferraz

Uso de Algoritmos Online e Programação Inteira Mista em Problemas de Pickup and Delivery

RELATÓRIO DE PROJETO FINAL

DEPARTAMENTO DE ENGENHARIA ELÉTRICA E DEPARTAMENTO DE INFORMÁTICA

Programa de graduação em Engenharia de Computação

Rio de Janeiro
Novembro de 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Pedro Maia de Sampaio Ferraz

Uso de Algoritmos Online e Programação Inteira Mista em Problemas de Pickup and Delivery

Relatório de Projeto Final

Relatório de Projeto Final, apresentado ao programa de Engenharia de Computação da PUC-Rio como requisito parcial para a obtenção do titulo de Engenheiro de Computação.

Orientador: Prof. Marco Serpa Molinaro

Rio de Janeiro
Novembro de 2021

Resumo

Maia de Sampaio Ferraz, Pedro; Serpa Molinaro, Marco. **Uso de Algoritmos Online e Programação Inteira Mista em Problemas de Pickup and Delivery**. Rio de Janeiro, 2021. 41p. Projeto de Graduação – Departamento de Engenharia Elétrica e Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Esse projeto teve como principal objetivo a implementação e experimentação de algoritmos online para a solução de problemas de *pickup* e *delivery*. A partir da análise dos resultados de dois algoritmos propostos recentemente na literatura, foram elaborados três novos algoritmos online. Os novos algoritmos foram implementados e analisados em bases de dados existentes e em instâncias sintéticas, apresentando resultados superiores aos anteriores. Além disso, é discutida a solução da versão offline do problema através de busca via força-bruta e programação inteira-mista.

Palavras-chave

Algoritmos Online, Programação Inteira Mista, Pickup, Delivery

Sumário

1
Introdução

Nos últimos anos, a demanda por serviços de *delivery* vem aumentado consideravelmente. Esse processo tornou-se ainda mais acelerado devido à pandemia de COVID-19. Por um lado, a necessidade de isolamento fez com que mais pessoas aderissem a esses serviços. Por outro, comércios e restaurantes viram tais serviços como forma de continuar em operação. Além disso, profissionais autônomos, que tiveram seu faturamento prejudicado nesse período, viram o *delivery* como uma boa oportunidade para ampliar suas fontes de renda. (Valor Investe 2020).

Nesse contexto, as plataformas de *crowdsourcing delivery* provêm a solução para uma entrega mais rápida e eficiente, tanto para os vendedores, quanto para os consumidores. Nessas plataformas os pedidos são realizados em tempo real por consumidores através de *smartphones* e são enviados para os entregadores disponíveis com rotas sugeridas. Alguns exemplos de plataformas que seguem esse modelo são Rappi e Uber Eats no Brasil, e GrubHub, Instacart e DoorDash nos EUA.

Conforme a demanda por serviços de *delivery* aumenta, o desafio enfrentado por essas plataformas se torna cada vez maior. Desse modo, nunca foi tão relevante buscar maneiras de otimizar a eficiência das plataformas para que seja possível atender todos os consumidores no menor tempo possível.

Com esse enfoque, Yu et al. (Yu et al. 2020) propuseram e estudaram o problema online de *pickup* e *delivery* do ponto de vista das plataformas de *crowdsourcing*. O problema considerado pelos autores é chamado de *Online pickup and delivery problem with constrained capacity to minimize latency* (OnlinePickDel). Esse problema é o foco deste trabalho. Na Seção 2 definimos mais formalmente esse problema.

1.1
Motivação

O artigo de Yu et al. (Yu et al. 2020) identificou quatro principais características de algumas plataformas de *pickup* e *delivery*: (1) todos os pedidos são realizados online em tempo real, o que implica que as decisões de agendamento e roteamento devem ser feitas apenas com informações sobre os

pedidos passados; (2) cada pedido tem um local único de coleta e de entrega, o que implica que entregadores devem buscar o pedido no local de coleta e levá-lo até o local de entrega, levando em consideração que os locais de coleta estão majoritariamente em centros comerciais, o que faz com que estes não sejam geograficamente dispersos; (3) os consumidores desejam receber suas entregas o mais rápido possível, o que faz com que o tempo de entrega seja um critério importante para vendedores e consumidores escolherem entre diferentes plataformas; e (4) os entregadores não são capazes de levar muitos pedidos simultaneamente.

Considerando as características acima, os autores estudam o problema em que um entregador com capacidade restrita busca os pedidos liberados ao longo do tempo em um único ponto de coleta e leva-os até seus destinos. O objetivo é minimizar a soma dos tempos de entrega dos pedidos. Esse objetivo também é referido como latência.

1.2 Situação atual

O problema OnlinePickDel pertence à vasta família de problemas de Roteamento de Veículos. Um exemplo clássico de problema desta família é o Problema do Caixeiro Viajante, um problema que tenta determinar a menor rota para percorrer todos os vértices de um grafo.

Um grande número de pesquisadores estudou variações desses problemas por décadas, e uma excelente revisão da literatura foi feita por Mor et al. (Mor et al. 2020). Também pode-se encontrar mais estudos sobre problemas do tipo *dial-a-ride* em Ho et al. (Ho et al. 2018).

Além disso, OnlinePickDel é similar a alguns outros problemas já estudados na literatura. Em particular, os problemas *Online Travelling Repairman Problem* (OLTRP) e *Latency Online Dial-A-Ride Problem* (L-OLDARP) têm o objetivo comum de minimizar a latência e são casos particulares do Problema de Roteamento de Veículos Online. Apesar de compartilharem o mesmo objetivo, os problemas OLTRP e L-OLDARP têm características diferentes do OnlinePickDel. Em OLTRP, os pedidos são considerados servidos quando o servidor visita o local do pedido após seu tempo de liberação, de forma que não é necessário fazer a coleta destes pedidos anteriormente. Já em L-OLDARP, os pedidos podem ter que ser coletados em diferentes localizações. Assim, o servidor de ambos os problemas não é adequado para modelar um entregador de *crowdsourcing delivery* com as características descritas.

Por fim, o problema L-OLDARP estuda apenas os casos em que a capacidade do servidor é 1 ou infinita, não contemplando a possibilidade de

uma capacidade genérica c, diferente do problema OnlinePickDel.

1.3 Objetivos do trabalho

Os objetivos deste trabalho consistem no desenvolvimento e implementação de soluções para o problema OnlinePickDel. Inicialmente, foram implementados os algoritmos descritos no artigo (Yu et al. 2020) e, em seguida, foram propostas variações desses algoritmos buscando a melhoria da qualidade das soluções. As variações foram os novos algoritmos "Naive Ignore", "Naive Return" e "Compute Return".

Para que fosse possível implementar os algoritmos online descritos no artigo, era necessário primeiramente resolver a versão offline do problema, o que não é trivial. Nesta versão, o servidor tem conhecimento completo sobre a instância desde o início, incluindo os locais de entrega e o tempo de liberação de cada pedido. Para resolver este problema, foi implementado um algoritmo de busca por força bruta com algumas técnicas de pré-processamento. Para que fosse possível encontrar soluções em instâncias maiores, também projetamos um modelo de *Programação Inteira Mista* (Conforti et al. 2014).

Para realizar o projeto, primeiramente foi feito um estudo da teoria de Algoritmos Online através das aulas gravadas do professor Marco Molinaro na disciplina Algoritmos e Incerteza. Em seguida, foi necessário estudar o artigo (Yu et al. 2020) e em particular, os algoritmos *Wait and Return* e *Wait and Ignore*. Além disso, foi realizado um estudo de Programação Inteira por meio das aulas do professor Alexandre Street, a ser utilizado para resolver o problema offline.

Todos os algoritmos foram desenvolvidos utilizando a linguagem Julia, que é uma linguagem de alto nível e alto desempenho que se destaca no contexto de computação científica. Para resolver o problema offline formulado por programação inteira, utilizamos o JuMP (Dunning et al. 2017), um pacote de programação matemática que permite escrever modelos de forma simples e suporta diversos *solvers* comerciais para problemas de diversas classes. Em particular, optamos por utilizar o Gurobi (Gurobi 2021) como solver devido ao seu alto desempenho e agilidade em encontrar boas soluções.

Além disso, para garantir a qualidade, corretude e manutenibilidade do código, elaboramos testes unitários para todos os algoritmos desenvolvidos.

Por fim, os diferentes algoritmos online implementados foram testados em bases de dados de problemas de Roteamento de Veículos Online e em instâncias geradas de forma aleatória. Os testes indicam a eficácia ou não de diferentes estratégias de espera e retorno à origem em diferentes tipos de instância.

2
Definição do problema

Neste capítulo iremos apresentar a motivação e a definição formal para o problema OnlinePickDel (Yu et al. 2020). Em seguida, apresentaremos um exemplo de instância do problema.

2.1
Definição formal

Considere um grafo simples direcionado com pesos $G = (V, E)$ onde cada aresta $(u, v) \in E$ caracteriza o tempo de viagem do vértice u até o vértice v. Seja $o \in V$ o vértice de origem onde os pedidos serão coletados. Seja $l(u, v)$ a menor distância do vértice u para o vértice v no grafo.

Considere também uma sequência de pedidos $\sigma = q_1, \ldots, q_m$, onde cada pedido $q_i = (r_i, d_i)$ é definido pelo vértice de destino $d_i \in V$ e pelo tempo de liberação r_i (i.e., o instante de tempo em que o pedido é liberado para ser levado até o destino). Seja l_i a menor distância (em tempo de viagem) da origem o até o vértice de destino d_i.

Considere um servidor (i.e., um entregador) com capacidade c que pode carregar consigo até c pedidos simultaneamente. O servidor tem como posição inicial o vértice de origem o. Além disso, cada pedido q_i deve ser buscado no vértice de origem o após o tempo r_i e não pode ser deixado em nenhum vértice que não o seu respectivo vértice de destino d_i.

O algoritmo online toma conhecimento do pedido q_i **apenas após o seu tempo de liberação** r_i. Em particular, ele não tem conhecimento sobre pedidos que são liberados no futuro e nem sobre o número total de pedidos.

Dada uma sequência de pedidos σ, uma rota viável para σ é uma sequência de movimentos do servidor que satisfaz todas as restrições impostas acima e na qual todos os pedidos em σ são servidos.

Seja C_i^S o tempo de conclusão do pedido q_i em uma rota viável S. O problema OnlinePickDel consiste em encontrar de forma online uma rota viável S que minimize $\sum_{i=1}^{m} C_i^S$ para uma dada sequência de pedidos online $\sigma = q_1, \ldots, q_m$. A função objetivo $\sum_{i=1}^{m} C_i^S$ também é chamada de *latência*.

A versão offline do problema é definida como aquela onde todos os pedidos σ são conhecidos desde o princípio.

Por fim, dada uma rota S, definiremos que uma sub-rota desta rota é um caminho em que o servidor sai da origem, entrega pedidos e retorna à origem. Note que uma rota pode ter uma ou mais sub-rotas.

2.2 Exemplos de instâncias e soluções

Para o melhor entendimento do problema, ilustraremos uma instância a seguir.

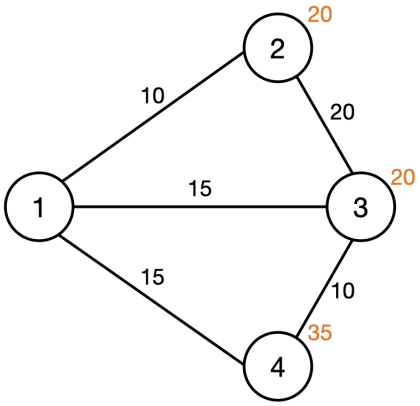

Figura 2.1: Instância exemplo

Na Figura 2.1, o vértice 1 representa a origem (onde os pedidos são coletados) enquanto os demais vértices representam pedidos. Os pesos das arestas representam o tempo de viagem do servidor. Os números em laranja no canto superior direito de cada pedido representam o tempo em que o pedido é liberado. Nesse exemplo assumiremos que a capacidade do servidor é igual a 3, de forma que o servidor poderia levar todos os pedidos simultaneamente caso desejasse.

Utilizando a notação apresentada na Seção 2.1, temos que $\sigma = q_1, q_2, q_3$, onde $q_1 = (20, 2)$, $q_2 = (20, 3)$ e $q_3 = (35, 4)$, e $c = 3$.

Na Figura 2.2 podemos visualizar a informação disponível para o algoritmo offline em cada instante de tempo. Como os primeiros pedidos são liberados apenas em $t = 20$, antes disso o algoritmo online não tem conhecimento de nenhum pedido. Em $t = 20$, o algoritmo online passa a ter informação sobre os pedidos dos vértices 2 e 3 e, por fim, em $t = 30$ passa a conhecer o pedido do vértice 4.

Na Figura 2.3 estão ilustradas duas soluções diferentes para o problema. Em ambas as soluções, as cores verde e azul representam a primeira e segunda sub-rotas, respectivamente. Na primeira solução S_1, o servidor leva os pedidos dos vértices 2 e 3 assim que se tornam disponíveis em $t = 20$, volta à origem após entregá-los em $t = 65$ e entrega o pedido restante do vértice 4. Em

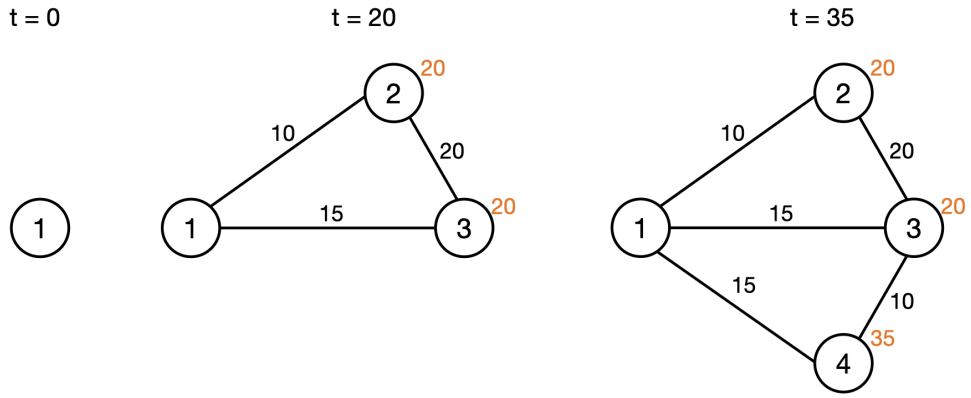

Figura 2.2: Instância exemplo ao longo do tempo

contraste, na segunda solução S_2 o servidor leva apenas o pedido do vértice 2 em $t = 20$, volta à origem em $t = 40$ e por fim entrega os pedidos restantes dos vértices 3 e 4.

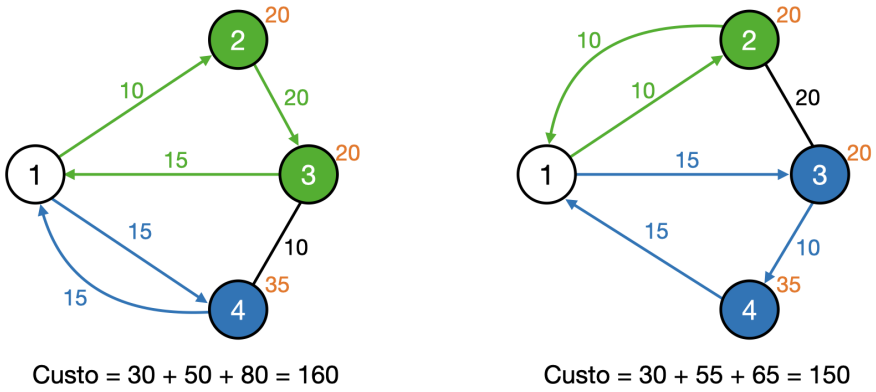

Figura 2.3: Soluções para o problema

Note que os tempos de conclusão dos pedidos na primeira rota são $C_1^{S_1} = 30$, $C_2^{S_1} = 50$, $C_3^{S_1} = 80$, enquanto na segunda rota são $C_1^{S_2} = 30$, $C_2^{S_2} = 55$, $C_3^{S_2} = 65$. Portanto, os custos das rotas são dados por $C^{S_1} = \sum_{i=1}^{3} C_i^{S_1} = 160$ e $C^{S_2} = \sum_{i=1}^{3} C_i^{S_2} = 150$. Dessa forma, a segunda rota tem menor custo e, portanto, é mais vantajosa. No entanto, note que o algoritmo online não sabia à respeito da existência do pedido de vértice 4 em $t = 20$.

3
Algoritmos online para problema OnlinePickDel

Neste capítulo iremos apresentar diferentes algoritmos online para resolver o problema OnlinePickDel e analisaremos exemplos e diferenças entre os diferentes algoritmos.

3.1
Algoritmo online "Wait and Ignore"

Uma primeira ideia para resolver o problema OnlinePickDel é fazer com que o servidor saia da origem com os pedidos disponíveis assim que forem liberados. No entanto, pode ser mais vantajoso esperar para que mais pedidos sejam liberados antes de sair para a entrega, seja para preencher a capacidade ou para escolher uma ordenação melhor de entrega dos pedidos.

Assim, a ideia do algoritmo "Wait and Ignore" proposto por Yu et al. (Yu et al. 2020) é estabelecer um critério para esperar até que alguns pedidos tenham sido liberados e apenas então levá-los para entrega, ignorando todos os novos pedidos que chegarem no meio do caminho.

3.1.1
Descrição do algoritmo

A fim de estabelecer o critério para quando o servidor deve sair para entrega com os pedidos, definiremos o conceito de *tempo ativo* t_i de um pedido q_i. Um pedido q_i poderá ser levado pelo entregador se e somente se $t \geq t_i$, onde t é o tempo decorrido.

Definimos $t_i = \max\{r_i, l_i\}$ como o *tempo ativo do pedido* q_i. Esse critério estabelece que o servidor pode coletar um pedido se, e somente se, o tempo decorrido for maior do que o tempo que leva para entregar o pedido a partir da origem. Assim, caso algum pedido esteja muito distante da origem, este critério tenta resguardar o servidor de entregá-lo antes de considerar outros pedidos possivelmente mais próximos.

Seja $Q_t^a = \{(r_i, l_i) \mid t \geq \max(r_i, l_i)\}$ o conjunto de pedidos ativos no tempo t e Q_t^S o conjunto de pedidos que foram coletados pelo servidor, mas ainda não foram entregues. Seja $|Q|$ o número de elementos de Q.

Com as definições acima, podemos especificar o algoritmo "Wait and Ignore" conforme o pseudo-código abaixo:

Algoritmo 1: Wait and Ignore

Entrada: Grafo $G = (V, E)$ com vértice de origem $o \in V$, capacidade c do servidor, sequência de pedidos $\sigma = q_1, \ldots, q_n$ liberados ao longo do tempo

Saída: Rota viável S, valor da função objetivo $\sum_{i=1^m} C_i^S$

Inicialização: Conjuntos Q_t^a e Q_t^S iniciam vazios no tempo $t = 0$ e são atualizados caso algum pedido se torne ativo ou o estado de algum pedido mude

Passo 1: Caso $Q_t^a = \emptyset$, o servidor online permanece no vértice de origem o. Caso contrário, vá para o passo 2.

Passo 2: Calcule a rota ótima $S(Q_t^a)$ para o conjunto de pedidos ativos através de um algoritmo offline.

Seja S_1 a primeira sub-rota de $S(Q_t^a)$. Seja $q = \min\{|Q_t^a|, c\}$. Escolha os $\min\{|S_1|, q\}$ primeiros pedidos da sub-rota S_1 e insira-os em Q_t^S.

Atualize o conjunto Q_t^a como $Q_t^a - Q_t^S$. Sirva todos os pedidos de Q_t^S de acordo com a rota ótima para os primeiros q pedidos de $S(Q_t^a)$ ignorando todos os pedidos que forem liberados ou se tornarem ativos.

Calcule o custo $\sum_{q_i \in S(q)} C_i^S$. Por fim, volte para a origem o e atualize o tempo decorrido t. Vá para o passo 1.

3.1.2 Exemplos e observações

A Figura 3.1 ilustra uma instância e a solução encontrada pelo algoritmo "Wait and Ignore". Na solução ilustrada as cores verde e azul representam a primeira e segunda sub-rotas, respectivamente.

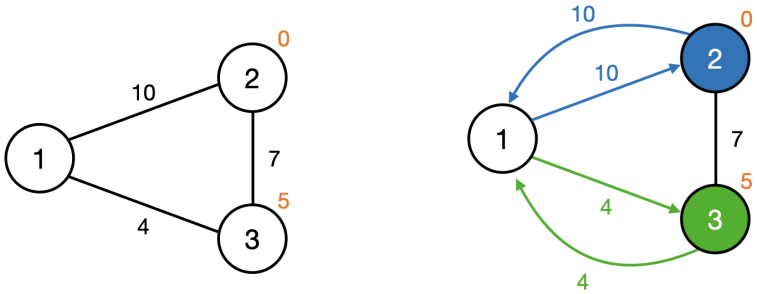

Figura 3.1: Exemplo do algoritmo "Wait and Ignore"

Note que o pedido do vértice 2 está liberado desde o instante $t = 0$. Logo, o servidor já poderia coletar tal pedido e sair da origem em $t = 0$. No entanto, como a distância até o pedido é $l_2 = 10$, o tempo ativo do pedido deste vértice é $t_2 = 10$. Assim, o pedido ainda não é ativo e o servidor espera na origem.

No instante $t = 5$ o pedido de vértice 3 é liberado. Como a distância $l_3 = 4$ é menor do que o tempo em que é liberado, o pedido 3 é ativo em $t = 5$. No entanto, o pedido de vértice 2 permanece inativo. Assim, o servidor coleta apenas o pedido de vértice 3 e leva-o para entrega.

Após entregar o pedido de vértice 3, o servidor volta à origem em $t = 13$. Nesse instante de tempo o pedido de vértice 2 já está ativo. Assim, o servidor coleta e entrega o pedido de vértice 2.

Note que a rota encontrada pelo algoritmo "Wait and Ignore" tem custo $C_1 = 9 + 23 = 32$. Caso o servidor levasse o pedido 2 em $t = 0$, o custo teria sido $C_2 = 10 + 24 = 34$. Assim, nesse caso o critério de espera foi vantajoso se comparado com a solução mais "gulosa".

3.2 Algoritmo online "Wait and Return"

O segundo algoritmo proposto por Yu et al. (Yu et al. 2020) é o "Wait and Return". Assim como no algoritmo "Wait and Ignore", o algoritmo "Wait and Return" busca se resguardar do caso em que o servidor deixa a origem antecipadamente por meio do *active time*. No entanto, ao invés de ignorar os novos pedidos que se tornam ativos, é considerada a possibilidade de retornar à origem para buscar novos pedidos caso seja vantajoso.

3.2.1 Descrição do algoritmo

Considere que um novo pedido se torna ativo no tempo t enquanto o servidor está fora da origem percorrendo a aresta $(u, v) \in E$. Neste momento o algoritmo decide se continua na sua rota atual ou se volta à origem. Seja t_u o instante de tempo em que o servidor percorreu o vértice u. Consideraremos o tempo de volta até a origem y como sendo o tempo para voltar até o vértice u mais o tempo de voltar do vértice u até a origem, isto é,

$$y = (t - t_u) + l(u, o).$$

Para o critério de decisão de voltar à origem, suponha que existam k pedidos ativos que ainda não foram coletados pelo servidor. Seja $l_m = \max\{l_i \mid q_i \in Q_t^S\}$ a distância máxima de qualquer vértice da rota atual até

a origem. Seja $r = |Q_t^S|$ o número de pedidos que estão em rota e ainda não foram entregues. Então, o algoritmo decide voltar à origem se, e somente se,

$$\frac{y}{l_m} \leq \frac{k}{k+r}$$

Note que lado esquerdo da inequação do critério leva em conta uma noção de distância relativa para retornar à origem, enquanto o lado direito considera o número de pedidos em rota e esperando na origem para construir um senso de "urgência" para o retorno. Assim, quanto menor a distância relativa à origem, maior o número de pedidos novos e menor o número de pedidos restantes na rota, maior será a chance do servidor voltar à origem.

O algoritmo "Wait and Return" é descrito pelo seguinte pseudo-código:

Algoritmo 2: Wait and Return

Entrada: Grafo $G = (V, E)$ com vértice de origem $o \in V$, capacidade c do servidor, sequência de pedidos $\sigma = q_1, \ldots, q_n$ liberados ao longo do tempo

Saída: Rota viável S, valor da função objetivo $\sum_{i=1^m} C_i^S$

Inicialização: Conjuntos Q_t^a e Q_t^S iniciam vazios no tempo $t = 0$ e são atualizados caso algum pedido se torne ativo ou o estado de algum pedido mude

Passo 1: Caso $Q_t^a = \emptyset$, o servidor online permanece no vértice de origem o. Caso contrário, vá para o passo 2.

Passo 2: Calcule a rota ótima $S(Q_t^a)$ para o conjunto de pedidos ativos através de um algoritmo offline.

Seja S_1 a primeira sub-rota de $S(Q_t^a)$. Seja $q = \min\{|Q_t^a|, c\}$. Escolha os $\min\{|S_1|, q\}$ primeiros pedidos da sub-rota S_1 e insira-os em Q_t^S. Atualize o conjunto Q_t^a como $Q_t^a - Q_t^S$. Vá para o passo 3.

Passo 3: Sirva todos os pedidos de Q_t^S de acordo com a rota ótima dada pelo algoritmo offline $S(Q_t^s)$.
Caso o pedido q_i seja entregue, remova-o de Q_t^S.
Caso $Q_t^S = \emptyset$, o servidor retorna à origem. Vá para o passo 1.
Caso algum pedido se torne ativo, vá para o passo 4.

Passo 4: Considere a possibilidade de voltar para a origem.
Caso $\frac{y}{l_m} \leq \frac{k}{k+r}$, o servidor retorna à origem. Vá para o passo 5.
Caso contrário, prossiga com o passo 3.

Passo 5: Quando o servidor retorna à origem durante a rota:
Caso $|Q_t^a| + |Q_t^S| \geq c$, considere $Q_t = Q_t^a \cup Q_t^S$. Calcule a rota ótima $S(Q_t)$ através do algoritmo offline. Seja $S(Q_t)^c$ o conjunto composto pelos c primeiros pedidos da rota ótima $S(Q_t)$. Atualize Q_t^S como $S(Q_t)^c$, e Q_t^a como $Q_t - Q_t^S$.
Caso contrário, atualize o conjunto Q_t^S como $Q_t^S \cup Q_t^a$ e $Q_t^a = \emptyset$.
Prossiga com o passo 3.

3.2.2 Exemplos e observações

Para o melhor entendimento do algoritmo, descreveremos o passo a passo seguido pelo algoritmo "Wait and Return" em uma instância. Na Figura 3.2, o vértice 1 representa a origem (onde os pedidos são coletados) enquanto os demais vértices representam pedidos. Os números em laranja no canto superior de cada pedido representam o tempo em que o pedido é liberado. Nesse exemplo assumiremos que a capacidade do servidor é igual a 5, podendo levar todos os pedidos simultaneamente se desejado.

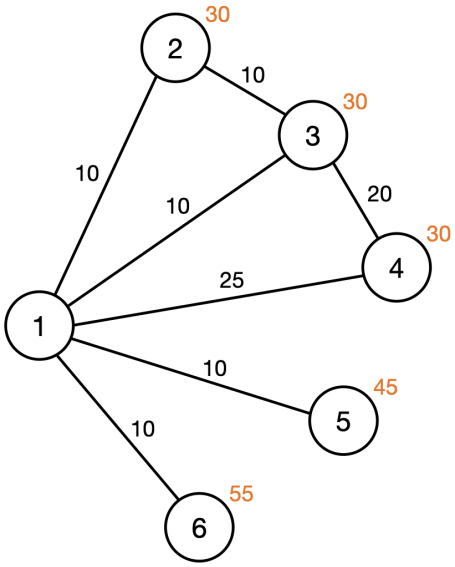

Figura 3.2: Instância exemplo do "Wait and Return"

Note que os primeiros pedidos a serem liberados correspondem aos vértices 2, 3 e 4, todos liberados no instante $t = 30$. Como a distância da origem até cada um desses vértices é menor do que 30, o tempo ativo dos pedidos em já foi atendido em $t = 30$. Note também que o algoritmo online ainda não tem conhecimento dos pedidos dos vértices 5 e 6 nesse instante. A rota ótima calculada pelo algoritmo offline para entregar os pedidos é a ordem crescente, i.e., 2, 3 e 4, respectivamente. Nessa rota, o servidor sairia da origem em $t = 30$, entregaria o pedido 2 em $t = 40$, o pedido 3 em $t = 50$ e o pedido 4 em $t = 70$, voltando à origem no instante $t = 95$. Assim, o entrega destes pedidos contribuiria com $40 + 50 + 70 = 160$ para o custo (latência).

No entanto, note que nos instantes de tempo $t = 45$ e $t = 55$ os pedido de vértice 5 e 6, respectivamente, são liberados e o algoritmo online deve considerar a possibilidade de retornar à origem. A Figura 3.3 ilustra tal instante de tempo. Nesta figura, os números em verde no canto de cada vértice correspondem ao instante de tempo em que o servidor deixou aquele vértice,

enquanto os números em verde nas arestas correspondem à distância percorrida pelo servidor naquela aresta.

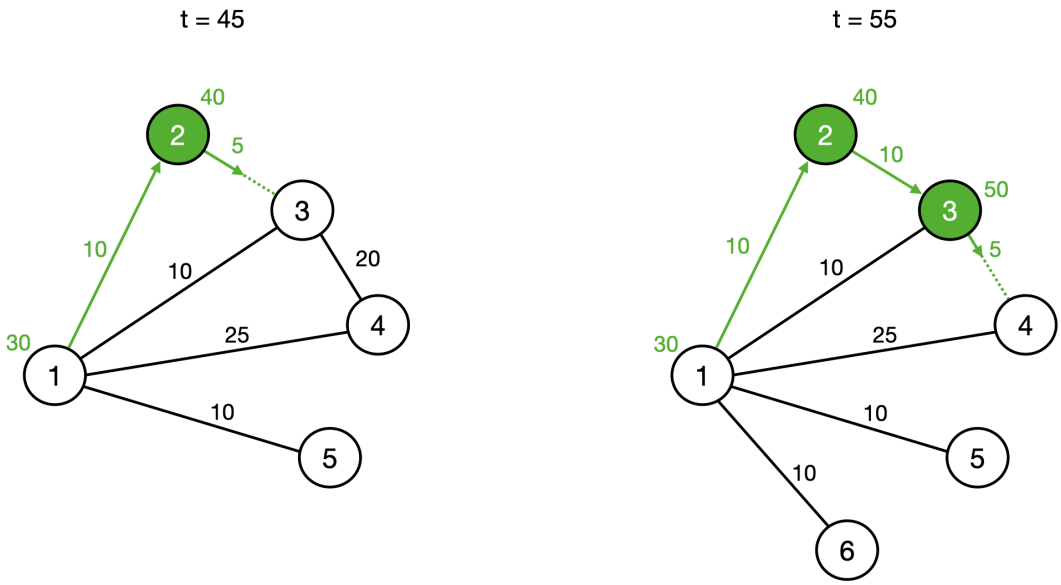

Figura 3.3: Passo a passo do algoritmo "Wait and Return"

No instante de tempo $t = 45$ temos que a distância y para retornar à origem é dada por $y = (t - t_2) + l(2, 1) = (45 - 40) + 10 = 15$, a distância máxima de qualquer vértice até a origem é dada por $l_m = 25$, o número de novos pedidos que se tornaram ativos é $k = 1$ e o número de pedidos em rota que ainda não foram entregues é $r = 2$. Assim, temos que

$$\frac{y}{l_m} = \frac{15}{25} = \frac{3}{5} > \frac{1}{3} = \frac{1}{1+2} = \frac{k}{k+r}$$

e, portanto, o servidor não deve voltar à origem e deve continuar a rota. Entretanto, em $t = 55$ temos que $y = (t - t_3) + l(3, 1) = (55 - 50) + 10 = 15$, $l_m = 25$, $k = 2$ e $r = 1$. Assim,

$$\frac{y}{l_m} = \frac{15}{25} = \frac{3}{5} < \frac{2}{3} = \frac{2}{2+1} = \frac{k}{k+r}$$

de forma que o servidor decide voltar à origem em $t = 55$. Ao fazer isso, o servidor retorna à origem em $t = 70$ e entrega os pedidos dos vértices 4, 5 e 6 de acordo com o algoritmo offline (primeiramente entregando os pedidos dos vértices 5 e 6 e, por último, o pedido do vértice 4).

3.3 Algoritmos online "Naive Ignore" e "Naive Return"

Com o intuito de verificar a eficácia do critério de espera de *tempo ativo* adotado por Yu et al. (Yu et al. 2020) nos algoritmos "Wait and Ignore" e

"Wait and Return", propomos variações desses algoritmos sem tempo de espera. Estes foram chamados de "Naive Ignore" e "Naive Return", respectivamente. Em cada um desses algoritmos, o servidor sai da origem para levar pedidos assim que há pelo menos um pedido liberado, não havendo o conceito de *tempo ativo*. Apesar do *tempo ativo* possibilitar o algoritmo se resguardar de atender pedidos muito distantes precipitadamente, ao remover o tempo de espera o servidor é capaz de entregar pedidos mais cedo, o que compensa em muitos casos.

Dessa forma, o conjunto de pedidos ativos $Q_t^a = \{(r_i, l_i) \mid t \geq \max(r_i, l_i)\}$ deve ser substituído pelo conjunto $Q_t^L = \{(r_i, l_i) \mid t \geq r_i\}$. Essa é a única modificação necessária no pseudo-código dos algoritmos "Wait and Ignore" e "Wait and Return" para descrever os algoritmos "Naive Ignore" e "Naive Return".

3.4 Algoritmo online "Compute Return"

Outro algoritmo online que propomos é o "Compute Return". Neste, não há tempo de espera, e o critério de retorno é calculado de maneira diferente dos algoritmos "Wait and Return" e "Naive Return".

A única diferença deste algoritmo para o "Naive Return" é o critério de retorno. Assim, ainda podemos utilizar como base o pseudo-código do algoritmo "Wait and Return". A diferença se dará no passo 4 e será descrita a seguir.

3.4.1 Critério de retorno

A ideia do critério de retorno do algoritmo "Compute Return" é simular as situações do servidor continuar na rota e de voltar para a origem e verificar qual tem menor custo. Assim, quando um novo pedido for liberado, iremos comparar o custo C^I caso o servidor ignore os novos pedidos e continue na rota e o custo C^R caso este retorne para a origem na chegada de um novo pedido. Caso o custo da rota com o retorno seja menor, o servidor retornará para a origem.

Seja Q_t^S o conjunto de pedidos que estão em rota no tempo t. Seja Q_t^L o conjunto de pedidos que já foram liberados, mas estão na origem no tempo t. Note que $Q_t^S \cup Q_t^L$ é o conjunto de todos os pedidos que são de conhecimento do algoritmo online no tempo t que ainda não foram entregues. Portanto, iremos comparar os custos de servir os pedidos em $Q_t^S \cup Q_t^L$ nos casos em que servidor continua com a rota e volta para a origem.

Seja $C^S(Q) = \sum_{q_i \in Q} C_i^S$ o custo de entregar todos os pedidos no conjunto Q através de uma rota viável S já estabelecida. Seja Offline(Q, t) o custo calculado pelo algoritmo offline para entregar todos os pedidos no conjunto Q considerando que o servidor está coletando-os na origem no tempo t.

Caso o servidor continue com a rota S, deverá primeiramente terminar de entregar todos os pedidos em Q_t^S e, em seguida, retornar à origem para coletar e entregar os pedidos restantes em Q_t^L. O custo de entrega dos pedidos em Q_t^S é facilmente calculado através dos tempos de entrega dos pedidos na rota S e é dado por $C^S(Q_t^S)$. Para calcular o custo Q_t^L, considere que o servidor volta à origem após entregar os pedidos em Q_t^S no instante de tempo t^I. O custo de entrega dos pedidos em Q_t^L será dado pelo algoritmo offline com tempo inicial t^I, i.e., Offline(Q_t^L, t^I). Assim, o custo de continuar com a rota S é dado pelo custo de terminar de entregar os pedidos em Q_t^S acrescentado ao custo de coletar e entregar os pedidos em Q_t^L calculado pelo algoritmo offline, isto é,

$$C^I = C^S(Q_t^S) + \text{Offline}(Q_t^L, t^I)$$

Em contrapartida, caso o servidor retorne à origem, este deverá entregar todos os pedidos em $Q_t^S \cup Q_t^L$ a partir da origem. Considere o instante de tempo t^R em que o servidor retorna à origem caso decida retornar imediatamente no tempo t (sem concluir a rota). O custo de retorno C^R será dado pelo algoritmo offline para entregar os pedidos em $Q_t^S \cup Q_t^L$ com tempo inicial t^R, isto é,

$$C^R = \text{Offline}(Q_t^S \cup Q_t^L, t^R)$$

Assim, é feita uma simulação dos dois cenários e é tomada a decisão correspondente ao mínimo entre C^I e C^R.

3.4.2 Exemplos e observações

No exemplo abaixo, mostraremos o critério de retorno do algoritmo "Compute Return" e compararemos com o critério do algoritmo "Naive Return". Na Figura 3.4 podemos ver a instância a ser analisada nesse exemplo.

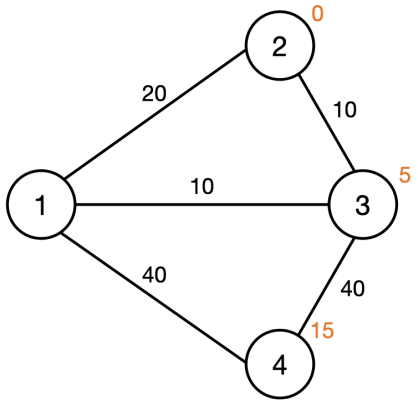

Figura 3.4: Instância para exemplificar o algoritmo “Compute Return”

Note que em $t = 0$ o pedido do vértice 2 já está liberado, de forma que o servidor pode coletá-lo imediatamente. Conforme ilustrado na primeira imagem da Figura 3.5, em $t = 5$ o pedido do vértice 3 é liberado e o algoritmo deve considerar a possibilidade de retorno à origem. Para tomar esta decisão, o servidor deve considerar os custos C^I e C^R.

Vamos primeiramente considerar a possibilidade de continuar em rota. Note que em $t = 5$ o único pedido em rota é o do vértice 2. Assim, caso o servidor continue, o custo de entregar os pedidos em rota será $C^S(Q_5^S) = 20$. Após a entrega, o servidor retornaria à origem no instante de tempo $t = 40$, logo $t^I = 40$. Além disso, o único pedido restante na origem de conhecimento do servidor em $t = 5$ é o pedido do vértice 3. Assim, o custo calculado pelo algoritmo offline para entregar os pedidos restantes na origem coletando-os em $t = 40$ é $\text{Offline}(Q_5^L, 40) = 50$ (dado que o servidor chegaria no vértice 3 em $t = 50$). Assim, o custo associado a continuar na rota é

$$C^I = C^S(Q_5^S) + \text{Offline}(Q_5^L, 40) = 20 + 50 = 70$$

Agora vamos considerar o custo associado ao retorno. Caso o algoritmo decida pelo retorno em $t = 5$, o servidor retornaria à origem em $t = 10$, isto é, $t^R = 10$. Assim, o custo associado ao retorno seria o custo de entregar os pedidos dos vértices 2 e 3 coletando-os em $t = 10$. Note que nesse caso a melhor rota é entregar primeiramente o pedido do vértice 3 (em $t = 20$) e depois o pedido do vértice 2 (em $t = 30$) totalizando um custo de $20 + 30 = 50$. Logo, temos que o custo associado ao retorno é

$$C^R = \text{Offline}(Q_5^S \cup Q_5^L, 10) = 50$$

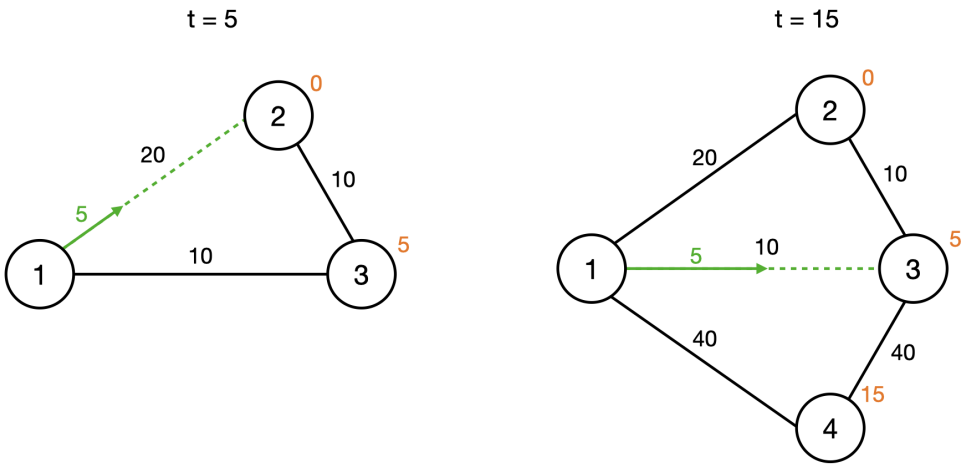

Figura 3.5: Momentos de decisão de retorno ou não à origem

Portanto, $C^R < C^I$, e o algoritmo "Compute Return" decide retornar para a origem em $t = 5$. Ao retornar à origem, o algoritmo sai para entregar os pedidos 2 e 3 conforme calculado anteriormente.

No entanto, em $t = 15$ o pedido do vértice 4 é liberado e o algoritmo deve considerar a possibilidade de retornar para a origem para coletá-lo. Dessa vez, o custo associado a continuar em rota é dado por $C^I = C^S(Q_1 5^S) + \text{Offline}(Q_1 5^L, 50) = 50 + 90 = 140$, enquanto o custo associado ao retorno é dado por $C^R = \text{Offline}(Q_1 5^S \cup Q_1 5^L, 20) = 160$. Assim, $C^I < C^R$, e o algoritmo "Compute Return" decide continuar com a rota e ignorar o novo pedido.

Podemos comparar o algoritmo "Compute Return" com o "Naive Return" avaliando as decisões do "Naive Return" nos instantes de tempo $t = 5$ e $t = 15$. Note que em $t = 5$ temos que

$$\frac{y}{l_m} = \frac{5}{20} < \frac{1}{2} = \frac{k}{k+r}$$

e, portanto, o algoritmo também decide voltar, tomando a mesma decisão que o algoritmo "Compute Return". Entretanto, em $t = 15$ temos que

$$\frac{y}{l_m} = \frac{5}{20} < \frac{1}{3} = \frac{k}{k+r}$$

e o "Naive Return" decide voltar, contrariando a decisão do "Compute Return".

Note que se o servidor tivesse retornado à origem em $t = 15$, o custo teria sido 160 invés de 140, conforme já havia sido calculado pelo "Compute Return". Assim, podemos ver que o algoritmo "Compute Return" sempre toma decisões melhores baseado no que é conhecido da instância no momento da decisão.

Na Figura 3.6 podemos visualizar a rota completa realizada pelo algoritmo "Compute Return". As cores verde, azul e vermelho representam a pri-

meira, segunda e terceira sub-rotas, respectivamente.

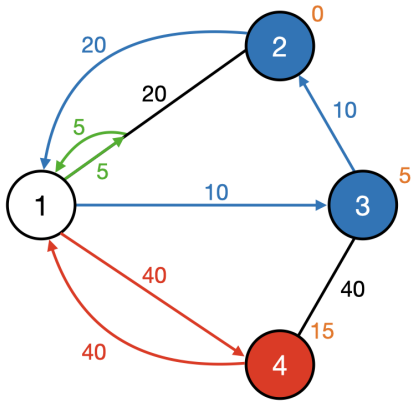

Figura 3.6: Rota realizada pelo algoritmo “Compute Return”

4
Versão offline do problema OnlinePickDel

No capítulo anterior, analisamos diferentes algoritmos online para solucionar o problema OnlinePickDel. Entretanto, tais algoritmos realizam chamadas para a versão offline do problema considerando os pedidos que já foram liberados. Nesta versão, o servidor tem conhecimento completo sobre a instância desde o início, incluindo os locais de entrega e o tempo de liberação de cada pedido.

Dessa forma, ter uma implementação eficiente do algoritmo offline é essencial para que o algoritmo online possa ser executado em instâncias maiores. Além disso, uma implementação geral do algoritmo offline pode ajudar a realizar experimentos computacionais e avaliar a competitividade empírica dos algoritmos online.

4.1
Algoritmo offline via busca por força bruta

A primeira versão implementada do algoritmo offline foi através da busca por força bruta.

O algoritmo offline é composto por duas rotinas - uma função principal (**Algoritmo 3**) que funciona como "casca" para o algoritmo, e uma função recursiva (**Algoritmo 4**) que é chamada pela "casca". O **Algoritmo 3** realiza uma chamada para o **Algoritmo 4** para cada permutação possível de vértices, que, por sua vez, calcula a rota de custo mínimo para entrega dos vértices naquela ordem. Por fim, o **Algoritmo 3** retorna a solução de menor custo.

O **Algoritmo 4** recebe uma permutação de pedidos e avalia qual a rota de menor custo para entregar os pedidos na ordem especificada. Para isto, o algoritmo considera todos os tempos de liberação de pedidos e o tempo inicial como possíveis instantes de tempo R^L para o servidor sair da origem e realizar entregas. Para cada instante de tempo $r^L \in R^L$ possível, são avaliados quais pedidos já foram liberados naquele instante de tempo e qual a quantidade máxima destes pedidos $k_{\max}$ que podem ser coletados levando em conta o limite de capacidade. Em seguida, é testado para cada possível quantidade de pedidos $k \leq k_{\max}$ qual o custo C_k de entregar os k primeiros pedidos em uma sub-rota, deixando os $k - k_{\max}$ pedidos para serem entregados posteriormente. O custo

de entrega dos pedidos restantes C_{rec} é calculado recursivamente considerando que os k primeiros pedidos já foram entregues. Assim, são testadas todas as possibilidades de instante de saída e de quantidade de pedidos coletados, e é retornada a solução de menor custo.

Além disso, para que a menor distância entre quaisquer dois vértices pudesse ser encontrada rapidamente, foi utilizado como pré-processamento o algoritmo de Floyd-Warshall para construir a matriz de menor distância entre quaisquer dois vértices do grafo.

O pseudo-código do algoritmo offline é dado a seguir:

Algoritmo 3: Função principal do algoritmo offline

Entrada: Grafo $G = (V, E)$ com vértice de origem $o \in V$,
capacidade c do servidor, sequência de pedidos
$\sigma = q_1, \ldots, q_n$, instante de tempo inicial t_0

Saída: Rota ótima viável S, valor mínimo da função objetivo
$\sum_{i=1^m} C_i^S$,
tempo de fim t_f

Inicialização: Pré-processar grafo G obtendo matriz de menor distância

Seja $\mathcal{P}(\sigma)$ o conjunto de todas as permutações de pedidos.

para cada $p \in \mathcal{P}(\sigma)$ **faça**

Faça uma chamada ao **Algoritmo 4** passando o grafo G, a capacidade c do servidor, a permutação de pedidos p e o instante de tempo inicial t_0.

Caso a solução retornada pelo **Algoritmo 4** tenha custo menor que a melhor solução conhecida até então, atualize a rota ótima S, o valor mínimo $\sum_{i=1^m} C_i^S$ e o tempo de fim t_f para a nova solução correspondente à permutação p.

fim

Algoritmo 4: Função recursiva do algoritmo offline

Entrada: Grafo $G = (V, E)$ com vértice de origem $o \in V$, capacidade c do servidor, permutação dos pedidos $p \in \mathcal{P}(\sigma)$, instante de tempo inicial t_0

Saída: Rota viável S, valor da função objetivo $\sum_{i=1^m} C_i^S$, tempo de fim t_f

Seja R^L o conjunto de instantes de tempo em que o servidor pode sair da origem para levar os pedidos.

Note que $R^L = \{r_i \mid (r_i, d_i) \in p \ \wedge \ r_i \geq t_0\} \cup \{t_0\}$.

para cada $r^L \in R^L$ **faça**

- Seja $Q^L = \{q_i \mid r_i \leq r^L\}$ o conjunto de pedidos liberados no instante de tempo r^L.
- Seja $k_{\max} = \min\{c, |Q^L|\}$.
- **para cada** $k \in 1, \ldots, k_{\max}$ **faça**
  - Faça a entrega dos k primeiros pedidos de p calculando o custo C_k e tempo de término t_k.
  - Faça a entrega de todos os pedidos restantes recursivamente. Isto é, faça uma chamada ao **Algoritmo 4** passando grafo G, a capacidade c do servidor, os pedidos restantes de p e o tempo inicial para os próximos pedidos t_k.
  - Sejam C_{rec} e t_{rec} o custo e tempo de fim calculados pela chamada recursiva para a entrega dos demais pedidos, respectivamente. O custo para entrega associado a essa permutação será dado por $C_p = C_k + C_{\text{rec}}$.
  - Caso C_p seja menor do que o menor custo atualmente conhecido pelo algoritmo, devemos atualizar a melhor solução conhecida considerando esta nova rota S, custo e tempo de fim t_{rec}.
- **fim**

fim

Como o algoritmo offline precisa executar a mesma rotina para um grande número de permutações, há a possibilidade de utilizar paralelismo para acelerar a busca. Assim, utilizamos *multi-threading* para dividir as permutações entre os núcleos disponíveis na máquina. Em uma máquina com n núcleos, cada núcleo fica responsável por processar uma fração de aproximadamente $\frac{1}{n}$ das permutações e, por fim, é escolhida a permutação de menor custo dentre as n melhores de cada núcleo. Como o código é altamente paralelizável, tal abordagem proporcionou um *speedup* consideravelmente alto. Em um

computador de 4 núcleos, obteve-se um *speedup* médio de 3.6.

Para verificar a corretude da implementação do algoritmo offline, foram realizados alguns testes unitários em grafos pequenos. O grafo escolhido para cada teste busca contemplar diferentes situações que podem ocorrer. Para cada grafo foram realizados testes utilizando diferentes quantidades de pedidos e capacidade.

Além disso, foram realizados alguns testes de desempenho para determinar o maior tamanho de grafo e número de pedidos que são atendidos em uma quantia de tempo razoável. Tais testes foram realizados em grafos euclideanos gerados aleatoriamente (vértices sendo pontos sorteados uniformemente em $[0, 1]^2$ e arestas sendo a distância euclideana entre os pontos).

Os testes de desempenho indicam que o algoritmo offline via força bruta descrito acima é capaz de resolver instâncias com até 8 pedidos em menos de 10 segundos (para qualquer capacidade). Instâncias com mais pedidos demoram uma quantia de tempo consideravelmente maior.

4.2 Algoritmo offline via programação inteira mista

Programação Inteira (PI) (Conforti et al. 2014) é uma ferramenta de otimização muito utilizada para modelar diversos problema de decisão, especialmente aqueles envolvendo decisões discretas. Um exemplo clássico é o problema da mochila. Neste exemplo, há uma mochila com capacidade de peso W, e N itens de valor v_i e peso w_i cada. O objetivo do problema é escolher os itens de forma a maximizar o valor carregado na mochila sem exceder a capacidade de peso disponível. Para modelar tal problema através de PI, podemos introduzir variáveis binárias x_i que indicam se a solução inclui ($x_i = 1$) ou não ($x_i = 0$) o i-ésimo item na mochila. Dessa forma, o problema da mochila é expresso por

$$\begin{aligned} \max_{x} \quad & \sum_{i=1}^{N} x_i v_i \\ \text{s.a.} \quad & \sum_{i=1}^{N} x_i w_i \leq W \\ & x_i \in \{0, 1\} \quad \forall i = 1, \ldots, N \end{aligned}$$

Para que fosse possível resolver a versão offline do problema OnlinePickDel em instâncias maiores, foi desenvolvida uma formulação do problema por meio de programação inteira mista. Esta foi baseada na formulação do problema de roteamento de veículos com janelas de tempo proposto por Cordeau et al. (Cordeau et al. 2002). Uma importante observação é que essa

formulação considera que todos os pedidos já foram liberados e que, portanto, qualquer pedido pode ser levado imediatamente para entrega. Note que essa limitação não influencia a qualidade das soluções obtidas durante o algoritmo online, uma vez que as chamadas ao algoritmo offline são feitas apenas com pedidos que já foram liberados.

A variável de decisão $x_{i,j,k}$ é uma variável binária que indica se a aresta i, j é utilizada na k-ésima rota. A variável de decisão $\omega_{i,k}$ representa o tempo em que o pedido do vértice i é atendido na k-ésima rota. Além disso: d_i indica se o vértice i possui algum pedido a ser atendido, $t_{i,j}$ indica o tempo de viagem de menor custo do vértice i até o vértice j, C representa a capacidade do servidor, t_0 representa o tempo de início, e $K_{\max}$ representa o número de rotas que serão utilizadas. Considere também que $\Delta^+(i)$ e $\Delta^-(i)$ representam os conjuntos de vértices adjacentes a i por meio de uma aresta de saída e de entrada, respectivamente.

Note que serão necessárias pelo menos $\lceil \frac{m}{C} \rceil$ rotas para servir todos os pedidos, onde m é o número total de pedidos. No entanto, há casos em que a melhor solução exige um número maior que $\lceil \frac{m}{C} \rceil$ rotas. Portanto, executaremos a PI para todos os valores de $K_{\max}$ de $\lceil \frac{m}{C} \rceil$ até m e escolheremos a melhor solução.

Os vértices 1 e $N + 1$ representam a origem. O grafo é pré-processado removendo todos os vértices que não possuem nenhum pedido ativo, o que faz com que o número de vértices do grafo seja sempre igual a $m + 2$, onde m é o número de pedidos.

Com as definições acima, podemos formular o problema por programação inteira mista como a seguir:

$$\min_{x,\omega} \quad \sum_{i=2}^{N} \sum_{k=1}^{K_{\max}} \omega_{i,k} \tag{4-1}$$

$$\text{s.a.} \quad \sum_{k=1}^{K_{\max}} \sum_{j \in \Delta^+(i)} x_{i,j,k} = 1 \quad \forall i = 2, \ldots, N \text{ tal que } d_i = 1 \tag{4-2}$$

$$\sum_{j \in \Delta^+(1)} x_{1,j,k} = 1 \quad \forall k = 1, \ldots, K_{\max} \tag{4-3}$$

$$\sum_{i \in \Delta^+(j)} x_{i,j,k} - \sum_{i \in \Delta^-(j)} x_{j,i,k} = 0 \quad \forall k = 1, \ldots, K_{\max},\ \forall j = 2, \ldots, N \tag{4-4}$$

$$\sum_{i \in \Delta^-(N+1)} x_{i,N+1,k} = 1 \quad \forall k = 1, \ldots, K_{\max} \tag{4-5}$$

$$x_{i,j,k}(\omega_{i,k} + t_{i,j} - \omega_{j,k}) \leq 0 \quad \forall k = 1, \ldots, K_{\max},\ \forall (i,j) \in E \tag{4-6}$$

$$\sum_{i=2}^{N} d_i \sum_{j \in \Delta^+(i)} x_{i,j,k} \leq C \quad \forall k = 1, \ldots, K_{\max} \tag{4-7}$$

$$\omega_{1,k+1} = \omega_{N+1,k} \quad \forall k = 1, \ldots, K_{\max} - 1 \tag{4-8}$$

$$\omega_{1,1} = t_0 \tag{4-9}$$

$$x_{i,j,k} \in \{0, 1\} \tag{4-10}$$

$$\omega_{i,k} \geq 0 \tag{4-11}$$

A função objetivo (4-1) representa a minimização da latência dos pedidos. A restrição (4-2) impõe que cada pedido deve ser atendido em uma única rota. As restrições (4-3), (4-4) e (4-5) caracterizam as restrições de fluxo da k-ésima rota no grafo. A restrição (4-6) é uma restrição não-linear que impõe que se a aresta (i, j) é utilizada na rota k, então $\omega_{i,k} + t_{i,j} \leq \omega_{j,k}$. A restrição (4-7) garante que em cada rota o servidor não levará mais pedidos do que seu limite de capacidade. A restrição (4-8) faz com que o tempo de término de cada rota seja igual ao tempo de início da próxima rota. Por fim, a restrição (4-9) garante que o tempo de início será igual a t_0.

Para linearizar a restrição (4-6), poderíamos substituí-la por $\omega_{i,k} + t_{i,j} - \omega_{j,k} \leq (1 - x_{i,j,k})M$ para algum M constante grande o suficiente. No entanto, o solver utilizado (Gurobi) permite a adição de restrições indicadoras diretamente, o que possibilita uma melhor incorporação dessa restrição na solução.

Além dos testes em grafos pequenos, a implementação do problema offline por programação inteira mista foi testada exaustivamente contra o algoritmo offline via busca por força bruta em diversas instâncias geradas aleatoriamente. Utilizamos grafos euclideanos gerados a partir da amostragem de pontos em $[0, 1]^2$ com probabilidade uniforme. Para que as soluções fossem comparáveis, todos os pedidos tinham tempo de liberação igual a zero. Em cada teste,

verificou-se que a solução encontrada pela PI foi a mesma que a encontrada pelo força bruta.

Realizamos também alguns testes de desempenho para verificar o maior tamanho de instância para o qual o algoritmo é capaz de encontrar a melhor solução em uma quantia de tempo razoável. Verificou-se que o solver é capaz de prover garantia de otimalidade (i.e., gap de otimalidade igual a zero) para instâncias com até 8 pedidos em menos de dez segundos. Para instâncias maiores, frequentemente o solver parece encontrar a melhor solução em pouco tempo, mas passa a demorar muito para diminuir o gap de otimalidade.

Dessa forma, em instâncias grandes podemos adotar um determinado limite de tempo e considerar a melhor solução que o solver for capaz de encontrar no dado limite de tempo como uma boa heurística.

No entanto, note que se o objetivo do algoritmo offline for exclusivamente ser executado pelo algoritmo online, o gargalo no tempo de execução passa a ser apenas o número de pedidos liberados/ativos em um dado instante de tempo. Isso ocorre devido ao fato de que o algoritmo online faz chamadas ao algoritmo offline apenas para encontrar a melhor rota para os pedidos liberados/ativos. Assim, é possível executar o algoritmo online em grafos grandes com um grande volume de pedidos, desde que poucos pedidos tornem-se liberados/ativos simultaneamente.

5
Experimentos computacionais e resultados

Neste capítulo descreveremos os experimentos computacionais realizados e os resultados e conclusões obtidas. Realizamos experimentos em bases de dados conhecidas para problemas similares e em instâncias sintéticas.

5.1
Ambiente de execução e limitações

Todos os experimentos foram executados em um MacBook Pro 2015 com 16 GB de memória RAM e processador de 4 núcleos Intel Core i7 Quad-Core de 2.2 GHz com 256 KB de memória cache L2 por núcleo e 6 MB de memória cache L3 compartilhada. Todos os algoritmos foram implementados na linguagem de programação Julia. O solver utilizado para resolver o problema de programação inteira foi o Gurobi.

Devido às limitações de tempo de execução do algoritmo offline, que só é capaz de prover soluções com garantia de otimalidade rapidamente com até 8 pedidos, decidimos separar os experimentos em duas subseções: experimentos em instâncias menores, para as quais é possível encontrar soluções ótimas; e experimentos em instâncias maiores, em que são realizadas flexibilizações.

Nas instâncias de até 8 pedidos o algoritmo offline via força-bruta é capaz de encontrar a solução ótima mais rapidamente do que o algoritmo offline via programação inteira mista. Dessa forma, a programação inteira mista é chamada apenas para entradas com 9 ou mais pedidos.

Ainda assim, para que fosse possível executar os algoritmos online em instâncias maiores com um grande número de pedidos, foi necessário incluir algumas limitações no o algoritmo offline via programação inteira mista. Com esse intuito, foi estabelecido um limite de tempo para a execução de cada chamada da PI de 20 segundos. Além disso, foi fixado o valor de $K_{\max} = \lceil \frac{m}{C} \rceil$ para diminuir o número de chamadas necessárias, apesar da solução ótima possivelmente utilizar mais rotas do que isso. Por fim, inicializamos as variáveis de decisão com uma solução viável para prover um *warm start*.

Para isso, foi implementado um algoritmo guloso. O algoritmo sempre realiza $\lceil \frac{m}{C} \rceil$ rotas e, em cada rota, serve os pedidos que estiverem mais próximos do nó corrente. Como exemplo, considere uma instância com pedidos 2, 3, 4 e

5. O algoritmo primeiramente verifica qual destes é mais próximo à origem - suponha que é o pedido 4. Ele então serve o pedido 4 e verifica qual o pedido mais próximo do 4 para seguir em diante. Sempre que o servidor atinge o limite de capacidade, volta à origem e começa uma nova rota.

Esse algoritmo é muito limitado, mas ao menos provê uma solução viável para o solver utilizar como *warm start*. Tal solução é fundamental especialmente em instâncias grandes, nas quais o solver tem dificuldade de encontrar uma primeira solução viável.

5.2 Base de dados

Os experimentos apresentados nesta seção foram realizados em uma base de dados de Roteamento de Veículos. A base de dados utilizada foi a "goeke 2018" (Goeke 2018) com 92 instâncias, disponível no repositório de bases de dados VRP-REP.

Cada instância possui uma série localizações. Para cada localização, a base de dados provê as seguintes informações:

- StringID: identificador único da localização
- Type: tipo da localização, podendo ser depósito, local de coleta, local de entrega ou estação de reabastecimento
- Coordenadas (x, y): indicam a localização no espaço euclideano (distâncias são consideradas euclideanas também)
- Demand: quantidade de carga necessária (sendo positivo para coleta e negativo para entrega)
- ReadyTime e DueDate: início e fim da janela de tempo para entrega, respectivamente
- Service time: tempo de serviço (tempo que leva para o servidor coletar ou entregar o pedido na localização)
- PartnerID: em locais de coleta ou entrega, o PartnerID corresponde ao StringID associado ao local correspondente de entrega ou coleta, respectivamente

Dentre estes, utilizamos apenas os atributos "Type", "ReadyTime" e as coordenadas (x, y). No nosso problema, assumimos que todos os pedidos serão coletados na origem (i.e., no depósito). Dessa forma, consideramos que a origem será o local de depósito indicado pelo "Type". Além disso, todos os locais com "Type" coleta ou entrega são considerados como locais de entrega, enquanto os locais com "Type" estação de reabastecimento são desconsiderados.

Cada pedido é criado na posição especificada pelas coordenadas (x, y) associadas e é liberado no instante de tempo dado por "ReadyTime".

5.2.1 Resultados em instâncias menores da base de dados

Nesta subseção, consideramos instâncias "pequenas" da base de dados. Como o gargalo dos algoritmos online é a chamada do algoritmo offline, a noção de "pequeno" consiste em não realizar chamadas para o algoritmo offline com 9 ou mais pedidos. Assim, se durante a execução algum dos algoritmos chama o algoritmo offline com 9 ou mais pedidos, a instância foi eliminada. Com isso, do total de 92 instâncias, apenas 28 foram utilizadas.

Na Figura 5.1 está ilustrado o custo médio dos algoritmos nas 28 instâncias em função da capacidade. Note que o algoritmo "Compute Return" tem o menor custo médio para todos os valores de capacidade.

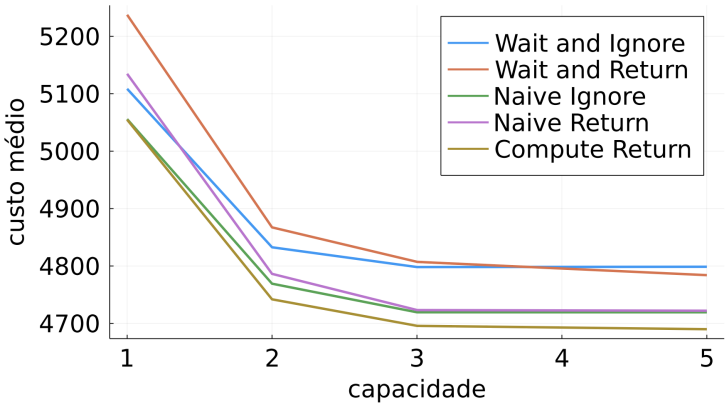

Figura 5.1: Custo médio em função da capacidade.

Além disso, pode-se perceber que os algoritmos do tipo "Wait" tiveram um custo médio mais alto que seus análogos "Naive" para todos os valores de capacidade. Isso indica que o critério de espera utilizado, em média, não foi eficaz, apesar de prover soluções melhores em alguns casos.

Pode-se perceber também que os algoritmos "Wait and Return" e "Naive Return" tem maior custo médio que seus análogos "Wait and Ignore" e "Naive Ignore" para valores de capacidade menores. No entanto, na medida com que a capacidade do servidor aumenta, a qualidade das soluções encontradas pelos algoritmos de retorno aumenta e tal diferença diminui.

Na Figura 5.2, podemos ver em quantas instâncias cada algoritmo foi capaz de prover a solução de menor custo para diferentes valores de capacidade. Em caso de empate, a "vitória" é contabilizada para todos os algoritmos que proveram a solução de menor custo.

Tal figura reforça os resultados da Figura 5.1, mas também demonstra que muitas vezes algoritmos diferentes encontram a mesma solução, e também que nem sempre o algoritmo "Compute Return" encontra soluções com menor custo que os demais.

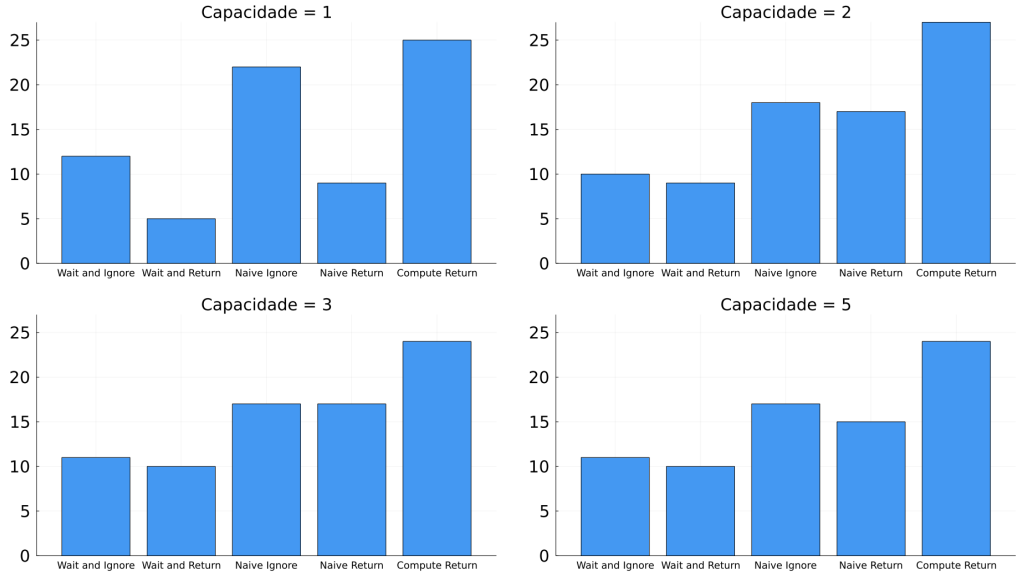

Figura 5.2: Número de "vitórias" dos algoritmos para diferentes valores de capacidade.

5.2.2 Resultados em instâncias maiores da base de dados

Nesta subseção foram consideradas algumas instâncias maiores. Para executar tais instâncias, foi estabelecido um limite de até 25 pedidos para a chamada do algoritmo offline via programação inteira.

Usamos aqui uma metodologia um pouco diferente da usada na Seção 5.2.1. Como poucas instâncias adicionais foram capazes de executar em todos os valores de capacidade dada a limitação de 25 pedidos, foi necessário admitir que algumas instâncias não possuíssem resultados para alguns valores de capacidade.

Assim, as instâncias que excederam o limite com *warm start* para algum valor de capacidade tiveram esse valor de capacidade desconsiderado. Instâncias que excederam o limite em todos os limites de capacidade foram totalmente desconsideradas. Com isso, foi possível analisar 12 instâncias, cujos resultados podem ser observados na Figura 5.3.

Dentre as 12 instâncias, 6 puderam ser executadas com capacidade $c = 1$, 10 com capacidade $c = 2$, 10 com capacidade $c = 3$ e 10 com capacidade $c = 5$. Analisando a figura, pode-se concluir que nestas instâncias o critério de espera

do algoritmo "Wait" também não foi eficaz, enquanto o algoritmo "Compute Return" teve os melhores resultados.

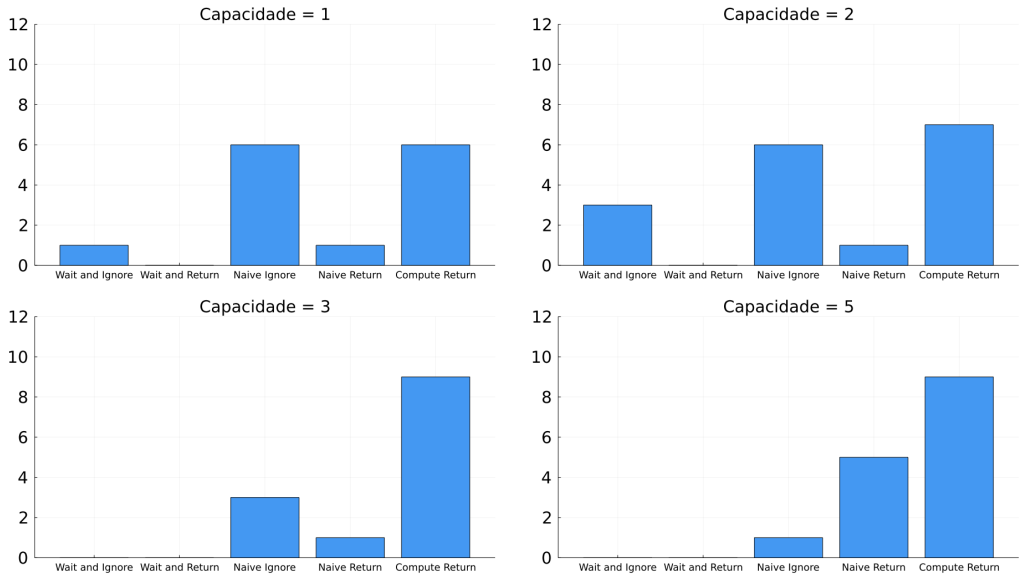

Figura 5.3: Número de "vitórias" dos algoritmos para diferentes valores de capacidade.

5.3 Instâncias sintéticas

Para ter um entendimento melhor do impacto de diferentes fatores como distância e intervalo de tempo médio entre pedidos, geramos instâncias aleatórias variando tais parâmetros e comparamos o desempenho dos algoritmos online.

Para gerar uma instância com N pedidos, amostramos $N + 1$ pontos uniformemente em $[0, L]^2$, onde $L \in \mathbb{R}$. O primeiro ponto amostrado é utilizado como origem (ponto de coleta dos pedidos), enquanto os N pontos restantes são associados a pedidos. Dessa forma, geramos um grafo em que a distância entre dois vértices é igual à distância euclideana entre os pontos associados a cada um dos vértices. Gerando os pontos dessa forma, a distância média entre dois pontos é de aproximadamente $0.52L$.

Para determinar o intervalo de tempo entre a liberação de cada pedido, foi utilizada uma distribuição de probabilidade exponencial. Utilizamos a função densidade de probabilidade parametrizada em β, conforme a seguir:

$$f(t;\beta) = \begin{cases} \frac{1}{\beta} e^{-\frac{t}{\beta}}, & t \geq 0 \\ 0, & t < 0 \end{cases}$$

Com essa parametrização, temos que $T \sim \text{Exp}(\beta) \implies \mathbb{E}[T] = \beta$. Assim, o tempo médio de espera entre pedidos é igual a β. A distribuição exponencial foi escolhida devida à sua propriedade de falta de memória $\Pr(T > s + t \mid T > s) = \Pr(T > t), \forall s, t \geq 0$. Essa propriedade é ideal para modelar tempos de espera e é muito utilizada em Teoria das Filas. Mais detalhes podem ser encontrados no livro "Probability, Random Variables, and Stochastic Processes", de Athanasios Papoulis (Papoulis 2002).

Dessa forma, a partir dos parâmetros N, L e β podemos gerar instâncias com diferentes distribuições de espaço e tempo para os pedidos.

Foram realizados diversos testes fixando o valor de L e variando β para diferentes valores de N. Dessa forma é possível explorar como o tempo médio entre a liberação de pedidos impacta na tomada de decisão dos algoritmos. Quanto menor o valor de β, mais frequente será a liberação de pedidos.

5.3.1 Resultados em instâncias sintéticas menores

Nesta subseção, analisaremos experimentos realizados com $N = 8$ pedidos. Assim, em todos os experimentos realizados o algoritmo offline foi capaz de encontrar a solução ótima. Para cada valor de β, foram geradas 100 instâncias aleatórias que foram utilizadas para testar os algoritmos utilizando diferentes valores de capacidade. Assim, os resultados apresentados nesta subseção são uma média de 100 realizações.

Na Figura 5.4 podemos visualizar o desempenho de cada um dos algoritmos online em função da capacidade para diferentes valores de β. Note que como β é responsável por determinar o tempo médio entre a liberação dos pedidos, o valor de β impacta significativamente a escala das soluções. A partir dos gráficos apresentados nesse figura, é possível realizar várias observações.

Primeiramente, pode-se perceber que o algoritmo "Compute Return" teve um custo médio menor que os algoritmos anteriores independente dos valores de capacidade e β, indicando a eficácia geral do critério de retorno proposto.

Ademais, note que os algoritmos do tipo "Wait" tiveram um custo médio mais alto que seus análogos "Naive" para praticamente todos os valores de capacidade e β. Note também que essa diferença se mantém aproximadamente a mesma para diferentes valores de capacidade dado um mesmo β. Isso indica que o critério de espera utilizado nos algoritmos do tipo "Wait", apesar de beneficial em alguns casos, não é efetivo em média.

Além disso, note que para valores menores de capacidade os algoritmos "Wait and Return" e "Naive Return" têm um custo médio mais alto que seus análogos "Wait and Ignore" e "Naive Ignore". Isso indica que o critério de

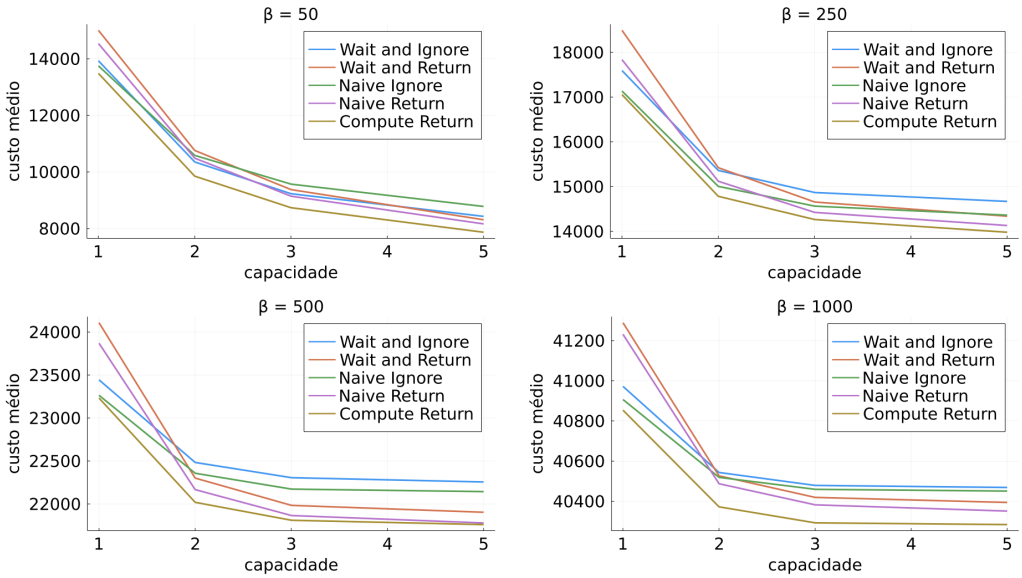

Figura 5.4: Custo médio em função da capacidade para diferentes valores de β.

retorno utilizado por tais algoritmos não é eficaz em média para valores baixos de capacidade. No entanto, para valores maiores de capacidade o critério de retorno obtém resultados mais próximos aos obtidos pelo algoritmo "Compute Return".

Outra observação importante é que o efeito do aumento da capacidade no custo médio é maior para valores menores de β. Isso pode ser explicado pelo fato de que nos casos em que o tempo médio entre a liberação de pedidos β é menor, uma quantia maior de pedidos acumula na origem, fazendo com que a capacidade faça uma diferença maior no custo. É importante ressaltar que como as localizações dos pedidos e da origem foram amostrados de $[0, 500]^2$, a distância média entre dois pontos é de aproximadamente 260. Assim, para valores maiores de β, é menos comum que uma quantidade grande de pedidos acumulem na origem, fazendo com que valores maiores de capacidade se tornem menos relevantes.

A Figura 5.5 ajuda com a compreensão do comportamento dos diferentes algoritmos na medida com que variamos β. Nesta figura, podemos perceber que apesar do número de pedidos ser o mesmo, o custo aumenta conforme aumentamos β devido ao tempo de espera entre os pedidos ser maior.

Além disso, note que para valores pequenos de β a dispersão de custo médio entre os diferentes algoritmos é maior, especialmente para valores de capacidade pequenos também. Isso indica que quanto menor o tempo entre pedidos, melhor deve ser o critério do algoritmo online para lidar com o maior volume de pedidos, i.e., mais "esperto" o algoritmo deve ser.

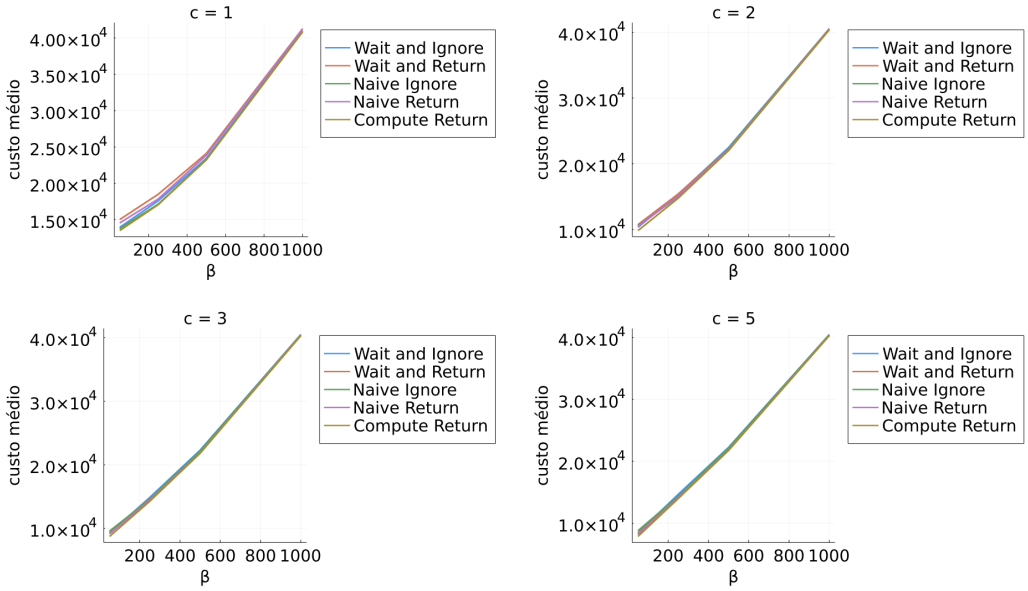

Figura 5.5: Custo médio em função de β para diferentes valores de capacidade.

Por fim, podemos visualizar o comportamento dos diferentes algoritmos de forma resumida na Figura 5.6.

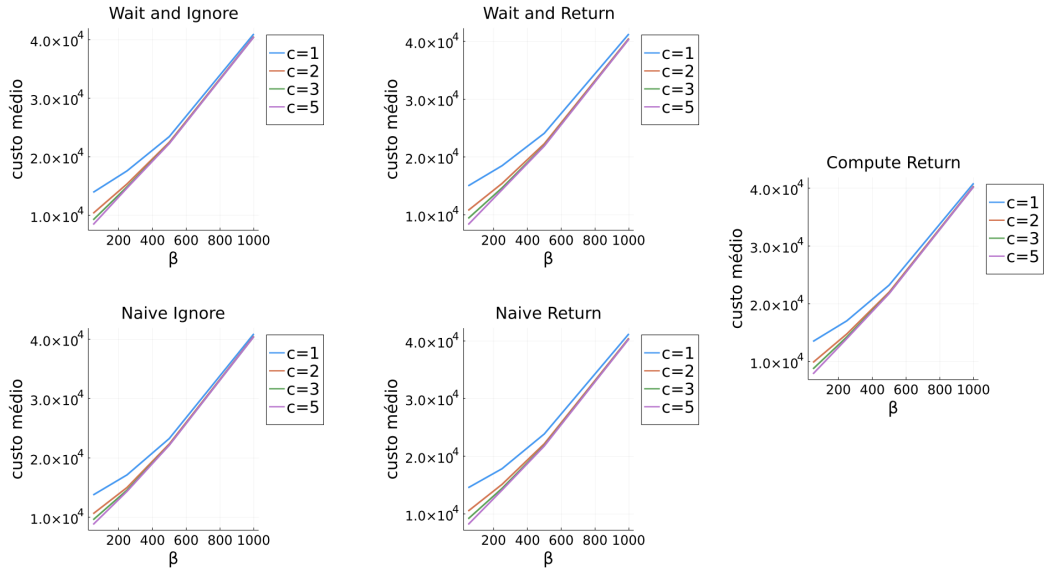

Figura 5.6: Custo médio em função de β para os diferentes algoritmos.

5.3.2
Resultados em instâncias sintéticas maiores

Nesta subseção serão analisados experimentos realizados com $N = 20$ pedidos. Assim como na subseção anterior, foram geradas 100 instâncias aleatórias para cada valor de β para testar os algoritmos utilizando diferentes valores de capacidade. No entanto, devido ao alto custo computacional de

executar tais instâncias com valores pequenos de β, foram utilizados apenas os valores $\beta = 500$ e $\beta = 1000$. Além disso, nas chamadas para o algoritmo offline com 9 ou mais pedidos, foi utilizada a implementação por programação inteira conforme especificado na Seção 5.1.

Analisando a Figura 5.7, podemos perceber que o comportamento dos algoritmos em instâncias maiores é similar ao comportamento observado nas instâncias menores (com $N = 8$) na subseção anterior.

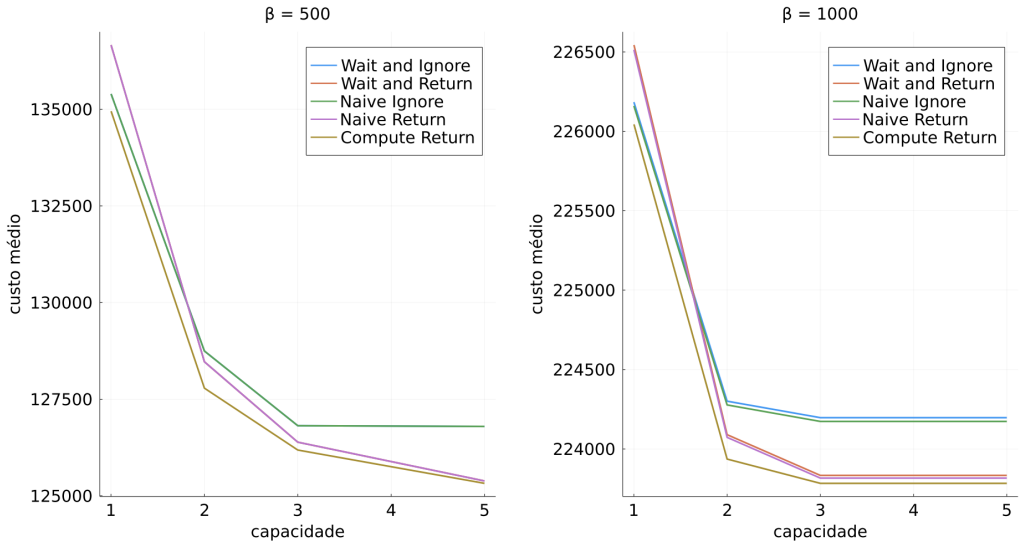

Figura 5.7: Custo médio em função da capacidade para diferentes valores de β.

5.3.3 Resultados de competitividade empírica

Nesta subseção, analisaremos resultados de *competitive ratio* médio empíricos. Seja C_{on} o custo de uma solução do algoritmo online em uma determinada instância e C_{off} o custo da solução ótima do algoritmo offline na mesma instância. O *competitive ratio* do algoritmo online nessa instância é dado por $\frac{C_{\text{on}}}{C_{\text{off}}}$. Assim, quanto mais próximo de 1 o *competitive ratio*, mais próximo o algoritmo online está da solução ótima.

Para que fosse possível executar os experimentos e comparar com a solução obtida pelo algoritmo offline com garantia de otimalidade, foram utilizadas 100 instâncias aleatórias para cada valor de β, cada uma com $N = 6$ pedidos. Cada valor de *competitive ratio* é a média dos *competitive ratios* nas 100 instâncias.

Na Figura 5.8 podemos comparar os *competitive ratios* dos algoritmos em função da capacidade para diferentes valores de β. Nesta figura, podemos

perceber que o *competitive ratio* do algoritmo "Compute Return" tende a aumentar conforme o aumento da capacidade. Tal resultado reforça a ideia de que para valores maiores de capacidade do servidor há uma maior complexidade na tomada de decisão, o que faz com que o algoritmo online tenha um pior desempenho.

Além disso, os algoritmos "Wait and Ignore" e "Naive Ignore" tiveram um aumento de *competitive ratio* ainda maior conforme o aumento de capacidade, o que indica a ineficácia da estratégia de ignorar pedidos novos. Ao analisar os algoritmos "Wait" com seus correspondentes "Naive", pode-se concluir que o critério de espera utilizado é ineficaz.

Podemos, ainda, avaliar o desempenho dos algoritmos "Wait and Return" e "Naive Return". Nestes, percebe-se que o *competitive ratio* se mantém aproximadamente o mesmo, se aproximando do "Compute Return" na medida em que a capacidade aumenta. Tal resultado indica que o critério de retorno dos algoritmos anteriores é ineficaz para valores mais baixos de capacidade, mas se aproxima do "Compute Return" para valores mais altos.

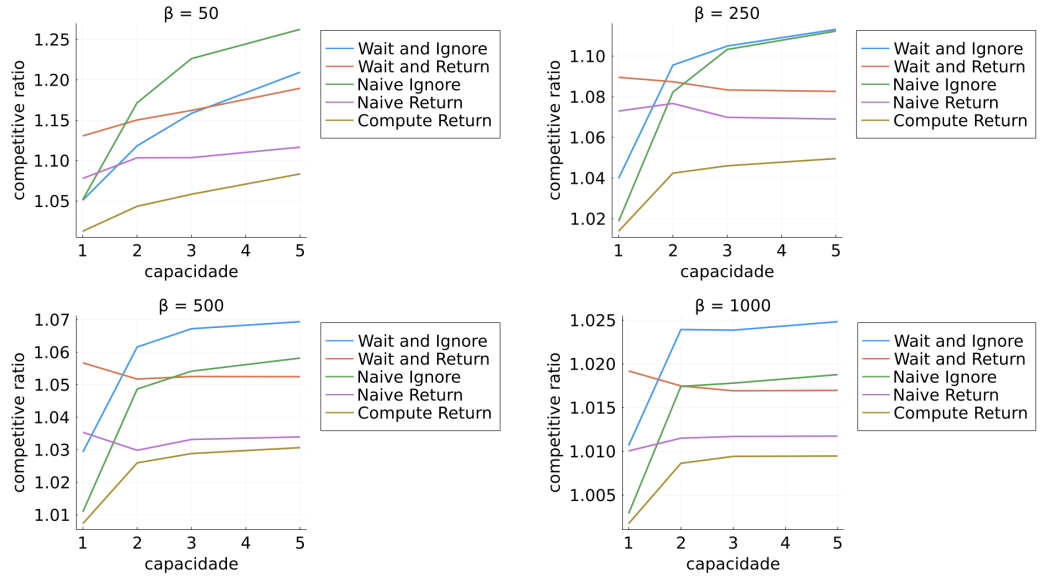

Figura 5.8: *Competitive ratio* em função da capacidade para diferentes valores de β.

Na Figura 5.9 podemos visualizar as informações anteriores de uma outra perspectiva, com o *competitive ratio* em função de β para diferentes valores de capacidade. Nesta figura, fica claro que conforme o valor de β aumenta, o *competitive ratio* de todos os algoritmos diminui significativamente. Isto reforça a ideia de que há uma maior complexidade quando há uma maior taxa de liberação de pedidos e, consequentemente, um maior volume de pedidos acumulados.

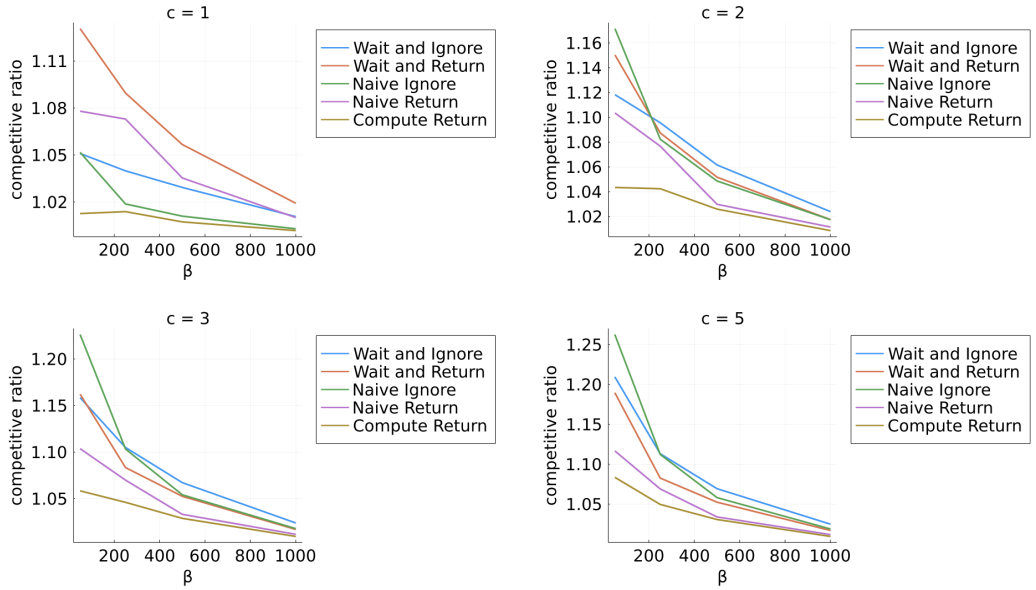

Figura 5.9: *Competitive ratio* em função da β para diferentes valores de capacidade.

Por fim, podemos visualizar o *competitive ratio* de cada algoritmo online em função de β para diferentes valores de capacidade na Figura 5.10.

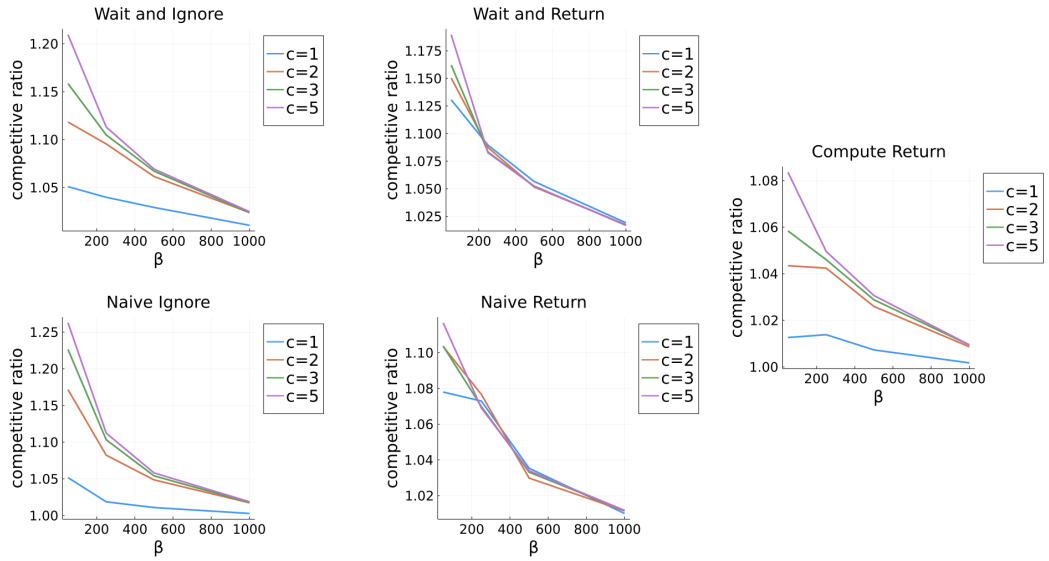

Figura 5.10: *Competitive ratio* dos diferentes algoritmos online.

6
Conclusão

Neste trabalho, analisamos diferentes algoritmos online e propusemos novos critérios para melhorar a qualidade das soluções obtidas. A partir dos experimentos computacionais, pode-se concluir que o critério de espera dos algoritmos do tipo "Wait" não é eficaz em média. Também pode-se observar que o critério de retorno dos algoritmos "Wait and Return" e "Naive Return" é eficaz apenas quando a capacidade do servidor é grande, ou quando a frequência de liberação de pedidos é pequena. Por fim, o algoritmo proposto "Compute Return" é robusto a essas situações e possui um desempenho médio melhor que os demais algoritmos.

Devido à amplitude do problema, há uma série de oportunidades para trabalhos futuros. Uma possibilidade é tentar experimentar diferentes critérios de espera. Em particular, o conceito de *active time* utilizado só afeta os primeiros pedidos liberados, de forma que, a partir de um certo instante de tempo, se torna obsoleto.

Outra direção interessante é pensar em otimizações da formulação por programação inteira e/ou em heurísticas para que seja possível resolver o algoritmo offline em instâncias maiores em um tempo razoável.

Por fim, trabalhos futuros podem também cogitar trabalhar em variações do problema. Pode-se, por exemplo, adotar vários locais de coleta invés de apenas uma origem, considerar vários servidores, ou buscar otimizar uma função objetivo diferente (como minimizar a "demora" máxima de entrega).

Referências bibliográficas

[Conforti et al. 2014] CONFORTI, M.; CORNUÉJOLS, G. ; ZAMBELLI, G.. **Integer programming**, volumen 271. Springer, 2014.

[Cordeau et al. 2002] CORDEAU, J.; DESAULNIERS, G.; DESROSIERS, J.; SOLOMON, M. M. ; F., S.. **VRP with time windows**. The Vehicle Routing Problem, p. 157–193, 2002.

[Dunning et al. 2017] DUNNING, I.; HUCHETTE, J. ; LUBIN, M.. **Jump: A modeling language for mathematical optimization**. SIAM Review, 59(2):295–320, 2017.

[Goeke 2018] DOMINIK GOEKE, 2021. [Online; acessado em 21 de Outubro de 2021].

[Gurobi 2021] GUROBI OPTIMIZATION, L.. **Gurobi optimizer reference manual**, 2021.

[Ho et al. 2018] HO, S. C.; SZETO, W.; KUO, Y.-H.; LEUNG, J. M.; PETERING, M. ; TOU, T. W.. **A survey of dial-a-ride problems: Literature review and recent developments**. Transportation Research Part B: Methodological, 111(C):395–421, 2018.

[Mor et al. 2020] MOR, A.; SPERANZA, M. G.. **Vehicle routing problems over time: a survey**. 4OR, 18(2):129–149, Jun 2020.

[Papoulis 2002] PAPOULIS, A.; PILLAI, S. U.. **Probability, Random Variables, and Stochastic Processes**. McGraw Hill, Boston, fourth edition, 2002.

[Valor Investe 2020] NATHÁLIA LARGHI. **Com quarentena, apps de entregas são oportunidade para trabalhadores e comércios**, 2021. [Online; acessado em 11 de Abril de 2021].

[Yu et al. 2020] YU, H.; LUO, X. ; WU, T.. **Online pickup and delivery problem with constrained capacity to minimize latency**. Journal of Combinatorial Optimization, June 2020.