UFSC - Universidade Federal de Santa Catarina
Departamento de Informática e Estatística

Matheus Leonel Balduino

Aplicação do Problema do Comprador Viajante em uma lista de compras

Florianópolis
2023

Matheus Leonel Balduino

Aplicação do Problema do Comprador Viajante em uma lista de compras

Tese submetida à CURSO DE CIÊNCIA DA COMPUTAÇÃO para a obtenção do Grau de Bacharel em Ciência da Computação.

Orientador: Prof. Álvaro Junio Pereira Franco

Florianópolis
2023

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE REDUÇÕES

ATPP: *Capacitated Asymmetric Traveling Purchaser Problem* (Problema do comprador viajante assimétrico)
C-ATPP: *Capacitated Asymmetric Traveling Purchaser Problem* (Problema do comprador viajante capacitado assimétrico)
LKH: *Lin-Kernighan Heuristic* (Heurística de Lin-Kernighan)
LSH: *Local Search Heuristic* (Heurística de busca local)
TPP: *Traveling Purchaser Problem* (Problema do comprador viajante)
TSP: *Traveling Salesman Problem* (Problema do vendedor viajante)

RESUMO

O problema do comprador viajante (TPP - *Traveling Purchaser Problem*), possui várias aplicações no mundo real. Neste trabalho, iremos explorar sua aplicação na solução de uma lista de compras, na busca de encontrar quais são os mercados de uma determinada região que oferecem os melhores preços para os produtos, levando em consideração o valor do deslocamento entre estes mercados. Este cálculo é algo extremamente custoso computacionalmente, pois trata-se de uma solução para um problema NP-DIFÍCIL (especialização do TSP - *Traveling Salesman Problem*), portanto, grande parte dos trabalhos propostos anteriormente apresentam heurísticas para o problema. Desenvolver uma solução para problemas difíceis em tempo aceitável para o usuário, geralmente acostumado com tempos de resposta extremamente baixos, é um desafio, e por isso o foco será nas heurísticas e não nos algoritmos que devolvem uma solução ótima. Existem diversas variações do TPP, porém a natureza do problema apresentado no presente trabalho exige que o algoritmo tenha suporte para grafos direcionados, e que a quantidade de produtos disponível em cada mercado seja limitada (conhecido como ”C-ATPP Capacitated Asymmetric Traveling Purchaser Problem”). O foco da contribuição deste trabalho está na busca e apresentação dos melhores parâmetros para o algoritmo implementado, utilizando-se de experimentos com instâncias de testes previamente solucionadas, para fins de comparação.

Palavras-chave: TPP, ATPP, C-ATPP, TSP (Traveling Salesman Problem), grafos, algoritmos

INTRODUÇÃO

As compras de supermercado, por muitas vezes, podem ser um problema para famílias com orçamento reduzido. A variação de preços de um supermercado para outro pode fazer uma grande diferença no valor final da lista de compras, e isso pode afetar drasticamente a qualidade de vida de uma família. Estas famílias, por vezes, precisam realizar pesquisas de preços em vários supermercados diferentes, em busca dos melhores descontos.

O ato de pesquisar os preços de produtos em uma lista de compras, em um conjunto de mercados, é algo muito custoso computacionalmente quando a lista de compras possui alguns itens e o conjunto de mercados possui alguns elementos. Nessas condições, o processo de encontrar a melhor solução para a compra dos itens nos mercados se torna praticamente impossível de ser realizado, sem o uso de métodos computacionais. Utilizando-se apenas de papel e caneta pode-se dizer que se trata de uma tarefa impossível. O problema é NP-Difícil, e o tempo necessário para se obter a solução ótima pode ser muito alto (generalização do TSP - *Traveling Salesman Problem*).

Este problema é amplamente estudado pela comunidade científica, visto que as soluções para um problema da classe NP podem ser utilizadas para solucionar outros problemas da mesma classe. Existem diversas contribuições recentes que se utilizam de técnicas diferentes das apresentadas neste trabalho, como inteligência artificial, algoritmos genéticos, etc. O foco deste trabalho está na apresentação, discussão, e testes, de um algoritmo de busca local iterada.

De maneira mais formal, o problema pode ser descrito da seguinte forma: "Encontrar o melhor orçamento final para uma lista de compras **L**, onde cada produto **P** precisa ser necessariamente comprado em um mercado pertencente ao conjunto **M**, levando em consideração o valor do deslocamento, um grafo dirigido, e que a quantidade de produtos que cada mercado oferece é limitada". Esse problema foi primeiramente proposto por Ramesh, em 1981, tendo sido chamado de "**TPP - *Traveling Purchaser Problem***" [2]. Por se tratar de uma generalização do TSP, o problema possui uma complexidade alta. Apenas para fins de comparação, observe a Figura 1 que ilustra uma tabela contendo um exemplo do número de combinações possíveis em relação ao número de vértices do problema.

Number of Cities	Number of Possible Solutions
10	3,628,800
11	39,916,800
12	479,001,600
13	6,227,020,800
14	87,178,291,200
15	1,307,674,368,000

Figura 1 - Relação entre o tamanho da entrada e as combinações possíveis

Problemas NP-difíceis não podem ser resolvidos em tempo satisfatório quando tratamos de instâncias com tamanho acima do trivial. Algoritmos que encontram uma solução ótima acabam que, na prática, não são úteis, uma vez que o tempo de resposta é excessivamente alto (utilizando um tempo de, no máximo, 1 minuto entre início de uma requisição e o recebimento da resposta como referência). Nesse caso, algoritmos que oferecem algum resultado não-ótimo acabam se tornando mais interessantes (heurísticas).

Encontrar uma solução para problemas deste tipo é uma forma de melhorar a qualidade de vida de famílias de baixa renda, ao mesmo tempo em que se aumenta o fluxo de dinheiro que é movimentado pelas grandes empresas do setor de varejo/atacado, por exemplo.

A principal contribuição deste trabalho está na busca dos melhores parâmetros de execução para o algoritmo utilizado como base na implementação. Esses valores são buscados através de testes que se utilizam de instâncias de teste previamente resolvidas, para que seja possível realizar comparações na qualidade da solução oferecida. O principal parâmetro, chamado ***dropLen***, recebeu um valor de 39% como o melhor valor (tendo em vista o conjunto de testes realizados).

OBJETIVOS

Objetivo geral

Realizar um estudo sobre o estado da arte, e implementar um protótipo de algoritmo para a solução do problema do comprador viajante, aplicado na solução de uma lista de compras de supermercado simples. Note que a implementação do protótipo visa a construção de uma heurística que possa ser utilizada em tempo real, para solução das listas, em oposição a um algoritmo que encontre a solução ótima mas possua tempo demasiado alto. A heurística deve dar suporte para grafos direcionados, e levar em consideração a quantidade de cada produto disponível em cada mercado.

Objetivos Específicos

- Desenvolver um breve estudo sobre o estado da arte dos algoritmos utilizados atualmente para a solução do problema do comprador viajante;
- Desenvolver um algoritmo (protótipo), implementado na linguagem GO (justificativa logo abaixo), que encontre uma solução viável em tempo real para o Problema do Vendedor Viajante.

JUSTIFICATIVA

Diariamente, em todos os lugares do mundo no qual existem mercados de varejo, podendo-se incluir até mesmo os mercados de atacado, as pessoas necessitam realizar compras de supermercado. É muito comum que famílias façam uma lista de compras e vão ao mercado comprar os itens.

Em busca de economia, é comum que as famílias identifiquem os melhores preços, nos melhores mercados, ao longo do tempo, e com isso passem a visitar mais de um mercado quando vão às compras. Ao escolherem passar em mais de um mercado, no entanto, acabam por não levar em consideração várias métricas importantes, como gasto de combustível, localização dos supermercados, tempo, etc.

A busca pela melhor solução, dada uma lista de produtos, é um problema difícil de ser resolvido. Tentar solucionar sem auxílio computacional é uma tarefa impossível quando a lista de produtos não possui um tamanho trivial, pois trata-se de um problema NP-Difícil (variação do Problema do Caixeiro Viajante). É em busca de melhores preços, desempenho, economia, que a criação e execução de um algoritmo se justifica. Para viabilizar toda esta estrutura, é preciso que o algoritmo responsável pelo cálculo tenha um tempo de resposta aceitável, visto que a experiência do usuário exige interatividade.

O foco deste trabalho está na solução de listas de compras, mas repare que os algoritmos relacionados com essa tarefa podem ser generalizados e utilizados em uma série de outras situações. É comum encontrar aplicação para este mesmo algoritmo em situações como a compra de matéria-prima para uma indústria específica, por exemplo.

Imagine uma indústria de montagem de veículos. Para montar um veículo, é necessário uma série de itens, sendo que cada um destes itens é vendido por uma empresa diferente. Pode ser interessante, em alguma situação, que a montadora precise buscar esses itens diretamente com os vendedores, ao invés de receber diretamente nas suas instalações. Nesse caso, é necessário fazer a aplicação dos algoritmos que tratamos aqui para solucionar essa lista de compras.

MÉTODOS

Para a construção dos modelos iniciais, e estudos de artigos publicados anteriormente, foram realizadas pesquisas através do serviço oferecido pela Google, chamado Google Scholar.

Para implementação optou-se por GO, ao invés de uma linguagem com mais desempenho (C ou Rust, por exemplo), pois ela oferece um excelente balanço entre produtividade e eficiência. Trata-se de uma linguagem compilada para código nativo, com uso de garbage collector (GC) para gerenciamento de memória.

A princípio foi levado em consideração a possibilidade do uso de dados 100% reais para a execução e teste dos algoritmos, mas o desafio de logística associado a ir até os supermercados fazer a coleta dos preços em tempo real para alimentar a base foi considerado inviável (e encontrar a solução dessas instâncias reais seria

mais difícil ainda). Portanto, para implementação e testes dos algoritmos, serão utilizados os dados do trabalho proposto por Laporte et al. em 2000 [1].

Todos os detalhes e discussões referentes às tecnologias envolvidas no processo de desenvolvimento e testes, podem ser visualizados nas seções seguintes. Na próxima seção, é descrita uma revisão de literatura para o TPP.

REVISÃO DE LITERATURA

Esta seção é dividida em dois grandes grupos de algoritmos: aqueles que possuem solução exata e outros que tratam-se de heurísticas. Por um lado, algoritmos que solucionam o problema encontrando o mínimo global são interessantes pois temos a certeza de que a resposta não tem como ser melhor. No entanto, o tempo exigido para o cálculo, em geral, não é aceitável. Por esse motivo, o foco estará mais nos algoritmos que fornecem soluções que tentam “aproximar-se” do mínimo global, com alguma porcentagem de distanciamento.

Algoritmos que oferecem mínimos globais

O primeiro autor a apresentar o problema da forma que encaramos aqui, foi Ramesh, em 1981, com seu artigo "*Traveling Purchaser Problem*" [2]. Neste artigo, Ramesh apresenta um algoritmo que soluciona o problema através de uma busca lexicográfica nos ramos de uma árvore de decisão, realizando alguns cortes em alguns ramos dependendo do valor do deslocamento. Como esperado, este algoritmo possui um tempo de execução alto, sendo ineficaz para qualquer instância que contenha mais de 12 mercados e 10 produtos [2].

Após o trabalho de Ramesh, Singh e Oudheusden (1997) [3] apresentaram uma proposta que se utilizava de um algoritmo de *Branch and Bound*, capaz de solucionar instâncias de tamanho um pouco maiores, com 25 mercados (ou cidades, como tratado no texto original) e 100 produtos. No intervalo entre esses dois trabalhos existiram algumas propostas de heurísticas também, que até então não tinham sido verificadas pois na época não existiam métodos bons o bastante para encontrar as soluções (mínimos globais) das instâncias de teste, para fins de comparação.

No ano 2000, Laporte et al. apresentaram um novo algoritmo, baseado em uma técnica de *Branch and Cut* [1], que soluciona instâncias de tamanho ainda maiores, com até 200 mercados e 200 produtos. É importante ressaltar que o algoritmo apresentado por Laporte tem como foco grafos não-direcionados, e leva em consideração a quantidade de produtos disponível em cada mercado, ou seja, trabalha com a versão capacitada do problema.

Anos depois, em 2006, Ledesma e González apresentaram um novo algoritmo para solucionar o mesmo problema de forma ótima, com "*Solving the Asymmetric Traveling Purchaser Problem*" [4]. Como o título sugere, este algoritmo dá suporte para grafos direcionados, bem como leva em consideração a disponibilidade de cada produto em cada supermercado. Esta nova proposta consegue solucionar instâncias de tamanho semelhante a de trabalhos anteriores, algo em torno de 200 mercados e 200 produtos.

Apesar de todas essas soluções serem interessantes, pois entregam uma solução final ótima, o tempo de execução acaba por ser insatisfatório (como esperado em problemas NP-Difíceis). Todos os algoritmos citados anteriormente apresentam tempos de execução que em geral são considerados muito altos, para instâncias de tamanho moderado (cerca de 25 mercados, e 50 produtos), e instâncias maiores possuem tempos de execução ainda maiores (podendo chegar a horas, e até dias, no processamento de instâncias grandes [4]).

Heurísticas

A primeira heurística proposta partiu do próprio Ramesh, que no mesmo trabalho no qual cunhou o termo, apresentou uma heurística baseada em uma técnica que explorava mínimos locais, na seção "*A Near Neighbour Algorithm*" [2]. Não são apresentados tempos de execução na publicação original, porém a solução possui um intervalo de 20% a 98% de distanciamento do mínimo global.

A partir de então, diversos outros algoritmos foram aparecendo ao longo do tempo. Serão destacados apenas aqueles considerados os principais, ou que tiveram os melhores tempos de execução versus qualidade de resposta. Para cada algoritmo, no entanto, existem diversos tipos de variação. O foco vai ser em

apresentar a ideia geral de cada um dos algoritmos, sem aprofundar em suas variações.

Golden et al propôs um algoritmo iterativo, chamado de "*Generalized Savings Heuristic - GSH*" [5], no qual o algoritmo vai gradualmente adicionando novos mercados ao percurso, de acordo com o impacto que essa modificação vai trazer para a solução final do problema.

Ong e Hoon Liong, em 1982, publicou algumas alterações no algoritmo proposto por Golden, criando então um novo algoritmo intitulado "*Tour Reduction Heuristic*" [6]. Este novo algoritmo, ao invés de adicionar novos mercados ao trajeto, remove. Ao iniciar, é selecionado um percurso inicial, utilizando alguma heurística para o TSP, e a cada iteração é calculado o impacto que remover algum dos mercados trará para a solução final.

Outra proposta partiu de Pearn et al, em 1998, com um algoritmo chamado de "*Commodity-Adding Heuristic*" [7]. O algoritmo inicia com o mercado que possui o menor preço para o primeiro produto, e a cada iteração é analisado o próximo produto. Se adicionar um mercado ao trajeto para comprar um dos produtos reduz o custo total, esse mercado é adicionado. Quando todos os produtos forem comprados, o algoritmo para.

Laporte et al, mais recentemente, apresenta uma nova heurística, na mesma publicação em que apresenta seu algoritmo de *Branch and Cut*: "*Market adding heuristic* (MAH)" [1]. Este algoritmo se assemelha bastante ao GSH, de Golden, pois insere novos mercados à rota, a cada iteração. A cada iteração o mercado que possui o produto pelo maior preço é adicionado à rota, e então esta nova rota (agora com mais um mercado) é melhorada para buscar redução no valor dos produtos. Este algoritmo, na verdade, é utilizado para dar direção para uma solução ótima que foi desenvolvida no mesmo artigo, sendo utilizado algumas vezes como subrotina do algoritmo principal.

Até então, o algoritmo com o melhor custo-benefício entre tempo de execução e qualidade de resposta era uma variação do CAH (*Commodity-Adding Heuristic*), proposta por Boctor et al em 2003 [8].

Em 2005, no entanto, Jorge Riera-Ledesma e Juan José Salazar apresentam um novo algoritmo que realiza uma busca local, sendo batizado de "*Local-search*" [9], encontrando resultados bem próximos do ótimo, com um tempo de execução dentro do aceitável. A ideia principal do algoritmo é selecionar um trajeto utilizando

alguma heurística para o TSP, e então adicionar/remover mercados do trajeto, utilizando-se de mercados não selecionados para a rota, para reduzir o custo final. Este algoritmo, de acordo com as análises realizadas, entrega o melhor custo-benefício entre tempo de execução e qualidade de resposta atualmente.

DESENVOLVIMENTO

A proposta principal deste trabalho é apresentar uma implementação (protótipo) que solucione o TPP em tempo real, para um usuário final. Importante ressaltar que o algoritmo deve considerar a quantidade de produtos disponíveis em cada um dos mercados da lista, bem como situações na qual um produto P não está disponível para compra em nenhum mercado (produtos em falta).

Essa implementação se dá na forma de um algoritmo, que recebe entradas em um formato bem definido, e retorna uma resposta com uma solução para o problema. Repare que por se tratar de um problema NP-Difícil, o algoritmo irá retornar soluções viáveis (porém, muito provavelmente, não ótimas).

Nas seções subsequentes, serão definidos os tipos de entrada que a implementação deve aceitar, e quais tipos de dados devem ser retornados, para que seja possível atingir esse objetivo.

Dados de entrada

É importante que as entradas do algoritmo sejam as mais semelhantes possíveis com a realidade, portanto será necessário informar dados que são esperados na construção de uma lista de compras domésticas, como preço dos produtos em cada um dos mercados, e o valor de deslocamento entre cada um deles. Para isso, será necessário:

- Uma matriz **MxM**, onde cada entrada da matriz contém um valor **d** que representa o deslocamento de cada mercado para cada um dos outros mercados. Observe um exemplo de entrada na Figura 2;

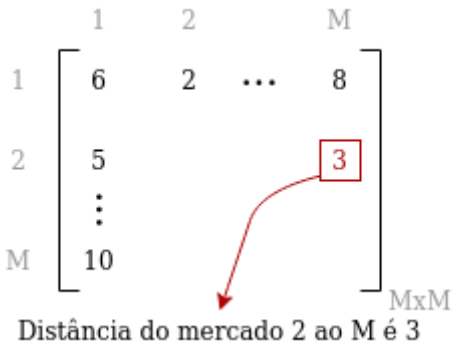

Figura 2 - Matriz de custos de deslocamento entre mercados

- Uma matriz **MxP**, onde cada entrada da matriz contém uma tupla **(c,q)** sendo **c** o custo e **q** a quantidade de cada produto em cada um dos mercados listados (estamos trabalhando com a versão capacitada do problema). Observe um exemplo na Figura 3; e

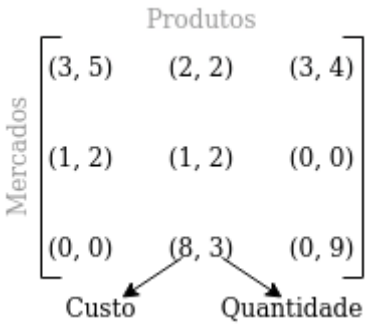

Figura 3 - Matriz de ofertas dos produtos em cada supermercado

- Uma lista **P**, onde cada entrada da lista contém uma tupla **(p,q)** sendo **p** um produto a ser comprado e **q** a quantidade deste produto. Observe um exemplo na Figura 4.

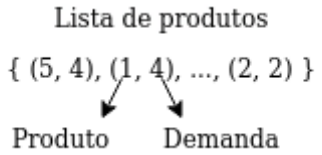

Figura 4 - Array de tuplas, representando a demanda

Dados de saída

Como resposta ao usuário, no entanto, será necessário um pouco mais de informação, visto que procuramos informar todo o procedimento necessário para maximizar os descontos (seja no preço dos produtos ou no deslocamento):

- Uma lista **F**, com a demanda faltante (produtos que não estavam disponíveis para compra em algum mercado na quantidade desejada);
- Uma lista **R**, que deve possuir os mercados que devem ser visitados, em ordem, do primeiro ao último juntamente com quais produtos comprar em cada um dos mercados;
- Um número inteiro positivo **CD,** que representa o custo de deslocamento;
- Um número inteiro positivo **CP**, que representa o custo total da compra dos produtos;
- Um número inteiro positivo **CT**, que representa o custo total da lista, sendo a soma do custo dos produtos e do deslocamento; e
- Uma lista **T**, que contem quais produtos vão ser comprados, por qual valor, e em qual mercado

Observações

1. Observe que o primeiro e o último elemento da lista **R** são o ponto de partida e o ponto final, respectivamente, e não mercados (podendo ser o mesmo ponto ou não);
2. Observe que é possível que um produto apareça tanto na lista **R** como na lista **F**, caso a quantidade disponível de algum dos produtos seja inferior à solicitada.

O algoritmo

O algoritmo selecionado como base para a implementação da proposta de heurística é o LSH, de Ledesma e Salazar [9]. A ideia principal deste algoritmo é selecionar uma rota inicial, contendo todos os mercados, e realizar uma sequência de remoções baseada em uma técnica muito utilizada por algoritmos que solucionam o TSP, chamada "*k-opt exchange*". Nesta técnica, **k** mercados são

substituídos por outro conjunto de mercados, para tentar escapar de mínimos locais, a cada iteração. Nas próximas seções, será descrita a implementação do algoritmo e algumas de suas características. O algoritmo termina quando admite-se, através de uma heurística, que não há mais como melhorar a solução atual (ótimo local).

k-opt exchange

Essa técnica é central para a execução do algoritmo, pois é com ela que é possível "escapar" dos mínimos locais que existem na solução do problema. Dado um conjunto M_1 de mercados que pertencem à rota, e um conjunto M_2 de mercados que não pertencem à rota, **k** mercados do conjunto $\mathbf{M_1}$ serão substituídos por **m** mercados do conjunto $\mathbf{M_2}$. Esse processo é repetido, e em cada execução, decrementa-se o número de mercados substituídos. Ao final, a versão que possui o menor custo é selecionada.

Repare que no caso do TPP, não é necessário que a quantidade de mercados adicionados seja igual à quantidade de mercados removidos, como de costume nos algoritmos que utilizam a mesma técnica na solução do TSP. A única obrigatoriedade é que a lista de compras deve continuar sendo satisfeita após a operação ser realizada. A imagem da Figura 5 descreve uma rota imaginária que parte de um ponto inicial (ponto mais à esquerda), passa por alguns mercados, e termina em um ponto final (ponto mais à direita).

Heurística para o TSP

Durante a execução do algoritmo será necessário encontrar soluções viáveis, por diversas vezes, do problema TSP. Toda vez que mercados são adicionados/removidos da rota, é necessário recalcular os custos da mesma. Esse algoritmo será executado por diversas vezes, tendo em vista que serão executadas diversas operações do tipo "***k-opt exchange***", por isso precisa ser relativamente eficiente. Importante notar que a operação de "k-opt exchange", no contexto do TPP, é um pouco diferente em relação ao TSP, visto que a troca de arestas acontece de forma indireta (com a troca de vértices).

A heurística escolhida nesta implementação foi a desenvolvida por Lin e Kernighan [10], disponível em http://webhotel4.ruc.dk/~keld/research/LKH/ (**LKH**). Trata-se de uma das heurísticas mais famosas para o problema do TSP e também é

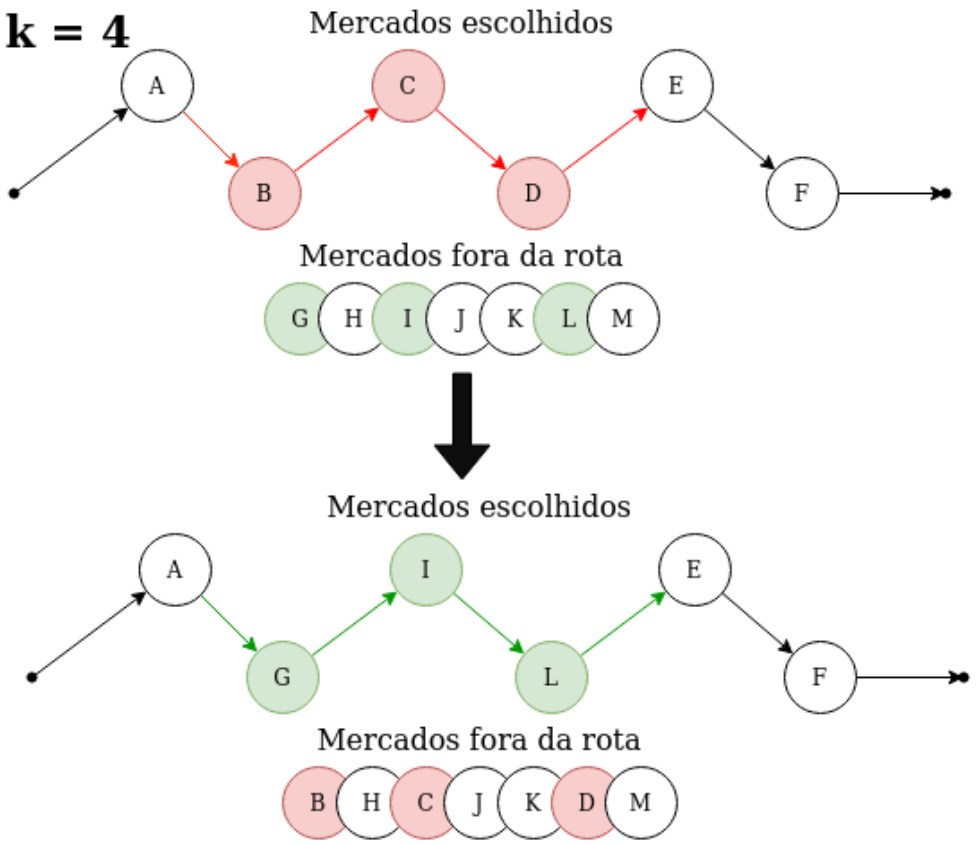

Figura 5 - 4-opt exchange, para o TPP, na prática (4 arestas alteradas)

é bastante eficiente no sentido de uso de memória e CPU. Utiliza-se da mesma técnica já comentada anteriormente, “k-opt exchange”, para otimizar a rota.

A ideia principal deste algoritmo é muito semelhante aquela descrita no item anterior, tendo como única diferença o fato de que todos os mercados devem obrigatoriamente pertencer à rota, ou seja, os mercados nunca são removidos, e sim realocados em novas posições (índice) na rota. Essa transformação costuma diminuir o valor final da rota. Abaixo, é ilustrado a aplicação de um ***2-opt exchange***, que nada mais é do que um ***k-opt exchange*** onde **k** é igual a **2**. Observe a Figura 6, que ilustra esse processo.

Contato com os autores originais do *Local search*

Durante a construção do algoritmo, foi realizada uma tentativa de contato direto com os autores do artigo utilizado como base (*Local search* [9]), para discutir alguns detalhes de implementação.

As únicas respostas que foram obtidas não puderam ser utilizadas no desenvolvimento do projeto, pois os autores informaram não possuir mais a imple-

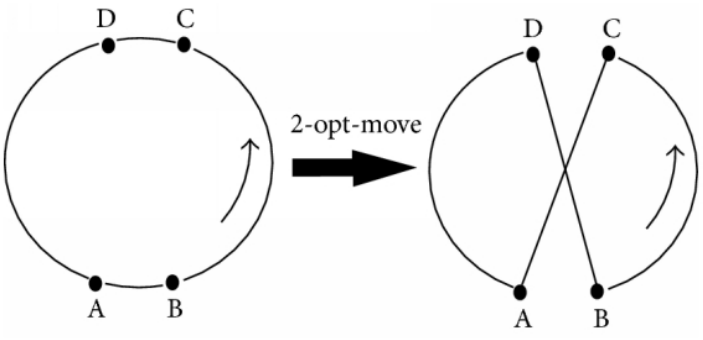

Figura 6 - 2-opt exchange, para o TSP, na prática

mentação original, nem meios de auxiliar na implementação que foi realizada neste TCC.

Isso foi um fator que prejudicou o plano de desenvolvimento, pois não foi possível ter acesso a alguns parâmetros de configuração que são muito importantes durante a execução. Esses parâmetros serão discutidos com mais profundidade nos tópicos a seguir, mas de forma geral controlam número de iterações e alcance dos passos utilizados para escapar de mínimos locais.

Para compensar por esse infortúnio, foi necessário realizar uma série de testes e ajustes, buscando encontrar, possivelmente, os valores originais utilizados na primeira implementação. Esses problemas serão discutidos com mais profundidade nos próximos tópicos, com apresentação de alguns dos testes realizados e os melhores valores obtidos.

Algoritmo em alto-nível

Esta seção, e as próximas, dedicam-se ao detalhamento do algoritmo apresentado por Ledesma e Salazar [1] (e implementado neste trabalho), com a apresentação da estrutura do algoritmo em alto nível. As especificidades de cada uma das funções internas, utilizadas e apresentadas no algoritmo abaixo, serão discutidas com a utilização de uma seção específica para cada.

A ideia principal do algoritmo está no laço de repetição externo, que controla o número de vezes que o algoritmo será executado, que por sua vez é controlado pela variável ***factor***. Essa variável controla a tolerância no aumento do valor do deslocamento, ao adicionar novos mercados na rota, através da função ***shaking***.

Essa variável é arbitrária, porém decidiu-se por utilizar o mesmo valor descrito no artigo original como sendo o melhor (detalhes serão discutidos nos tópicos a seguir).

No interior deste laço de repetição, existe outro laço responsável por realizar as diversas substituições de mercados (remoção e adição). Dentro dele, é calculada outra variável arbitrária chamada ***dropLen***, que define quantos mercados serão removidos/adicionados. O valor exato dessa variável não é mencionado no artigo original, portanto uma abordagem específica foi desenvolvida. Após realizar todas as remoções/adições necessárias, verifica-se se foi possível reduzir o valor total da melhor solução encontrada até então, saindo deste laço interno na primeira vez que uma redução no custo **não** for possível (enquanto for possível reduzir, reduza).

```
missingDemands := computeMissingDemand(input)
solution := firstFeasibleSolution(input)
curr := solution
factor := 35
for {
  for improved := true; improved; {
     dropLen := calculateDropLen(curr)
     s := consecutiveExchange(dropLen, curr)
     s = insertion(s)
     if s.totalCost < curr.totalCost
        curr = s
     else
        improved = false
  }
  if curr.totalCost < solution.totalCost
     solution = curr
  if factor == 0
     break
  curr_, shouldStop := shaking(factor, curr)
  if shouldStop
     break
  curr = curr_
  factor -= 1
}
return solution, missingDemand
```

Uma das condições de parada do algoritmo é decidida dentro da função ***shaking*** (a outra é controlada pela variável ***factor)***, que retorna um booleano informando se novos mercados foram adicionados ou não. Caso não haja nenhuma novidade na rota, o laço principal é interrompido e o algoritmo acaba. Caso algum mercado tenha sido, de fato, adicionado à rota, então a variável ***factor*** é decrementada, interrompendo o laço caso a mesma chegue a zero.

Nos tópicos a seguir, todas as funções descritas no algoritmo acima serão explicadas e discutidas com maior profundidade, bem como a razão pela qual cada um dos valores das variáveis arbitrárias foram escolhidos.

Sobre a função: computeMissingDemand

Esta função é responsável por identificar se existem algumas demandas, para alguns produtos, que não poderão ser satisfeitas mesmo que todos os mercados possíveis sejam selecionados. Essas demandas, que extrapolam a oferta total, são excluídas da entrada e armazenadas em uma estrutura de dados separada, para que seja possível informar ao usuário tudo que não será possível ser comprado.

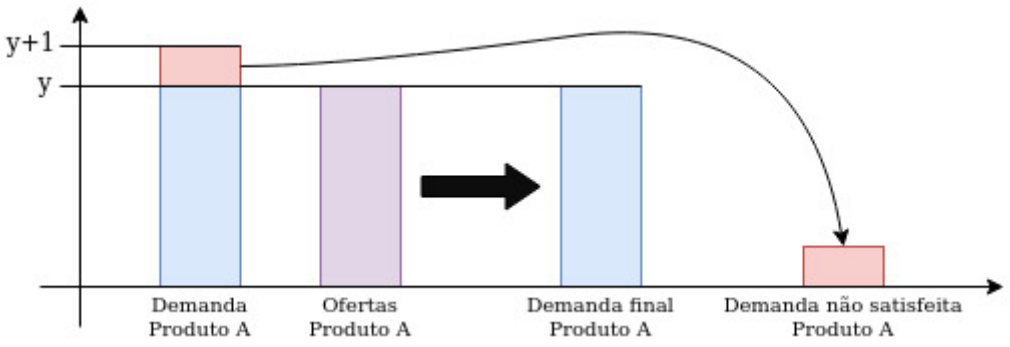

Figura 7 - Normalização das demandas

Esse processo é importante pois o algoritmo, durante a execução de seus estágios posteriores, é programado para sempre descartar qualquer solução que não satisfaça completamente todas as demandas, para todos os produtos presentes na lista.

Sobre a função: firstFeasibleSolution

Nessa etapa, é necessário construir uma rota inicial, para que o algoritmo principal possa executar usando-a como base. Para isso, todos os mercados são

considerados membros da rota, e a lista de produtos é completamente satisfeita com a compra de todos os produtos pelo menor preço possível.

Repare que nesse ponto, os mercados que não estão participando ativamente da solução do problema, são removidos da rota e colocados na lista de mercados não selecionados. Isso pode acontecer devido ao fato de que um mercado x pode não oferecer nenhum dos produtos pelo melhor preço.

Após selecionar quais mercados vão participar da primeira solução, é necessário construir uma rota melhorada que possibilite a viagem entre o ponto inicial, compra dos produtos nos mercados, e o ponto final. Para essa etapa, como já comentado anteriormente, é utilizado a heurística **LKH**, que soluciona problemas para o **TSP**.

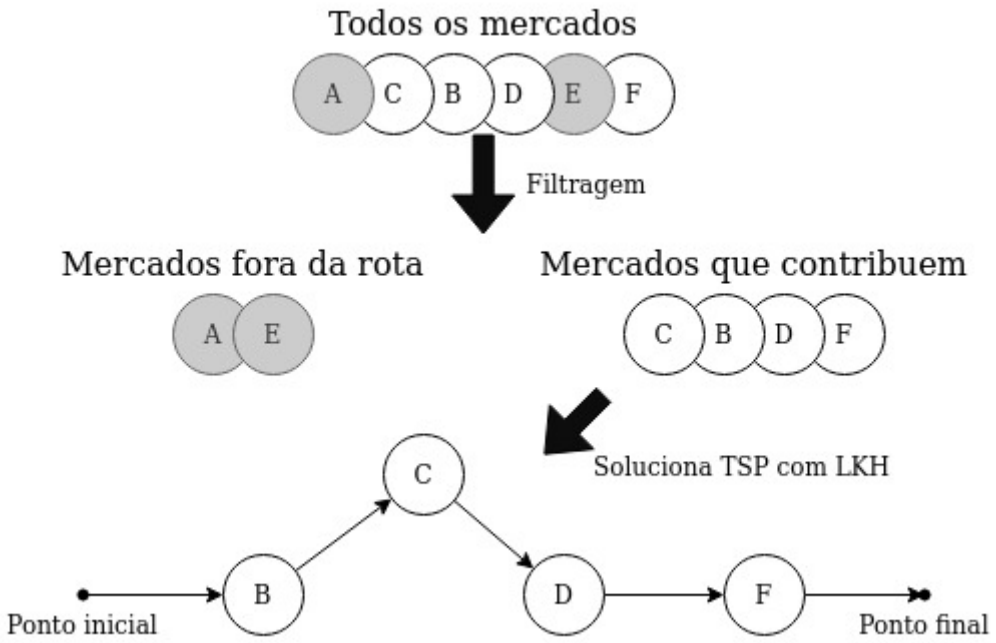

Figura 8 - Filtragem dos mercados que não contribuem

Sobre a função: calculateDropLen

Como já comentado anteriormente, essa função irá decidir quantos mercados vão ser removidos/adicionados da rota atual, na busca de melhores soluções. Esse valor é decidido de forma dinâmica, podendo ser alterado com o avanço da execução do algoritmo.

A regra que permite o comportamento dinâmico desse parâmetro não foi discutida no artigo original, e os autores do artigo não se recordam como foi feita

essa configuração (conforme elucidado anteriormente, o contato com os autores originais não solucionou todas as dúvidas referentes à implementação do algoritmo). A abordagem escolhida, portanto, foi utilizar o tamanho da rota atual (quantidade de mercados selecionados), multiplicada por um fator arbitrário. Após a execução de diversos testes, verificou-se que **39%** parece ser o melhor custo-benefício (sempre arredondando para baixo).

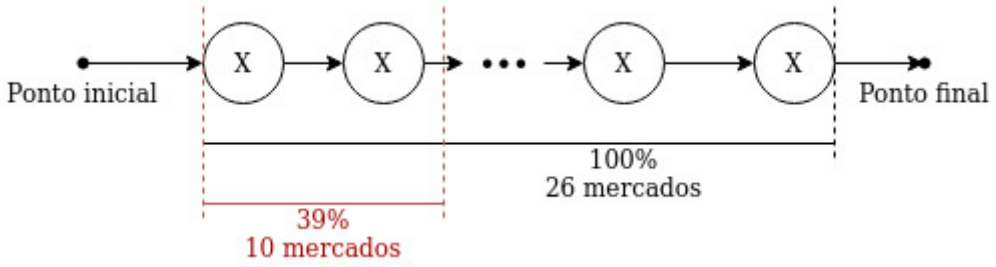

Figura 9 - Significado da variável *dropLen*

No exemplo acima (Figura 9), imaginando-se uma rota que contém 26 mercados, o número de mercados a serem removidos/adicionados pelas próximas etapas do algoritmo é definido como sendo 10, já considerando o arredondamento para baixo. Nessa etapa, não importa quais mercados serão substituídos, apenas a quantidade.

É possível que existam alguns casos específicos, como a quantidade de mercados ser muito pequena, ou muito grande. Em suma, o valor de ***dropLen*** deve ser zero para tamanho de rota igual a 2 ou menos, e nunca deve ultrapassar 25 mesmo em rotas de tamanho muito grande. Note que ao aumentar o tamanho de ***dropLen***, o tempo de execução do algoritmo tende a aumentar consideravelmente, como será possível observar nas próximas seções.

Sobre a função: consecutiveExchange

Essa é uma das principais funções, pois contém dentro de si mais dois laços de repetição internos que são muito importantes na execução do algoritmo. Dentro dessa função, o laço de repetição mais externo irá executar as remoções/adições em sequência, reduzindo o valor da variável ***dropLen*** em 1 (um) a cada iteração, enquanto que o laço interno é responsável por testar diferentes combinações da remoção de tamanho ***dropLen*** através da rota.

Na prática, essas operações vão descrever uma espécie de “caminhada de ***dropLen*** remoções” na rota, testando todas as combinações de remoção sequencial possíveis, com ***dropLen*** sendo decrementado até que chegue a zero. A Figura 10 a seguir descreve esse processo de maneira visual:

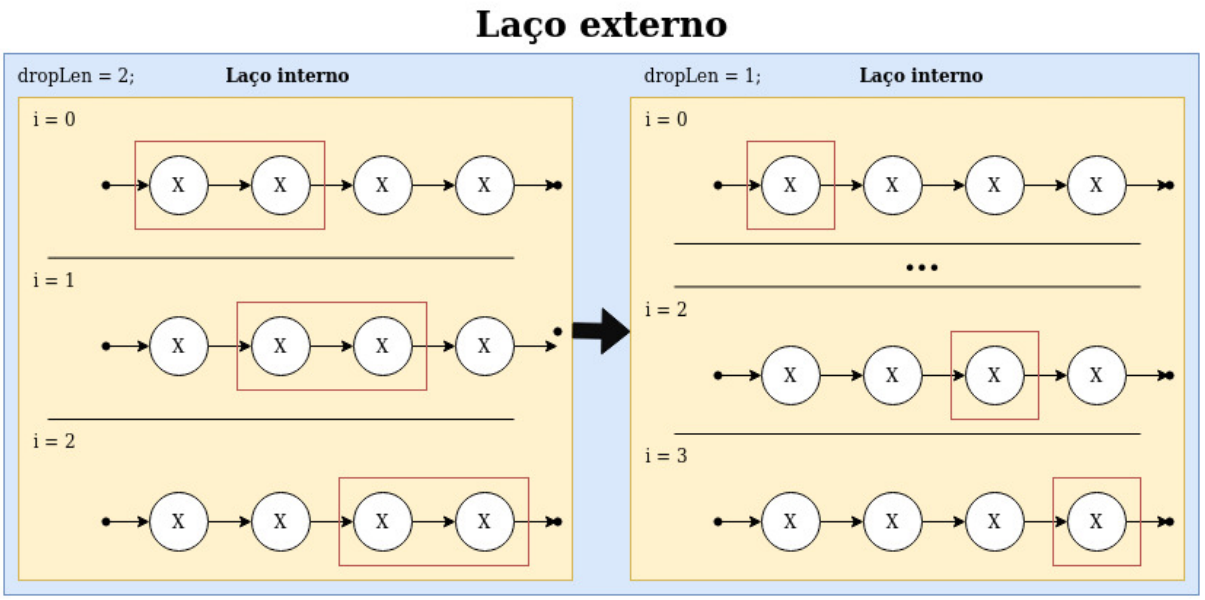

Figura 10 - Laços de repetição externo/interno

A melhor solução encontrada, no laço interno, é selecionada e retornada para o laço externo após ser aplicado o algoritmo **LKH**. Esse, por sua vez, realiza duas checagens: viabilidade da solução encontrada no laço interno, e redução do valor total em comparação com a melhor solução encontrada até então, pelo laço externo.

Caso a solução do laço interno não seja viável (alguma demanda de algum produto não pôde ser satisfeita), é necessário adicionar alguns mercados até que essa solução torne-se viável, para então poder ser comparada com a melhor solução atual do laço externo. Nesse processo, somente os mercados que não pertenciam à rota original (antes das remoções) são candidatos, e sobre essa rota o **LKH** também é aplicado, novamente. No artigo original observou-se que essa abordagem possibilita melhores resultados em contraste com a outra possibilidade, na qual todos os mercados fora da rota são candidatos. Observe a Figura 11.

Ao final da execução dessa função (**consecutiveExchange**), uma nova solução candidata será devolvida para o laço interno global. Antes de comparar essa nova solução com a melhor solução encontrada, alguns outros processos, descritos no próximo tópico, são aplicados nessa nova solução.

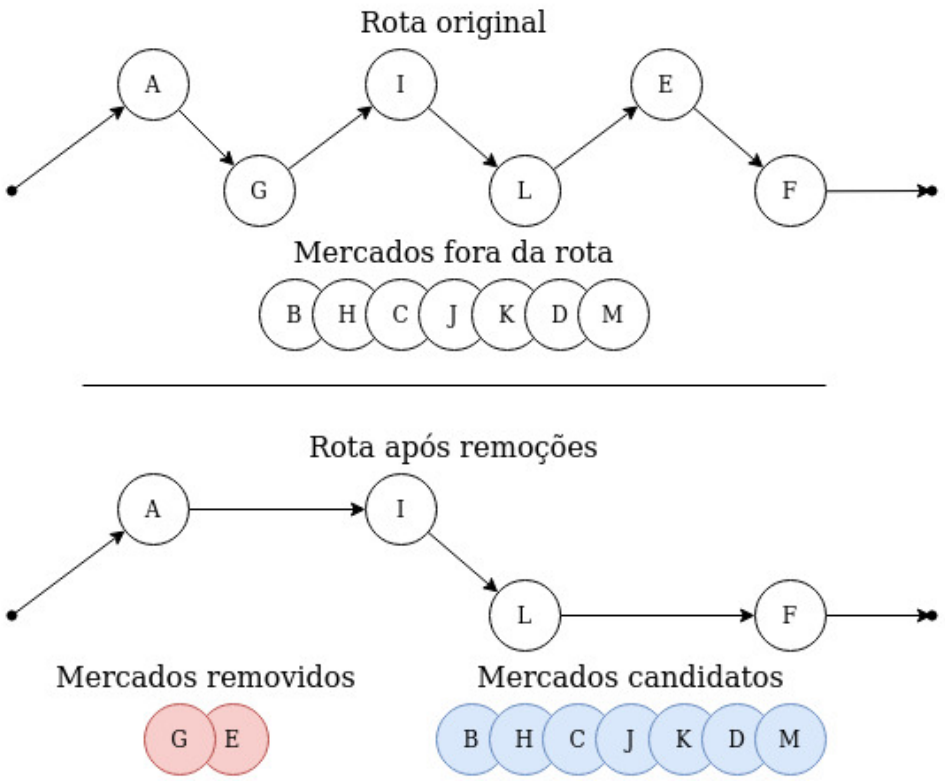

Figura 11 - Processo de escolha de novos mercados

Sobre a função: insertion

Essa é uma função muito simples, que tem como responsabilidade a tarefa de inserir novos mercados na rota devolvida pela função ***consecutiveExchange***, caso estes mercados propiciem uma diminuição no valor final da solução. Esse processo pode ser visualizado através da Figura 8.

Repare que diferente da função ***consecutiveExchange***, não há restrições em quais mercados podem ou não ser adicionados. Todos os mercados não pertencentes à rota, são candidatos. Por fim, sobre essa nova rota o **LKH** é mais uma vez aplicado.

Sobre a função: shaking

Esta é uma função de suporte ao algoritmo principal, visto que não visa alterar de forma direta a composição da melhor solução encontrada até então, mas busca perturbar a mesma, tendo como objetivo “escapar” de alguns possíveis mínimos locais. Observe um exemplo gráfico na Figura 12.

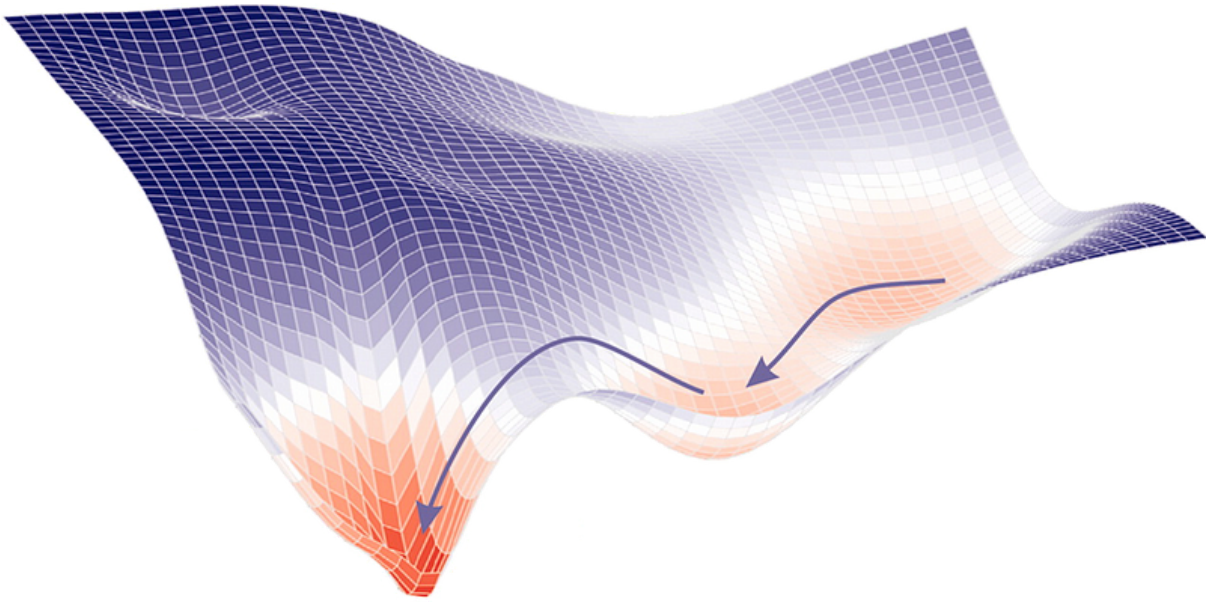

Figura 12 - Fuga de mínimos locais

O grau dessa perturbação é controlada através da variável ***factor***, que indica o fator de tolerância no aumento do custo ao inserir novos mercados na rota. Repare que todas as vezes que algum novo e bom mercado é adicionado à rota, o custo dos produtos nunca pode aumentar, e o custo de deslocamento sempre deve aumentar ou permanecer igual. Portanto esse fator controla, na prática, o quanto de aumento no deslocamento é tolerado, ao adicionar novos mercados. É importante notar que o **LKH não** é aplicado na rota resultante dessa etapa.

A cada iteração do laço principal, esse fator vai se reduzindo, de forma que em determinado momento nenhum aumento no valor de deslocamento é tolerado, e portanto nenhum mercado é adicionado. Quando a função ***shaking*** não consegue adicionar nenhum novo mercado à rota, então a execução do algoritmo é interrompida de forma global.

A outra opção, no entanto, é também possível de acontecer. Caso a variável ***factor*** ainda esteja com um valor próximo do inicial (alto), ainda é possível que nenhum novo mercado seja adicionado, pois o custo excederia a tolerância, e portanto o algoritmo será interrompido na primeira iteração.

Essa técnica é muito conhecida na literatura como *Simulated Annealing*, pois é permitido que o algoritmo utilize soluções piores ao longo da execução, para que seja mais fácil escapar de mínimos locais e aproximar-se do mínimo global. Isso é realizado de forma controlada, e a tolerância é reduzida a cada iteração.

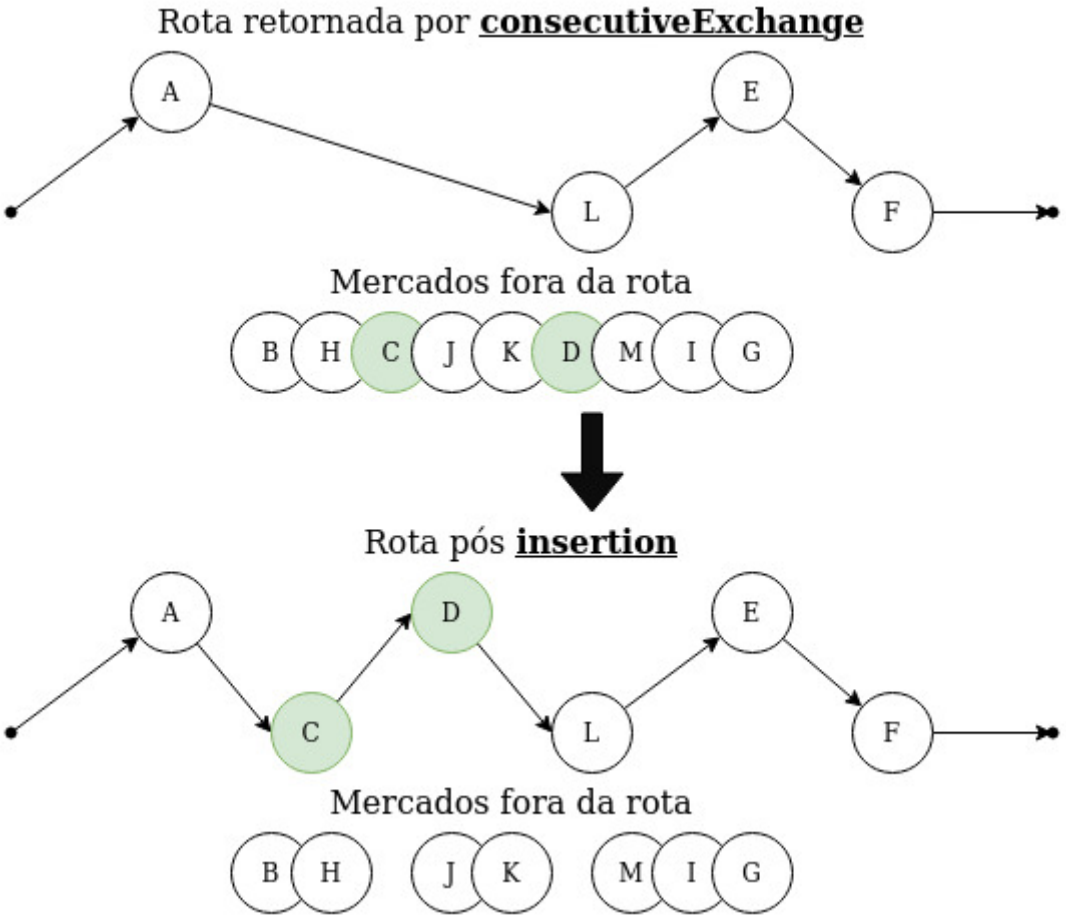

Figura 13 - Inserção de novos mercados na rota

EXPERIMENTOS

Para realizar testes com o algoritmo, após o término da implementação, foi necessário encontrar algumas instâncias de problemas do tipo TPP, para que pudessem ser resolvidas e comparadas com os resultados ótimos obtidos pela comunidade acadêmica.

As instâncias que foram utilizadas nos testes deste TCC são as mesmas utilizadas pelo artigo que serviu como base (*Local Search [9]*). Essas instâncias estão disponíveis para download através do link: https://jriera.webs.ull.es/TPP.htm. Repare que existem diversas categorias de instâncias, que cobrem os 4 possíveis casos: instâncias simétricas e não capacitadas, simétricas capacitadas, assimétricas não capacitadas e assimétricas capacitadas.

O foco deste trabalho se encontra nas instâncias que mais se assemelham a problemas de rota no mundo real: assimétricas capacitadas. Infelizmente, para esta categoria de problema, as instâncias disponibilizadas não possuem solução ótima e nem documentação. Por esse motivo optou-se pelo uso das instâncias simétricas capacitadas, que estão devidamente documentadas e possuem solução para a maior parte das instâncias.

Problemas simétricos são um pouco mais fáceis de resolver, tendo em vista que todos os custos de deslocamento entre mercados são iguais tanto saindo de A para B quanto de B para A. No mundo real essa regra claramente não se mantém verdadeira, pois é comum que as regras de trânsito acabem definindo que a distância saindo de A até B seja diferente de B até A, em alguns casos.

Estas instâncias foram todas geradas aleatoriamente. O número de mercados e a quantidade de produtos são definidos de forma arbitrária, enquanto que a demanda e oferta dos produtos é feita da seguinte forma:

> *Para cada produto e cada mercado, a oferta do produto em cada mercado é definida aleatoriamente no intervalo [1, 15], e a demanda é calculada da seguinte forma:*
>
> - *Seleciona-se a oferta mais alta, disponível em algum dos mercados, e multiplica-se esse valor por λ (0 <= λ <= 1). Vamos chamar esse valor de "**X**"*
> - *Faz-se o somatório da oferta que cada mercado possui para o determinado produto, e multiplica-se o valor final do somatório por (1 - λ). Vamos chamar esse valor de "**Y**"*
> - *Soma-se os dois valores resultantes anteriores, e aplica-se uma função de arredondamento para cima: **demanda = ceil(X + Y)***

Note que quanto maior o valor de λ, menor o tamanho do trajeto, pois com λ=1 a instância se torna um TPP irrestrito (oferta de produtos infinita), e com λ=0 se torna um TSP (obrigatória a passagem por todos os mercados). Foram testadas **487** instâncias com as seguintes características:

- 50, 100, 150 e 200 mercados
- 50, 100, 150 e 200 produtos
- Com λ (lambda) igual a 0.1, 0.5, 0.7, 0.8, 0.9, 0.95 e 0.99

Todos os arquivos das instâncias foram gerados seguindo o padrão utilizado pela biblioteca TPPLIB, que nada mais é do que uma extensão do padrão utilizado pela biblioteca TSPLIB. Esta biblioteca define uma série de seções que seus arquivos devem ter, de modo a manter a compatibilidade na entrada dos dados a serem testados por algoritmos de resolução do TSP, e TPP (após a extensão).

O padrão da TSPLIB define um arquivo que pode conter diversas seções, para diferentes variantes do problema, mas este trabalho possui interesse apenas nas seguintes seções, que estão presentes nos arquivos de teste utilizados:

- `NAME`: define um nome para a instância do problema
- `TYPE`: define o tipo de problema
- `COMMENT`: algum comentário, caso necessário
- `DIMENSION`: quantidade de vértices para o TSP
- `EDGE_WEIGHT_TYPE`: método utilizado para calcular a distância entre os vértices. Nos arquivos escolhidos: distância euclidiana no plano 2D
- `NODE_COORD_SECTION`: coordenadas que representam a posição de cada um dos mercados, no plano 2D

Já o padrão TPPLIB define duas novas seções:

- `DEMAND_SECTION`: informa a qual a demanda desejada de cada um dos produtos
- `OFFER_SECTION`: informa qual mercado oferece qual produto, a quantidade dos produtos ofertados, e seu preço

A seguir é possível observar um arquivo de exemplo, contendo uma instância pequena de um problema TPP:

```
NAME                : SMALL TPP
TYPE                : TPP
COMMENT             : EUCLIDEAN SMALL CTPP
DIMENSION           : 5
EDGE_WEIGHT_TYPE    : EUC_2D
NODE_COORD_SECTION :
1 445 258
2 155 958
3 383 849
4 406 523
5 463 414
DEMAND_SECTION :
5
1  2
2  3
3  7
4 11
5 20
OFFER_SECTION :
1 0
2 2   3 8  7   2 10 12
3 3   5 9  3   4  9 10   1 1 3
4 2   5 3 12   1  1  7
5 2   5 8 14   4  4  3
EOF
```

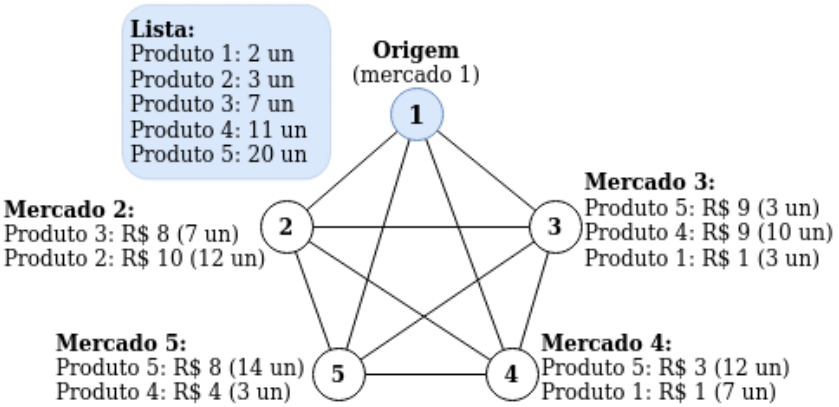

Figura 14 - Visualização do arquivo de exemplo acima

(o posicionamento dos vértices no plano 2D é apenas ilustrativo)

Dito isso, as 487 instâncias utilizadas, aproximadamente, a documentação completa destas instâncias, e a documentação completa sobre os formatos de arquivo, estão disponíveis em:

- ZIP: http://webpages.ull.es/users/jriera/TPPLIB/SymmetricEuclideanCapLaporteRieraSalazar.zip
- Soluções (mínimos globais): https://jriera.webs.ull.es/TPPLIB/class4.pdf
- Documentação sobre os arquivos TPPLIB: https://jriera.webs.ull.es/TPPLIB/TPPLIBFormat.htm
- Documentação sobre como interpretar as soluções: https://jriera.webs.ull.es/TPPLIB/Description.pdf

Todos os testes foram realizados em um computador pessoal, com as seguintes configurações:

- Processador Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz;
- 32GB de memória RAM;
- Linux Mint 19.3 (Tricia);
- Linux Kernel 5.0.0-32-generic; e
- Go build tool 1.19.4.

Para execução dos testes, utilizou-se o máximo de paralelismo possível, sendo que cada instância de teste executou em sua própria *go-routine* por todo o seu ciclo de vida. O tempo de execução de todas as instâncias de teste foi variado, dependendo dos parâmetros escolhidos, sendo que durante esse tempo o poder de processamento do computador foi deixado todo à disposição da execução do algoritmo.

Resultados

O método utilizado para encontrar os melhores parâmetros para a execução das instâncias foi o de tentativa e erro. Ao longo da implementação do algoritmo, foi-se percebendo uma afinidade maior de determinados parâmetros, com determinadas instâncias de teste. O método para cálculo do *gap* de cada uma das

respostas, em relação ao mínimo global, é realizado através do cálculo da porcentagem na qual o valor ficou acima do mínimo. Se o mínimo global for 100, por exemplo, e a resposta foi 120, temos um *gap* de 20%.

O parâmetro ***factor*** está diretamente ligado com o tempo de execução do algoritmo. Caso esse valor seja alto demais, o algoritmo irá executar por muito tempo, sem a garantia de encontrar valores melhores. Com um valor muito baixo, a perturbação causada nas soluções parciais vai ser muito baixa, tornando-se irrelevante. No caso deste parâmetro, optou-se por utilizar a mesma métrica definida no artigo original (*Local Search* [9]): 35%. Neste caso, na primeira iteração, é admitido um aumento de até 35% no custo, ao perturbar a solução parcial adicionando novos mercados.

O parâmetro ***dropLen*** é um pouco mais complicado de realizar os ajustes, pois é muito mais sensível e desencadeia em uma série de alterações nas soluções encontradas. A princípio, tentou-se utilizar 50% do tamanho da solução atual, porém esse valor demonstrou-se ser muito ineficaz na questão de tempo de execução. Valores muito pequenos como 10% e 20% não demonstraram um desempenho satisfatório na questão de redução de custos. Com base nos experimentos realizados, o melhor intervalo, portanto, parece ser entre 35% e 45% do tamanho da rota atual.

Note que esses parâmetros estão intimamente ligados com o formato das instâncias de teste, e portanto a seleção destes parâmetros está intimamente ligada com o objetivo que está sendo buscado. No caso deste TCC, a busca é pelo equilíbrio entre o tempo de execução e a redução nos custos. Nas instâncias de teste que foram executadas aqui, o valor com melhor custo-benefício ficou próximo de ~39%, e por isso esse foi o valor escolhido, mas isso pode se alterar de acordo com as instâncias de teste ou objetivo.

A seguir serão descritos alguns gráficos (na forma de figuras) e uma tabela, contendo os valores dos experimentos realizados, juntamente com alguns parágrafos que buscam analisar esses resultados.

Após analisar atentamente os dados coletados na Tabela 1, é possível observar que o melhor valor para ***dropLen***, considerando-se apenas o custo, seria algo em torno de 34% (para esse conjunto de instâncias). Como o objetivo é obter um algoritmo que produza uma resposta em um tempo satisfatório, também, optou-se por escolher 39%, visto que este oferece um tempo de execução menor, e

um custo semelhante. Isso é visualizado com mais facilidade através do gráfico de linha, representado pela Figura 15.

dropLen	***gap* mínimo**	***gap* médio**	***gap* máximo**	**Desvio padrão do *gap***	**Tempo**
20%	0.03%	3.33%	119.75%	9.3%	7h22m
24%	0.03%	3.29%	119.75%	9.1%	7h54m
28%	0.03%	3.09%	119.75%	7.7%	9h23m
30%	0.03%	3.05%	119.75%	7.7%	9h31m
32%	0.03%	3.05%	119.75%	7.8%	8h33m
34%	0.03%	2.90%	119.75%	7.6%	8h49m
36%	0.03%	2.91%	119.75%	7.5%	8h18m
38%	0.03%	2.92%	119.75%	7.6%	7h11m
39%	0.03%	2.96%	119.75%	7.6%	6h54m
40%	0.03%	2.99%	119.75%	7.6%	9h20m
42%	0.03%	2.95%	119.75%	7.6%	8h34m
44%	0.03%	2.95%	119.75%	7.6%	11h14m
46%	0.03%	2.94%	119.75%	7.6%	14h36m
48%	0.03%	2.98%	119.75%	7.6%	10h29m
50%	0.03%	2.97%	119.75%	7.6%	13h37m
54%	0.03%	2.97%	119.75%	7.6%	13h58m
58%	0.03%	2.99%	119.75%	7.6%	16h42m

Tabela 1 - Custo e tempo de execução para cada valor de *dropLen*

O gráfico da Figura 15 é uma sobreposição dos dados referentes ao *gap* médio (azul), para cada valor de ***dropLen***, juntamente com o tempo gasto (vermelho) para calcular todas as 487 instâncias de teste. No eixo **y** da esquerda temos a porcentagem de *gap* médio, e no eixo **y** da direita temos o tempo de execução. Foram desenhadas, também, duas linhas de tendência: uma para o *gap* médio e outra para o tempo de execução.

É possível observar que o tempo de execução tende a aumentar conforme o valor de ***dropLen*** é aumentado, através da linha de tendência vermelha. Isso é esperado, visto que esse parâmetro controla a quantidade de vezes que alguns laços de repetição vão executar, como comentado nas seções anteriores. O inverso, no entanto, também pode ocorrer (redução no tempo de execução), mas em geral a tendência é de aumento.

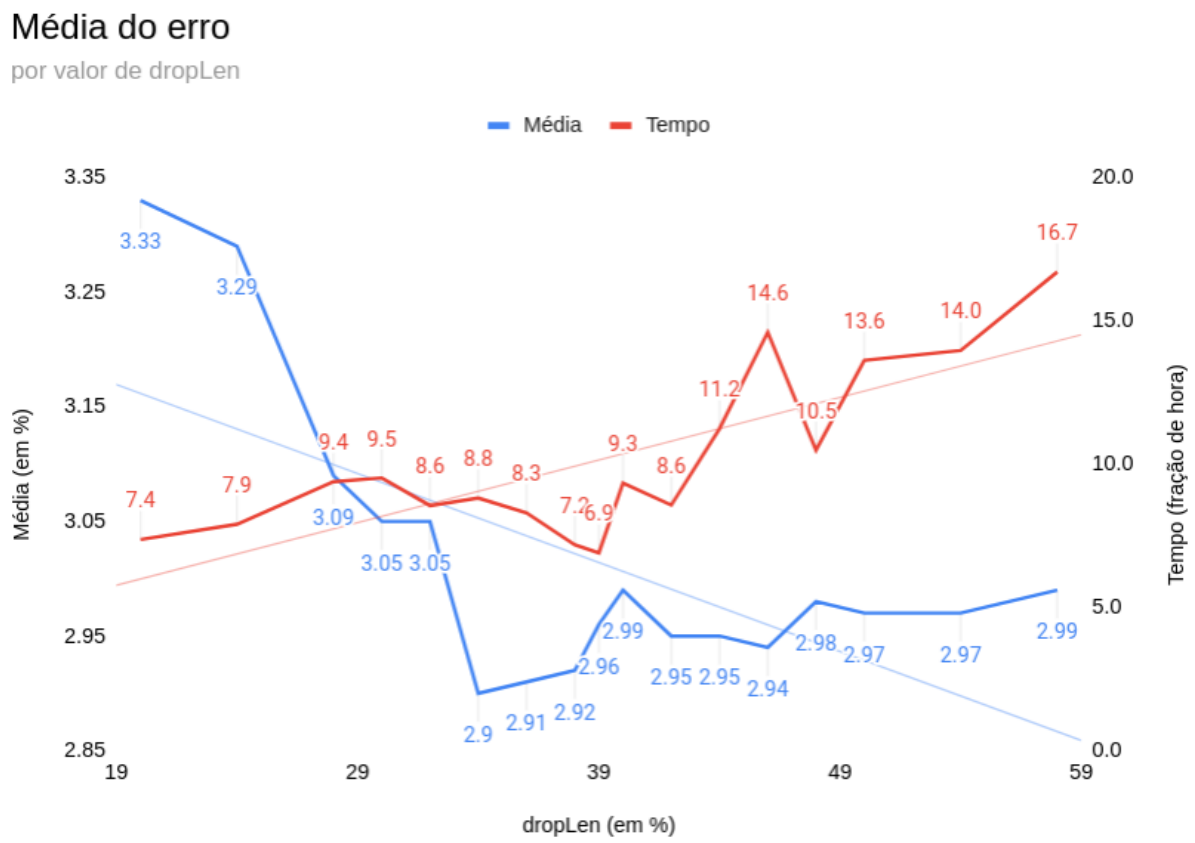

Figura 15 - Gráfico de linha, com linhas de tendência

O *gap* médio atinge o mínimo quando ***dropLen*** é 34%, porém o tempo de execução ainda pode ser melhorado. Somente perto dos 39% que o tempo de execução reduz, ainda sem muito prejuízo na qualidade do *gap* médio. Esse equilíbrio é o motivo da escolha de 39% como o melhor valor, neste trabalho.

Através do gráfico representado pela Figura 16, pode-se observar que mesmo com a aplicação de diferentes valores para ***dropLen***, os dados estatísticos permanecem muito próximos uns dos outros. Em situações na qual o *gap* médio se altera, os valores mínimos, máximos e desvio padrão permanecem o mesmo, indicando apenas um deslocamento do *gap* em relação ao valor ótimo, em

praticamente todas as instâncias (visto que a dispersão dos valores continua a mesma). Os *outliers* foram removidos pois tratam-se sempre das mesmas instâncias (que serão comentadas nas seções seguintes), e isso melhora significativamente a visualização do gráfico.

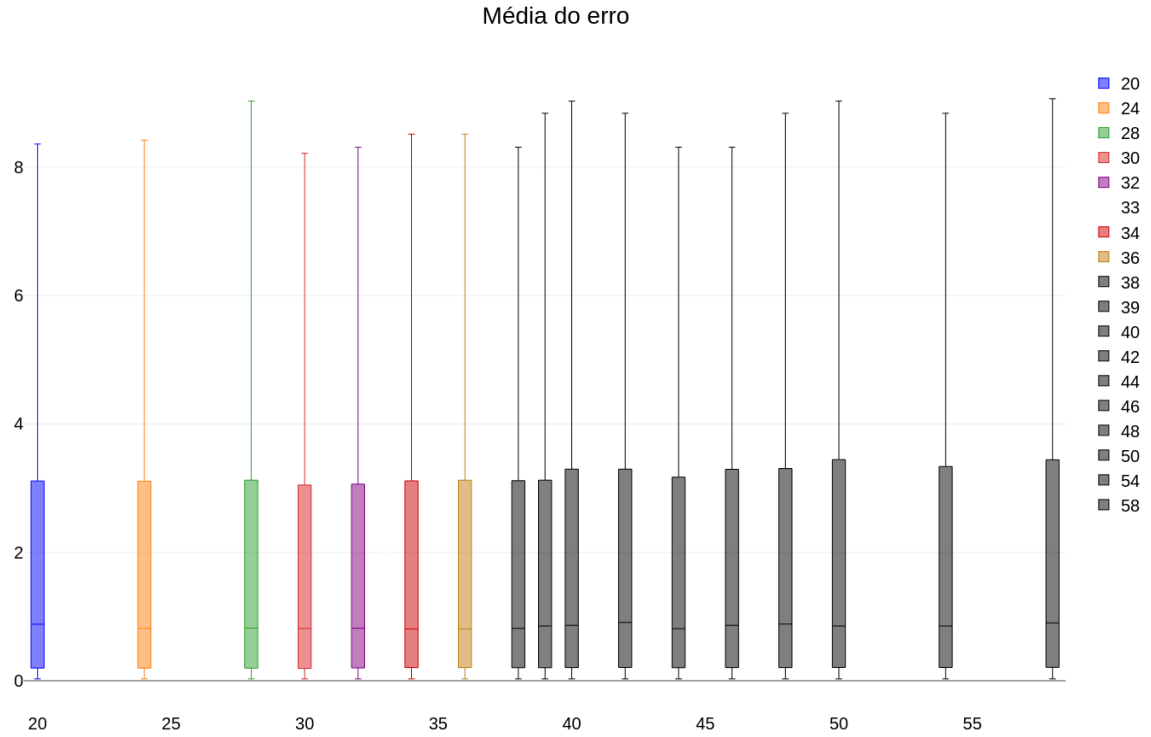

Figura 16 - Gráfico em formato boxplot, contendo a média dos *gaps* (valores em %), com outliers removidos para melhor visualização

Observações importantes

Em alguns casos isolados, a redução de ***dropLen*** pode aumentar o tempo de execução, algo que pode parecer uma surpresa em um primeiro momento. Isso ocorre devido ao fato de que diferentes valores desse parâmetro vão causar a exploração de ramos completamente diferentes na árvore de possibilidades de cada um dos problemas. É completamente possível, e esperado, que em alguns casos a redução deste valor provoque alguns saltos (na tentativa de escapar de mínimos locais), que caiam em “vales” que sejam mais demorados para otimizar do que outros.

Em suma, aumentar o valor de ***dropLen*** tende a aumentar o tempo de execução, mas essa afirmação pode não ser verdadeira em alguns casos

específicos, como esperado. Diminuir esse valor também tende a reduzir o tempo de execução, mas isso também não se mantém verdade em todos casos.

A qualidade da solução obtida possui uma tendência de melhora conforme ***dropLen*** é aumentado, pois o número de iterações aumenta juntamente com esse parâmetro. O acontece, inversamente, para o tempo de execução, que tende a aumentar conforme ***dropLen*** aumenta.

Instâncias “fáceis”

Algumas instâncias de problemas são ligeiramente mais fáceis de serem resolvidas, pois reduzem-se ao próprio TSP. Caso todos os mercados disponíveis precisem necessariamente fazer parte da rota, então o problema torna-se um TSP, sendo necessário apenas comprar os produtos pelo melhor valor.

Mercados	Produtos	λ (lambda)	Mercados na rota final	*Gap*
50	200	0.1	50/50	0.02832%
150	200	0.1	150/150	0.02807%
50	200	0.1	50/50	0.02794%
150	200	0.1	150/150	0.02752%
50	200	0.1	50/50	0.02745%

Tabela 2 - Melhores instâncias de teste (*gap* se mantém igual para todos *dropLen*)

Observe na Tabela 2 que as instâncias que se mantêm como as melhores soluções, não por acaso, são aquelas que obrigam a presença de todos os mercados disponíveis na rota final (coluna “*Mercados na rota final*”). Esse comportamento se deve ao fato do fator lambda ser próximo de zero (o valor mais baixo usado para lambda, na verdade), que quase transforma essa instância em um TSP.

Por tratarem-se de um problemas quase idênticos ao TSP, o *gap* acima pode se tratar de erro numérico, como será comentado na seção “**Erro numérico**”. Essas instâncias são solucionadas muito mais rapidamente e com muito mais precisão, por focarem apenas na parte TSP do problema. Infelizmente é difícil encontrar situações

como essa no mundo real, na solução de uma lista de compras, pois a situação inversa é mais comum: poucos mercados são necessários para completar uma lista.

Instâncias difíceis, ou “problemas perversos”

O método utilizado para solucionar as instâncias de teste, como já comentado anteriormente, é uma generalização da técnica ***k-opt***, usada por diversos algoritmos que solucionam o TSP. Essa técnica possui os melhores resultados, atualmente, porém por tratar-se de uma heurística também contém algumas falhas.

Papadimitriou e Steiglitz [11] demonstraram, em 1978, que existe uma classe de problemas TSP gerados aleatoriamente no qual algoritmos de busca local (a técnica ***k-opt*** é uma busca local), são extremamente ineficazes. Essa classe de problemas foi chamada de “problemas perversos”, na descrição original.

No conjunto de testes realizados no presente trabalho, é possível notar que algumas instâncias sempre são solucionadas com uma alta taxa de *gap*. Abaixo são listadas as 5 instâncias que obtiveram os piores resultados, permanecendo como pior resultado em todas as iterações:

Mercados	**Produtos**	**λ (lambda)**	**Mercados na rota final**	***Gap***
150	50	0.99	55/150	119.75%
50	150	0.99	43/50	74.96%
50	200	0.99	36/50	71.79%
150	200	0.99	84/150	70.78%
50	200	0.99	38/50	70.65%

Tabela 3 - Piores instâncias de teste (*gap* se mantém igual para todos *dropLen*)

Repare na Tabela 3 que todas as instâncias que obtiveram os piores resultados possuem um lambda igual a **0.99**. Isso acontece pelo fato de que a dificuldade em solucionar os problemas cresce conforme o λ (lambda), pois como já comentado anteriormente, com um valor alto para lambda é necessário otimizar os dois aspectos do problema: a escolha dos produtos e a escolha dos mercados, ao mesmo tempo.

Note que em algumas situações, quando a demanda é igual (ou superior) a quantidade global ofertada, todos os mercados obrigatoriamente deverão pertencer à rota (na prática, um TSP). O inverso também pode ocorrer, caso apenas 1 mercado ofereça toda a demanda. Isso é controlado através do parâmetro λ (lambda), já discutido anteriormente. No fim das contas, todas as instâncias que envolvem a decisão de ambos fatores tendem a ser mais difíceis de serem resolvidas. Essa classe de "problemas perversos", no entanto, tende a não se apresentar em problemas do mundo real, na maioria das vezes.

Erro numérico

O posicionamento dos mercados no plano cartesiano 2D é feito de forma aleatória, as distâncias são calculadas de acordo com a distância Euclidiana, e ao mesmo tempo todos os valores de custo (deslocamento e preço de produto) são definidos como valores inteiros.

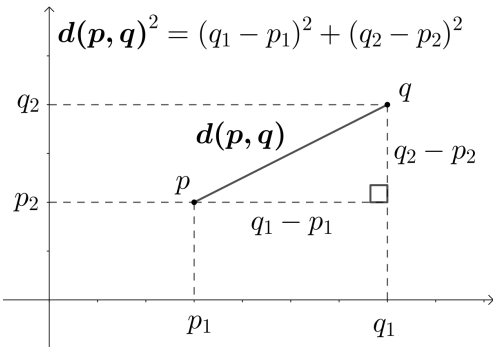

Figura 17 - Exemplo de distância euclidiana entre dois pontos

A configuração descrita acima acaba por provocar uma taxa de erro numérico, que se acumula ao longo das iterações e execução do algoritmo. Como as distâncias são euclidianas, e os mercados são distribuídos aleatoriamente no plano, é inevitável que a maioria das distâncias sejam números reais. Observe a Figura 17 para visualizar o cálculo da distância entre dois pontos.

No artigo original que descreve o algoritmo usado como base para este trabalho [1], nenhuma informação sobre a normalização desses erros é comentada. Nesta implementação, optou-se por um arredondamento para baixo (truncamento), ao calcular a matriz de distâncias entre as coordenadas. Isso pode ser uma das razões pela qual algumas instâncias possuem um *gap* pequeno e constante, para todos os valores de ***dropLen*** utilizados.

CONCLUSÃO

No presente trabalho foi analisado o contexto histórico da comunidade acadêmica, no que diz respeito ao TPP. Vários autores diferentes já se dedicaram ao estudo deste problema, pois sua solução é de extrema importância para uma série de setores da sociedade e indústria.

Por se tratar de um algoritmo NP-Difícil, com certa semelhança com o TSP, solucionar esse problema também significa avançar na solução de diversos outros problemas semelhantes (como o próprio TSP).

A alta complexidade dos algoritmos envolvidos acabam por limitar bastante a aplicação dos mesmos, pois qualquer aplicação que venha a interagir com usuários finais, hoje em dia, necessita de tempos de resposta rápidos. Um dos algoritmos mais promissores para implementação em um sistema de tempo real é a proposta de Ledesma e Salazar [9].

O presente trabalho apresentou uma implementação do mesmo algoritmo, com algumas diferenças na seleção dos parâmetros de configuração original, bem como um estudo de caso em cima dos resultados obtidos. O algoritmo demonstra ser extremamente flexível, podendo ser aplicado com diferentes objetivos, apenas com a alteração dos seus parâmetros.

Próximos trabalhos, relacionados com esse mesmo problema, podem incluir estudos relacionados com a otimização dos laços de repetição mais utilizados por este algoritmo, em busca de diminuição no tempo de execução e/ou melhoria nas soluções. É possível, por exemplo, a utilização de técnicas mais recentes de computação para implementar algumas dessas otimizações, como inteligência artificial ou algoritmos genéticos.

REFERÊNCIAS

[1] Laporte, Gilbert; Jorge Riera-Ledesma; e Juan-José Salazar-González. "A branch-and-cut algorithm for the undirected traveling purchaser problem." *Operations Research* 51.6 (2003): 940-951.

[2] T., Ramesh. “Travelling Purchaser Problem.” Operations Research 18.2 (1981): 78-91.

[3] Singh, Kashi N.; e Dirk L. van Oudheusden. "A branch and bound algorithm for the traveling purchaser problem." *European Journal of operational research* 97.3 (1997): 571-579.

[4] Jorge Riera-Ledesma; e Juan-José Salazar-González. "Solving the asymmetric traveling purchaser problem." *Annals of Operations Research* 144.1 (2006): 83-97.

[5] Golden, Bruce; Larry Levy; e Roy Dahl. "Two generalizations of the traveling salesman problem." *Omega* 9.4 (1981): 439-441.

[6] Ong, Hoon Liong. "Approximate algorithms for the travelling purchaser problem." *Operations Research Letters* 1.5 (1982): 201-205.

[7] Pearn, Wen Lea; and R. C. Chien. "Improved solutions for the traveling purchaser problem." *Computers & Operations Research* 25.11 (1998): 879-885.

[8] Boctor, Fayez F.; Laporte, Gilbert; e Jacques Renaud. "Heuristics for the traveling purchaser problem." *Computers & Operations Research* 30.4 (2003): 491-504.

[9] Riera-Ledesma, Jorge; e Juan José Salazar-González. "A heuristic approach for the travelling purchaser problem." *European Journal of Operational Research* 162.1 (2005): 142-152.

[10] Lin, Shen; Brian W. Kernighan. "An effective heuristic algorithm for the traveling-salesman problem." *Operations research* 21.2 (1973): 498-516.

[11] C. H. Papadimitriou & K. Steiglitz. “Some Examples of Difficult Traveling Salesman Problems.” Operations research 26 (1978): 434-443.

APÊNDICES

APÊNDICE A - ARTIGO

Aplicação do Problema do Comprador Viajante em uma lista de compras

Matheus Leonel Balduino

Instituto de Informática - Universidade Federal de Santa Catarina (UFSC)
88040420 - Florianópolis - SC - Brasil

(matheusl.balduino@gmail.com)

Abstract. *The Traveling Purchaser Problem (TPP) has several applications in the real world, and in this work we will explore its application in the solution of a supermarket shopping list. Because it is an NP-Hard problem, TSP generalization, much of the literature focuses on heuristics, but a brief study is also made on exact algorithms. The focus of this work is on the search and presentation of the best parameters for the algorithm (heuristic) implemented, from the existing literature, using experiments with previously solved test instances for comparison purposes.*

Resumo. *O problema do comprador viajante (TPP - Traveling Purchaser Problem), possui várias aplicações no mundo real, e neste trabalho iremos explorar sua aplicação na solução de uma lista de compras de supermercado. Por se tratar de um problema NP-Difícil, generalização do TSP, grande parte da literatura se concentra em heurísticas, porém também é feito um breve estudo sobre os algoritmos exatos. O foco deste trabalho está na busca e apresentação dos melhores parâmetros para o algoritmo (heurística) implementado, a partir da literatura existente, utilizando-se de experimentos com instâncias de testes previamente solucionadas para fins de comparação.*

1. Introdução

As compras de supermercado, por muitas vezes, podem ser um problema para famílias com orçamento reduzido. A variação de preços de um supermercado para outro pode fazer uma grande diferença no valor final da lista de compras, e isso pode afetar drasticamente a qualidade de vida de uma família. Estas famílias, por vezes, precisam realizar pesquisas de preços em vários supermercados diferentes, em busca dos melhores descontos.

O ato de pesquisar os preços de produtos em uma lista de compras, em um conjunto de mercados, é algo muito custoso computacionalmente quando a lista de compras possui alguns itens e o conjunto de mercados possui alguns elementos. Nessas condições, o processo de encontrar a melhor solução para a compra dos itens nos mercados se torna praticamente impossível de ser realizado, sem o uso de métodos computacionais. Utilizando-se apenas de papel e caneta pode-se dizer que se trata de uma tarefa impossível. O problema é NP-Difícil, e o tempo necessário para se obter a solução ótima pode ser muito alto (generalização do **TSP** - Traveling Salesman Problem).

Este problema é amplamente estudado pela comunidade científica, visto que as soluções para um problema da classe NP podem ser utilizadas para solucionar outros problemas da mesma classe. Existem diversas contribuições recentes que se utilizam de técnicas diferentes das apresentadas neste trabalho, como inteligência artificial, algoritmos genéticos, etc. O foco deste trabalho está na apresentação, discussão, e testes, de um algoritmo de busca local iterada.

De maneira mais formal, o problema pode ser descrito da seguinte forma: "Encontrar o melhor orçamento final para uma lista de compras **L**, onde cada produto **P** precisa ser necessariamente comprado em um mercado pertencente ao conjunto **M**, levando em consideração o valor do deslocamento, um grafo dirigido, e que a quantidade de produtos que cada mercado oferece é limitada". Esse problema foi primeiramente proposto por Ramesh, em 1981, tendo sido chamado de "TPP - Traveling Purchaser Problem". Por se tratar de uma generalização do **TSP**, o problema possui uma complexidade alta.

Problemas NP-difíceis não podem ser resolvidos em tempo satisfatório quando tratamos de instâncias com tamanho acima do trivial. Algoritmos que encontram uma solução ótima acabam que, na prática, não são úteis, uma vez que o tempo de resposta é excessivamente alto (utilizando um tempo de, no máximo, 1 minuto entre início de uma requisição e o recebimento da resposta como referência). Nesse caso, algoritmos que oferecem algum resultado não-ótimo acabam se tornando mais interessantes (heurísticas).

Encontrar uma solução para problemas deste tipo é uma forma de melhorar a qualidade de vida de famílias de baixa renda, ao mesmo tempo em que se aumenta o fluxo de dinheiro que é movimentado pelas grandes empresas do setor de varejo/atacado, por exemplo.

A principal contribuição deste trabalho está na busca dos melhores parâmetros de execução para o algoritmo utilizado como base na implementação. Esses valores são buscados através de testes que se utilizam de instâncias de teste previamente resolvidas, para que seja possível realizar comparações na qualidade da solução oferecida. O principal parâmetro, chamado *dropLen*, recebeu um valor de 39% como o melhor valor (tendo em vista o conjunto de testes realizados).

2. Objetivos

2.1. Objetivo geral

Realizar um estudo sobre o estado da arte, e implementar um protótipo de algoritmo para a solução do problema do comprador viajante, aplicado na solução de uma lista de compras de supermercado simples. Note que a implementação do protótipo visa a construção de uma heurística que possa ser utilizada em tempo real, para solução das listas, em oposição a um algoritmo que encontre a solução ótima mas possua tempo demasiado alto. A heurística deve dar suporte para grafos direcionados, e levar em consideração a quantidade de cada produto disponível em cada mercado.

2.2. Objetivos Específicos

- Desenvolver um breve estudo sobre o estado da arte dos algoritmos utilizados atualmente para a solução do problema do comprador viajante;
- Desenvolver um algoritmo (protótipo), implementado na linguagem *GO*, que encontre uma solução viável em tempo real para o Problema do Vendedor Viajante.

3. Desenvolvimento

A proposta principal deste trabalho é apresentar uma implementação (protótipo) que solucione o **TPP** em tempo real, para um usuário final. Importante ressaltar que o algoritmo deve considerar a quantidade de produtos disponíveis em cada um dos mercados da lista, bem como situações na qual um produto **P** não está disponível para compra em nenhum mercado (produtos em falta).

Essa implementação se dá na forma de um algoritmo, que recebe entradas em um formato bem definido, e retorna uma resposta com uma solução para o problema. Repare que por se tratar de um problema NP-Difícil, o algoritmo irá retornar soluções viáveis (porém, muito provavelmente, não ótimas).

Nas seções subsequentes, serão definidos os tipos de entrada que a implementação deve aceitar, e quais tipos de dados devem ser retornados, para que seja possível atingir esse objetivo.

3.1. Dados de entrada

É importante que as entradas do algoritmo sejam as mais semelhantes possíveis com a realidade, portanto será necessário informar dados que são esperados na construção de uma lista de compras domésticas, como preço dos produtos em cada um dos mercados, e o valor de deslocamento entre cada um deles. Para isso, será necessário:

- Uma matriz **M**x**M**, onde cada entrada da matriz contém um valor **d** que representa o deslocamento de cada mercado para cada um dos outros mercados;
- Uma matriz **M**x**P**, onde cada entrada da matriz contém uma tupla (c, q) sendo **c** o custo e **q** a quantidade de cada produto em cada um dos mercados listados (estamos trabalhando com a versão capacitada do problema)
- Uma lista **P**, onde cada entrada da lista contém uma tupla (p, q) sendo **p** um produto a ser comprado e **q** a quantidade deste produto

3.2. Dados de saída

Como resposta ao usuário, no entanto, será necessário um pouco mais de informação, visto que procuramos informar todo o procedimento necessário para maximizar os descontos (seja no preço dos produtos ou no deslocamento):

- Uma lista **F**, com a demanda faltante (produtos que não estavam disponíveis para compra em algum mercado na quantidade desejada);
- Uma lista **R**, que deve possuir os mercados que devem ser visitados, em ordem, do primeiro ao último juntamente com quais produtos comprar em cada um dos mercados;
- Um número inteiro positivo **CD**, que representa o custo de deslocamento;
- Um número inteiro positivo **CP**, que representa o custo total da compra dos produtos;
- Um número inteiro positivo **CT**, que representa o custo total da lista, sendo a soma do custo dos produtos e do deslocamento; e
- Uma lista **T**, que contem quais produtos vão ser comprados, por qual valor, e em qual mercado

Observações:

- Observe que o primeiro e o último elemento da lista **R** são o ponto de partida e o ponto final, respectivamente, e não mercados (podendo ser o mesmo ponto ou não);
- Observe que é possível que um produto apareça tanto na lista **R** como na lista **F**, caso a quantidade disponível de algum dos produtos seja inferior à solicitada.

4. O algoritmo

O algoritmo selecionado como base para a implementação da proposta de heurística é o **LSH**, de Ledesma e Salazar (2005). A ideia principal deste algoritmo é selecionar uma rota inicial, contendo todos os mercados, e realizar uma sequência de remoções baseada em uma técnica muito utilizada por algoritmos que solucionam o **TSP**, chamada "*k-opt exchange*". Nesta técnica, **k** mercados são substituídos por outro conjunto de mercados, para tentar escapar de mínimos locais, a cada iteração. Nas próximas seções, será descrita a implementação do algoritmo e algumas de suas características. O algoritmo termina quando admite-se, através de uma heurística, que não há mais como melhorar a solução atual (ótimo local).

4.1. k-opt exchange

Essa técnica é central para a execução do algoritmo, pois é com ela que é possível "escapar" dos mínimos locais que existem na solução do problema. Dado um conjunto **M1** de mercados que pertencem à rota, e um conjunto **M2** de mercados que não pertencem à rota, **k** mercados do conjunto **M1** serão substituídos por **m** mercados do conjunto **M2**. Esse processo é repetido, e em cada execução, decrementa-se o número de mercados substituídos. Ao final, a versão que possui o menor custo é selecionada.

Repare que no caso do **TPP**, não é necessário que a quantidade de mercados adicionados seja igual à quantidade de mercados removidos, como de costume nos algoritmos que utilizam a mesma técnica na solução do **TSP**. A única obrigatoriedade é que a lista de compras deve continuar sendo satisfeita após a operação ser realizada.

4.2. Heurística para o TSP

Durante a execução do algoritmo será necessário encontrar soluções viáveis, por diversas vezes, do problema **TSP**. Toda vez que mercados são adicionados/removidos da rota, é necessário recalcular os custos da mesma. Esse algoritmo será executado por diversas vezes, tendo em vista que serão executadas diversas operações do tipo "*k-opt exchange*", por isso precisa ser relativamente eficiente. Importante notar que a operação de "*k-opt exchange*", no contexto do **TPP**, é um pouco diferente em relação ao **TSP**, visto que a troca de arestas acontece de forma indireta (com a troca de vértices).

A heurística escolhida nesta implementação foi a desenvolvida por Lin e Kernighan (1973), disponível em http://webhotel4.ruc.dk/~keld/research/LKH/ (LKH). Trata-se de uma das heurísticas mais famosas para o problema do **TSP** e também é bastante eficiente no sentido de uso de memória e CPU. Utiliza-se da mesma técnica já comentada anteriormente, "*k-opt exchange*", para otimizar a rota.

A ideia principal deste algoritmo é muito semelhante aquela descrita no item anterior, tendo como única diferença o fato de que todos os mercados devem obrigatoriamente

pertencer à rota, ou seja, os mercados nunca são removidos, e sim realocados em novas posições (índice) na rota. Essa transformação costuma diminuir o valor final da rota.

5. Experimentos

Para realizar testes com o algoritmo, após o término da implementação, foi necessário encontrar algumas instâncias de problemas do tipo TPP, para que pudessem ser resolvidas e comparadas com os resultados ótimos obtidos pela comunidade acadêmica.

As instâncias que foram utilizadas nos testes deste TCC são as mesmas utilizadas pelo artigo que serviu como base [Laporte 2000]. Essas instâncias estão disponíveis para download através do link: https://jriera.webs.ull.es/TPP.htm. Repare que existem diversas categorias de instâncias, que cobrem os 4 possíveis casos: instâncias simétricas e não capacitadas, simétricas capacitadas, assimétricas não capacitadas e assimétricas capacitadas.

O foco deste trabalho se encontra nas instâncias que mais se assemelham a problemas de rota no mundo real: assimétricas capacitadas. Infelizmente, para esta categoria de problema, as instâncias disponibilizadas não possuem solução ótima e nem documentação. Por esse motivo optou-se pelo uso das instâncias simétricas capacitadas, que estão devidamente documentadas e possuem solução para a maior parte das instâncias.

Problemas simétricos são um pouco mais fáceis de resolver, tendo em vista que todos os custos de deslocamento entre mercados são iguais tanto saindo de **A** para **B** quanto de **B** para **A**. No mundo real essa regra claramente não se mantém verdadeira, pois é comum que as regras de trânsito acabem definindo que a distância saindo de **A** até **B** seja diferente de **B** até **A**, em alguns casos.

Estas instâncias foram todas geradas aleatoriamente. O número de mercados e a quantidade de produtos são definidos de forma arbitrária, enquanto que a demanda e oferta dos produtos é feita da seguinte forma: Para cada produto e cada mercado, a oferta do produto em cada mercado é definida aleatoriamente no intervalo [1, 15], e a demanda é calculada através da fórmula:

- Seleciona-se a oferta mais alta, disponível em algum dos mercados, e multiplica-se esse valor por **λ** ($0 <= \lambda <= 1$). Vamos chamar esse valor de "**X**"
- Faz-se o somatório da oferta que cada mercado possui para o determinado produto, e multiplica-se o valor final do somatório por $(1 - \lambda)$. Vamos chamar esse valor de "**Y**"
- Soma-se os dois valores resultantes anteriores, e aplica-se uma função de arredondamento para cima: $demanda = ceil(X + Y)$

Note que quanto maior o valor de **λ**, menor o tamanho do trajeto, pois com **λ**=1 a instância se torna um **TPP** irrestrito (oferta de produtos infinita), e com **λ**=0 se torna um **TSP** (obrigatória a passagem por todos os mercados). Foram testadas 487 instâncias com as seguintes características:

- 50, 100, 150 e 200 mercados
- 50, 100, 150 e 200 produtos
- Com **λ** (*lambda*) igual a 0.1, 0.5, 0.7, 0.8, 0.9, 0.95 e 0.99

Todos os arquivos das instâncias foram gerados seguindo o padrão utilizado pela biblioteca **TPPLIB**, que nada mais é do que uma extensão do padrão utilizado pela biblioteca **TSPLIB**. Esta biblioteca define uma série de seções que seus arquivos devem ter, de modo a

manter a compatibilidade na entrada dos dados a serem testados por algoritmos de resolução do **TSP**, e **TPP** (após a extensão).

Todos os testes foram realizados em um computador pessoal, com as seguintes configurações:

- Processador Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz;
- 32GB de memória RAM;
- Linux Mint 19.3 (Tricia);
- Linux Kernel 5.0.0-32-generic; e
- Go build tool 1.19.4.

Para execução dos testes, utilizou-se o máximo de paralelismo possível, sendo que cada instância de teste executou em sua própria *go-routine* por todo o seu ciclo de vida. O tempo de execução de todas as instâncias de teste foi variado, dependendo dos parâmetros escolhidos, sendo que durante esse tempo o poder de processamento do computador foi deixado todo à disposição da execução do algoritmo.

5.1. Resultados

O método utilizado para encontrar os melhores parâmetros para a execução das instâncias foi o de tentativa e erro. Ao longo da implementação do algoritmo, foi-se percebendo uma afinidade maior de determinados parâmetros, com determinadas instâncias de teste. O método para cálculo do *gap* de cada uma das respostas, em relação ao mínimo global, é realizado através do cálculo da porcentagem na qual o valor ficou acima do mínimo. Se o mínimo global for 100, por exemplo, e a resposta foi 120, temos um *gap* de 20%.

O parâmetro *factor* está diretamente ligado com o tempo de execução do algoritmo. Caso esse valor seja alto demais, o algoritmo irá executar por muito tempo, sem a garantia de encontrar valores melhores. Com um valor muito baixo, a perturbação causada nas soluções parciais vai ser muito baixa, tornando-se irrelevante. No caso deste parâmetro, optou-se por utilizar a mesma métrica definida no artigo original (Local Search [9]): 35%. Neste caso, na primeira iteração, é admitido um aumento de até 35% no custo, ao perturbar a solução parcial adicionando novos mercados.

O parâmetro *dropLen* é um pouco mais complicado de realizar os ajustes, pois é muito mais sensível e desencadeia em uma série de alterações nas soluções encontradas. A princípio, tentou-se utilizar 50% do tamanho da solução atual, porém esse valor demonstrou-se ser muito ineficaz na questão de tempo de execução. Valores muito pequenos como 10% e 20% não demonstraram um desempenho satisfatório na questão de redução de custos. Com base nos experimentos realizados, o melhor intervalo, portanto, parece ser entre 35% e 45% do tamanho da rota atual.

Note que esses parâmetros estão intimamente ligados com o formato das instâncias de teste, e portanto a seleção destes parâmetros está intimamente ligada com o objetivo que está sendo buscado. No caso deste artigo, a busca é pelo equilíbrio entre o tempo de execução e a redução nos custos. Nas instâncias de teste que foram executadas aqui, o valor com melhor custo-benefício ficou próximo de ~39%, e por isso esse foi o valor escolhido, mas isso pode se alterar de acordo com as instâncias de teste ou objetivo.

A seguir serão descritos alguns gráficos (na forma de figuras) e uma tabela, contendo os valores dos experimentos realizados, juntamente com alguns parágrafos que buscam analisar esses resultados.

Após analisar atentamente os dados coletados na Tabela 1, é possível observar que o melhor valor para *dropLen*, considerando-se apenas o custo, seria algo em torno de 34% (para esse conjunto de instâncias). Como o objetivo é obter um algoritmo que produza uma resposta em um tempo satisfatório, também, optou-se por escolher 39%, visto que este oferece um tempo de execução menor, e um custo semelhante. Isso é visualizado com mais facilidade através do gráfico de linha, representado pela Figura 1.

O gráfico da Figura 1 é uma sobreposição dos dados referentes ao gap médio (azul), para cada valor de *dropLen*, juntamente com o tempo gasto (vermelho) para calcular todas as 487 instâncias de teste. No eixo y da esquerda temos a porcentagem de *gap* médio, e no eixo y da direita temos o tempo de execução. Foram desenhadas, também, duas linhas de tendência: uma para o *gap* médio e outra para o tempo de execução.

É possível observar que o tempo de execução tende a aumentar conforme o valor de *dropLen* é aumentado, através da linha de tendência vermelha. Isso é esperado, visto que esse parâmetro controla a quantidade de vezes que alguns laços de repetição vão executar, como comentado nas seções anteriores. O inverso, no entanto, também pode ocorrer (redução no tempo de execução), mas em geral a tendência é de aumento.

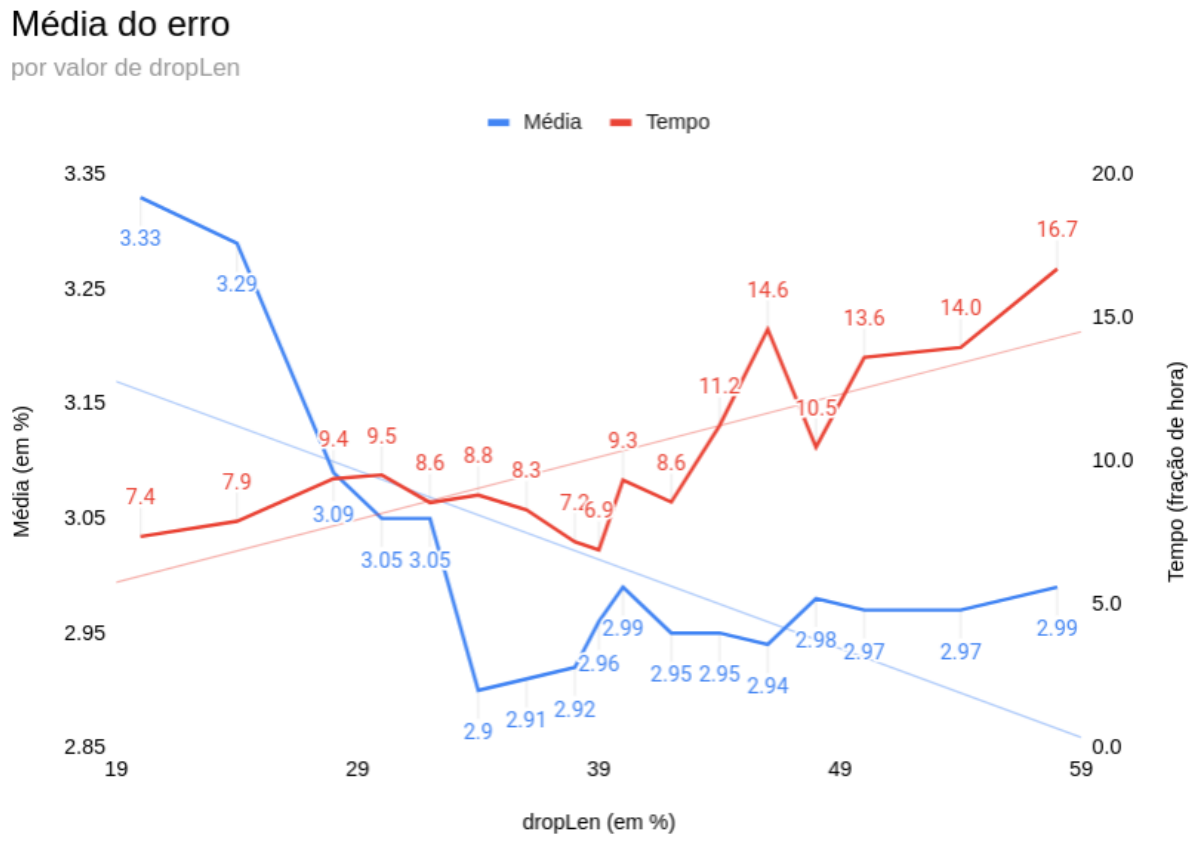

Figura 1 - Gráfico de linha, com linhas de tendência

O *gap* médio atinge o mínimo quando *dropLen* é 34%, porém o tempo de execução ainda pode ser melhorado. Somente perto dos 39% que o tempo de execução reduz, ainda sem muito prejuízo na qualidade do *gap* médio. Esse equilíbrio é o motivo da escolha de 39% como o melhor valor, neste trabalho.

Através do gráfico representado pela Figura 2, pode-se observar que mesmo com a aplicação de diferentes valores para *dropLen*, os dados estatísticos permanecem muito próximos uns dos outros. Em situações na qual o *gap* médio se altera, os valores mínimos, máximos e desvio padrão permanecem o mesmo, indicando apenas um deslocamento do *gap* em relação ao valor ótimo, em praticamente todas as instâncias (visto que a dispersão dos valores continua a mesma). Os outliers foram removidos pois tratam-se sempre das mesmas instâncias (que serão comentadas nas seções seguintes), e isso melhora significativamente a visualização do gráfico.

Tabela 1 - Custo e tempo de execução para cada valor de dropLen

dropLen	***gap* mínimo**	***gap* médio**	***gap* máximo**	**Desvio padrão do *gap***	**Tempo**
20%	0.03%	3.33%	119.75%	9.3%	7h22m
24%	0.03%	3.29%	119.75%	9.1%	7h54m
28%	0.03%	3.09%	119.75%	7.7%	9h23m
30%	0.03%	3.05%	119.75%	7.7%	9h31m
32%	0.03%	3.05%	119.75%	7.8%	8h33m
34%	0.03%	2.90%	119.75%	7.6%	8h49m
36%	0.03%	2.91%	119.75%	7.5%	8h18m
38%	0.03%	2.92%	119.75%	7.6%	7h11m
39%	0.03%	2.96%	119.75%	7.6%	6h54m
40%	0.03%	2.99%	119.75%	7.6%	9h20m
42%	0.03%	2.95%	119.75%	7.6%	8h34m
44%	0.03%	2.95%	119.75%	7.6%	11h14m
46%	0.03%	2.94%	119.75%	7.6%	14h36m
48%	0.03%	2.98%	119.75%	7.6%	10h29m
50%	0.03%	2.97%	119.75%	7.6%	13h37m
54%	0.03%	2.97%	119.75%	7.6%	13h58m
58%	0.03%	2.99%	119.75%	7.6%	16h42m

5.2. Observações importantes

Em alguns casos isolados, a redução de *dropLen* pode aumentar o tempo de execução, algo que pode parecer uma surpresa em um primeiro momento. Isso ocorre devido ao fato de que diferentes valores desse parâmetro vão causar a exploração de ramos completamente diferentes na árvore de possibilidades de cada um dos problemas. É completamente possível, e esperado, que em alguns casos a redução deste valor provoque alguns saltos (na tentativa de escapar de mínimos locais), que caiam em "vales" que sejam mais demorados para otimizar do que outros.

Em suma, aumentar o valor de *dropLen* tende a aumentar o tempo de execução, mas essa afirmação pode não ser verdadeira em alguns casos específicos, como esperado. Diminuir esse valor também tende a reduzir o tempo de execução, mas isso também não se mantém verdade em todos casos.

A qualidade da solução obtida possui uma tendência de melhora conforme *dropLen* é aumentado, pois o número de iterações aumenta juntamente com esse parâmetro. O acontece, inversamente, para o tempo de execução, que tende a aumentar conforme dropLen aumenta.

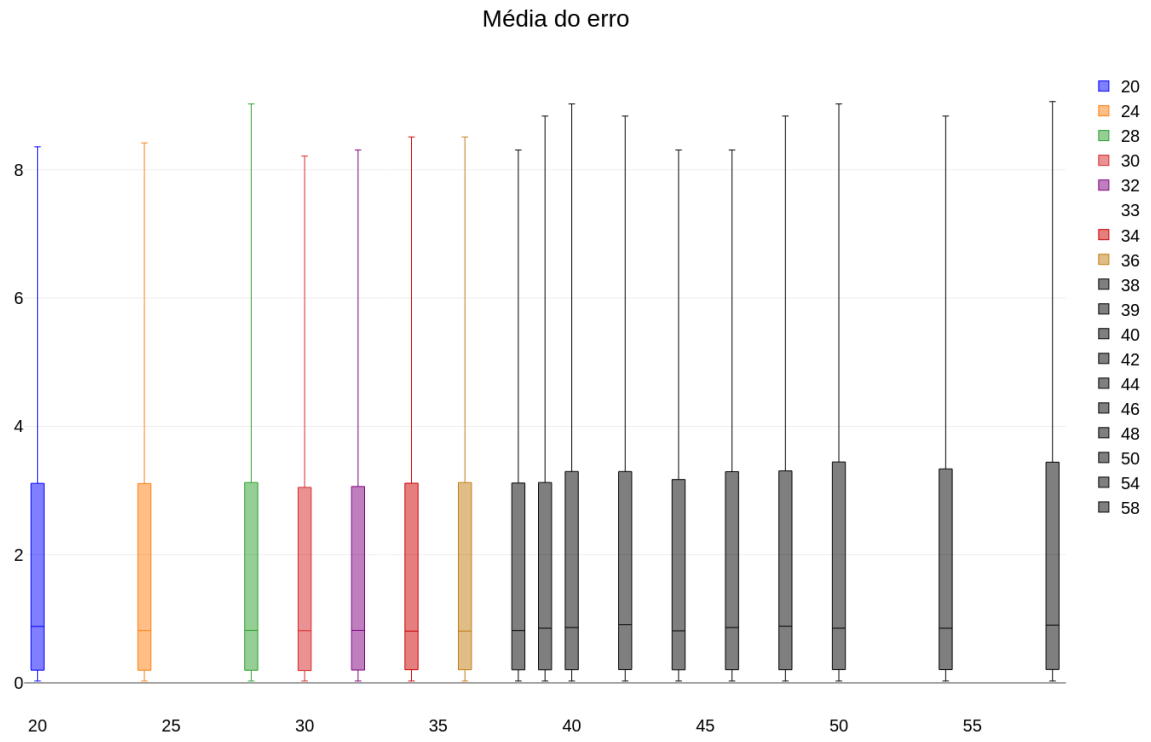

Figura 2 - Gráfico em formato *boxplot*, contendo a média dos gaps (valores em %), sem *outliers*

5.3. Instâncias "fáceis"

Algumas instâncias de problemas são ligeiramente mais fáceis de serem resolvidas, pois reduzem-se ao próprio **TSP**. Caso todos os mercados disponíveis precisem necessariamente fazer parte da rota, então o problema torna-se um **TSP**, sendo necessário apenas comprar os produtos pelo melhor valor.

Observe na Tabela 2 que as instâncias que se mantêm como as melhores soluções, não por acaso, são aquelas que obrigam a presença de todos os mercados disponíveis na rota final

Tabela 2 - Melhores instâncias de teste (*gap* se mantém igual para todos *dropLen*)

Mercados	Produtos	λ (lambda)	Mercados na rota final	*Gap*
50	200	0.1	50/50	0.02832%
150	200	0.1	150/150	0.02807%
50	200	0.1	50/50	0.02794%
150	200	0.1	150/150	0.02752%
50	200	0.1	50/50	0.02745%

(coluna "Mercados na rota final"). Esse comportamento se deve ao fato do fator lambda ser próximo de zero (o valor mais baixo usado para lambda, na verdade), que quase transforma essa instância em um **TSP**.

Por tratarem-se de um problemas quase idênticos ao **TSP**, o *gap* acima pode se tratar de erro numérico, como será comentado na seção "Erro numérico". Essas instâncias são solucionadas muito mais rapidamente e com muito mais precisão, por focarem apenas na parte **TSP** do problema. Infelizmente é difícil encontrar situações como essa no mundo real, na solução de uma lista de compras, pois a situação inversa é mais comum: poucos mercados são necessários para completar uma lista.

5.4. Instâncias difíceis, ou "problemas perversos"

O método utilizado para solucionar as instâncias de teste, como já comentado anteriormente, é uma generalização da técnica *k-opt*, usada por diversos algoritmos que solucionam o **TSP**. Essa técnica possui os melhores resultados, atualmente, porém por tratar-se de uma heurística também contém algumas falhas.

Papadimitriou e Steiglitz (1978) demonstraram que existe uma classe de problemas **TSP** gerados aleatoriamente no qual algoritmos de busca local (a técnica *k-opt* é uma busca local), são extremamente ineficazes. Essa classe de problemas foi chamada de "problemas perversos", na descrição original.

No conjunto de testes realizados, é possível notar que algumas instâncias sempre são solucionadas com uma alta taxa de *gap*. Abaixo são listadas as 5 instâncias que obtiveram os piores resultados, permanecendo como pior resultado em todas as iterações:

Repare na Tabela 3 que todas as instâncias que obtiveram os piores resultados possuem um lambda igual a 0.99. Isso acontece pelo fato de que a dificuldade em solucionar os problemas cresce conforme o λ (lambda), pois como já comentado anteriormente, com um valor alto para lambda é necessário otimizar os dois aspectos do problema: a escolha dos produtos e a escolha dos mercados, ao mesmo tempo.

Note que em algumas situações, quando a demanda é igual (ou superior) a quantidade global ofertada, todos os mercados obrigatoriamente deverão pertencer à rota (na prática, um **TSP**). O inverso também pode ocorrer, caso apenas 1 mercado ofereça toda a demanda. Isso é

Tabela 3 - Piores instâncias de teste (*gap* se mantém igual para todos *dropLen*)

Mercados	Produtos	λ (lambda)	Mercados na rota final	*Gap*
150	50	0.99	55/150	119.75%
50	150	0.99	43/50	74.96%
50	200	0.99	36/50	71.79%
150	200	0.99	84/150	70.78%
50	200	0.99	38/50	70.65%

controlado através do parâmetro **λ** (lambda), já discutido anteriormente. No fim das contas, todas as instâncias que envolvem a decisão de ambos fatores tendem a ser mais difíceis de serem resolvidas. Essa classe de "problemas perversos", no entanto, tende a não se apresentar em problemas do mundo real, na maioria das vezes.

5.5. Erro numérico

O posicionamento dos mercados no plano cartesiano 2D é feito de forma aleatória, as distâncias são calculadas de acordo com a distância Euclidiana, e ao mesmo tempo todos os valores de custo (deslocamento e preço de produto) são definidos como valores inteiros.

A configuração descrita acima acaba por provocar uma taxa de erro numérico, que se acumula ao longo das iterações e execução do algoritmo. Como as distâncias são euclidianas, e os mercados são distribuídos aleatoriamente no plano, é inevitável que a maioria das distâncias sejam números reais. Observe a Figura 3 para visualizar o cálculo da distância entre dois pontos.

No artigo original que descreve o algoritmo usado como base para este trabalho [Ledesma e Salazar 2005], nenhuma informação sobre a normalização desses erros é comentada. Nesta implementação, optou-se por um arredondamento para baixo (truncamento), ao calcular a matriz de distâncias entre as coordenadas. Isso pode ser uma das razões pela qual algumas instâncias possuem um gap pequeno e constante, para todos os valores de dropLen utilizados.

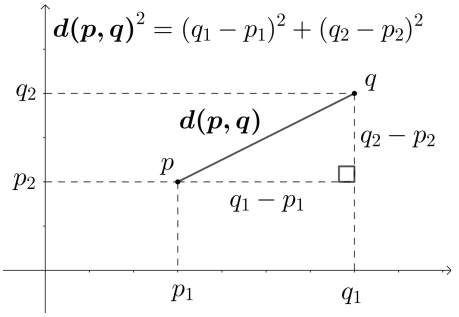

Figura 3 - Exemplo de distância euclidiana entre dois pontos

6. Conclusão

No presente trabalho foi analisado o contexto histórico da comunidade acadêmica, no que diz respeito ao **TPP**. Vários autores diferentes já se dedicaram ao estudo deste problema, pois sua solução é de extrema importância para uma série de setores da sociedade e indústria.

Por se tratar de um algoritmo NP-Difícil, com certa semelhança com o **TSP**, solucionar esse problema também significa avançar na solução de diversos outros problemas semelhantes (como o próprio **TSP**).

A alta complexidade dos algoritmos envolvidos acaba por limitar bastante a aplicação dos mesmos, pois qualquer aplicação que venha a interagir com usuários finais, hoje em dia, necessita de tempos de resposta rápidos.

O presente trabalho apresentou uma implementação do algoritmo **LSH** de Ledesma e Salazar (2005), com algumas diferenças na seleção dos parâmetros de configuração original, bem como um estudo de caso em cima dos resultados obtidos. O algoritmo demonstra ser extremamente flexível, podendo ser aplicado com diferentes objetivos, apenas com a alteração dos seus parâmetros.

7. Referências

Laporte, Gilbert; Jorge Riera-Ledesma; e Juan-José Salazar-González. "A branch-and-cut algorithm for the undirected traveling purchaser problem." Operations Research 51.6 (2003): 940-951.

T., Ramesh. “Travelling Purchaser Problem.” Operations Research 18.2 (1981): 78-91.

Boctor, Fayez F.; Laporte, Gilbert; e Jacques Renaud. "Heuristics for the traveling purchaser problem." Computers & Operations Research 30.4 (2003): 491-504.

Riera-Ledesma, Jorge; e Juan José Salazar-González. "A heuristic approach for the travelling purchaser problem." European Journal of Operational Research 162.1 (2005): 142-152.

Lin, Shen; Brian W. Kernighan. "An effective heuristic algorithm for the traveling-salesman problem." Operations research 21.2 (1973): 498-516.

C. H. Papadimitriou & K. Steiglitz. “Some Examples of Difficult Traveling Salesman Problems.” Operations research 26 (1978): 434-443.

APÊNDICE B - CÓDIGO-FONTE

```
// CheapestInsertionTSP will return a tuple containing the cost of the
tour, and the tour itself.
// Each element of the tour array represents an index of the cost
matrix
func CheapestInsertionTSP(costMatrix private.TppMatrix) (uint64, []int)
{
  if len(costMatrix) == 0 {
     return math.MaxUint64, nil
  }
  if len(costMatrix) == 1 {
     return 0, []int{0}
  }
  if len(costMatrix) == 2 {
     return costMatrix[0][1].Cost, []int{0, 1}
  }

  tour := make([]int, 0, len(costMatrix))
  tour = append(tour, 0)
  tour = append(tour, len(costMatrix)-1)
  totalCost := costMatrix[0][len(costMatrix)-1].Cost
  marketsInserted := make(map[int]bool, len(costMatrix)-2)
  for len(tour) < len(costMatrix) {
     marketToInsert := -1
     insertMarketAt := -1
     marketsBestCost := uint64(math.MaxUint64)
     for i_ := range costMatrix[1 : len(costMatrix)-1] {
        i := i_ + 1
        if marketsInserted[i] {
           continue
        }
        indexToInsert, bestCost := BestSavingTSP(i, tour, costMatrix)
        if bestCost < marketsBestCost {
           marketToInsert = i
           marketsBestCost = bestCost
           insertMarketAt = indexToInsert
        }
     }
     totalCost += marketsBestCost
                                                    totalCost          -=
costMatrix[tour[insertMarketAt-1]][tour[insertMarketAt]].Cost
     marketsInserted[marketToInsert] = true
     aux := marketToInsert
     tour = append(tour, -1)
     for i_, currentIndex := range tour[insertMarketAt:] {
        i := insertMarketAt + i_
        aux__ := currentIndex
        tour[i] = aux
```

```
            aux = aux__
        }
    }
    return totalCost, tour
}

func TestRun(t *testing.T) {
    t.Run("Run all instances", func(t *testing.T) {
        now := fmt.Sprintf("%s", time.Now())
        notifyTelegram(now)
        runAll(t, private.LSH, toIgnore{})
    })
    t.Run("Run selected instances", func(t *testing.T) {
        now := fmt.Sprintf("%s", time.Now())
        notifyTelegram(now)
        runAllSelected(t, private.LSH, toSelect{
            //map[uint64]bool{3: true},
            //map[uint64]map[uint64]bool{3: {1: true}},
            //map[uint64]map[uint64]map[uint64]bool{3: {1: {1: true}}},
              //map[uint64]map[uint64]map[uint64]map[uint64]bool{3: {1: {1:
{1: true}}}},
            map[uint64]bool{50: true},
            map[uint64]map[uint64]bool{50: {50: true}},
            map[uint64]map[uint64]map[uint64]bool{50: {50: {99: true}}},
             map[uint64]map[uint64]map[uint64]map[uint64]bool{50: {50: {99:
{1: true}}}},
        })
    })
    t.Run("Run missing instances", func(t *testing.T) {
        files, e := os.ReadDir(RESULTS_FOLDER)
        if e != nil {
            t.Fatal(e)
        }
            m := map[string]int{"all": 0, "nMarket": 1, "nProduct": 2,
"lambda": 3, "variation": 4}
        sort.Slice(files, func(i, j int) bool {
            fileI, fileJ := files[i], files[j]
                    nameI, nameJ := strings.Split(fileI.Name(), ".")[0],
strings.Split(fileJ.Name(), ".")[0]
            orderI, orderJ := m[nameI], m[nameJ]
            return orderI < orderJ
        })
        ignore := toIgnore{
            map[uint64]bool{},
            map[uint64]map[uint64]bool{},
            map[uint64]map[uint64]map[uint64]bool{},
            map[uint64]map[uint64]map[uint64]map[uint64]bool{}}
        for _, file := range files {
            if strings.HasPrefix(file.Name(), "all.") {
                t.Logf("There's no missing tests to execute\n")
                return
```

```
        }
        if strings.HasPrefix(file.Name(), "nMarket.") {
           nMarkets := uint64(0)
           fmt.Sscanf(file.Name(), "nMarket.%d.solution", &nMarkets)
           ignore.Nmarkets[nMarkets] = true
           t.Logf("nMarket %d instances skipped\n", nMarkets)
           continue
        }
        if strings.HasPrefix(file.Name(), "nProduct.") {
           nMarkets, nProducts := uint64(0), uint64(0)
                     fmt.Sscanf(file.Name(),  "nProduct.%d.%d.solution",
&nMarkets, &nProducts)
           if ignore.Nmarkets[nMarkets] {
              continue
           }
           if ignore.Nproducts[nMarkets] == nil {
              ignore.Nproducts[nMarkets] = map[uint64]bool{}
           }
           ignore.Nproducts[nMarkets][nProducts] = true
                t.Logf("nMarket  %d,  nProduct  %d  instances  skipped\n",
nMarkets, nProducts)
           continue
        }
        if strings.HasPrefix(file.Name(), "lambda.") {
                 nMarkets,  nProducts,  lambda  :=  uint64(0),  uint64(0),
uint64(0)
                    fmt.Sscanf(file.Name(),  "lambda.%d.%d.%d.solution",
&nMarkets, &nProducts, &lambda)
           if ignore.Nmarkets[nMarkets] {
              continue
           }
                           if  ignore.Nproducts[nMarkets]  !=  nil  &&
ignore.Nproducts[nMarkets][nProducts] {
              continue
           }
           if ignore.Nlambdas[nMarkets] == nil {
              ignore.Nlambdas[nMarkets] = map[uint64]map[uint64]bool{}
           }
           if ignore.Nlambdas[nMarkets][nProducts] == nil {
              ignore.Nlambdas[nMarkets][nProducts] = map[uint64]bool{}
           }
           ignore.Nlambdas[nMarkets][nProducts][lambda] = true
                 t.Logf("nMarket  %d,  nProduct  %d,  lambda  %d  instances
skipped\n", nMarkets, nProducts, lambda)
           continue
        }
        if strings.HasPrefix(file.Name(), "variation.") {
                 nMarkets,  nProducts,  lambda,  variation  :=  uint64(0),
uint64(0), uint64(0), uint64(0)
              fmt.Sscanf(file.Name(), "variation.%d.%d.%d.%d.solution",
&nMarkets, &nProducts, &lambda, &variation)
```

```
            if ignore.Nmarkets[nMarkets] {
                continue
            }
                            if  ignore.Nproducts[nMarkets]  !=  nil  &&
ignore.Nproducts[nMarkets][nProducts] {
                continue
            }
                             if  ignore.Nlambdas[nMarkets]  !=  nil  &&
ignore.Nlambdas[nMarkets][nProducts]         !=         nil         &&
ignore.Nlambdas[nMarkets][nProducts][lambda] {
                continue
            }
            if ignore.Nvariation[nMarkets] == nil {
                                            ignore.Nvariation[nMarkets]   =
map[uint64]map[uint64]map[uint64]bool{}
            }
            if ignore.Nvariation[nMarkets][nProducts] == nil {
                                    ignore.Nvariation[nMarkets][nProducts]  =
map[uint64]map[uint64]bool{}
            }
            if ignore.Nvariation[nMarkets][nProducts][lambda] == nil {
                            ignore.Nvariation[nMarkets][nProducts][lambda]  =
map[uint64]bool{}
            }
              ignore.Nvariation[nMarkets][nProducts][lambda][variation] =
true
              t.Logf("nMarket %d, nProduct %d, lambda %d, variation %d
skipped\n", nMarkets, nProducts, lambda, variation)
            continue
          }
          t.Fatalf("unexpected filename: %s\n", file.Name())
      }
      runAll(t, private.LSH, ignore)
  })
}

type toIgnore struct {
  Nmarkets   map[uint64]bool
  Nproducts  map[uint64]map[uint64]bool
  Nlambdas   map[uint64]map[uint64]map[uint64]bool
  Nvariation map[uint64]map[uint64]map[uint64]map[uint64]bool
}

type allResult struct {
  Results    map[uint64]marketResult `json:"-"`
  Statistics statistics[intervalStatistic]
}

func  runAll(t  *testing.T,  algorithm  private.TppAlgorithm,  ignore
toIgnore) allResult {
  t.Helper()
```

```
  c := make(chan map[uint64]marketResult, len(solutions))
  results := make(map[uint64]marketResult, len(solutions))
  n := 0
  for nMarkets_ := range solutions {
     nMarkets := nMarkets_
     if ignore.Nmarkets[nMarkets] {
        continue
     }
     n += 1
     go func() {
         c <- map[uint64]marketResult{nMarkets: runMarket(t, algorithm,
nMarkets, ignore)}
     }()
  }
  for i := 0; i < n; i++ {
     r := <-c
     for lambda := range r {
        results[lambda] = r[lambda]
     }
  }

  nResults := 0.0
       sumCost,    sumMarkets,    sumSeconds    :=    statistic[int64]{},
statistic[int64]{}, statistic[int64]{}
  minCost := minMaxValue{statistic[int64]{Percentage: math.MaxFloat64},
0, 0, 0, 0}
  maxCost := minMaxValue{statistic[int64]{Percentage: 0}, 0, 0, 0, 0}
          minMarkets     :=     minMaxValue{statistic[int64]{Percentage:
math.MaxFloat64}, 0, 0, 0, 0}
   maxMarkets := minMaxValue{statistic[int64]{Percentage: 0}, 0, 0, 0,
0}
          minSeconds     :=     minMaxValue{statistic[int64]{Percentage:
math.MaxFloat64}, 0, 0, 0, 0}
   maxSeconds := minMaxValue{statistic[int64]{Percentage: 0}, 0, 0, 0,
0}
  for _, result := range results {
     for _, productResults := range result.Results {
        for _, lambdaResults := range productResults.Results {
           nResults += float64(len(lambdaResults.Results))
        }
     }
     sumCost.Percentage += result.Statistics.Cost.Sum.Percentage
     sumCost.Absolute += result.Statistics.Cost.Sum.Absolute
     sumMarkets.Percentage += result.Statistics.Markets.Sum.Percentage
     sumMarkets.Absolute += result.Statistics.Markets.Sum.Absolute
     sumSeconds.Percentage += result.Statistics.Seconds.Sum.Percentage
     sumSeconds.Absolute += result.Statistics.Seconds.Sum.Absolute
                  if    result.Statistics.Cost.Min.Value.Percentage    <
minCost.Value.Percentage {
        minCost.Variation = result.Statistics.Cost.Min.Variation
        minCost.Lambda = result.Statistics.Cost.Min.Lambda
```

```
        minCost.Nproducts = result.Statistics.Cost.Min.Nproducts
        minCost.Nmarkets = result.Statistics.Cost.Min.Nmarkets
                                            minCost.Value.Absolute     =
result.Statistics.Cost.Min.Value.Absolute
                                          minCost.Value.Percentage     =
result.Statistics.Cost.Min.Value.Percentage
     }
                    if    result.Statistics.Cost.Max.Value.Percentage    >
maxCost.Value.Percentage {
        maxCost.Variation = result.Statistics.Cost.Max.Variation
        maxCost.Lambda = result.Statistics.Cost.Max.Lambda
        maxCost.Nproducts = result.Statistics.Cost.Max.Nproducts
        maxCost.Nmarkets = result.Statistics.Cost.Max.Nmarkets
                                            maxCost.Value.Absolute     =
result.Statistics.Cost.Max.Value.Absolute
                                          maxCost.Value.Percentage     =
result.Statistics.Cost.Max.Value.Percentage
     }
                 if    result.Statistics.Markets.Min.Value.Percentage   <
minMarkets.Value.Percentage {
        minMarkets.Variation = result.Statistics.Markets.Min.Variation
        minMarkets.Lambda = result.Statistics.Markets.Min.Lambda
        minMarkets.Nproducts = result.Statistics.Markets.Min.Nproducts
        minMarkets.Nmarkets = result.Statistics.Markets.Min.Nmarkets
                                          minMarkets.Value.Absolute     =
result.Statistics.Markets.Min.Value.Absolute
                                       minMarkets.Value.Percentage     =
result.Statistics.Markets.Min.Value.Percentage
     }
                 if    result.Statistics.Markets.Max.Value.Percentage   >
maxMarkets.Value.Percentage {
        maxMarkets.Variation = result.Statistics.Seconds.Max.Variation
        maxMarkets.Lambda = result.Statistics.Seconds.Max.Lambda
        maxMarkets.Nproducts = result.Statistics.Seconds.Max.Nproducts
        maxMarkets.Nmarkets = result.Statistics.Seconds.Max.Nmarkets
                                          maxMarkets.Value.Absolute     =
result.Statistics.Markets.Max.Value.Absolute
                                       maxMarkets.Value.Percentage     =
result.Statistics.Markets.Max.Value.Percentage
     }
                 if    result.Statistics.Seconds.Min.Value.Percentage   <
minSeconds.Value.Percentage {
        minSeconds.Variation = result.Statistics.Markets.Min.Variation
        minSeconds.Lambda = result.Statistics.Markets.Min.Lambda
        minSeconds.Nproducts = result.Statistics.Markets.Min.Nproducts
        minSeconds.Nmarkets = result.Statistics.Markets.Min.Nmarkets
                                          minSeconds.Value.Absolute     =
result.Statistics.Seconds.Min.Value.Absolute
                                       minSeconds.Value.Percentage     =
result.Statistics.Seconds.Min.Value.Percentage
     }
```

```
                if   result.Statistics.Seconds.Max.Value.Percentage   >
maxSeconds.Value.Percentage {
        maxSeconds.Variation = result.Statistics.Seconds.Max.Variation
        maxSeconds.Lambda = result.Statistics.Seconds.Max.Lambda
        maxSeconds.Nproducts = result.Statistics.Seconds.Max.Nproducts
        maxSeconds.Nmarkets = result.Statistics.Seconds.Max.Nmarkets
                                          maxSeconds.Value.Absolute     =
result.Statistics.Seconds.Max.Value.Absolute
                                       maxSeconds.Value.Percentage     =
result.Statistics.Seconds.Max.Value.Percentage
     }
  }
     averageCost   :=   statistic[float64]{float64(sumCost.Absolute)   /
nResults, sumCost.Percentage / nResults}
   averageMarkets  := statistic[float64]{float64(sumMarkets.Absolute)  /
nResults, sumMarkets.Percentage / nResults}
   averageSeconds  := statistic[float64]{float64(sumSeconds.Absolute)  /
nResults, sumSeconds.Percentage / nResults}
      costSum,   marketsSum,   secondsSum   :=   statistic[float64]{},
statistic[float64]{}, statistic[float64]{}
  for _, result := range results {
            diff   :=   float64(result.Statistics.Cost.Sum.Absolute)   -
averageCost.Absolute
     costSum.Absolute += diff * diff
           perc   :=   float64(result.Statistics.Cost.Sum.Percentage)   -
averageCost.Percentage
     costSum.Percentage += perc * perc
           diff   =   float64(result.Statistics.Markets.Sum.Absolute)   -
averageMarkets.Absolute
     marketsSum.Absolute += diff * diff
          perc   =   float64(result.Statistics.Markets.Sum.Percentage)   -
averageMarkets.Percentage
     marketsSum.Percentage += perc * perc
           diff   =   float64(result.Statistics.Seconds.Sum.Absolute)   -
averageSeconds.Absolute
     secondsSum.Absolute += diff * diff
          perc   =   float64(result.Statistics.Seconds.Sum.Percentage)   -
averageSeconds.Percentage
     secondsSum.Percentage += perc * perc
  }
    costStdError   :=   statistic[float64]{math.Sqrt(costSum.Absolute   /
nResults), math.Sqrt(costSum.Percentage / nResults)}
  marketsStdError := statistic[float64]{math.Sqrt(marketsSum.Absolute /
nResults), math.Sqrt(marketsSum.Percentage / nResults)}
  secondsStdError := statistic[float64]{math.Sqrt(secondsSum.Absolute /
nResults), math.Sqrt(secondsSum.Percentage / nResults)}
  r := allResult{
     results,
     statistics[intervalStatistic]{
           intervalStatistic{sumCost,  minCost,  averageCost,  maxCost,
costStdError},
```

```
            intervalStatistic{sumSeconds, minSeconds, averageSeconds,
maxSeconds, secondsStdError},
            intervalStatistic{sumMarkets, minMarkets, averageMarkets,
maxMarkets, marketsStdError},
      },
  }

  s := strings.Builder{}
  s.WriteString("#ALL\n")
  s.WriteString(fmt.Sprintf("%s\n", algorithm))
  intervalStatisticToTelegram(s.String(), r.Statistics)
  f, e := os.Create(fmt.Sprintf("%s/all.solution", RESULTS_FOLDER))
  if e != nil {
     notifyTelegram("ERROR: " + e.Error())
     return r
  }
  j, e := json.Marshal(r)
  if e != nil {
     notifyTelegram("ERROR: " + e.Error())
     return r
  }
  _, e = f.Write(j)
  if e != nil {
     notifyTelegram("ERROR: " + e.Error())
     return r
  }
  e = f.Close()
  if e != nil {
     notifyTelegram("ERROR: " + e.Error())
     return r
  }
  return r
}

type toSelect = toIgnore

func  runAllSelected(t  *testing.T,  algorithm  private.TppAlgorithm,
selected toSelect) allResult {
  ignore := toIgnore{
     map[uint64]bool{},
     map[uint64]map[uint64]bool{},
     map[uint64]map[uint64]map[uint64]bool{},
     map[uint64]map[uint64]map[uint64]map[uint64]bool{}}
  for nMarkets := range solutions {
     ignore.Nmarkets[nMarkets] = !selected.Nmarkets[nMarkets]
     for nProducts := range solutions[nMarkets] {
        if ignore.Nproducts[nMarkets] == nil {
           ignore.Nproducts[nMarkets] = map[uint64]bool{}
        }
                              ignore.Nproducts[nMarkets][nProducts]    =
!selected.Nproducts[nMarkets][nProducts]
```

```
        for lambda := range solutions[nMarkets][nProducts] {
           if ignore.Nlambdas[nMarkets] == nil {
              ignore.Nlambdas[nMarkets] = map[uint64]map[uint64]bool{}
           }
           if ignore.Nlambdas[nMarkets][nProducts] == nil {
              ignore.Nlambdas[nMarkets][nProducts] = map[uint64]bool{}
           }
                           ignore.Nlambdas[nMarkets][nProducts][lambda]   =
!selected.Nlambdas[nMarkets][nProducts][lambda]
                                          for    variation_    :=    range
solutions[nMarkets][nProducts][lambda] {
              variation := uint64(variation_ + 1)
              if ignore.Nvariation[nMarkets] == nil {
                                           ignore.Nvariation[nMarkets]   =
map[uint64]map[uint64]map[uint64]bool{}
              }
              if ignore.Nvariation[nMarkets][nProducts] == nil {
                                    ignore.Nvariation[nMarkets][nProducts]  =
map[uint64]map[uint64]bool{}
              }
                if ignore.Nvariation[nMarkets][nProducts][lambda] == nil
{
                           ignore.Nvariation[nMarkets][nProducts][lambda]  =
map[uint64]bool{}
              }
               ignore.Nvariation[nMarkets][nProducts][lambda][variation]
= !selected.Nvariation[nMarkets][nProducts][lambda][variation]
           }
        }
     }
  }
  return runAll(t, algorithm, ignore)
}

type lambdaResult struct {
  Results    []variationResult `json:"-"`
  Statistics statistics[intervalStatistic]
}

func runLambda(t *testing.T, algorithm private.TppAlgorithm, nMarkets,
nProducts, lambda uint64, ignore toIgnore) lambdaResult {
  t.Helper()
                  c         :=         make(chan         variationResult,
len(solutions[nMarkets][nProducts][lambda]))
                results        :=        make([]variationResult,        0,
len(solutions[nMarkets][nProducts][lambda]))
  n := 0
  for i_ := range solutions[nMarkets][nProducts][lambda] {
     i := uint64(i_) + 1
     if ignore.Nvariation[nMarkets][nProducts][lambda][i] {
        continue
```

```
      }
      n += 1
      go func() {
         c <- runVariation(t, algorithm, nMarkets, nProducts, lambda, i)
      }()
   }
   for i := 0; i < n; i++ {
      r := <-c
      results = append(results, r)
   }

   nResults := float64(len(results))
        sumCost,    sumMarkets,    sumSeconds    :=    statistic[int64]{},
statistic[int64]{}, statistic[int64]{}
   minCost := minMaxValue{statistic[int64]{Percentage: math.MaxFloat64},
nMarkets, nProducts, lambda, 0}
             maxCost      :=      minMaxValue{statistic[int64]{Percentage:
-math.MaxFloat64}, nMarkets, nProducts, lambda, 0}
            minMarkets     :=     minMaxValue{statistic[int64]{Percentage:
math.MaxFloat64}, nMarkets, nProducts, lambda, 0}
            maxMarkets     :=     minMaxValue{statistic[int64]{Percentage:
-math.MaxFloat64}, nMarkets, nProducts, lambda, 0}
            minSeconds     :=     minMaxValue{statistic[int64]{Percentage:
math.MaxFloat64}, nMarkets, nProducts, lambda, 0}
            maxSeconds     :=     minMaxValue{statistic[int64]{Percentage:
-math.MaxFloat64}, nMarkets, nProducts, lambda, 0}
   for _, result := range results {
      sumCost.Percentage += result.Statistics.Cost.Percentage
      sumCost.Absolute += result.Statistics.Cost.Absolute
      sumMarkets.Percentage += result.Statistics.Markets.Percentage
      sumMarkets.Absolute += result.Statistics.Markets.Absolute
      sumSeconds.Percentage += result.Statistics.Seconds.Percentage
      sumSeconds.Absolute += result.Statistics.Seconds.Absolute
      if result.Statistics.Cost.Percentage < minCost.Value.Percentage {
         minCost.Variation = result.ExpectedSolution.Variation
         minCost.Value.Absolute = result.Statistics.Cost.Absolute
         minCost.Value.Percentage = result.Statistics.Cost.Percentage
      }
      if result.Statistics.Cost.Percentage > maxCost.Value.Percentage {
         maxCost.Variation = result.ExpectedSolution.Variation
         maxCost.Value.Absolute = result.Statistics.Cost.Absolute
         maxCost.Value.Percentage = result.Statistics.Cost.Percentage
      }
                           if     result.Statistics.Markets.Percentage     <
minMarkets.Value.Percentage {
         minMarkets.Variation = result.ExpectedSolution.Variation
         minMarkets.Value.Absolute = result.Statistics.Markets.Absolute
                                        minMarkets.Value.Percentage     =
result.Statistics.Markets.Percentage
      }
```

```
                        if     result.Statistics.Markets.Percentage     >
maxMarkets.Value.Percentage {
        maxMarkets.Variation = result.ExpectedSolution.Variation
        maxMarkets.Value.Absolute = result.Statistics.Markets.Absolute
                                          maxMarkets.Value.Percentage     =
result.Statistics.Markets.Percentage
     }
                        if     result.Statistics.Seconds.Percentage     <
minSeconds.Value.Percentage {
        minSeconds.Variation = result.ExpectedSolution.Variation
        minSeconds.Value.Absolute = result.Statistics.Seconds.Absolute
                                          minSeconds.Value.Percentage     =
result.Statistics.Seconds.Percentage
     }
                        if     result.Statistics.Seconds.Percentage     >
maxSeconds.Value.Percentage {
        maxSeconds.Variation = result.ExpectedSolution.Variation
        maxSeconds.Value.Absolute = result.Statistics.Seconds.Absolute
                                          maxSeconds.Value.Percentage     =
result.Statistics.Seconds.Percentage
     }
  }
     averageCost   :=   statistic[float64]{float64(sumCost.Absolute)   /
nResults, sumCost.Percentage / nResults}
   averageMarkets  :=  statistic[float64]{float64(sumMarkets.Absolute)  /
nResults, sumMarkets.Percentage / nResults}
   averageSeconds  :=  statistic[float64]{float64(sumSeconds.Absolute)  /
nResults, sumSeconds.Percentage / nResults}
      costSum,    marketsSum,    secondsSum    :=    statistic[float64]{},
statistic[float64]{}, statistic[float64]{}
  for _, result := range results {
               diff   :=   float64(result.Statistics.Cost.Absolute)   -
averageCost.Absolute
     costSum.Absolute += diff * diff
              perc   :=   float64(result.Statistics.Cost.Percentage)   -
averageCost.Percentage
     costSum.Percentage += perc * perc
              diff   =   float64(result.Statistics.Markets.Absolute)   -
averageMarkets.Absolute
     marketsSum.Absolute += diff * diff
             perc   =   float64(result.Statistics.Markets.Percentage)   -
averageMarkets.Percentage
     marketsSum.Percentage += perc * perc
              diff   =   float64(result.Statistics.Seconds.Absolute)   -
averageSeconds.Absolute
     secondsSum.Absolute += diff * diff
             perc   =   float64(result.Statistics.Seconds.Percentage)   -
averageSeconds.Percentage
     secondsSum.Percentage += perc * perc
  }
```

```
    costStdError  :=  statistic[float64]{math.Sqrt(costSum.Absolute  /
nResults), math.Sqrt(costSum.Percentage / nResults)}
  marketsStdError := statistic[float64]{math.Sqrt(marketsSum.Absolute /
nResults), math.Sqrt(marketsSum.Percentage / nResults)}
  secondsStdError := statistic[float64]{math.Sqrt(secondsSum.Absolute /
nResults), math.Sqrt(secondsSum.Percentage / nResults)}

  r := lambdaResult{
     results,
     statistics[intervalStatistic]{
            intervalStatistic{sumCost, minCost, averageCost, maxCost,
costStdError},
             intervalStatistic{sumSeconds, maxSeconds, averageSeconds,
maxSeconds, secondsStdError},
             intervalStatistic{sumMarkets, maxMarkets, averageMarkets,
maxMarkets, marketsStdError},
     },
  }

  s := strings.Builder{}
  s.WriteString("#LAMBDA\n")
  s.WriteString(fmt.Sprintf("%s\n", algorithm))
   s.WriteString(fmt.Sprintf("|V|: %d, |K|: %d, L: %d\n", nMarkets,
nProducts, lambda))
  intervalStatisticToTelegram(s.String(), r.Statistics)
    f,   e   :=   os.Create(fmt.Sprintf("%s/lambda.%d.%d.%d.solution",
RESULTS_FOLDER, nMarkets, nProducts, lambda))
  if e != nil {
     notifyTelegram("ERROR: " + e.Error())
     return r
  }
  j, e := json.Marshal(r)
  if e != nil {
     notifyTelegram("ERROR: " + e.Error())
     return r
  }
  _, e = f.Write(j)
  if e != nil {
     notifyTelegram("ERROR: " + e.Error())
     return r
  }
  e = f.Close()
  if e != nil {
     notifyTelegram("ERROR: " + e.Error())
     return r
  }
  return r
}

type intervalStatistic struct {
  Sum      statistic[int64]
```

```
  Min      minMaxValue
  Avg      statistic[float64]
  Max      minMaxValue
  StdError statistic[float64]
}

type minMaxValue struct {
  Value     statistic[int64]
  Nmarkets  uint64
  Nproducts uint64
  Lambda    uint64
  Variation uint64
}


func readTppFile(filename string) private.TppProblem {
  tppInput := private.TppProblem{}
  scanner := openFile(filename)
  markets := readMarketsCoordinates(scanner)
  rootQueues, filtersIndex := readProductsDemand(scanner)
  tppInput.RootQueues = rootQueues
   rootQueues, tppInput.Supplies = assignSuppliesToRootQueues(scanner,
markets, rootQueues, filtersIndex)
  markets = append(markets, markets[0]) // The last market in the cost
matrix is the final destination
  tppInput.CostMatrix = buildCostMatrix(markets)
  tppInput.Markets = markets[1 : len(markets)-1]
  return tppInput // the first and the last one aren't markets
}

func buildCostMatrix(markets []entities.Market) private.TppMatrix {
  costMatrix := make(private.TppMatrix, len(markets))
  for i := range costMatrix {
     costMatrix[i] = make([]private.TppMatrixCell, len(markets))
     for j := range costMatrix[i] {
        costMatrix[i][j].CoordI = i
        costMatrix[i][j].CoordJ = j

        if i == j {
           costMatrix[i][j].Cost = math.MaxUint64
        } else {
           marketA := markets[i]
           marketB := markets[j]
           a := math.Abs(marketA.Latitude - marketB.Latitude)
           b := math.Abs(marketA.Longitude - marketB.Longitude)
           distance := math.Sqrt(a*a + b*b)
           costMatrix[i][j].Cost = uint64(math.Ceil(distance))
        }
     }
  }
  return costMatrix
```

```
}

func    assignSuppliesToRootQueues(scanner    *bufio.Scanner,    markets
[]entities.Market,    rootQueues    []private.ORqueues,    filtersIndex
map[uint64]int) ([]private.ORqueues, map[uint64]*entities.Supply) {
  supplyByID := make(map[uint64]*entities.Supply, len(rootQueues)*10)
  suppliesID := uint64(0)
  for scanner.Scan() {
     txt := scanner.Text()
     if strings.HasPrefix(txt, "EOF") {
        scanner.Scan()
        break
     }
     nOffers := 0
     marketIndex := 0
     marketAndOffers := strings.Split(txt, "  ")
     if len(marketAndOffers) != 2 {
        panic("error reading market offers")
     }
       n, e := fmt.Sscanf(marketAndOffers[0], "%d %d ", &marketIndex,
&nOffers)
     if e != nil {
        panic(e)
     }
     if n != 2 {
        panic("error reading demand of product")
     }
     marketIndex -= 1 // we need a zero-based index
     if nOffers > 0 {
        integersStr := strings.Split(marketAndOffers[1], " ")
        for i_ := 0; i_ < nOffers; i_++ {
           i := 3 * i_
           filterID, _ := strconv.ParseUint(integersStr[i], 10, 64)
           price, _ := strconv.ParseUint(integersStr[i+1], 10, 64)
           quantity, _ := strconv.ParseUint(integersStr[i+2], 10, 64)
           for j := uint64(0); j < quantity; j++ {
              suppliesID++
              s := &entities.Supply{
                 suppliesID,
                 1,
                 1,
                 price,
                 time.Time{},
                 sql.NullTime{},
                 0,
                 markets[marketIndex].ID,
                 time.Now()}
              supplyByID[s.ID] = s
              k := filtersIndex[filterID]
                                          rootQueues[k][0].Supplies    =
append(rootQueues[k][0].Supplies, s)
```

```
                                                rootQueues[k][1].Supplies    =
append(rootQueues[k][1].Supplies, s)
            }
        }
    }
  }
  return rootQueues, supplyByID
}

func  readProductsDemand(scanner  *bufio.Scanner)  ([]private.ORqueues,
map[uint64]int) {
  scanner.Scan()
  nProducts := 0
  n, e := fmt.Sscanf(scanner.Text(), "%d", &nProducts)
  if e != nil {
     panic(e)
  }
  if n != 1 {
     panic("error reading size of products")
  }
  type demand_ struct {
     demand   uint64
     filterID uint64
  }
  productsDemand := make([]demand_, 0, nProducts)
  for scanner.Scan() {
     txt := scanner.Text()
     if strings.HasPrefix(txt, "OFFER_SECTION") {
        break
     }
     quantity := uint64(0)
     filterID := uint64(0)
     n, e = fmt.Sscanf(txt, "%d %d", &filterID, &quantity)
     if e != nil {
        panic(e)
     }
     if n != 2 {
        panic("error reading demand of product")
     }
          productsDemand  =  append(productsDemand,  demand_{quantity,
filterID})
  }
  if len(productsDemand) == 0 {
     panic("error reading products demand")
  }
  rootQueues := make([]private.ORqueues, 0, len(productsDemand))
  filtersIndex := make(map[uint64]int, len(productsDemand))
  for _, f := range productsDemand {
     filterID, demand := f.filterID, f.demand
     filtersIndex[filterID] = len(rootQueues)
     rootQueues = append(rootQueues, private.ORqueues{{
```

```
            demand,
            private.UNITS,
            make([]*entities.Supply, 0, 2*demand)}})
    }
    for i := len(productsDemand) - 1; i >= 0; i-- {
        f := productsDemand[i]
        filterID, demand := f.filterID, f.demand
        j := filtersIndex[filterID]
        rootQueues[j] = append(rootQueues[j], private.SuppliesQueue{
            demand,
            private.UNITS,
            make([]*entities.Supply, 0, 2*demand)})
    }
    return rootQueues, filtersIndex
}

func readMarketsCoordinates(scanner *bufio.Scanner) []entities.Market {
    markets := readMarketsSize(scanner)
    for scanner.Scan() {
        if !strings.HasPrefix(scanner.Text(), "NODE_COORD_SECTION") {
            continue
        }
        for scanner.Scan() {
            txt := scanner.Text()
            if strings.HasPrefix(txt, "DEMAND_SECTION") {
                break
            }
            market := entities.Market{}
                    n,  e  :=  fmt.Sscanf(txt,  "%d  %f  %f",  &market.ID,
&market.Longitude, &market.Latitude)
            if e != nil {
                panic(e)
            }
            if n != 3 {
                panic("error reading market coordinate")
            }
            markets = append(markets, market)
        }
        break
    }
    if len(markets) != cap(markets) {
        panic("error reading the markets coordinates")
    }
    return markets
}

func readMarketsSize(scanner *bufio.Scanner) []entities.Market {
    nMarkets := 0
    for scanner.Scan() {
        if !strings.HasPrefix(scanner.Text(), "DIMENSION") {
            continue
```

```
      }
      n, e := fmt.Sscanf(scanner.Text(), "DIMENSION : %d", &nMarkets)
      if e != nil {
         panic(e)
      }
      if n != 1 {
         panic("error reading the markets size")
      }
      break
   }
   if nMarkets == 0 {
      panic("error reading the markets size")
   }
   return make([]entities.Market, 0, nMarkets)
}

func openFile(filename string) *bufio.Scanner {
   f, e := os.Open(fmt.Sprintf("./files/%s", filename))
   if e != nil {
      panic(e)
   }
   return bufio.NewScanner(f)
}


func BestSavingTSP(marketMatrixIndex int, tour []int, costMatrix
private.TppMatrix) (int, uint64) {
   indexToInsert := -1
   bestCost := uint64(math.MaxUint64)
   for j := 1; j < len(tour); j++ {
      prev := tour[j-1]
      next := tour[j]
                cost := costMatrix[prev][marketMatrixIndex].Cost +
costMatrix[marketMatrixIndex][next].Cost
      if cost < bestCost {
         bestCost = cost
         indexToInsert = j
      }
   }
   return indexToInsert, bestCost
}

func Subsets[T any](slice []T, cb func(subset []T)) {
   for i := 0; i < int(math.Pow(2, float64(len(slice)))); i++ {
      subset := make([]T, 0, len(slice))
      for j := 0; j < len(slice); j++ {
         if i&int(math.Pow(2, float64(j))) > 0 {
            subset = append(subset, slice[j])
         }
      }
      cb(subset)
```

```
  }
}

func  filterCostMatrix(costMatrix  private.TppMatrix,  selectedMarkets
map[uint64]bool,    markets    []outputs.Market)    (private.TppMatrix,
map[int]int) {
  newIndexToOld := make(map[int]int, len(selectedMarkets)+2)
          filteredCostMatrix     :=     make(private.TppMatrix,     0,
len(selectedMarkets)+2)
  for i, row := range costMatrix {
     if i > 0 && i < len(costMatrix)-1 {
        market := markets[i-1]
        if !selectedMarkets[market.ID] {
           continue
        }
     }

     newIndexToOld[len(filteredCostMatrix)] = i
     newRow := make([]private.TppMatrixCell, 0, len(selectedMarkets)+2)
     for j, cell := range row {
        if j > 0 && j < len(costMatrix)-1 {
           market := markets[j-1]
           if !selectedMarkets[market.ID] {
              continue
           }
        }
        cell.CoordI = len(filteredCostMatrix)
        cell.CoordJ = len(newRow)
        newRow = append(newRow, cell)
     }
     filteredCostMatrix = append(filteredCostMatrix, newRow)
  }

  return filteredCostMatrix, newIndexToOld
}

type Market struct {
  Coordinate
  ID           uint64    `json:"id"`
  Name         string    `json:"name"`
  Description  string    `json:"description"`
  Brand        Brand     `json:"brand"`
  Supplies     []*Supply `json:"supplies"`
  CreationDate time.Time `json:"creation_date"`
}

func NewMarket(m entities.Market, c Coordinate, b Brand, s []*Supply)
Market {
  return Market{
     c,
     m.ID,
```

```
      m.Name,
      m.Description,
      b,
      s,
      m.CreationDate,
   }
}

func      RunLKH(costMatrix      private.TppMatrix,      selectedMarkets
map[uint64]bool, markets []outputs.Market) (uint64, []int) {
   var atspFile *os.File
   var solutionFile *os.File
   var parameterFile *os.File
   defer func() {
      if atspFile != nil {
         atspFile.Close()
         os.Remove(atspFile.Name())
      }
      if solutionFile != nil {
         solutionFile.Close()
         os.Remove(solutionFile.Name())
      }
      if parameterFile != nil {
         parameterFile.Close()
         os.Remove(parameterFile.Name())
      }
   }()

       newCostMatrix,   newIndexToOld   :=   filterCostMatrix(costMatrix,
selectedMarkets, markets)
   atspFile = lkhAtspFile(newCostMatrix)
   parameterFile = lkhParameterFile(atspFile.Name())
   c := exec.Command("LKH", parameterFile.Name())
   _, e := c.Output()
   if e != nil {
      panic(e)
   }
   solutionFile, e = os.Open(atspFile.Name() + ".tour")
   if e != nil {
      panic(e)
   }

   cost := uint64(0)
   tour := make([]int, 0, len(newCostMatrix))
   scanner := bufio.NewScanner(solutionFile)
   for scanner.Scan() {
      txt := scanner.Text()
      if strings.HasPrefix(txt, "COMMENT : Length = ") {
         fmt.Sscanf(txt, "COMMENT : Length = %d", &cost)
         continue
      }
```

```
      if strings.HasPrefix(txt, "TOUR_SECTION") {
         for scanner.Scan() {
            txt = scanner.Text()
            if strings.HasPrefix(txt, "-1") {
               break
            }
            i := 0
            fmt.Sscanf(txt, "%d", &i)
            tour = append(tour, i-1)
         }
         break
      }
   }
   if err := scanner.Err(); err != nil {
      panic(err)
   }

   if len(tour) != len(newCostMatrix)+1 {
      panic("unexpected solution tour length")
   }
   if tour[len(newCostMatrix)] != len(newCostMatrix) {
      panic("the last market should be dummy one")
   }
   tour = tour[:len(newCostMatrix)]
   for i, newCostMatrixIndex := range tour {
      tour[i] = newIndexToOld[newCostMatrixIndex]
   }
   return cost, tour
}

type variationResult struct {
   Filename         string
   Algorithm        private.TppAlgorithm
   ExpectedSolution solution
   GivenSolution    tppSolution
   Statistics       statistics[statistic[int64]]
}

func runVariation(t *testing.T, algorithm private.TppAlgorithm,
nMarkets, nProducts, lambda, variation uint64) variationResult {
   t.Helper()
   tpp_, _ := tpp.NewTPP()
      filename := fmt.Sprintf("CapEuclideo.%d.%d.%d.%d.tpp", nMarkets,
nProducts, lambda, variation)
   input := readTppFile(filename)
   s := solutions[nMarkets][nProducts][lambda][variation-1]

   now := time.Now()
   givenSolution, e := tpp_.Run(algorithm, input)
   elapsed := math.Round(time.Since(now).Seconds())
   if e != nil {
```

```
      t.Fatalf("%v\n", e)
   }

   r := variationResult{
      filename,
      algorithm,
      s,
      tppSolution{givenSolution, uint64(elapsed)},
      statistics[statistic[int64]]{
         statistic[int64]{
            int64(givenSolution.Cost - s.OptimalValue),
                              float64(givenSolution.Cost-s.OptimalValue)   /
float64(s.OptimalValue) * 100},
         statistic[int64]{
            int64(elapsed) - int64(s.Seconds),
                               float64(int64(elapsed)-int64(s.Seconds))   /
float64(s.Seconds) * 100},
         statistic[int64]{
            int64(len(givenSolution.Markets)+1) - int64(s.Nstops),
             float64(int64(len(givenSolution.Markets)+1)-int64(s.Nstops))
/ float64(s.Nstops) * 100}}}

   variationResultToTelegram(r)
      f,   e   :=   os.Create(fmt.Sprintf("%s/%s.solution",   RESULTS_FOLDER,
strings.ReplaceAll(filename, "CapEuclideo", "variation")))
   if e != nil {
      notifyTelegram("ERROR: " + e.Error())
      return r
   }
   j, e := json.Marshal(r)
   if e != nil {
      notifyTelegram("ERROR: " + e.Error())
      return r
   }
   _, e = f.Write(j)
   if e != nil {
      notifyTelegram("ERROR: " + e.Error())
      return r
   }
   e = f.Close()
   if e != nil {
      notifyTelegram("ERROR: " + e.Error())
      return r
   }

   return r
}

type tppSolution struct {
   private.TppSolution
   Seconds uint64
```

```
}

type statistics[T any] struct {
  Cost    T
  Seconds T
  Markets T
}

type statistic[T any] struct {
  Absolute   T
  Percentage float64
}

func notifyTelegram(msg string) {
  type params struct {
     ChatID string `json:"chat_id"`
     Text   string `json:"text"`
  }
  body, e := json.Marshal(params{"CHAT_ID", msg})
  if e != nil {
     return
  }
     url   :=  fmt.Sprintf("https://api.telegram.org/bot%s/sendMessage",
"<BOT_ID>")
  http.Post(url, "application/json", bytes.NewBuffer(body))
}

func variationResultToTelegram(r variationResult) {
  s := strings.Builder{}
  s.WriteString("#VARIATION\n")
  s.WriteString(fmt.Sprintf("%s\n", r.Algorithm))
  s.WriteString(fmt.Sprintf("%s\n", r.Filename))
  knownSolution := ""
  if r.ExpectedSolution.Seconds >= 18000 {
     knownSolution = "*"
  }
  signal := ""
  if r.Statistics.Cost.Absolute > 0 {
     signal = "+"
  }
    s.WriteString(fmt.Sprintf("       %sCost:  %d,  %s%0.2f%%  (%s%d)\n",
knownSolution,               r.GivenSolution.Cost,               signal,
r.Statistics.Cost.Percentage, signal, r.Statistics.Cost.Absolute))
  signal = ""
  if r.Statistics.Markets.Absolute > 0 {
     signal = "+"
  }
    s.WriteString(fmt.Sprintf("      Markets:  %d,  %s%0.2f%%  (%s%d)\n",
len(r.GivenSolution.Markets)+1,                                  signal,
r.Statistics.Markets.Percentage,                                 signal,
r.Statistics.Markets.Absolute))
```

```
  signal = ""
  if r.Statistics.Seconds.Absolute > 0 {
     signal = "+"
  }
    s.WriteString(fmt.Sprintf("       Seconds:  %d,  %s%0.2f%%  (%s%d)\n",
r.GivenSolution.Seconds,    signal,    r.Statistics.Seconds.Percentage,
signal, r.Statistics.Seconds.Absolute))

  if r.Statistics.Cost.Absolute == 0 {
     s.WriteString("\n## EXACT SOLUTION ##")
  }

  msg := s.String()
  fmt.Println(msg)
  notifyTelegram(msg)
}

func         intervalStatisticToTelegram(header         string,         r
statistics[intervalStatistic]) {
  s := strings.Builder{}
  s.WriteString(header)
  s.WriteString(lambdaStatisticToString("  Cost:\n", r.Cost))
  s.WriteString(lambdaStatisticToString("  Markets:\n", r.Markets))
  s.WriteString(lambdaStatisticToString("  Seconds:\n", r.Seconds))
  if r.Cost.Avg.Absolute == 0 {
     s.WriteString("\n## EXACT SOLUTION ##")
  }
  msg := s.String()
  fmt.Println(msg)
  notifyTelegram(msg)
}

func  lambdaStatisticToString(header  string,  stats  intervalStatistic)
string {
  s := strings.Builder{}
  s.WriteString(header)
  signal := ""
  if stats.Min.Value.Absolute > 0 {
     signal = "+"
  }
    s.WriteString(fmt.Sprintf("       Min:  %s%0.2f%%  (%s%d),  variation
%d\n",         signal,         stats.Min.Value.Percentage,         signal,
stats.Min.Value.Absolute, stats.Min.Variation))
  signal = ""
  if stats.Avg.Absolute > 0 {
     signal = "+"
  }
    s.WriteString(fmt.Sprintf("       Average:  %s%0.2f%%  (%s%0.2f)\n",
signal, stats.Avg.Percentage, signal, stats.Avg.Absolute))
  signal = ""
  if stats.Max.Value.Absolute > 0 {
```

```
      signal = "+"
   }
    s.WriteString(fmt.Sprintf("       Max:  %s%0.2f%%  (%s%d),  variation
%d\n",         signal,         stats.Max.Value.Percentage,         signal,
stats.Max.Value.Absolute, stats.Max.Variation))
      s.WriteString(fmt.Sprintf("            SD:   %0.2f%%   (%0.2f)\n",
stats.StdError.Percentage, stats.StdError.Absolute))
   return s.String()
}

func first[T comparable, U any](m map[T]U) U {
   for _, u := range m {
      return u
   }
   panic("index out of bounds")
}


func lkhParameterFile(atspFilename string) *os.File {
   parameterFile, e := os.CreateTemp("", "TPP_*.par")
   if e != nil {
      panic(e)
   }
    s  :=  fmt.Sprintf("PROBLEM_FILE  =  %s\nTOUR_FILE  =  %s.tour\nRUNS  =
1\n", atspFilename, atspFilename)
   _, e = parameterFile.WriteString(s)
   if e != nil {
      panic(e)
   }
   return parameterFile
}

func lkhAtspFile(costMatrix private.TppMatrix) *os.File {
   fileContents := strings.Builder{}
   fileContents.WriteString("TYPE : ATSP\n")
         fileContents.WriteString(fmt.Sprintf("DIMENSION     :     %d\n",
len(costMatrix)+1))
   fileContents.WriteString("EDGE_WEIGHT_TYPE: EXPLICIT\n")
   fileContents.WriteString("EDGE_WEIGHT_FORMAT: FULL_MATRIX\n")
   fileContents.WriteString("EDGE_WEIGHT_SECTION\n")
   for i, row := range costMatrix {
      for j, cell := range row {
         if i == j {
            fileContents.WriteString(" 0")
            continue
         }
         if cell.Cost >= lkhIntMax {
               panic("LKH's implementation limits the max cost for the
edges")
         }
         fileContents.WriteString(fmt.Sprintf(" %d", cell.Cost))
```

```
        }
        if i == len(costMatrix)-1 {
            fileContents.WriteString(" 0")
        } else {
            fileContents.WriteString(fmt.Sprintf(" %d", lkhIntMax))
        }
        fileContents.WriteString("\n")
    }
    for i := 0; i < len(costMatrix)+1; i++ {
        if i == 0 || i == len(costMatrix) {
            fileContents.WriteString(" 0")
        } else {
            fileContents.WriteString(fmt.Sprintf(" %d", lkhIntMax))
        }
    }
    fileContents.WriteString("\nEOF\n")

    atspFile, e := os.CreateTemp("", "TPP_*.atsp")
    if e != nil {
        panic(e)
    }
    _, e = atspFile.WriteString(fileContents.String())
    if e != nil {
        panic(e)
    }
    return atspFile
}

const lkhIntMax uint64 = 10000000

func Run(input private.TppProblem) (private.TppSolution, error) {
    h := buildHelpers(input)
    missingDemands, h := ignoreMissingDemand(input, h)
    solution := firstFeasibleSolution(h)
    currentSolution := solution
    factor := uint64(35)
    for {
        for improved := true; improved; {
            dropLen := newDropLen(len(currentSolution.selectedMarkets))
            s := consecutiveExchange(dropLen, currentSolution, h)
            s = insertion(s, h)
            if s.totalCost < currentSolution.totalCost {
                currentSolution = s
            } else {
                improved = false
            }
        }
        if currentSolution.totalCost < solution.totalCost {
            solution = currentSolution
        }
        if factor == 0 {
```

```
            break
        }
         currentSolution_, shouldStop := shaking(factor, currentSolution,
h)
        if shouldStop {
            break
        }
        currentSolution = currentSolution_
        factor -= 1
    }
    s := buildSolution(solution, h)
    s.MissingDemand = missingDemands
    return s, nil
}

func removeNotSelectedMarkets(tour []int, selectedMarkets
map[uint64]bool, h helpers) []int {
    newTour := make([]int, 0, len(selectedMarkets)+2)
    newTour = append(newTour, tour[0])
    for _, curr := range tour[1 : len(tour)-1] {
        market := h.markets[curr-1]
        if selectedMarkets[market.ID] {
            newTour = append(newTour, curr)
        }
    }
    newTour = append(newTour, tour[len(tour)-1])
    return newTour
}

type marketResult struct {
    Results    map[uint64]productResult `json:"-"`
    Statistics statistics[intervalStatistic]
}

func runMarket(t *testing.T, algorithm private.TppAlgorithm, nMarkets
uint64, ignore toIgnore) marketResult {
    t.Helper()
    c := make(chan map[uint64]productResult, len(solutions[nMarkets]))
    results := make(map[uint64]productResult, len(solutions[nMarkets]))
    n := 0
    for nProducts_ := range solutions[nMarkets] {
        nProducts := nProducts_
        if ignore.Nproducts[nMarkets][nProducts] {
            continue
        }
        n += 1
        go func() {
                c <- map[uint64]productResult{nProducts: runProduct(t,
algorithm, nMarkets, nProducts, ignore)}
        }()
```

```
  }
  for i := 0; i < n; i++ {
     r := <-c
     for lambda := range r {
        results[lambda] = r[lambda]
     }
  }

  nResults := 0.0
       sumCost,    sumMarkets,    sumSeconds    :=    statistic[int64]{},
statistic[int64]{}, statistic[int64]{}
  minCost := minMaxValue{statistic[int64]{Percentage: math.MaxFloat64},
nMarkets, 0, 0, 0}
            maxCost      :=      minMaxValue{statistic[int64]{Percentage:
-math.MaxFloat64}, nMarkets, 0, 0, 0}
          minMarkets     :=     minMaxValue{statistic[int64]{Percentage:
math.MaxFloat64}, nMarkets, 0, 0, 0}
          maxMarkets     :=     minMaxValue{statistic[int64]{Percentage:
-math.MaxFloat64}, nMarkets, 0, 0, 0}
          minSeconds     :=     minMaxValue{statistic[int64]{Percentage:
math.MaxFloat64}, nMarkets, 0, 0, 0}
          maxSeconds     :=     minMaxValue{statistic[int64]{Percentage:
-math.MaxFloat64}, nMarkets, 0, 0, 0}
  for _, result := range results {
     for _, lambdaResults := range result.Results {
        nResults += float64(len(lambdaResults.Results))
     }
     sumCost.Percentage += result.Statistics.Cost.Sum.Percentage
     sumCost.Absolute += result.Statistics.Cost.Sum.Absolute
     sumMarkets.Percentage += result.Statistics.Markets.Sum.Percentage
     sumMarkets.Absolute += result.Statistics.Markets.Sum.Absolute
     sumSeconds.Percentage += result.Statistics.Seconds.Sum.Percentage
     sumSeconds.Absolute += result.Statistics.Seconds.Sum.Absolute
                  if    result.Statistics.Cost.Min.Value.Percentage    <
minCost.Value.Percentage {
        minCost.Variation = result.Statistics.Cost.Min.Variation
        minCost.Lambda = result.Statistics.Cost.Min.Lambda
        minCost.Nproducts = result.Statistics.Cost.Min.Nproducts
                                            minCost.Value.Absolute     =
result.Statistics.Cost.Min.Value.Absolute
                                          minCost.Value.Percentage     =
result.Statistics.Cost.Min.Value.Percentage
     }
                  if    result.Statistics.Cost.Max.Value.Percentage    >
maxCost.Value.Percentage {
        maxCost.Variation = result.Statistics.Cost.Max.Variation
        maxCost.Lambda = result.Statistics.Cost.Max.Lambda
        maxCost.Nproducts = result.Statistics.Cost.Max.Nproducts
                                            maxCost.Value.Absolute     =
result.Statistics.Cost.Max.Value.Absolute
```

```
                                        maxCost.Value.Percentage     =
result.Statistics.Cost.Max.Value.Percentage
     }
                if   result.Statistics.Markets.Min.Value.Percentage   <
minMarkets.Value.Percentage {
        minMarkets.Variation = result.Statistics.Markets.Min.Variation
        minMarkets.Lambda = result.Statistics.Markets.Min.Lambda
        minMarkets.Nproducts = result.Statistics.Markets.Min.Nproducts
                                        minMarkets.Value.Absolute     =
result.Statistics.Markets.Min.Value.Absolute
                                    minMarkets.Value.Percentage     =
result.Statistics.Markets.Min.Value.Percentage
     }
                if   result.Statistics.Markets.Max.Value.Percentage   >
maxMarkets.Value.Percentage {
        maxMarkets.Variation = result.Statistics.Seconds.Max.Variation
        maxMarkets.Lambda = result.Statistics.Seconds.Max.Lambda
        maxMarkets.Nproducts = result.Statistics.Seconds.Max.Nproducts
                                        maxMarkets.Value.Absolute     =
result.Statistics.Markets.Max.Value.Absolute
                                    maxMarkets.Value.Percentage     =
result.Statistics.Markets.Max.Value.Percentage
     }
                if   result.Statistics.Seconds.Min.Value.Percentage   <
minSeconds.Value.Percentage {
        minSeconds.Variation = result.Statistics.Markets.Min.Variation
        minSeconds.Lambda = result.Statistics.Markets.Min.Lambda
        minSeconds.Nproducts = result.Statistics.Markets.Min.Nproducts
                                        minSeconds.Value.Absolute     =
result.Statistics.Seconds.Min.Value.Absolute
                                    minSeconds.Value.Percentage     =
result.Statistics.Seconds.Min.Value.Percentage
     }
                if   result.Statistics.Seconds.Max.Value.Percentage   >
maxSeconds.Value.Percentage {
        maxSeconds.Variation = result.Statistics.Seconds.Max.Variation
        maxSeconds.Lambda = result.Statistics.Seconds.Max.Lambda
        maxSeconds.Nproducts = result.Statistics.Seconds.Max.Nproducts
                                        maxSeconds.Value.Absolute     =
result.Statistics.Seconds.Max.Value.Absolute
                                    maxSeconds.Value.Percentage     =
result.Statistics.Seconds.Max.Value.Percentage
     }
  }
     averageCost   :=   statistic[float64]{float64(sumCost.Absolute)   /
nResults, sumCost.Percentage / nResults}
   averageMarkets  :=  statistic[float64]{float64(sumMarkets.Absolute)  /
nResults, sumMarkets.Percentage / nResults}
   averageSeconds  :=  statistic[float64]{float64(sumSeconds.Absolute)  /
nResults, sumSeconds.Percentage / nResults}
```

```
      costSum,   marketsSum,   secondsSum   :=   statistic[float64]{},
statistic[float64]{}, statistic[float64]{}
  for _, result := range results {
            diff   :=   float64(result.Statistics.Cost.Sum.Absolute)   -
averageCost.Absolute
     costSum.Absolute += diff * diff
           perc   :=   float64(result.Statistics.Cost.Sum.Percentage)   -
averageCost.Percentage
     costSum.Percentage += perc * perc
           diff   =   float64(result.Statistics.Markets.Sum.Absolute)   -
averageMarkets.Absolute
     marketsSum.Absolute += diff * diff
          perc   =   float64(result.Statistics.Markets.Sum.Percentage)   -
averageMarkets.Percentage
     marketsSum.Percentage += perc * perc
           diff   =   float64(result.Statistics.Seconds.Sum.Absolute)   -
averageSeconds.Absolute
     secondsSum.Absolute += diff * diff
          perc   =   float64(result.Statistics.Seconds.Sum.Percentage)   -
averageSeconds.Percentage
     secondsSum.Percentage += perc * perc
  }
    costStdError   :=   statistic[float64]{math.Sqrt(costSum.Absolute   /
nResults), math.Sqrt(costSum.Percentage / nResults)}
  marketsStdError := statistic[float64]{math.Sqrt(marketsSum.Absolute /
nResults), math.Sqrt(marketsSum.Percentage / nResults)}
  secondsStdError := statistic[float64]{math.Sqrt(secondsSum.Absolute /
nResults), math.Sqrt(secondsSum.Percentage / nResults)}
  r := marketResult{
     results,
     statistics[intervalStatistic]{
           intervalStatistic{sumCost,  minCost,  averageCost,  maxCost,
costStdError},
            intervalStatistic{sumSeconds,  minSeconds,  averageSeconds,
maxSeconds, secondsStdError},
            intervalStatistic{sumMarkets,  minMarkets,  averageMarkets,
maxMarkets, marketsStdError},
     },
  }

  s := strings.Builder{}
  s.WriteString("#MARKET\n")
  s.WriteString(fmt.Sprintf("%s\n", algorithm))
  s.WriteString(fmt.Sprintf("|V|: %d\n", nMarkets))
  intervalStatisticToTelegram(s.String(), r.Statistics)
       f,    e    :=    os.Create(fmt.Sprintf("%s/nMarket.%d.solution",
RESULTS_FOLDER, nMarkets))
  if e != nil {
     notifyTelegram("ERROR: " + e.Error())
     return r
  }
```

```
  j, e := json.Marshal(r)
  if e != nil {
     notifyTelegram("ERROR: " + e.Error())
     return r
  }
  _, e = f.Write(j)
  if e != nil {
     notifyTelegram("ERROR: " + e.Error())
     return r
  }
  e = f.Close()
  if e != nil {
     notifyTelegram("ERROR: " + e.Error())
     return r
  }
  return r
}


type productResult struct {
  Results    map[uint64]lambdaResult `json:"-"`
  Statistics statistics[intervalStatistic]
}

func runProduct(t *testing.T, algorithm private.TppAlgorithm, nMarkets,
nProducts uint64, ignore toIgnore) productResult {
  t.Helper()
               c       :=       make(chan       map[uint64]lambdaResult,
len(solutions[nMarkets][nProducts]))
                  results         :=         make(map[uint64]lambdaResult,
len(solutions[nMarkets][nProducts]))
  n := 0
  for lambda_ := range solutions[nMarkets][nProducts] {
     lambda := lambda_
     if ignore.Nlambdas[nMarkets][nProducts][lambda] {
        continue
     }
     n += 1
     go func() {
           c <- map[uint64]lambdaResult{lambda: runLambda(t, algorithm,
nMarkets, nProducts, lambda, ignore)}
     }()
  }
  for i := 0; i < n; i++ {
     r := <-c
     for lambda := range r {
        results[lambda] = r[lambda]
     }
  }

  nResults := 0.0
```

```
        sumCost,    sumMarkets,    sumSeconds    :=    statistic[int64]{},
statistic[int64]{}, statistic[int64]{}
  minCost := minMaxValue{statistic[int64]{Percentage: math.MaxFloat64},
nMarkets, nProducts, 0, 0}
           maxCost      :=      minMaxValue{statistic[int64]{Percentage:
-math.MaxFloat64}, nMarkets, nProducts, 0, 0}
          minMarkets     :=     minMaxValue{statistic[int64]{Percentage:
math.MaxFloat64}, nMarkets, nProducts, 0, 0}
          maxMarkets     :=     minMaxValue{statistic[int64]{Percentage:
-math.MaxFloat64}, nMarkets, nProducts, 0, 0}
          minSeconds     :=     minMaxValue{statistic[int64]{Percentage:
math.MaxFloat64}, nMarkets, nProducts, 0, 0}
          maxSeconds     :=     minMaxValue{statistic[int64]{Percentage:
-math.MaxFloat64}, nMarkets, nProducts, 0, 0}
  for _, result := range results {
     nResults += float64(len(result.Results))
     sumCost.Percentage += result.Statistics.Cost.Sum.Percentage
     sumCost.Absolute += result.Statistics.Cost.Sum.Absolute
     sumMarkets.Percentage += result.Statistics.Markets.Sum.Percentage
     sumMarkets.Absolute += result.Statistics.Markets.Sum.Absolute
     sumSeconds.Percentage += result.Statistics.Seconds.Sum.Percentage
     sumSeconds.Absolute += result.Statistics.Seconds.Sum.Absolute
                  if    result.Statistics.Cost.Min.Value.Percentage    <
minCost.Value.Percentage {
        minCost.Variation = result.Statistics.Cost.Min.Variation
        minCost.Lambda = result.Statistics.Cost.Min.Lambda
                                            minCost.Value.Absolute     =
result.Statistics.Cost.Min.Value.Absolute
                                          minCost.Value.Percentage     =
result.Statistics.Cost.Min.Value.Percentage
     }
                  if    result.Statistics.Cost.Max.Value.Percentage    >
maxCost.Value.Percentage {
        maxCost.Variation = result.Statistics.Cost.Max.Variation
        maxCost.Lambda = result.Statistics.Cost.Max.Lambda
                                            maxCost.Value.Absolute     =
result.Statistics.Cost.Max.Value.Absolute
                                          maxCost.Value.Percentage     =
result.Statistics.Cost.Max.Value.Percentage
     }
                if    result.Statistics.Markets.Min.Value.Percentage   <
minMarkets.Value.Percentage {
        minMarkets.Variation = result.Statistics.Markets.Min.Variation
        minMarkets.Lambda = result.Statistics.Markets.Min.Lambda
                                         minMarkets.Value.Absolute     =
result.Statistics.Markets.Min.Value.Absolute
                                       minMarkets.Value.Percentage     =
result.Statistics.Markets.Min.Value.Percentage
     }
                if    result.Statistics.Markets.Max.Value.Percentage   >
maxMarkets.Value.Percentage {
```

```
        maxMarkets.Variation = result.Statistics.Seconds.Max.Variation
        maxMarkets.Lambda = result.Statistics.Seconds.Max.Lambda
                                            maxMarkets.Value.Absolute     =
result.Statistics.Markets.Max.Value.Absolute
                                           maxMarkets.Value.Percentage     =
result.Statistics.Markets.Max.Value.Percentage
     }
                if   result.Statistics.Seconds.Min.Value.Percentage   <
minSeconds.Value.Percentage {
        minSeconds.Variation = result.Statistics.Markets.Min.Variation
        minSeconds.Lambda = result.Statistics.Markets.Min.Lambda
                                            minSeconds.Value.Absolute     =
result.Statistics.Seconds.Min.Value.Absolute
                                           minSeconds.Value.Percentage     =
result.Statistics.Seconds.Min.Value.Percentage
     }
                if   result.Statistics.Seconds.Max.Value.Percentage   >
maxSeconds.Value.Percentage {
        maxSeconds.Variation = result.Statistics.Seconds.Max.Variation
        maxSeconds.Lambda = result.Statistics.Seconds.Max.Lambda
                                            maxSeconds.Value.Absolute     =
result.Statistics.Seconds.Max.Value.Absolute
                                           maxSeconds.Value.Percentage     =
result.Statistics.Seconds.Max.Value.Percentage
     }
  }
     averageCost   :=   statistic[float64]{float64(sumCost.Absolute)   /
nResults, sumCost.Percentage / nResults}
   averageMarkets  :=  statistic[float64]{float64(sumMarkets.Absolute)  /
nResults, sumMarkets.Percentage / nResults}
   averageSeconds  :=  statistic[float64]{float64(sumSeconds.Absolute)  /
nResults, sumSeconds.Percentage / nResults}
      costSum,   marketsSum,   secondsSum   :=   statistic[float64]{},
statistic[float64]{}, statistic[float64]{}
  for _, result := range results {
            diff   :=   float64(result.Statistics.Cost.Sum.Absolute)   -
averageCost.Absolute
     costSum.Absolute += diff * diff
           perc   :=   float64(result.Statistics.Cost.Sum.Percentage)   -
averageCost.Percentage
     costSum.Percentage += perc * perc
           diff   =   float64(result.Statistics.Markets.Sum.Absolute)   -
averageMarkets.Absolute
     marketsSum.Absolute += diff * diff
          perc   =   float64(result.Statistics.Markets.Sum.Percentage)   -
averageMarkets.Percentage
     marketsSum.Percentage += perc * perc
           diff   =   float64(result.Statistics.Seconds.Sum.Absolute)   -
averageSeconds.Absolute
     secondsSum.Absolute += diff * diff
```

```
            perc  =  float64(result.Statistics.Seconds.Sum.Percentage)  -
averageSeconds.Percentage
        secondsSum.Percentage += perc * perc
    }
        costStdError  :=  statistic[float64]{math.Sqrt(costSum.Absolute  /
nResults), math.Sqrt(costSum.Percentage / nResults)}
    marketsStdError := statistic[float64]{math.Sqrt(marketsSum.Absolute /
nResults), math.Sqrt(marketsSum.Percentage / nResults)}
    secondsStdError := statistic[float64]{math.Sqrt(secondsSum.Absolute /
nResults), math.Sqrt(secondsSum.Percentage / nResults)}
    r := productResult{
        results,
        statistics[intervalStatistic]{
                intervalStatistic{sumCost,  minCost,  averageCost,  maxCost,
costStdError},
                intervalStatistic{sumSeconds,  minSeconds,  averageSeconds,
maxSeconds, secondsStdError},
                intervalStatistic{sumMarkets,  minMarkets,  averageMarkets,
maxMarkets, marketsStdError},
        },
    }

    s := strings.Builder{}
    s.WriteString("#PRODUCT\n")
    s.WriteString(fmt.Sprintf("%s\n", algorithm))
    s.WriteString(fmt.Sprintf("|V|: %d, |K|: %d\n", nMarkets, nProducts))
    intervalStatisticToTelegram(s.String(), r.Statistics)
        f,    e    :=    os.Create(fmt.Sprintf("%s/nProduct.%d.%d.solution",
RESULTS_FOLDER, nMarkets, nProducts))
    if e != nil {
        notifyTelegram("ERROR: " + e.Error())
        return r
    }
    j, e := json.Marshal(r)
    if e != nil {
        notifyTelegram("ERROR: " + e.Error())
        return r
    }
    _, e = f.Write(j)
    if e != nil {
        notifyTelegram("ERROR: " + e.Error())
        return r
    }
    e = f.Close()
    if e != nil {
        notifyTelegram("ERROR: " + e.Error())
        return r
    }
    return r
}
```

```
func tourCost(tour []int, h helpers) uint64 {
  newTravelCost := uint64(0)
  for i, curr := range tour[1:] {
     prev := tour[i]
     newTravelCost += h.costMatrix[prev][curr].Cost
  }
  return newTravelCost
}

func supplyUnits(supply *outputs.Supply, demandType private.UnitType)
uint64 {
  units := supply.ProductUnits
  if demandType != private.UNITS {
     units = supply.Units
  }
  return units
}

func switchMapsPointers[T comparable, U any](dest *map[T]U, source
*map[T]U) {
  aux := *dest
  *dest = *source
  *source = aux
}

func emptyMap[T comparable, U any](dest map[T]U) {
  for key := range dest {
     delete(dest, key)
  }
}

func moveMap[T comparable, U any](dest map[T]U, source map[T]U) {
  for key, value := range source {
     dest[key] = value
     delete(source, key)
  }
}

func sortSuppliesQueue(supplies []*outputs.Supply, missingDemand
uint64, demandType private.UnitType) {
  sort.Slice(supplies, func(i, j int) bool {
     supplyI, supplyJ := supplies[i], supplies[j]
                          priceI := float64(supplyI.Price) /
math.Min(float64(supplyI.ProductUnits), float64(missingDemand))
                          priceJ := float64(supplyJ.Price) /
math.Min(float64(supplyJ.ProductUnits), float64(missingDemand))
     if demandType != private.UNITS {
                             priceI = float64(supplyI.Price) /
math.Min(float64(supplyI.Units), float64(missingDemand))
                             priceJ = float64(supplyJ.Price) /
math.Min(float64(supplyJ.Units), float64(missingDemand))
```

```
      }
      return priceI < priceJ
  })
}

const (
  LSH = TppAlgorithm("LSH - Local Search Heuristic")
)

type TppAlgorithm string
type TppMatrix = [][]TppMatrixCell

type TppProblem struct {
  CostMatrix TppMatrix
  RootQueues RootQueues
  Markets    []outputs.Market
  Supplies   map[uint64]*outputs.Supply
}
type TppMatrixCell struct {
  CoordI   int
  CoordJ   int
  Cost     uint64
  Distance float64
  Duration float64
}
type TppSolution struct {
  FinalCost     uint64
  Cost          uint64
  Tax           uint64
  ProductsCost  uint64
  RouteCost     uint64
  RouteDistance float64
  RouteDuration float64
  Markets       []outputs.Market
  MissingDemand [][]uint64
  BestSingle    *TppSolution
}


func ignoreMissingDemand(input private.TppProblem, h helpers)
([][]uint64, helpers) {
      s := feasibleSolution{selectedMarkets: make(map[uint64]bool,
len(input.Markets))}
  for _, market := range input.Markets {
      s.selectedMarkets[market.ID] = true
  }
  _, missingDemands := computeMissingDemand(s, h)
  for rootQueue, ORqueues := range missingDemands {
      for ORqueue, missingDemand := range ORqueues {
```

```
                h.rootQueues[rootQueue].SuppliesQueues[ORqueue].Demand  -=
missingDemand
      }
   }
   return missingDemands, h
}

func firstFeasibleSolution(h helpers) feasibleSolution {
   s := feasibleSolution{
      0, 0, 0, nil,
      make(map[uint64]bool, len(h.markets)),
      make(map[uint64]bool, len(h.supplies))}
   s = calculateProductsCost(s, h)
    s.travelCost, s.tour = utils.RunLKH(h.costMatrix, s.selectedMarkets,
h.markets)
   s.totalCost = s.travelCost + s.productsCost
   return s
}
```