UNIVERSIDADE VEIGA DE ALMEIDA – UVA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

INTRODUÇÃO À HEURÍSTICA DA RECIPROCIDADE AO PROBLEMA DO CAIXEIRO VIAJANTE

LUCAS LOPES FELIPE

RIO DE JANEIRO

2017

UNIVERSIDADE VEIGA DE ALMEIDA - UVA

LUCAS LOPES FELIPE

Monografia apresentada ao curso de Ciência da Computação da Universidade Veiga de Almeida, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientadora: MSc ANA MARIA DOS SANTOS VIANNA

INTRODUÇÃO À HEURÍSTICA DA RECIPROCIDADE AO PROBLEMA DO CAIXEIRO VIAJANTE

RIO DE JANEIRO

2017

UVA | MEMBRO DA REDE ILUMNO

UNIVERSIDADE VEIGA DE ALMEIDA - UVA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS LOPES FELIPE

INTRODUÇÃO À HEURÍSTICA DA RECIPROCIDADE AO PROBLEMA DO CAIXEIRO VIAJANTE

Monografia apresentada como requisito parcial à conclusão do curso em Bacharel em Ciência da Computação.

APROVADA EM:

CONCEITO: 10,0 (DEZ)

BANCA EXAMINADORA:

Ana Maria Vianna

PROF. MSc ANA MARIA DOS SANTOS VIANNA
ORIENTADORA

PROF. MSc CAMILLA LOBO PAULINO

PROF. DSc LUIZ HENRIQUE DE SOUZA AGUIAR COUTINHO

Coordenação de Ciência da Computação

Rio de Janeiro

Pelo apoio e confiança, dedico este trabalho à minha noiva.

AGRADECIMENTOS

Todo grande feito é graças a colaboração de outros indivíduos, ainda que indiretamente, mas para este trabalho houve em especial o apoio direto de três pessoas, a quem devo o agradecimento.

A professora Ana Maria, minha amiga e professora, que me estendeu a mão no último dia que deveria decidir o orientador da monografia. E que não apenas se intitulou, mas que de fato orientou, dedicando seu tempo, esforço e conhecimento para o progresso deste trabalho.

O professor Daniel Shiffman do NYU, que apesar de não me conhecer, seus vídeos no *YouTube* foram de suma importância para meu aprendizado, foi ele que me apresentou e ensinou o *framework* que utilizo neste trabalho. Me despertou um novo jeito de enxergar a programação e inclusive reativou minha curiosidade quanto ao problema do caixeiro viajante.

Flávia Bechtlufft, a pessoa que quero estar junto pelo resto da minha vida, que soube administrar bem meu tempo, já que por mim eu o gastaria todo ao seu lado. Me motivou e me passou confiança, me cobrou e participou de todas as etapas do longo processo, que ouviu a todas minhas ideias e questionamentos.

Outras pessoas também foram de grande importância e participaram deste trajeto, como minha família e amigos, mas essas três foram as que tiveram maior impacto no desenvolvimento de minha monografia.

“A amizade é uma predisposição recíproca que torna dois seres igualmente ciosos da felicidade um do outro”

Platão

RESUMO

O Problema do Caixeiro Viajante é um dos problemas mais populares da matemática e consequentemente da computação, seu objetivo é encontrar a rota de menor distância que passe por um conjunto de cidades, mas sem repetir nenhuma. Sua popularidade se da por até hoje não ter sido encontrado um método que resulte na solução exata em um tempo viável, visto que por ser um problema de otimização combinatória, onde o número de possíveis rotas é dado pelo fatorial do número de cidades, é inviável, mesmo para um computador, comparar todas em um tempo hábil a fim de decidir qual é a melhor. Métodos alternativos que buscam soluções satisfatórias dentro de um tempo plausível, tornam-se populares, como por exemplo o uso de heurísticas. Nesta monografia é apresentado a heurística da reciprocidade, onde considera que cada cidade pertence a um grupo; grupos possuem o "desejo" de conectar-se com o que esteja mais próximo de si; dois grupos são unidos quando o desejo de conexão entre eles for mútuo; o processo é repetido até que haja apenas um único grupo contendo todas as cidades conectadas. A heurística foi implementada utilizando a linguagem de programação *JavaSript*, e apresentou desempenho bem satisfatório, resolvendo por exemplo, 5000 cidades em cerca de 5 segundos. Sua solução pode ser considerada boa de forma independente, porém, pode apresentar resultados ainda melhores se combinada com o método que utiliza Algoritmo Genético, servindo como base para a primeira geração.

Palavras-Chave: Problema do Caixeiro Viajante, Heurística, Otimização

ABSTRACT

The Travelling Salesman Problem is one of the most popular math, and consequently, computation problems, its objective is to find the shortest path that visit a set of cities, but without repeating any. Its popularity is because no method has been stated that results in the exact solution in a viable time, since for being a combinatorial optimization problem, where the number of possible routes is given by the factorial of the number of cities, it is impracticable, even for a computer, compare all routes in able time, in order to decide which is the best. Alternative methods that search for satisfactory solutions in a plausible time, have become popular, such as the use of heuristics. In this work is presented the heuristic of reciprocity, where it considers that each city belongs to a group; groups have the "desire" to connect with the closest to them; two groups are united when the desire for connection between them is mutual; the process is repeated until there is only one group containing all connected cities. The heuristic was implemented using the JavaSript programming language, and presented very satisfactory performance, solving, for example, 5000 cities in about 5 seconds. Its solution can be considered good independently, however, it can present even better results if combined with the method that uses Genetic Algorithm, serving as base for the first generation.

Keywords: Travelling Salesman Problem, Heuristic, Optimization

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

PCV – Problema do Caixeiro Viajante

PVC-PB – Problema do Caixeiro Viajante Preto e Branco

TSP – Travelling Salesman Problem

AG – Algoritmo Genético

SUMÁRIO

1 INTRODUÇÃO

O Problema do Caixeiro Viajante é um dos mais conhecidos da ciência da computação. Sua fama se justifica por ser antigo, amplamente explorado e, mesmo assim, ainda seguir em aberto. É uma referência para áreas de estudo como Teoria dos Grafos e Otimização, e o máximo que atingimos até hoje foram soluções que entregam uma resposta razoável dentro de um tempo delimitado.

Seu objetivo é simples de ser explicado, mas muito complexo de ser executado. Trata-se de encontrar a melhor trajetória a ser percorrida por um caixeiro viajante (vendedor ambulante) em um conjunto de cidades, passando por todas e sem repetir nenhuma, a fim de obter a rota com a menor distância total. A complexidade está no intenso número de possibilidades, pois para cada nova cidade adicionada ao problema, o número aumenta fatorialmente.

Mesmo utilizando computadores, que conseguem executar várias tarefas muito rápido, a vasta possibilidade de rotas torna impraticável listar todos os percursos e, assim, compara-los a fim de encontrar o melhor. Como por exemplo, de acordo com o Instituto de Matemática e Estatística da UFRGS[1] se utilizarmos um computador capaz de efetuar 1 bilhão de adições por segundo, o mesmo levaria 73 anos para comparar todas as possíveis rotas dentre apenas 20 cidades. E se aumentarmos o número de cidades para 25, o mesmo levaria 470 milhões de anos.

O problema disso é que grande parte dessas possíveis soluções não são convenientes, porque fazem uma trajetória com um considerável grau de ineficiência, consumindo grande poder computacional nessas comparações. Em contrapartida, se dermos o problema para ser resolvido por um humano, ainda que ele não trace a melhor rota possível, é provável que intuitivamente, a simples observação dos elementos do problema venha a promover a indicação de uma solução muito mais eficaz que as diversas rotas testadas pelo computador.

Esta monografia toma como hipótese que, se o humano não precisa comparar todas as rotas para indicar uma solução razoável, isto é, uma melhor que as inadequadas, o

[1] http://www.mat.ufrgs.br/~portosil/caixeiro.html - data de acesso: 01/06/17

computador também seria capaz de adotar o mesmo padrão. Dessa forma, como proposta, é apresentada uma heurística, ou seja, uma estratégia que visa facilitar a busca pela solução, ou como Talbi (2009) descreve, um método aproximado, que explore a relação entre a qualidade da resposta com o respectivo tempo de processamento, mesmo quando trabalhado com milhares de cidades.

Essa proposta surge como inspiração a resolução processada mentalmente por um humano racional, de modo que ao invés de programar um código que compare todas as rotas, sua execução visa simular os procedimentos realizados pelo humano, ou ao menos, características similares a tal.

O trabalho foi organizado para que no segundo capítulo seja abordado as características do problema, no terceiro, as soluções popularmente conhecidas, e no quarto, é apresentada a heurística em si.

2 SOBRE O PROBLEMA

Este capítulo visa explorar o problema em si, para tal, será abordado sua definição, história, cenário em que se encontra, categoria de problemas a qual pertence, descrição de sua complexidade e sua relevância para matemática e computação.

2.1 O CAIXEIRO VIAJANTE

Apesar do fácil entendimento sobre sua finalidade, a explicação dada na introdução não contempla todos aspectos que o problema tem. Dessa forma, exploremos mais sobre suas características, a começar por uma visão mais contextualizada de sua aplicabilidade, depois revisando como a atual literatura o descreve e, por fim, conhecendo sua história e suas variações.

2.1.1 Contextualizando

Com o objetivo de melhor ilustrar o problema, vamos supor a seguinte estória. É noite de Natal e o Papai Noel precisa se apressar, pois precisa entregar todos os presentes nas casas das crianças que se comportaram ao longo do ano em apenas uma noite. Ele sairá do Polo Norte e precisará passar por diversas casas por todo planeta, até que finamente retorne à sua.

O trajeto que ele terá de fazer precisa ser muito bem escolhido, afinal, há uma limitação de tempo para cumprir o trabalho por completo. Após alguns debates com seus duendes, chegaram à conclusão que o critério de decisão a ser seguido ao longo da noite, é, estando em uma casa, a seguinte a ser visitada deverá ser a mais próxima da atual, em termos de distância, podendo escolher livremente em casos de empate.

Afinal, seria um trajeto bem maior caso saísse, por exemplo, de São Paulo, fosse para Dubai, retornasse ao Brasil para entregar no Rio de Janeiro, e depois seguisse novamente para a Ásia, continuando as entregas.

Figura 1: Trajeto ineficiente realizado pelo Papai Noel
Fonte: autor

Ao mesmo tempo, sendo a escolha da próxima casa ser baseada na que possui a menor distância da atual, ele pode acabar traçando uma linha descendo o continente da América e depois ter que subir porque deixou as casas que eram a segunda melhor opção de lado, e descer novamente para compensar as outras que sobraram. Ele estaria tomando como decisão, apenas a melhor opção atual, não visando o resultado como um todo.

Esse critério adotado pelo Papai Noel e seus duendes é uma heurística, que busca adotar uma prática diferente do tradicional método de comparação entre todas as possibilidades. Mesmo não obtendo a melhor solução, ela encontra uma resposta razoável com muito mais rapidez, o que nesse caso, onde há inúmeras casa, é bem mais preferível que utilizar o método exato, (Talbi, 2009) que apesar de obter a melhor rota, gasta um tempo extraordinário comparando todas as possibilidades.

2.1.2 Na literatura

O exemplo do Papai Noel foi uma forma de abstrair o Problema do Caixeiro Viajante, nele o caixeiro é substituído pelo bom velhinho e as cidades pelas casas das crianças. Aliás, o próprio exemplo do caixeiro viajante é uma abstração do real problema matemático, cujo trata-se de um problema de otimização ilustrado através de grafos planares.

A Teoria dos Grafos surgiu em 1736 com a publicação do artigo *O Problema das Sete Pontes de Königsberg* por Leonhard Euler. É um ramo da matemática que estuda as relações entre os objetos de um determinado conjunto. As estruturas denominadas grafos podem ser descritas como *G(V, E)*, onde V representa um conjunto não vazio de vértices e E um subconjunto de V que representa as arestas.

Figura 2: O Problema das Sete Pontes de Königsberg
Fonte: Historic Cities Research Project

Trazendo para o PCV, as cidades seriam os vértices e as conexões entre elas, as arestas. E por se tratar de um grafo onde todas as cidades, isto é, vértices, devam ser visitadas ao percorrer de um circuito, dizemos, portanto, que é um circuito hamiltoniano. Podemos definir matematicamente o PCV como "Dados um grafo G e um custo c_e em $\mathbb{Q} \geq$ para cada aresta e , determina um circuito hamiltoniano C que minimize $c(C)$" (ANDRETTA, 2015, p.2)

Percorrer grafos não é exclusividade do PCV, Edsger Dijkstra, um cientista da compunção holandês, publicou em 1959 o artigo *A note on two problems in connection with graphs*, onde solucionava o problema de encontrar o caminho mais curto entre dois vértices de um grafo, dirigido ou não, isto é, quando as arestas possuem sentido de deslocamento, e com pesos não negativos, onde os pesos representam o custo de deslocamento por uma determinada aresta.

Na ocasião, o grafo já possui arestas definidas, e que pode não haver uma conexão direta entre dois vértices, necessitando assim encontrar um caminho por onde a partir de um, passar por outros, até chegar ao vértice de destino. O que não serve para resolver o PCV pois neste não há a exigência de percorrer todos os vértices, pelo contrário, o objetivo é encontrar o menor percurso entre dois.

O algoritmo que resolve o mesmo problema que o de Dijkstra, mas com pesos negativos é "o algoritmo de Ford-Moore-Bellman, assim denominado em homenagem aos trabalhos simultâneos dos pesquisadores Lester Ford (1956), Edward Moore (1957) e Richard Bellman (1958)" (GOLDBARG e GOLDBARG, 2012)

Há também o algoritmo de Kruskal (1956), onde dado um grafo com pesos, ele encontra a conexão mínima que inclua todos os vértices, similar o PCV, porém aqui não há a restrição ser realizado um circuito, isto é, um caminho único que passe por todos. Primeiro as arestas são

ordenadas de forma crescente de acordo com o valor de seus pesos, e em seguida são utilizadas as arestas até que todos os vértices pertençam a um grupo, mas ignorando aquelas que conectam dois vértices que já pertençam a um mesmo subgrupo, ou seja.

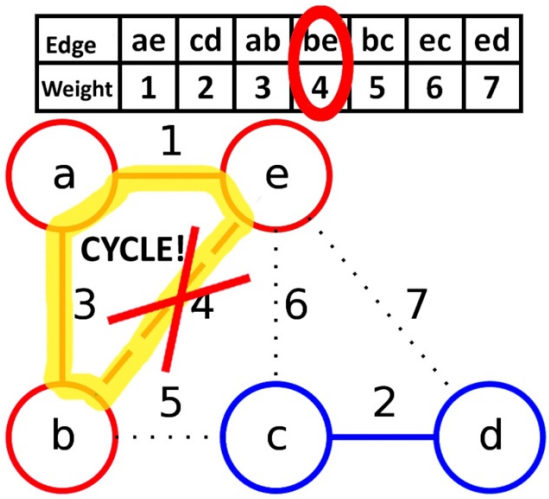

Figura 3: Aresta 4 é descartada pois une dois vértices de um mesmo grupo
Fonte: Wikipédia

Como pode ser observado na Figura 3, a aresta de peso um (que liga os vértices A com E) é utilizada, em seguida a de peso 2, 3, e quando chega na de peso 4, está é descartada pois conecta dois vértices que já pertençam a um grupo. O algoritmo pode ser descrito através do pseudocódigo:

```
1. Crie uma floresta F (um conjunto de árvores), onde cada
   vértice no grafo é uma árvore separada
2. Crie um conjunto S contendo todas as arestas do grafo
3. Enquanto S for não-vazio, faça:
   3.1.   Remova uma aresta com peso mínimo de S
   3.2.   Se essa aresta conecta duas árvores diferentes,
          adicione-a à floresta, combinando duas árvores numa única
          árvore parcial
   3.3.   Do contrário, descarte a aresta
```

Código 1: Pseudocódigo do algoritmo de Kruskal
Fonte: Wikipédia

Note na Figura 4 como, apesar de todos os vértices estarem conectados utilizando o mínimo de arestas possíveis através do algoritmo de Kruskal, este não representa o PCV, pois as conexões não são feitas através de um circuito hamiltoniano, onde cada vértice pode ter no máximo duas conexões, a de onde chegou até ele, e para onde ele leva.

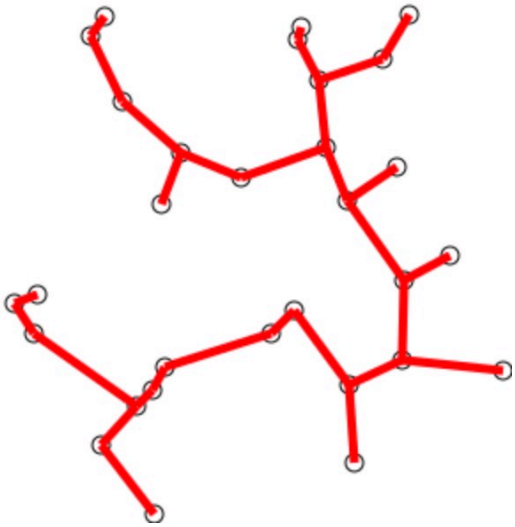

Figura 4: Grafo conectado de forma mínima através do algoritmo de Kruskal
Fonte: Wikipédia

O algoritmo de Kruskal é considerado um algoritmo guloso, isto é, um algoritmo que tenta resolver o problema fazendo a melhor escolha local em cada fase na expectativa de encontrar a melhor solução global, o que para problemas combinatórios, não é comum de acontecer. Como é o caso do PCV, onde escolher o próximo melhor caminho, acaba não resultando no melhor caminho possível.

2.1.3 Sua História

Caixeiro-viajante foi uma antiga e popular profissão em que a pessoa vendia produtos fora do local de onde eles eram produzidos, pois devido a dificuldade de transporte entre cidades, os caixeiros-viajantes eram uma forma de transportar produtos entre regiões diferentes das grandes cidades.

Mas a profissão em si não tem a ligação com a origem do problema, aliás, foi encontrado em um manual para caixeiros-viajantes de 1832 onde mencionava o problema, incluindo exemplos de trajetos pela Alemanha e Suiça, mas que não era de cunho matemático.

O problema foi matematicamente formalizado na década de 1800 pelo irlandês William Rowan Hamilton e pelo britânico Thomas Kirkman. Mas foi somente após 1930 que o problema começou a ficar globalmente reconhecido, especialmente devido o incentivo da Coorporação RAND em ter oferecidos prêmios por avanços no problema, nas décadas de 1950 e 1960.

2.1.4 Variações do problema

As regras determinadas sobre o problema podem variar a depender de cada versão. As variações dizem respeito sobre qual será a cidade inicial, se há o retorno à cidade de origem após visitar todas e quanto as características das cidades.

Se tratando da cidade inicial, há situações em que deseja descobrir a melhor rota, porém iniciando a partir de uma cidade pré-estabelecida. Nesse caso há grande chance da rota selecionada não ser a melhor possível, já que limita-se o início do trajeto a uma cidade em específico. Mas é útil em casos como uma transportadora que precisa partir de sua sede em direção aos locais de destino.

Quanto ao retorno à cidade de origem, é também uma questão a depender do cenário do problema, pois, caso seja necessário retornar ao ponto inicial, o deslocamento entre a última e primeira cidade precisa ser levado em conta no somatório da distância total percorrida. Já quando não há a necessidade, a prioridade é somente visitar as cidades, deixando a primeira e a última com apenas uma conexão, e as intermediárias com duas. É útil em casos onde por exemplo, um braço robótico precisa soldar um certo número de componentes em uma placa, onde neste caso, não é necessário o retorno ao primeiro componente soldado para que sua tarefa seja concluída, mas que o deslocamento entre as peças precisa ser bem otimizado para garantir eficiência.

Figura 5: Braço robótico de solda
Fonte: Golden Laser

Além desses atributos que podem ser considerados a partir da necessidade de cada problema, há também questões sobre as características das cidades, e que levam a uma generalização do problema principal, onde um bom exemplo é o Problema do Caixeiro Viajante Preto e Branco.

O PCV-PB funciona de forma similar ao PCV, o objetivo de concluir o circuito hamiltoniano ainda está presente, a diferença está na classificação dos vértices, onde há os brancos e os pretos. Existem duas restrições, a primeira é quanto a cardinalidade, que determinado a quantidade de vértices brancos, e a segunda, quando ao comprimento, que determina a distância máxima a ser percorrida entre dois vértices pretos.

> Uma aplicação para o PCV-PB pode ser observada na programação de companhias aéreas. Durante o trajeto de um avião é preciso realizar algumas operações como paradas para manutenção da aeronave e paradas relacionadas ao embarque e desembarque de passageiros. Neste contexto, é importante para as companhias aéreas determinar uma sequência de voos que minimize o custo das operações. (CAZETTA, 2015, p.2).

Figura 6: O PCV-PB em agendamentos de companhias aéreas
Fonte: Paôla Pinto Cazetta

Outro tipo de variação é quanto a simetria, há casos onde a distância entre dois vértices pode ser diferente dependendo do sentido ao qual está se deslocando até ele, como por exemplo, ao trafegar por uma cidade, onde há ruas de apenas um sentido, o que pode resultar em um

caminho de volta ser mais ou menos distante que o caminho de ida. Para esses casos, dizemos que é um problema assimétrico, e nos casos onde a distância entre dois vértices é a mesma, independente do sentido, dizemos que é simétrico.

Para o ambiente de resolução deste trabalho, será considerado que qualquer cidade pode ser a inicial, as arestas são simétricas e também não há necessidade de retorno à cidade de origem, pois a prioridade é apenas visita-las no menor trajeto possível.

2.2 SUA RELEVÂNCIA

O que há de especial no PCV, é o fato de que um pequeno acréscimo no valor de entrada (número de cidades), resulta em um drástico aumento no valor de saída (número de possibilidades). O que o torna um excelente problema de otimização combinatória, já que há diversas combinações possíveis dentre a ordem na qual as cidades são visitadas, e que tem por objetivo descobrir qual destas é a que possui a menor distância total.

2.2.1 O crescente número de possibilidades

Já que a partir de qualquer cidade é possível se deslocar para todas as outras que ainda não tenham sido visitadas, para a primeira cidade é possível escolher qualquer uma dentre elas, já que nenhuma ainda foi visitada. Portanto, se há 5 cidades, há justamente 5 possibilidades de escolha para o começo do trajeto, e após esta, restam outras 4, pois não pode conectar-se consigo mesma. E uma vez estando na segunda cidade, que é uma dentre estas 4 opções, restam agora 3, e depois de escolher alguma dessas, restarão duas, até que por fim sobrará apenas a última.

Eis aqui um crescimento fatorial, onde o número de possibilidades é obtido através do valor fatorial do número de cidades. Ou seja, sendo 5 cidades, o número total de rotas possíveis será de 120, pois é o resultado de $5 \times 4 \times 3 \times 2 \times 1$, ou simplesmente $5!$ (cinco fatorial). Se a quantidade de cidades for acrescida em 1, o número de possibilidades será o fatorial de 5 multiplicado por 6, que resulta em 720. Já para 7 cidades, será $7 \times 6! = 5040$, ou $7!$

É um crescimento extremamente significativo, pois para cada nova cidade adicionada, o aumento de possibilidades é muito maior do que o aumento que houve da antepenúltima para a penúltima. Portanto, o problema torna-se extremamente complexo mesmo para um pequeno valor de entrada, pois comparar todas estas para descobrir a melhor, é uma atividade que custa muito tempo.

2.2.2 Complexo até mesmo para o computador

Computadores em sua essência são executores extremamente rápidos de tarefas lógico-matemáticas. A partir disto é possível dar diversos propósitos em seu uso, como jogos, executar protocolos de comunicação para conceber a internet e assim acessar a sites, criar e salvar arquivos digitais, dentre outros. Mas o que ele de fato faz para executar todas essas aplicações, são cálculos matemáticos extremamente rápido.

Como contado no filme Hidden Figures (2016), durante a Corrida Espacial em meio a Guerra Fria, houve a transição do termo *computer*, pois até essa época, um computador era uma pessoa que executava cálculos matemáticos, ou "aquele que computa". Os EUA e a União Soviética tinham suas respectivas equipes de cientistas, que estavam tentando realizar uma expedição que levasse o homem até a lua e o trouxesse em segurança, e para isso precisavam realizar inúmeros cálculos, que eram feitos à mão.

Com a chegada dos computadores, dessa vez uma máquina semelhante ao que temos hoje, a profissão computador deixou de ser necessária, já que agora era possível realizar cálculos muito mais rápido do que uma equipe inteira de humanos, e ainda com mais precisão nos resultados. Porém houve a necessidade de ter pessoas que sabiam programar essas tarefas a serem executadas pelo computador, surgiu então os programadores, que no início não era uma profissão específica, mas sim os já matemáticos que aprenderam a opera-lo.

Essas tarefas, também chamados de algoritmos, são instruções a serem efetuadas para que obtenha um resultado esperado. Funciona como o conceito de funções, onde lhe é fornecida um valor de entrada, executa uma tarefa específica com este valor, e retorna seu respectivo resultado. A diferença é que um algoritmo é um agrupamento de funções, que ao decorrer destas, finaliza entregando o resultado ao qual foi programado.

Como por exemplo, uma máquina de *snacks*, recebe como valores de entrada, o dinheiro e a lanche desejado, e após seu processamento, onde confere o valor da moeda e identifica a posição do produto, seu valor de saída resulta no deslocamento necessário da máquina para que o cliente possa retira-lo.

Figura 7: Máquina de *snacks* como representação de função computacional
Fonte: autor

No caso do PCV, os valores de entrada são todas as possíveis rotas, a tarefa a ser executada é comparar a distância total de cada uma, e o resultado, ou valor de saída, é a rota mais eficiente. O problema é que, mesmo o computador sendo capaz de executar tarefas muito rapidamente, se a quantidade de tarefas for extraordinariamente grande, sua velocidade já não contempla com um tempo viável.

Por exemplo, supondo que para executar o algoritmo que compara todas as rotas, seja utilizado um computador capaz de realizar 1 bilhão de adições por segundo, o que é já é bastante se comparado com um computador casual. O tempo que ele levaria para terminar a execução e entregar o resultado, de acordo com a respectiva quantidade de cidades, seria como demonstrado na Tabela 1.

Tabela 1: Relação entre Tempo Gasto e Número de Cidades
Fonte: Instituto de Matemática e Estatística da UFRGS

Número de Cidades	Possibilidades de Rotas	Tempo de Execução
5	120	insignificante
10	3.628.800	0.003 segundos
15	1.307.674.368.000	20 minutos
20	2.432.902.008.176.640.000	73 anos
25	15.511.210.043.330.985.984.000.000	470 milhões de anos

Perceba que, um pequeno acréscimo ao número de cidades, já resulta num grande acréscimo ao tempo gasto para terminar todas as comparações, o que torna impraticável a resolução deste problema através deste método. Portanto é necessário encontrar outras formas de obter o mesmo resultado, porém sendo mais eficiente.

2.2.3 O objetivo da otimização

Quando se trata de otimização, a frase "os fins justificam os meios" de Nicolau Maquiavel não aplica, já que apesar do resultado ser o mesmo, a forma como chegou-se até ele é muito importante. Na natureza, eficiência é um fator chave para sobrevivência, aquele que consegue atingir um objetivo, consumindo o mínimo de recursos possível, torna-se o mais adaptado a sobreviver aquele meio.

Segundo o *Mathematical Programming Glossary*, otimização é a seleção do melhor elemento, a partir de algum critério, dentre um conjunto de alternativas disponíveis. Em computação, um algoritmo eficiente, ou otimizado, é aquele capaz de entregar o resultado consumindo o mínimo de memória e tempo possível. Para resolver o PCV deveria ser utilizado um método capaz de descobrir a melhor rota, porém sem precisar comparar todas, o que é um tanto quanto controverso pois é complicado dizer qual é a melhor opção sem antes conhecer todas.

Justamente para isso temos as heurísticas, que "são processos cognitivos empregados em decisões não racionais, sendo definidas como estratégias que ignoram parte da informação com o objetivo de tornar a escolha mais fácil e rápida" (GIGERENZER e GAISSMAIER, 2011).

Até hoje não foi apresentado nenhum algoritmo capaz de solucionar este problema, o que faz seguir em aberto se é porque ninguém ainda conseguiu implementar uma solução, ou se de fato problemas como este não possuem uma solução simples. À esta incógnita é atribuído um problema matemático, que tenta justamente classificar os problemas matemáticos quanto a sua complexidade de resolução.

2.2.4 P vs NP

O objetivo de um algoritmo é resolver um determinado problema, nele há instruções a serem executas de modo que ao seu término, entregue o resultado. Apesar de cada problema possuir suas respectivas tarefas para sua resolução, os valores de entrada, isto é, o que é recebido

pelo programa, e o que será feito com eles, pode variar de forma que o resultado seja obtido com mais eficiência.

Qualquer problema que possua uma sequência de passos, pode ser programado através de um algoritmo computacional, especialmente aqueles que são repetitivos, pois os computadores conseguem executa-los de forma rápida. Mas apesar disso, cada problema possui seu nível de complexidade, que pode variar o desempenho do algoritmo.

2.2.4.1 Do que se trata

Todo problema algorítmico possui seu custo computacional, que é medido através do quanto de memória o mesmo necessita para atingir a resolução, e o tempo que leva até sua conclusão. Problemas que possuem um tempo de resolução hábil, isto é, que conseguem entregar o resultado no tempo necessário, são denominados polinomiais, ou seja, sua complexidade não muda drasticamente a medida que aumenta o valor de entrada. De modo geral, podemos descrever que problemas Polinomiais, ou simplesmente do grupo P, são todos os problemas em que é possível resolve-lo de forma trivial.

Já os problemas NP, sigla para não-polinomial (non-polinomial), pertencem ao grupo de problemas em que para verificar se a resposta fornecida pelo algoritmo está correta, é rápido, porém para encontrar a solução do problema, não necessariamente. Ou seja, a complexidade de solucionar o problema é diferente da complexidade para verificar se alguma resposta dada está correta.

Logo, podemos concluir que todo problema que está em P, também está em NP, afinal, se um problema é simples de se encontrar a resposta, consequentemente também será simples verificar se ela está correta. O que não sabemos, entretanto, é se o mesmo vale de forma inversa, ou seja, se todo problema NP também é um problema P, onde apenas ainda não encontramos uma forma otimizada e viável para resolução deste.

Um exemplo de problema P é qualquer cálculo aritmético, onde, independente se for *2 + 2* ou *203045 x 3103503*, computadores conseguem rapidamente encontrar a resposta. E um exemplo de problema NP é o Sudoku, pois para verificar se um jogo já preenchido está correto, é bem simples, pois basta verificar se não há nenhuma incongruência com as regras. Mas para descobrir quais números devem ser preenchidos nos espaços vazios, é preciso testar inúmeras combinações até que alguma delas esteja correta.

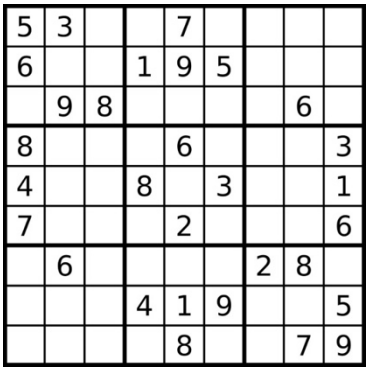

Figura 8: Sudoku é um problema Não Polinomial
Fonte: Kaggle

Há ainda o grupo NP-Difícil, que são todos os problemas que são no mínimo, tão difíceis quanto qualquer outro problema NP. Estes, portanto, não necessitam possuir tempo de verificação hábil, ou seja, não estão em P.

Os problemas que estão, tanto em NP-Difícil como em NP, são considerados NP-Completos. Estes possuem uma característica importante, pois servem como base para a resolução de qualquer outro problema NP. Isso significa que qualquer problema pode ser reduzido a ele, ou seja, se há uma solução para algum problema NP-Completo, logo, todos os outros NPs também serão solucionados. Um dos problemas mais populares que é NP-Completo, é justamente o Caixeiro Viajante.

2.2.4.2 Prêmio Millenium

Descobrir se todo problema NP é também um P, é um desafio altamente relevante para a matemática e que até hoje segue em aberto. Não é atoa que o Instituto Clay de Matemática (Clay Mathematics Institute)[2], uma fundação privada, sem fins lucrativos, localizada em Cambridge e que visa a disseminação do conhecimento matemático, oferece a recompensa de um milhão de dólares para aquele que der a completa solução para este, ou outros 6 problemas, definidos por eles como os maiores problemas da matemática atualmente, onde até então, apenas um foi resolvido[3]. São eles:

[2] https://www.claymath.org - data de acesso: 28/06/17
[3] http://bit.ly/2z9WWKJ - data de acesso: 28/06/17

- P versus NP (onde encontra-se o Problema do Caixeiro Viajante);
- A conjectura de Hodge;
- A conjectura de Poincaré (resolvido por Grigori Perelman);
- A hipótese de Riemann;
- A existência de Yang-Mills e a falha na massa;
- A existência e suavidade de Navier-Stokes;
- A conjectura de Birch e Swinnerton-Dyer.

Dentre esta lista, o de *P versus NP* é o de maior relevância para a Ciência da Computação, pois sua resolução trará grande impacto para forma como tratamos as soluções algorítmicas, que na atualidade está presente em praticamente todos os segmentos de ciência e tecnologia.

Seu objetivo é provar se o grupo de problemas que estão em NP, também estão, ou não, em NP, ou seja, provar se P é igual ou diferente de NP. Até hoje, apesar de muitos acreditarem ser diferente, não houve nenhuma prova completa que comprovasse nenhuma das duas possibilidades.

2.2.4.3 Consequências de sua resolução

Se caso seja provado que P é diferente de NP, não haverá grande impacto no que hoje temos na computação, pois esta afirmação é a atualmente especulada. Porém, caso seja provado que P é igual a NP, ou seja, todo problema que seja fácil verificar a resposta é também um problema fácil de obter a resposta, muita coisa mudará.

Uma forma de provar essa igualdade seria resolver qualquer problema que esteja no grupo dos NP-Completos, pois como já explicado, se um destes for resolvido, sua solução servirá de base para a resolução de todos os outros em NP, e assim, a forma como a pesquisa matemática é feita, mudará drasticamente.

Atualmente, matemáticos dedicam-se a vida inteira na tentativa de resolver um problema em específico, e em diversos casos acabam por não conseguir. Se fosse encontrada uma forma única de resolver todos estes complexos problemas, o esforço aplicado pelos matemáticos poderia ser dedicado a áreas mais abrangentes ao invés de dar atenção a um único problema durante sua carreira.

Além disso, o processamento computacional também sofreria grande impacto, uma vez que não mais seria necessário hardware muito potente para executar algoritmos, já que os mesmos estariam tão otimizados a ponto de serem efetuados em tempo polinomial. Consequentemente, a criptografia que hoje conhecemos, deixaria de existir, já que o processo de encriptação utiliza problemas NP.

Para garantir a seguridade da informação na era digital, é necessário que haja funções de mão única, isto é, a partir de um valor é fácil chegar a outro, mas a partir deste outro não é fácil chegar ao valor de origem. Este é o fundamento da criptografia, onde podemos citar como exemplo o RSA, sigla para seus três criadores, Ronald Rivest, Adi Shamir e Leonard Adleman. Esta técnica utiliza como princípio o produto de dois primos, onde a partir de dois números primos multiplicados obtem-se um número que só pode ser resultado do produto destes dois, e que a partir deste resultado é impraticável descobrir quais são seus fatores, especialmente para grandes números.

Como por exemplo, o produto de 3 e 5, que são dois números primos, resulta em 15, e não há outros dois números em que se multiplicados resultarão também em 15. Porém no caso do RSA, é utilizado números da ordem 10^{100}, no mínimo. Trata-se então de um problema Não Polinomial, e se P fosse igual a NP, a dificuldade de quebrar esta encriptação deixaria de existir, causando um grave problema de segurança. O mesmo serve para todos os problemas do grupo NP, e em suas diversas aplicações no mundo atual.

2.2.4.4 Sua relação com o PCV

Em 1972, Richard Manning Karp publicou o artigo *Reducibility Among Combinatorial Problems*, onde utilizando o teorema de Stephen Cook (1971), que identificou o Problema de Satisfatibilidade Booleana como o primeiro pertencente à classe de complexidade NP-Completo, provou que há uma redução por mapeamento, isto é, uma redução que converte instâncias de um problema em outro, para os 21 problemas de Análise Combinatória e Teoria dos Grafos do artigo, onde um deles é o Ciclo Hamiltoniano, ou PCV.

3 SOLUÇÕES EXISTENTES

Antes de apresentar a solução proposta desta monografia, é interessante passar pelos métodos comumente usados, como forma de obter não somente a comparação de desempenho entre eles, mas também entender as diferentes soluções que o problema possui.

Portanto, este capítulo tem como finalidade apresentar e explicar estes métodos, a fim de obter melhor entendimento sobre o processo de resolução e eficiência dos algoritmos.

3.1 MÉTODOS

As soluções escolhidas para serem tratadas nesta monografia vão de acordo com sua popularidade, ou seja, são métodos já bem explorados e que possuem melhor entendimento sobre seu desempenho. Entretanto, também é levado em consideração o desempenho humano, mesmo que este ainda não seja tão bem definido na literatura, pois é relevante utiliza-lo como critério de comparação em relação as soluções computacionais.

Foram abordados o método lexicográfico, que compara todas as possíveis rotas. O método randômico, cujo busca rotas de forma aleatória, armazenando a melhor encontrada até então. O método que utiliza algoritmo genético, onde busca aperfeiçoar as rotas através da "evolução das gerações". E uma breve análise quanto a resolução provida pelo humano.

3.1.1 Lexicográfico, ou Força Bruta

É a única forma, atualmente conhecida, que descobre a solução exata para o problema, ele compara todas as rotas a fim de encontrar a melhor, e como já é sabido, é inviável para de ser aplicado, especialmente conforme o número de cidades aumenta. Justamente por analisar todas as possibilidades a fim de obter a resposta, é que este método é de força-bruta, cujo é um termo não exclusivo ao PCV, mas sim de uma técnica de uso geral.

Já o termo lexicográfico, que em matemática significa uma estrutura de ordem natural do produto cartesiano de dois conjuntos ordenados, é um método capaz de listar todas as possíveis combinações de rotas do PCV. Pois sabendo que a as cidades não podem ser repetidas, apenas mudadas de ordem, é preciso estabelecer um critério para ir gradativamente passando por todas as combinações possíveis e em ordem. Seu pseudocódigo pode ser representado como:

```
1. Pegue o último número que é menor que seu sucessor (X) (se
   não encontrar, é porque terminou)
2. Pegue o último elemento que é menor que X (Y)
3. Troque-os de posição (X com Y)
4. Inverta a ordem de todos os números posteriores a X
```

Código 2: Pseudocódigo para o algoritmo lexicográfico
Fonte: Michal Forišek

Dessa forma, é possível definir uma ordem, tal qual a ordem alfabética, em que seja possível remanejar os elementos existes, neste caso as cidades, até que todas as combinações possíveis sejam realizadas, inclusive sabendo o progresso do quanto já foi verificado e quanto resta para terminar. Ou seja, é um algoritmo bem previsível e que entrega a solução desejada, seu único problema está no custo computacional e consequentemente, tempo gasto.

3.1.2 Randômico, ou Aleatório

Similar ao desafio de testar combinações para descobrir a senha de um cadeado, o método randômico tem por objetivo ir testando possibilidades de forma aleatória, porém, diferente do cadeado onde ao encontrar a solução correta, as tentativas são interrompidas, no PCV não há como saber se a solução testada é a melhor existente, afinal, não há um critério de parada uma vez que a melhor rota é desconhecida.

Ao invés disso, é armazenado a melhor possibilidade testada até então, onde a cada nova opção, é verificada se esta possui uma distância total menor que a atual melhor rota. Caso positivo, ela se torna a nova melhor opção, caso negativo, segue para a próxima tentativa.

Por se tratar de um método que depende da sorte, é possível que a melhor opção seja logo encontrada, mas como o número de possibilidades é vasto, as chances são muito remotas. E ainda assim, há o problema de não saber se esta tentativa encontrada é de fato a melhor, pois não há nenhum critério que determine qual é a menor distância possível, pois se tivesse, o problema já poderia ser resolvido através dele.

Há ainda outros dois problemas neste método, o primeiro é que as chances de ser selecionada uma rota completamente ruim é muito maior que uma boa, e que por sua vez é ainda mais difícil de encontrar uma que seja melhor do que a melhor descoberta até então. O segundo problema, está na possibilidade de repetir rotas, pois conforme as rotas vão sendo testadas, as chances de testar alguma repetida aumentam, de modo que quando for testada metades de todas as possibilidades, as chances de cair alguma repetida tornam-se maior que uma ainda não descoberta.

E não faz sentido armazenar todas as rotas testadas, para que assim seja evitado a redundância, pois nesse caso, para toda nova rota, será necessário verificar dentre todas essas já testadas se alguma delas é igual a nova da vez. O que aumenta e muito a complexidade do problema, tornando-o inviável.

3.1.3 Algoritmo Genético

Foi inspirado na *Teoria da Evolução por Seleção Natural* de Charles Darwin e Alfred Wallace, porém com algumas ressalvas, pois biologicamente, o termo evolução não significa que uma espécie é melhor que outra, pois se alguma foi extinta, não foi porque ela é inferior a uma que atualmente vive, e sim porque não se adaptou às variações do ambiente na época em que viveu. Diferentemente do A.G. que busca atingir um progresso nos resultados, a fim de obter o melhor possível.

A origem do algoritmo genético não tem uma data em específica, mas a técnica que é utilizada hoje tornou-se popular em 1975 através do livro *Adaptation in Natural and Artificial Systems* de John Henry Holland. Ele introduziu um método formalizado para predizer a qualidade da próxima geração, o que ficou conhecido como Teorema do Esquema de Holland (*Holland's Schema Theorem*).

Desde 1950, com a proposta de "aprendizagem de máquina" de Alan Turing, que seria paralelo aos princípios de evolução, há a ideia de realizar uma simulação computacional. Tanto que em 1954, Nils Aall Barricelli realizou uma simulação utilizando o computador do Instituto de Estudos Avançados de Nova Jersey, apesar de ser trabalho não ter se repercutido. Mas a pesquisa em A.G. permaneceu bastante teórica até meados dos anos 1980, quando a primeira Conferência Internacional em Algoritmo Genético foi realizada em Pitsburgo, Pensilvânia.

O conceito do A.G. funciona com base em ciclos (gerações), onde em cada ciclo há um conjunto (população) de tentativas para resolução ao problema (espécies). Um ciclo é gerado a partir das melhores tentativas do ciclo anterior (seleção), e já que não há nenhum antes do primeiro, este é iniciado com tentativas aleatórias.

As tentativas são informações referentes a cada problema, seria equivalente ao DNA de uma espécie. No caso do PCV, pode ser uma lista, ou *array*, contendo a ordem das cidades a serem visitadas. Durante uma geração, essas tentativas passam por um processo de seleção, cruzamento e mutação.

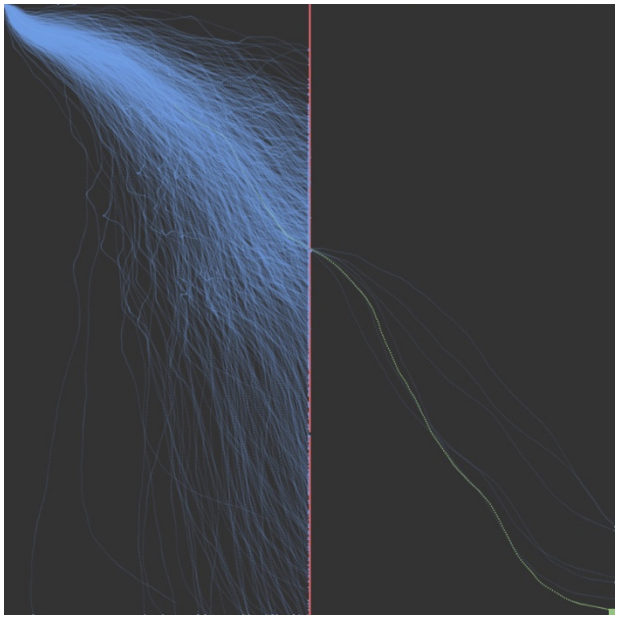

Figura 9: Exemplo de A.G. onde deve-se chegar ao destino passando pela barreira
Fonte: autor

3.1.3.1 Seleção

Para que uma nova população surja, é preciso selecionar os melhores membros da geração anterior a fim de obter um progresso a cada novo ciclo, até que as tentativas se encontrem num estado satisfatório para resolver o problema.

Para isso é preciso definir o critério que qualifica uma boa espécie, ou tentativa, de modo que ao ser avaliado, saiba o quão melhor este foi em relação aos outros membros da geração.

3.1.3.1.1 Cálculo do desempenho

Este critério varia de acordo com cada necessidade de problema, no caso do PCV é simples definir que uma tentativa é melhor que outra se esta possui uma distância total percorrida menor. Dessa forma é avaliado a distância de cada membro da geração para assim definir quais foram os melhores, e a partir destes constituir a nova população.

Porém se for definido que o valor da distância é o valor referente ao desempenho, ocorrerá uma inversão, já que quanto maior a distância, pior é desempenho daquela espécie. Para que seja garantido que uma menor distância resulte em um maior desempenho, basta efetuar o cálculo *1 / distância*.

Dessa forma uma espécie que teve distância total de 1000 resultará em um desempenho de 0.001, enquanto que uma que teve distância de 500 resultará em uma de 0.002 que é duas vezes melhor que a anterior.

Mas para que esse valor de desempenho não fique em casas decimais, é uma boa prática multiplicar esse valor por alguma constante, como 10000. Assim, o desempenho que era 0.001 passará a ser 10, e o de 0.002 será 20. Que fica mais legível de ser comparado. Então o cálculo de desempenho de cada espécie, para o PCV, fica da seguinte forma:

```
1. Verifique a distância total de cada espécie;
2. Determine o desempenho de cada a partir do cálculo:
   1 / distância * 10000.
```

Código 3: Cálculo de desempenho para o algoritmo genético
Fonte: autor

3.1.3.1.2 Critério de escolha

A seleção em si não está no cálculo de desempenho (*fitness score*), este é o que define o quão bem tão tentativa foi, ela é a forma que as tentativas são escolhidas para gerar as novas. Existem diferentes formas de selecionar as espécies, como por exemplo o torneio, onde é escolhido uma certa quantidade de tentativas de forma aleatória, e a partir destes ver o melhor. Como por exemplo, seleciona 3 tentativas de forma aleatória, e desses é escolhido o melhor para ir para o próximo estágio, que é a mutação.

No caso do PCV é comum utilizar ao invés do torneio, o método da roleta (*mating pool*), onde cada tentativa possui uma porcentagem referente ao seu desempenho, ou seja, aqueles que se saíram melhor, ocuparão um espaço maior da roleta, e o que se saíram pior, um espaço pequeno.

Uma vez definida a roleta, é selecionado de forma aleatória alguma tentativa desta, onde por os melhores constituírem uma parcela maior, estes têm maior chance de serem selecionados, mas que não é descartado a possibilidade de selecionar um que tenha ido mal, apenas será mais difícil.

Trantando-se de forma algorítmica, podemos definir a roleta como sendo uma lista de tentativas, onde os que possuem melhor desempenho terão mais cópias de si dentro do *array*. Para definir a quantidade de cópias que uma tentativa terá dentro da roleta, basta normalizar o desempenho de todas com base no melhor daquela geração.

Por exemplo, se a melhor tentativa de uma geração teve sua distância total de 500, seu desempenho será de *1 / 500 * 10000*, que resulta em 20, logo o desempenho de todos será dividido por este valor, de modo que resulte em um número entre 0 e 1, sendo 1 a melhor tentativa, já que *20 / 20* resulta em 1.

Após isso, basta multiplicar esse novo desempenho obtido, que antes era 1 dividido pelo comprimento da distância total, e que agora é um valor normalizado, por uma constante, como 100. Dessa forma a melhor espécie da geração terá 100 cópias de si na roleta, uma vez que 1 (melhor desempenho normalizado), multiplicado por 100 resulta no próprio 100. Enquanto que as outras terão algum valor entre 0 e 100, referente ao seu desempenho em comparação com o melhor.

Por exemplo, se uma espécie teve distância total de 1000, após calcular o desempenho (*1 / distância * 10000*) obtem-se 10, e normalizando pelo pelo melhor desempenho (*10 / 20*), seu novo desempenho será de 0,5 que multiplicado pela constante 100, serão 50 cópias de seu cromossoma na roleta, que é metade da quantidade do melhor, o que faz sentido já que seu desempenho é também a metade em relação ao primeiro.

3.1.3.2 Cruzamento

Após definido o critério de seleção, é necessário escolher os pares que terão seus genes cruzados a fim de resultar em um "filho" com as características herdadas de seus "pais". Esse processo de escolha é feito de forma aleatória, como no caso da roleta, onde são selecionadas duas espécies por vez e estas geram uma determinada quantidade de filhos.

A quantidade total de filhos gerados devem ser exatamente a quantidade da população, portanto, se forem gerados 2 filhos em cada cruzamento, haverão 500 *crossovers* para uma população de 1000 membros. Ou seja, serão selecionadas 1000 espécies, duas por vez, onde estas terão seus cromossomas cruzados resultando em dois filhos para preencher a nova população.

A técnica de mutação pode variar de acordo com a necessidade do problema, mas é comum ser utilizada a técnica do corte, onde uma posição aleatória do cromossoma é selecionada, e o gene do primeiro filho será a primeira parte do pai e a última da mãe. Já o segundo filho terá a primeira da mãe e o último do pai, como pode ser observado na Figura 10.

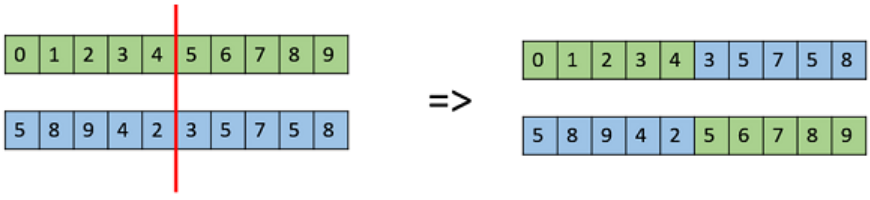

Figura 10: Representação do cruzamento dos genes
Fonte: TutorialPoint

3.1.3.3 Mutação

Uma vez sendo gerado uma nova espécie, ou “filho”, este deve passar por um processo de mutação, onde para cada elemento de seu gene é verificado se este receberá alguma modificação. A decisão de se ocorrerá ou não uma mutação, é novamente um processo aleatório, porém de baixíssima chance, mas que caso ocorra, é modificado aquele ponto do gene em que está sendo verificado.

No caso do PCV, a modificação que ocorre é uma possível troca na ordem ao qual os pontos são ligados, então é verificado *n* vezes, onde *n* é o número de cidades existentes, e para cada uma destas é testado se ocorrerá a mutação, que é uma chance baixíssima de acontecer, mas que caso positivo, duas posições, ou cidades, são selecionadas também de forma aleatória, e suas ordens são trocadas.

A mutação é fator importante para a evolução dos resultados, sem ela, o algoritmo apenas seria uma combinação entre as tentativas da primeira geração, estas que são geradas de forma aleatória, ou seja, sem grandes chances de possuir resultados positivos.

3.1.3.4 Outras caracteristicas do Algoritmo Genético

Como observado, o mesmo trabalha muito com aleatoriedade, portanto é importante possuir um bom gerador de números aleatórios, pois normalmente, os que vem por padrão nas linguagens de programação tende a padronizar os resultados quando ocorre muitas utilizações, como é o caso do A.G. Uma forma de resolver isso, seria modificar a *seed* do gerador a cada geração, ou utilizar uma biblioteca especializada em gerar números aleatórios.

O número de gerações existentes pode ser pré-determinado, este seria o critério de parada, onde ao completar esse total de ciclos, o programa é interrompido e a melhor tentativa até então será a dada resposta pelo algoritmo.

Apesar da técnica de seleção trabalhar de forma aleatória, é comum a cada nova geração passar o indivíduo de melhor desempenho para a seguinte, dessa forma evita que o perca durante os processos de cruzamento. Isso garante que as soluções sempre terão progresso, já que no mínimo o melhor de uma geração, será o da geração passada, podendo alguma espécie tendo o superado e então este se torna o novo melhor a ser passado a diante.

Uma prática bastante comum empregada no A.G. é de utilizar heurísticas antes de iniciar o processo de evolução, preocupando-se com o desempenho da primeira geração da população, onde ao invés de iniciar as tentativas de forma aleatória, é utilizado primeiro a heurística, para que a partir deste estágio comece o processo de evolução. Aproveitando que, em um tempo

muito curto, ela é capaz de entregar uma solução bem razoável, já que sua abordagem é de custo polinomial.

O intuito é aproveitar dos benefícios de evolução do algoritmo genético, mas pulando uma grandiosa etapa de tentativas ineficazes, que custariam tempo ao serem exploradas e evoluídas. É como se ao invés ter levado mais de 4 bilhões de anos para que houvesse vida inteligente habitando na Terra, já em seu primeiro ano de existência estivesse como hoje, e a partir deste ponto, inicia-se a evolução.

3.1.4 Desempenho humano

O desempenho na resolução do PCV, em especial a variação Euclidiana do problema, pode ser equiparado à resolução provida pelo computador através da utilização de heurísticas, e em certos casos até melhor (MACGREGOR e ORMEROD, 1996), o que atraí o problema para área de psicologia cognitiva.

Durante os experimentos realizados por MacGregor e Ormerod, onde testaram o desempenho de humanos em dois testes, um utilizando 10 pontos e outro com 20, sustentaram a hipótese que a complexidade do PCV é na verdade uma função do número de pontos não limitantes, e não do número total de pontos.

> Suas soluções se aproximaram das soluções mais conhecidas, tiveram uma ordem de magnitude melhor do que as soluções produzidas por três heurísticas bem conhecidas e, em média, ultrapassaram o percentual de 99.9 na distribuição de soluções aleatórias. O processo de solução parece ser baseado em percepção. (traduzido de MACGREGOR e ORMEROD, 1996)

Mas o método utilizado por um humano racional para a resolução de tal problema, é ainda uma incógnita, que vem sendo estudada não exclusivamente para a resolução do PCV, mas em diversas características cognitivas. Principalmente na área de Inteligência Artificial, mais especificamente no aprendizado de máquina ou *Machine Learning*, onde o estudo de como o cérebro humano funciona para aprendizagem e raciocínio, vem sendo bastante explorado para dar avanço em como as máquinas podem melhorar seu desempenho.

3.2 QUAL É O MELHOR?

Dependendo do critério de como definir o melhor, essa pergunta pode ter diferentes respostas, pois se o melhor for aquele que entrega o trajeto mais curto, força bruta é o único algoritmo até então capaz de dar a solução com precisão. Porém como já foi dito, o tempo que ele leva para tal não é viável na prática e, portanto, sua escolha não seria a mais indicada para lidar com problemas reais.

Para isso, devemos considerar que melhor é aquele que consegue, em um tempo necessário, entregar uma solução bem satisfatória. E nesses casos, métodos que fazem uso do algoritmo genético, obtêm bom proveito, afinal, eles são justamente para problemas de busca e otimização.

No trabalho de Mercês e Ladeira (2011), realizaram um estudo comparativo de técnicas de inteligência artificial, onde utilizaram uma rede neural auto-organizável, comumente chamada de Mapa de Kohonen ou *Self Organizing Map* (HAYKIN, 1999). "Em uma rede neural auto-organizável, os neurônios se organizam sem grupos, de acordo com informações que recebem. Essas informações não são recebidas somente por um único neurônio, mas pela influência dos outros que estão próximos." (MERCÊS e LADEIRA, 2011).

Em sua conclusão, citaram que a rede neural utilizada não obteve êxito, mas que a do algoritmo genético sim.

> A rede auto-organizável não obteve êxito ao gerar uma solução para o PCV. Como os neurônios se acumulam no mesmo ponto, após associar cada neurônio com a cidade mais próxima, muitos teriam que ser associados com uma cidade aleatória, pois a cidade mais próxima já estaria ocupada por outro neurônio coincidente. Isso invalidaria todo o treinamento da rede, gerando ao final do processo uma solução aleatória.
>
> Além disso, e baseando-se nos resultados obtidos, é possível concluir que o algoritmo genético com o operador de cruzamento guloso apresentou desempenho superior ao observado no EER e no algoritmo Lin-Kernighan implementados neste estudo. (MERCÊS e LADEIRA, 2011, p.11)

As heurísticas também são boas candidatas a melhores alternativas, a depender do desempenho de cada uma e do tipo de problema. Mas que para o PCV, técnicas em que combinam ambas, isto é, utilizar heurísticas para inserir como membro da população no A.G. apresentam ótimos resultados.

4 INTRODUÇÃO À HEURÍSTICA

Este capítulo trata todas as questões relacionadas ao desempenho dedicado para o desenvolvimento da heurística. Primeiro é relatado o progresso ao levou até chegar na versão final, em seguida é apresentada a heurística e seu funcionamento. E por fim, demonstrado a implementação e os resultados que ela gerou.

4.1 O PROGRESSO

Antes de descrever a heurística, é valido primeiro relatar o progresso no qual a fez chegar. O propósito no início era encontrar uma solução exata para o problema em tempo polinomial, mas após o insucesso, o foco levou-se a equiparar ao desempenho humano, implementando um algoritmo que descrevesse os processos que uma pessoa faz ao resolve-lo.

Uma observação que cabe ser ressaltada, é que durante as tentativas foram abordadas outras temáticas que melhor representassem os algoritmos, então deixou-se de lado a ilustração de conectar cidades, e foi utilizado algo mais simplista como apenas conectar pontos. O que não prejudica na proposta do problema, e sim foi apenas uma forma mais condizente de representa-lo.

4.1.1 Tentanto solucionar o problema

É justificável que por ser um problema que segue em aberto, e ao pesquisar a seu respeito, surja a curiosidade de tentar soluciona-lo com exatidão e em tempo hábil. Mas dizer qual é a melhor escolha dentre um grupo diverso de opções, e sem verificar todas, é uma tarefa complexa de ser executada, para isso, seria necessário um método que ao ser aplicado, independente da quantidade e posição dos pontos, fosse capaz de resultar para todos os casos, a melhor rota existente. Além disso, é preciso ainda encontrar uma forma de validar se a resposta fornecida pelo método, é de fato a melhor possível.

Se preocupando primeiro em encontrar um método que consiga entregar a melhor resposta, e depois em como provar se este de fato funciona, afinal, no pior dos casos, fazer uma comparação entre o método da força bruta e o método proposto, mesmo com poucas cidades, mas repetidas vezes, se caso o resultado de ambas for o mesmo para todos os testes realizados, é uma possível evidência de que o método proposto funcione também para grandes quantidades de cidades.

4.1.1.1 Método do Embrulho

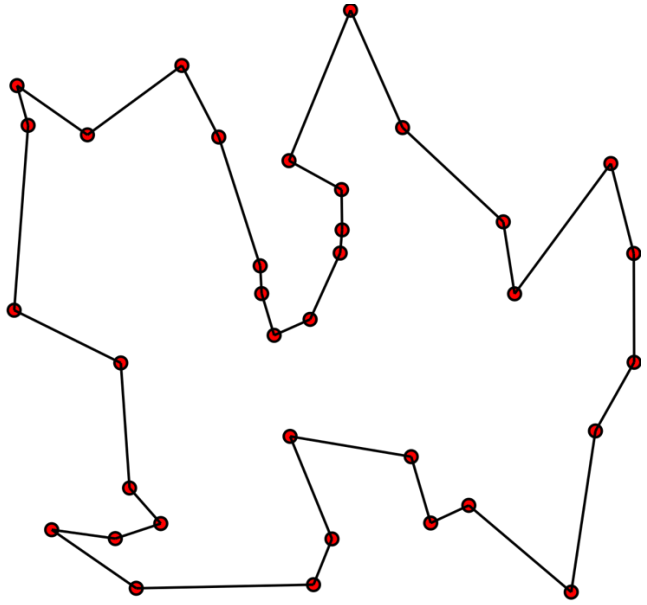

Figura 11: PCV com 35 cidades resolvido
Fonte: Wikipédia

Ao observar um PCV já resolvido, como visto na Figura 11, nota-se que a formação da rota mais eficiente gera uma forma geométrica, que circunda o perímetro dos pontos, conectando os externos até que se feche em uma forma contendo todos os vértices.

Outra forma de ilustrar isto, é imaginando que esta forma geométrica fosse um objeto a ser embrulhado para presente, se fosse uma caixa, é simples de conceber que a mesma teria 6 lados retos, mas sendo um objeto com sua "topologia mais acidentada" como um abajur, a folha de embrulho ficaria com mais quinas ao preenche-lo.

Mas enxergando de uma forma bidimensional, que possui mais similaridade com um problema de grafo planar como é o PCV, a caixa daria lugar a um retângulo, e o abajur, a um desenho com mais vértices, pois seria inviável desenhar um abajur ligando apenas 4 pontos, mas quanto mais pontos são dispostos, melhor sua figura é ilustrada. Da mesma forma com a representação de um círculo, quanto mais lados há, melhor será desenhado sua curvatura, afinal, um decágono está mais próximo de um círculo que um triângulo.

Para "embrulhar" os pontos a fim de chegar na forma geométrica mais eficiente, podemos partir de um retângulo cujo cada lado está exatamente no eixo referente ao ponto mais próximo das bordas. Ou seja, o lado esquerdo passará sobre o ponto que está mais a esquerda, o superior, sobre o que está mais acima, assim como o da direita e inferior. Dessa forma, independente de qualquer que seja a melhor rota, ela será alguma forma geométrica que está no interior deste quadrilátero formado.

Resta agora conectar os outros pontos, imaginando que cada um dos lados estivesse aproximando-se do centro e envolvendo os pontos que estiver neste caminho, até que todos estejam sendo visitados pelas bordas deste originalmente quadrilátero e que agora cada ponto se tornou um vértice de sua forma.

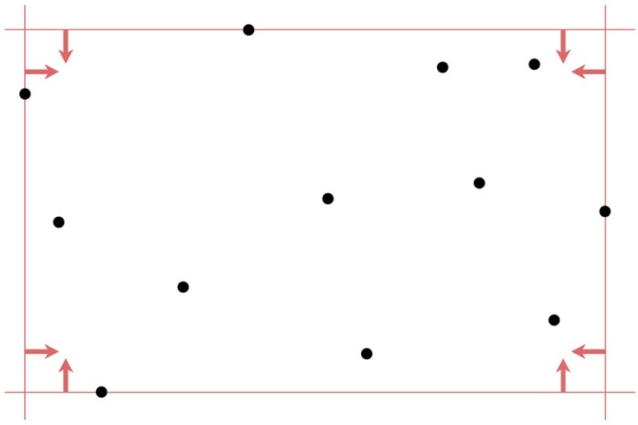

Figura 12: Bordas caminham ao centro “embrulhando” os pontos que passam
Fonte: autor

Mas há um problema, pois é necessário definir qual lado será responsável por preencher determinado ponto, pois como eles seguem em direção ao centro, todo ponto será visitado por duas retas, afinal, se cada lado se desloca até a metade do quadrilátero, há dois lados em cada eixo, horizontal e vertical, caminhando ao seu encontro e separando assim em 4 segmentos: superior esquerdo, superior direito, inferior esquerdo e inferior direito. Cada ponto está localizado em algum desses 4 cantos, e, portanto, será visitado pelos dois lados de seu respectivo lado, como por exemplo, o que está no superior esquerdo, será visitado pelo lado esquerdo e de cima.

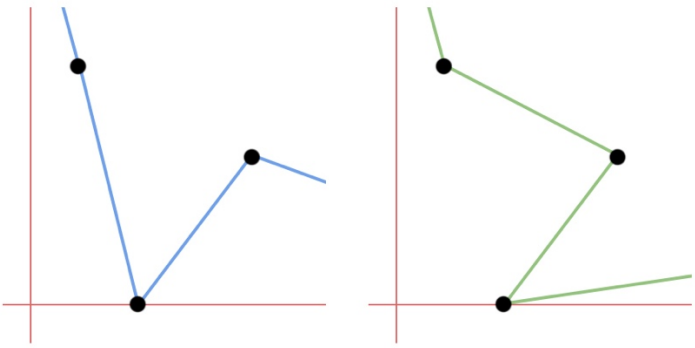

Figura 13: Para qual lado o ponto deverá dar prioridade?
Fonte: autor

Ao dar prioridade para o lado cujo o ponto está mais próximo, afinal, isto significa que ele está mais pertencente a este eixo, é então definido as arestas externas da forma geométrica. Resta definir como será realizada a conexão entre os pontos, e escolhendo seguir um caminho, seja no sentido horário ou anti-horário, e ligando cada ponto com o que está mais próximo e no mesmo lado que o seu, é então criado a seguinte heurística:

```
1. Defina as bordas do quadrilátero referente aos pontos mais
   externos de cada eixo;
2. Verifique a qual lado cada ponto está mais próximo;
3. Escolha um ponto inicial e caminhe em direção a algum
   sentido, horário ou anti-horário;
4. Ligue os pontos na ordem até que tenha retornado ao ponto
   de origem.
```

Código 4: Pseudocódigo para heurística do "embrulho"
Fonte: autor

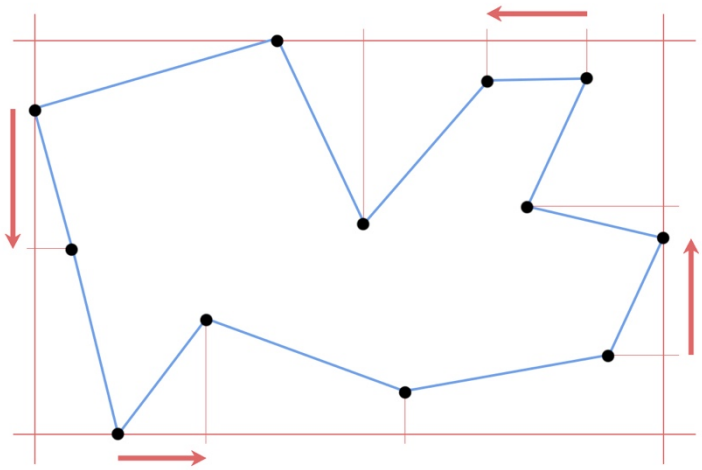

Figura 14: Ilustração da heurística do “embrulho”
Fonte: autor

Os resultados são satisfatórios, mas apenas para uma baixa quantidade de pontos, como demonstrado na Figura 15. Pois este algoritmo falha ao “embrulhar” os pontos que estão mais próximos ao centro, já que ele não considera que antes de conectar com um ponto com outro, que apesar de estarem próximos no eixo ao qual eles pertencem, pode haver um outro mais distante neste mesmo eixo, mas que esteja mais similar em relação ao outro eixo, como demonstrado na Figura 16. E quando há uma grande quantidade de pontos, as chances de ocorrer essa falha, aumentam significativa, tornando este método ineficiente.

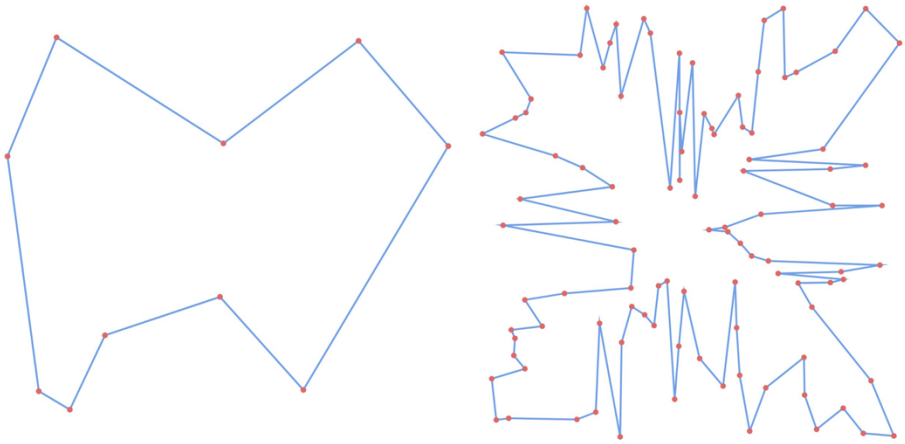

Figura 15: Resultado com 10 e 100 pontos
Fonte: autor

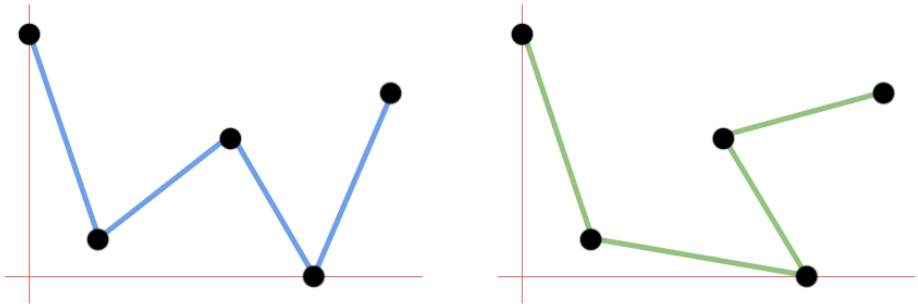

Figura 16: Como o algoritmo "embrulha" / Como deveria "embrulhar"
Fonte: autor

Talvez o método do embrulho traga resultados eficientes, mas não se implementado desta maneira, pois não contempla com exatidão como o modelo deveria se comportar. Mas já que não foi encontrado nenhuma forma alternativa a esta, a solução proposta foi abandona e dado seguimento a outro tipo de método.

4.1.1.2 Priorizando as melhores conexões

Quando dizemos que para 5 pontos há 120 possibilidades, pois equivale a 5 fatorial, significa que a ordem no qual os pontos são escolhidos, ou as conexões são selecionadas, é relevante para essa contagem do total de trajetos possíveis. Cujo é um valor bem diferente do total de conexões existentes, e que por sua vez é diferente do total de conexões utilizadas.

Já que cada ponto pode conectar-se com qualquer outro ponto, com exceção de si mesmo e as outras conexões que já foram utilizadas, já que, de A para B é o mesmo que de B para A. O total de conexões existentes, é o somatório de todas as conexões que a primeira cidade escolhida pode fazer (número de cidades existentes menos si própria), mais o total de todas que a segunda pode fazer (número de cidades existentes menos si própria e a primeira), e assim sucessivamente até que tenha chegado à última cidade.

Ou seja, para 5 cidades, o total de conexões existentes será $4 + 3 + 2 + 1 = 10$, que matematicamente falando podemos representar como uma Progressão Aritmética de razão 1, onde a quantidade de termos será a quantidade de cidades menos 1, pois a primeira cidade não pode conectar consigo própria, apenas com as outras restantes.

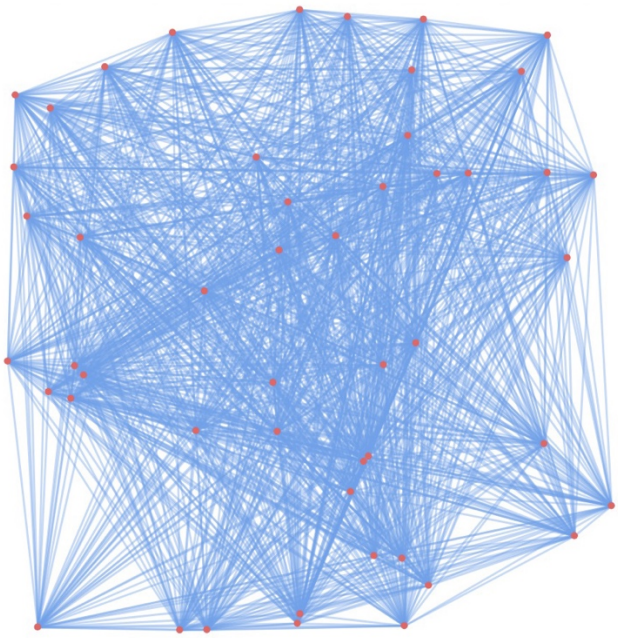

Figura 17: Conexões existentes entre 50 pontos
Fonte: autor

Já para descobrir o total de conexões utilizadas é ainda mais simples, afinal, se cada cidade leva a próxima até que ao término do circuito chegue ao mesmo ponto de partida, o total de conexões utilizadas em um circuito hamiltoniano, ou para resolver o PCV, é justamente a quantidade de cidades existentes.

Entender a diferença entre essas 3 quantidades é importante para conceber que dentre as conexões existentes estão as conexões que serão utilizadas, e a ordem na qual estas são conectadas está no total de conexões possíveis. Em um cenário ideal, a melhor rota possível

seria a que utilizasse as *N* melhores conexões existentes, onde *N* é a quantidade de rotas utilizadas, ou quantidade de cidades existentes.

Porém ao definir desta forma não está levando em consideração que uma conexão escolhida pode estar conectando duas cidades que já tem suas conexões definidas, afinal, cada cidade é ligada com outras duas, a que chega até ela e a próxima para onde ela leva. Se ao escolher apenas as melhores rotas, alguma que for selecionada conectar duas cidades que já possuem suas duas conexões, estaria rompendo com a restrição de que uma cidade só pode ser visitada uma vez, além de deixar cidades sem nenhuma conexão como demonstrado na Figura 18.

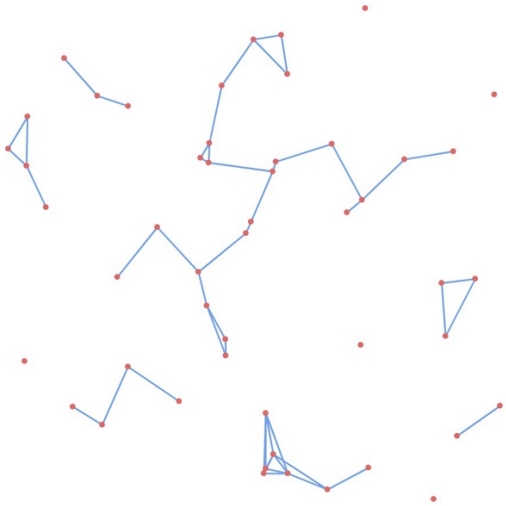

Figura 18: Priorizando as melhores conexões
Fonte: autor

Como forma de resolver esse problema, houve inspiração no algoritmo de Kruskal, já explicado na sessão 2.1.2, onde ao utilizar a próxima melhor conexão, é verificado se cada uma das cidades na qual ela conecta já possui suas 2 conexões, caso seja verdade, esta é descartada, e então a próxima melhor é testada. Somente se ambas as cidades a qual ela conecta estejam disponíveis (com no máximo 1 conexão) é que a mesma é utilizada. Os resultados desta heurística podem ser vistos na Figura 19, e sua definição é descrita da seguinte forma:

```
1. Pegue todas as conexões existentes com suas respectivas
   distâncias;
2. Coloque-as ordem crescente;
3. Partindo da primeira até a última, repita:
   3.1.  Verifique se alguma das duas cidades a qual esta
      conecta, já possui mais que 1 conexão
      3.1.1.   Caso verdade: Vá para a próxima conexão;
      3.1.2.   Caso falso: Utilize esta conexão.
```

Código 5: Pseudocódigo para heurística de "priorizando as melhores conexões"
Fonte: autor

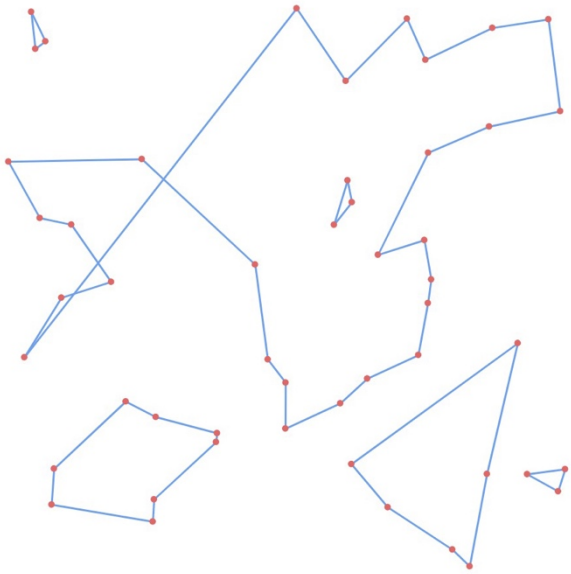

Figura 19: Resultados da heurística "priorizando as melhores conexões"
Fonte: autor

Como pode ser observado na Figura 19, apesar de não haver a violação de algum vértice possuir mais que 2 conexões, há o problema das conexões se fecharem isolando-se em grupos, mas que, mesmo após resolvido isso, criando uma função que verifica quando tal evento ocorre, o desempenho ainda assim não foi satisfatório, principalmente quando o número de cidade é aumentado.

Portanto, esta estratégia falha no que se compromete a entregar a melhor resposta, e até mesmo para uma resposta razoável ela não serve tão bem. Após esta tentativa, o objetivo de obter a resposta exata foi abandonado, dando lugar a uma busca por um método de proximidade, onde encontra um resultado satisfatório, mas em um tempo hábil. Tendo como referência o desempenho humano ao resolver o problema com uma baixa quantidade de cidades.

4.1.2 Tentando simular o desempenho humano

É interessante notar que ao perguntar a alguém cujo tenha acabado de resolver o problema com uma pequena quantidade de pontos, mesmo tendo acertado a melhor rota, sente dificuldade ao descrever o porquê de ter tomado tal decisão como observado na Figura 20. Por outro lado, quando há uma quantidade maior de pontos, é fácil perceber que a pessoa utiliza um critério semelhante ao de escolher uma cidade qualquer como ponto inicial, e a partir desta, ir escolhendo as mais próximas até ligar todas.

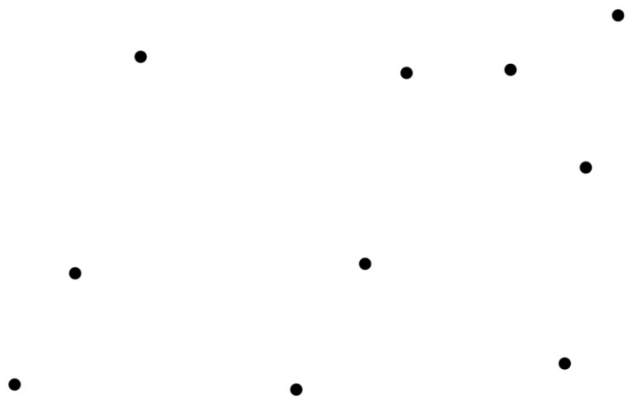

Figura 20: Tente explicar o critério que usou para ligar estes pontos
Fonte: autor

O problema é que conforme o número de pontos aumenta, a qualidade da resposta obtida diminui. Em suma pela maior quantidade de opções, que pode causar dúvida sobre qual caminho seguir. Porém o maior diferencial está em que, para uma quantidade pequena de pontos a pessoa consegue visualizar o caminho que fará antes de iniciar o trajeto, enquanto que para quantidade altas, ela seleciona um ponto e a partir dele vai tomando pequenas decisões.

Este diferencial é importante pois deduz-se que são critérios diferentes ao ser utilizados, visto que obter uma rota observando todos os pontos, é uma decisão mais precisa do que ir resolvendo de trecho em trecho, onde preocupa-se apenas em qual será a melhor próxima escolha, sem se questionar se tal decisão poderá acarretar em uma péssima rota ao término.

Por isso, é importante descobrir qual o critério utilizado na resolução de poucos pontos, mesmo que a pessoa diga “não sei o porquê, eu só olhei e fiz”. Para obter essa resposta é preciso primeiro descobrir em qual quantidade de pontos que a resposta deixa de ser intuitiva e a pessoa passa a adotar o método de ligar os mais próximos.

Levando em conta que este valor é relativo a depender de cada pessoa e da posição dos pontos, o foco deixa de ser em descobrir a exata quantidade, e passa a ser em analisar as soluções respondidas em uma quantidade que não seja, nem tão baixa a ponto de ser intuitivo, e nem tão alta a ponto de adotar o outro critério.

Após repetir o processo de resolução nesta condição, e para cada nova vez, os pontos estejam em posições diferentes, observa-se que há a preferência em conectar pontos que estejam próximos entre si, mas diferentemente de quando conecta-se a partir de uma linha, e sim focando nos pequenos agrupamentos e depois interligando-os.

4.1.2.1 Definindo o conceito de próximo

Para conectar um ponto que esteja próximo a outro, é necessário definir o que vem ser um ponto próximo, afinal, não há uma distância definida onde a partir de tal valor o ponto deixa de estar perto e passa a estar longe. É necessário um referencial, para comparar e definir que tal ponto está mais perto que outro em relação a um em específico, e não definir uma distância específica em que a partir desse valor o ponto deixa de estar perto e passa a estar longe.

Ou seja, a partir de um determinado ponto, é possível dizer quais estão mais próximos dele, ao comparar a distância entre este e cada um de todos os outros restantes. Dessa forma, é possível que cada ponto possua uma lista ordenada de outros pontos, onde o primeiro é aquele mais próximo em relação a si, e o último o mais distante.

Com esta lista obtida para cada ponto, resta definir qual critério será adotado para utilizar destas informações e assim efetuar as conexões. Neste trabalho foi testado duas abordagens, a primeira será introduzida na próxima seção, a segunda é a heurística principal da monografia.

4.1.2.2 O Pior Melhor

Na primeira heurística apresentada, o intuito estava na forma que a rota adquiria, na segunda, havia a preocupação quanto ao comprimento das conexões. Nesta heurística, o foco está no interesse do ponto ao se conectar, o que se aproxima à heurística principal deste trabalho.

Se quando foi priorizado as conexões a hipótese era “a melhor rota possível é aquela que possui as N melhores conexões”, nesta heurística, a hipótese está para “a melhor rota possível será aquela em que todos os pontos estejam conectados com o mais próximo de si”. Pois já que todos os pontos precisam ser conectados com outros dois, o melhor cenário será aquele em que todos possam se conectar com os que melhor lhe favorecem, isto é, com os dois que estejam mais próximos de cada um.

Novamente há o problema de que um ponto poderá se conectar com um outro que já possua suas duas conexões realizadas, e também deixando pontos sem nenhuma conexão. E como forma de resolver esse problema, é preciso definir qual ponto terá a prioridade de escolha, e nesse caso foi optado pela pior melhor opção.

A lógica é simples, se cada ponto possui seu respectivo ao qual gostaria de se conectar, baseado no que está mais próximo em relação a si, o ponto a ser priorizado é aquele em que sua melhor opção é a conexão mais distante dentre a melhor opção de todos os outros. Pois se a melhor alternativa que ele tem, já é relativamente ruim, caso seja priorizado os outros pontos, ele ficará com as opções restantes, que são ainda bem piores.

Portanto, o procedimento é verificar para cada ponto qual dentre todos os outros é o que está mais próximo de si, e a distância que os separa. Em seguida, é comparado a melhor escolha de cada um, e aquele que possui a opção de maior distância dentre o melhor de todos os pontos, é que terá sua conexão estabelecida.

Uma vez que este ponto se conectou com o que deseja, estes dois agora pertencem a um grupo, e portanto, agora devem escolher apenas um ponto ao qual desejam-se conectar. Para que isso seja determinado, é comparado a melhor opção entre os dois, e aquele que tiver sua conexão de desejo com a menor distância é o que irá representar a melhor conexão possível deste grupo.

Vale ressaltar que a partir de 3 pontos em um grupo, os pontos intermediários às extremidades, isto é, aqueles que estão entre as duas pontas do grupo, já possuem suas duas conexões efetuadas, e logo, não há necessidade de verificar a qual ponto estes desejam-se conectar. Então, sempre que foi “perguntado” a um grupo com qual ponto ele gostaria de se conectar, é verificado apenas as opções de escolha das duas extremidades.

Como um ponto isolado ainda não foi conectado com ninguém, ele por si só já é uma extremidade, e quando há dois pontos, eles também já são as extremidades, uma vez que não há ninguém entre eles.

O objetivo é repetir este processo até que reste apenas um único grupo contendo todos os pontos, como pode ser descrito pelo algoritmo abaixo:

```
1. Cada ponto é considerado um grupo, e este processo deve ser
   repetido até que reste apenas um grupo:
      1.1.1.   Verifica-se o ponto mais próximo referente a
         cada um dos grupos existes;
      1.1.2.   Aquele que possuir a pior melhor escolha, isto
         é, seu ponto mais próximo é o mais distante em relação
         ao dos outros, é o que terá a conexão realizada;
      1.1.3.   Após uma conexão ser efetuada, os integrantes
         dos dois grupos agora pertencem a apenas um, e
         portanto, só podem escolher uma opção que representa
         sua melhor conexão possível.
```

Código 6: Pseudocódigo para heurística do "pior melhor"
Fonte: autor

Este algoritmo é muito similar ao principal proposto nesta monografia, onde mantém-se o critério unir grupos até que restam apenas um, porém o critério ao qual uma conexão é estabelecida deixa de ser considerado o pior melhor e passa a selecionar as conexões em que haja reciprocidade entre os pontos, isto é, ambos possuem o desejo de conectar-se entre si. E os resultados forem bem mais satisfatórios.

4.2 A HEURÍSTICA DA RECIPOCIDADE

Cada ponto possui o “desejo” de se conectar com aquele que está mais próximo de si; Uma conexão é realizada se há reciprocidade entre os pontos, isto é, se ambos querem se conectar; Após estabelecida uma conexão entre pontos, estes agora pertencem a um grupo, e portanto, o desejo de se conectar aos outros se da pela melhor conexão possível entre suas extremidades, ou seja, verifica-se o desejo de conexão do ponto da esquerda e o da direita, e a que possuir a menor distância será o desejo de conexão deste grupo; Considerando que no início nenhum ponto está conectado, existem portanto n grupos, onde n é o número de pontos existentes; A heurística termina quando há apenas 1 grupo, ou seja, todos os pontos estão pertencentes a um mesmo grupo.

Como descrito na sessão Priorizando as melhores conexões o ideal seria com que todos os pontos pudessem se conectar com aqueles que desejam, isto é, com sua melhor opção, o mais perto de si. Porém, se não utilizar nenhum critério de definição, as conexões estabelecidas irão contra as regras do problema, pois haverá pontos com mais de uma conexão e outros com nenhuma.

O que esta heurística faz é definir um critério de quais as conexões devem ser realizadas, e leva-se como base que as que são recíprocas são as melhores, já que a conexão entre estes dois pontos não afeta nenhum outro ponto terceiro, eles estão em acordo com a união.

Porém, observa-se que nem todos os pontos possuem um mútuo à si, e se aplicado esse critério, estes ficarão sem nenhuma conexão. Mas uma vez que houve um agrupamento entre dois pontos, eles podem e devem continuar se conectando com os outros, porém dessa vez não faz sentido se cada ponto desde grupo puder escolher o seu, é preciso portanto, definir qual será a melhor conexão que este grupo pode fazer.

Isto é definido ao comparar a distância da conexão que cada ponto da extremidade, já que em um grupo de por exemplo 3 pontos, o do meio já está com suas duas conexões realizadas, enquanto os da extremidade apenas com um, e verificar qual dessas duas é a menor. Ou seja, o ponto da extremidade da esquerda possui seu desejo de conexão, e o mesmo com o da direita, é então comparado qual desses possui a menor distância, e este portanto, será a conexão que o grupo deseja realizar.

Vale observar que da mesma forma que não faz sentido verificar o desejo de conexão de um ponto que já esteja com duas conexões realizadas, também não faz sentido consultar a distância entre um ponto com outro que já esteja com as duas conexões. Dessa forma, todas as comparações verificadas são feitas apenas entre as extremidades de um grupo com as extremidades dos outros grupos, onde no caso de haver apenas um ponto isolado, ele já é uma extremidade por si só.

Isso justifica que ao iniciar a heurística a quantidade de grupos seja exatamente a quantidade de pontos, e conforme é estabelecida uma conexão, estes dois agora tornam-se um grupo. O processo então vai se repetindo até que reste apenas um único, cujo contém todos os pontos, formando assim uma lista ordenada dos pontos conforme devam ser conectados.

Para que essa ordem final esteja correta, é preciso que durante a união entre dois grupos, o novo gerado precisa condizer com a conexão estabelecida. Para isso não basta apenas saber quais os dois grupos a serem unidos, e sim por onde será feita a conexão entre eles. Considerado que um grupo pode ser conectado por sua extremidade da esquerda ou da direita, ou da frente e de trás dependendo de como é representado, e que uma conexão depende da união de dois grupos, há portanto quatro opções de conexões possíveis como ilustrado na Figura 21.

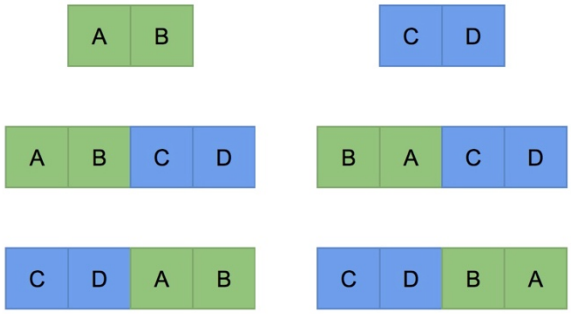

Figura 21: As 4 formas possíveis de conexão entre 2 grupos
Fonte: autor

Essa preocupação de como unir dois grupos é de suma importância para a ordem entre as cidades não seja prejudicando durante a união entre os grupos, já que representando visualmente é simples perceber por onde um grupo se conecta com outro, porém quando aplicado em algoritmo, onde cada ponto possui seu identificador e um grupo é uma ordem de pontos, durante a união é preciso representar corretamente por onde estes se conectaram.

Dessa forma, ao realizar todos os ciclos em que resultam em apenas um grupo, este estará na exata ordem em que os pontos devam ser conectados, respeitando a heurística de ir conectando

4.2.1 O Problema do Emparelhamento Estável

A heurística da Reciprocidade possui similaridades como o Problema do Emparelhamento Estável (*Stable Marriage Problem*) cujo retrata a questão de se é possível, dado dois conjuntos de tamanho igual, estabelecer conexões de entre dois elementos de cada conjunto, de modo que todos estejam pareados de forma estável. Onde estabilidade aqui é entendida como, em casos de não haver dois elementos que gostariam de se conectar, porém ambos já estão atrelados com outros elementos.

Em outras palavras, supondo que em uma vila há 5 homens e 5 mulheres, o problema questiona se é possível que todos se casem sem que haja nenhum casal onde gostariam de estar juntos, mas ambos já estejam casados com outra pessoa. Ou seja, apesar de estarem casados, eles preferem terminar seu atual relacionamento e ficarem juntos.

Lloyd S. Shapley e Alvin E. Roth receberam em 2012 o prêmio Nobel em economia ao responder essa questão no trabalho "*The Theory of Stable Allocations and the Practice of Market Design*", onde apresentaram a comprovação de que é sim possível estabelecer uma

união estável, dado dois conjuntos de mesma quantidade de elementos, onde cada um precisa se conectar com o outro. E apesar de não ser um problema que necessariamente envolva números, a solução pode ser descrita de forma algorítmica como no pseudocódigo abaixo:

```
funcao emparelhamentoEstavel {
    Inicialize todos os h ∈ H e m ∈ M como solteiros
    Enquanto (∃ um homem solteiro h que ainda foi chamado por
uma mulher m para namorar {
        m = primeira mulher na lista lista do homem m que ainda
não chamou ele para namorar
        Se m esta disponivel
            (h, m) começam a namorar
        Senao (h', m) já namoram
            Se m prefere o h do que o h'
                h' fica solteiro
                (h, m) começam a namorar
            Senao
                (h', m) continuam namorando
    }
}
```

Código 7: Pseudocódigo para resolução do "problema do emparelhamento estável"
Fonte: Wikipédia

A solução visa com que cada elemento possua um *ranking* das melhores opções que tem referente aos elementos aos quais pode se conectar, o que é similar à heurística da reciprocidade, no que se refere a que cada ponto possui uma "lista" das melhores conexões possíveis, onde a preferência se da pela menor distância de si.

O que é feito na solução de Shapley e Roth é que um primeiro conjunto envia propostas de união a sua primeira melhor opção, e aqueles do segundo conjunto que receberam alguma proposta, devem escolher a melhor opção dentre estas, porém, estas escolhas não estão definitivas, uma vez que o processo continua, onde o primeiro grupo agora escolher a próxima melhor opção, e o segundo grupo, dentre as propostas recebidas, compara com as decisões feitas anteriormente, e se forem mais vantajosas para eles, ocorre a mudança. Ao término dos ciclos, todos estarão unidos e sem que haja nenhum casal instável.

Porém, o que difere da heurística da reciprocidade, é que ao contrário de haver dois grupos em que se conectam entre si, aqui a conexão pode ser realizada com qualquer membro

do único conjunto. Além disso, uma conexão não se restringe a apenas dois elementos, pois o processo se repete até que todos estejam pertencentes a uma única conexão.

4.2.2 Generalizando a função

Durante o desenvolvimento, foi utilizado o paradigma Orientado a Objeto, onde havia as classes de Pontos, para representar as propriedades individuais de cada um, como posição e sua exibição, e a de Grupos, onde estes armazenavam o conjunto de pontos ao qual lhe pertenciam.

Nessas classes haviam os métodos que efetuavam os procedimentos da heurística, como identificar a melhor opção para cada ponto e efetuar a união entre os grupos. Foi inclusive utilizado temáticas como cada ponto ser na uma ilha, e após as uniões, arquipélagos eram formados. Ou então na temática de amigos ou família, em que cada ponto era uma pessoa e a após a união estes eram um grupo de amigos ou família. E novamente havia como objetivo formar um único arquipélago, grupo ou família que representasse a melhor conexão entre eles.

O problema disso estava na restrição de que a heurística só funcionaria para este determinado cenário, utilizando estes objetos e no caso da implementação, baseado em quantos *pixels* havia de distância entre cada ponto. Dessa forma, não seria simples utilizar o algoritmo caso mude necessidades do problema, como por exemplo, se cada ponto for de fato uma cidade com sua coordenada geográfica, seria preciso modificar o código para que se adeque a este cenário.

Surgiu então a necessidade de criar uma implementação mais genérica, que fosse capaz de servir para qualquer tipo de necessidade relacionado ao PCV. Porém, para isso, era preciso definir o que todos estes possuem em comum, e já que trata-se de um problema onde é preciso encontrar o trajeto de menor distância total entre os vértices, basta apenas justamente saber a distância entre cada um destes.

Dessa forma, saber todas as distâncias que cada vértice possuía em relação a todos os outros, já era informação suficiente para aplicar a heurística e determinar a ordem de conexão entre eles. Afinal, para que ela funcione é preciso saber a melhor conexão de cada grupo, e para formar um grupo, basta saber quais pontos tiveram o desejo de conexão em comum.

Ou seja, a heurística só precisa receber a distância entre todos os vértices para que seja capaz de retornar o resultado, e estas distâncias podem representar qualquer coisa, seja endereços de residências, coordenadas geográficas entre cidades, posições de solda em uma placa de circuito, ou qualquer outra necessidade. A única restrição é que todas as distâncias estejam na mesma unidade de medida, e assim, o algoritmo é capaz de aplicar sua heurística.

Uma forma de determinar todas as distâncias é através de uma tabela, onde tanto nas linhas como nas colunas, há todos os vértices existentes, e onde estes se interceptam, é a célula que contém a distância entre elas. O problema disso é que nessa tabela há a redundância de informação, já que, por exemplo, na célula da linha C coluna F encontra-se o mesmo valor da célula na linha F coluna C, uma vez que a distância entre C e F é a mesma de F para C.

Há cenários onde as conexões entre os vértices são assimétricas, isto é, a distância de ida não é a mesma que a de volta, como por exemplo, no deslocamento em uma cidade, há ruas de mão única onde para ir não passa pelo mesmo caminho que para voltar, e portanto, há divergência entre as distâncias dependendo do sentido ao qual é deslocado. Para estes casos a tabela completa se faz necessário, porém para este trabalho, como possui o intuito de buscar a otimização da resolução, foi considerado que um cenário de deslocamento simétrico, onde dessa forma, é possível economizar pela metade o gasto na memória de onde é armazenado a distância entre os pontos.

Para que essa economia seja possível, basta utilizar apenas metade da tabela das distâncias, uma vez que cada distância possui duas vezes o registro na tabela. Porém a divisão para obter-se a metade necessária, não está em cortar a tabela num eixo vertical ou horizontal, e sim na diagonal, mais precisamente através da diagonal principal, cujo começa no canto superior esquerdo e termina no inferior direito, como representado na Figura 22.

	A	B	C	D	E
A	0	284	950	318	943
B	284	0	320	102	439
C	950	320	0	403	750
D	318	102	403	0	449
E	943	439	750	449	0

Figura 22: As duas metades com valores duplicados
Fonte: autor

Assim, basta passar essa estrutura para a heurística que a mesma será capaz de efetuar as operações necessárias para retornar a ordem das conexões, onde cada vértice será identificado através de sua posição na linha, ou seja, a metade da tabela utilizada pela heurística

é a inferior esquerda, pois nesta que contém todas os vértices em ordem pelas linhas. Pois se fosse utilizada a metade superior direita, os vértices deveriam ser referenciados pelas colunas, o que seria menos intuitivo na hora de implementar o algoritmo.

Também não é necessário manter a própria diagonal principal, pois esta contém as distâncias entre os pontos com eles mesmos, e o problema não visa tal conexão. Porém, ao remove-la, a primeira linha, que contém o primeiro ponto, é também removida, resultando na omissão deste, ou seja, se há 10 pontos, haverão apenas 9 linhas já que a primeira foi removida.

Como forma de resolver isso, é preciso adicionar um novo elemento no topo, ou no caso, na primeira linha, para que este represente o primeiro ponto e assim a heurística seja capaz de identifica-lo.

Mas o valor que esta célula indica, é a distância entre o primeiro ponto com ele mesmo, que resulta em 0. Isso é um problema, já que a heurística busca pela melhor opção de cada ponto, e neste caso, o primeiro ponto não irá desejar conectar-se com nenhum outro, já que sempre preferirá conectar-se consigo próprio, pois esta distância é nula. Portanto, ao invés de inserir o valor 0, é preferível inserir o valor infinito, pois assim, nenhuma distância entre qualquer que seja o ponto, será superior a distância dele até ele mesmo. O resultado de como a estrutura fica, pode ser visualizado na Figura 23.

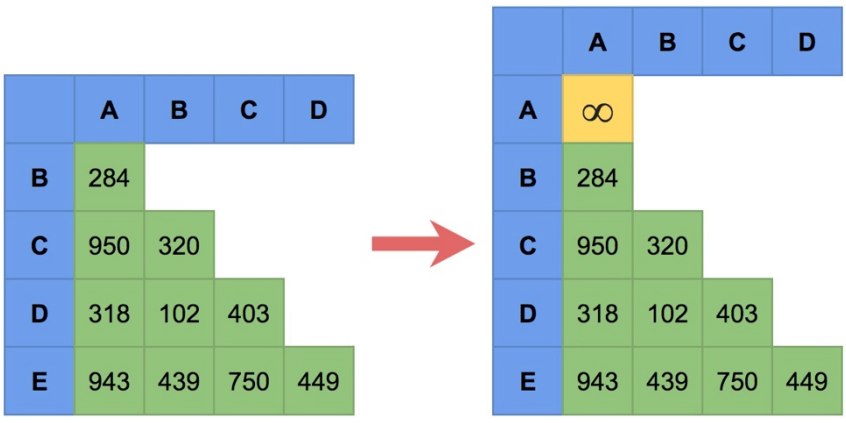

Figura 23: Inserir o valor infinito para ajustar a estrutura das distâncias
Fonte: autor

4.3 IMPLEMENTAÇÃO

Como forma de testar a eficiência da heurística, foi implementado um algoritmo capaz de executar os passos descritos pela mesma, e assim, coletar informações sobre seu desempenho. O código[4], bem como sua demonstração[5], estão disponíveis online através da hospedagem no *GitHub Pages*[6].

O intuito era criar uma interface onde o usuário pudesse inserir como valor de entrada a quantidade de cidades, ou neste caso pontos, a serem conectados pelo algoritmo. Para cada nova vez em que é executado, são gerados a exata quantidade de pontos fornecida, porém em posições aleatórias na tela, tendo como ambiente, um cenário bidimensional, onde cada ponto possui seus respectivos valores de cada um dos dois eixos, vertical e horizontal.

Ou seja, é uma implementação em ambiente planar, em que a conexão entre cada ponto pode ser feita em uma linha reta, não havendo restrições como ruas ao qual trafegar, mas que também não descarta a possibilidade de utiliza-las, já que o algoritmo define a rota com base nas distâncias entre cada vértice.

4.3.1 Ambiente de resolução

Para a implementação do método e visualização dos resultados foi utilizado a linguagem de programação *JavaScript* em conjunto com o *framework p5.js* [8]. Apesar de ser uma linguagem cujo não é disposta de um alto poder de processamento, visto que a mesma é interpretada pelo navegador, sua utilização possibilita a execução do código em praticamente qualquer ambiente, possibilitando assim uma maior gama de testes a serem efetuados em diferentes dispositivos. Além disso, o *framework* disponibiliza ferramentas que facilitam a visualização do problema, ou seja, com ele é possível ver os pontos e as rotas que o algoritmo define.

Porém, apesar da implementação possuir representação gráfica, o algoritmo principal, isto é, a heurística do trabalho, independe de qualquer ambiente ou interface, pois ela pode ser

---

[4] https://github.com/lucaslopes/tsp - data de acesso: 17/10/17
[5] https://lucaslopes.github.io/tsp - data de acesso: 17/10/17
[6] https://pages.github.com/ - data de acesso: 30/08/17

implementada em praticamente qualquer linguagem de programação. Ou seja, para este trabalho foi desenvolvido não somente a heurística, como também o ambiente ao qual será criado e representado os pontos e o circuito.

Isso significa que é possível implementar a heurística utilizando uma linguagem como C, cujo é uma ótima opção para executar tarefas que justamente demandam otimização. Além disso, a mesma também independe do cenário ao qual os vértices serão fornecidos, pois apenas precisa das distâncias entre os pontos em uma mesma unidade de medida, seja centímetro, metros, pixels, polegadas, etc, para que entregue seu resultado.

Cabe também observar, que apesar de utilizar uma linguagem interpretada e de algo nível como o *JavaScript*, os resultados não foram comprometidos, pelo contrário, foram bastante satisfatórios como será demonstrado ao final deste capítulo. Isso significa, que se neste ambiente já houve bom desempenho, em um ambiente de baixo nível, poderá apresentar resultados ainda melhores.

4.3.1.1 Estrutura do código

Foram necessários apenas 5 arquivos para a implementação de todo a heurística, onde, por ser utilizado a linguagem *JavaScript*, um deles é um arquivo HTML (*index.html*), ao qual serve como base para interpretação pelo navegador. Além disso, como é utilizado o *framework p5.js*, este é o segundo arquivo necessário para a implementação (*p5.min.js*).

Dos outros 3 restantes, temos a classe Ponto (*Point.js*), cujo tem o propósito de representar os vértices do problema. O arquivo principal (sketch.js), ao qual é através dele que são interligados os outros módulos e chamadas as funções em sua devida procedência. E por fim, a heurística em si (*reciprocity.js*), onde nele está todo o código necessário para resolução do problema, ou seja, os outros são apenas para a representação deste cenário.

Caso deseja-se implementar a solução em outra linguagem, é no arquivo *reciprocity.js* que está todas as funções necessárias para a resolução do problema, bastando apenas seguir o parâmetro ao qual ela necessita, cujo são as distâncias entre cada vértice, independente de como estes são representados.

4.3.1.2 Arquivo HTML

É um código extremamente simples, nele há apenas a referência para os outros 4 arquivos *JavaScript*. Todas as partes visuais são implementadas no arquivo *sketch.js* através das ferramentas fornecidas pelo *p5.min.js*. Apenas por uma questão uma questão de estética, foi inserido um título na página (TSP), e removido as bordas do *canvas*, cujo é onde os pontos e rota são "desenhados", deixando-o ocupar toda a tela do navegador.

```
<!DOCTYPE html>
<html>
  <head>
    <title>TSP</title>

    <script src="js/p5.min.js"></script>
    <script src="js/Point.js"></script>
    <script src="js/reciprocity.js"></script>
    <script src="js/sketch.js"></script>

    <style> body { padding: 0; margin: 0; } </style>
  </head>
  <body>
  </body>
</html>
```

Código 8: Arquivo HTML do código
Fonte: autor

4.3.1.3 Biblioteca p5.js

Criado por Lauren McCarthy e desenvolvido por uma comunidade de colaboradores, o *framework p5.js* é uma biblioteca de código aberto que tem o suporte da Universidade de Nova Iorque (NYU) e da fundação *Processing*, cujo tem por objetivo tornar a programação mais acessível à diversas pessoas, através de ferramentas que facilitam a criação de conteúdos visuais através da programação.

A iniciativa começou com a linguagem Java, e hoje tem suporte a Python e JavaScript com o framework *p5*. E hoje já obteve diversas conquistas ao levar programação para jovens e artistas, onde tem como foco explorar a criatividade e despertar interesse pela programação.

O professor do NYU, Daniel Shiffman, que está nos agradecimentos desta monografia, é um dos divulgadores do projetos através de seu canal no *YouTube*[7] onde ensina não somente utilizar a biblioteca, como também conceitos de programação, onde um deles foi justamente sobre o PCV. Foi através desta inspiração que o *framework p5.js* foi o utilizado neste trabalho, onde é responsável por ilustrar as cidades e rotas, e dessa forma, torna a visualização do processo da heurística muito mais legível, tanto para usuários leigos como programadores.

A biblioteca possui 3 módulos, o *core*, cujo é o arquivo principal; O *dom*, que tem por objetivo facilitar a interatividade com elementos do *Document Object Model* na página HTML; E o *sound*, que visa agregar recursos de áudio para os projetos. Há ainda outros módulos criados pela comunidade e que podem ser explorados no site do projeto[8].

Para este trabalho foi utilizado apenas o módulo principal, pois este já possui todos os recursos necessários para a ilustração da heurística. E como o código da biblioteca é extenso e não é de criação deste trabalho, seu código não será posto aqui, mas é possível acessa-lo no perfil da organização no *GitHub*[9].

4.3.1.4 Classe Ponto

É composta por apenas dois simples métodos, o construtor (*constructor*), cujo instância o objeto, e o responsável por exibir o ponto na tela (*show*).

O método construtor recebe como parâmetros dois valores, o primeiro para indicar a posição no eixo horizontal, e o segundo para o eixo vertical. Após receber estes valores, é criado um vetor, que é um objeto fornecido pelo *framework* onde possui funcionalidades como calcular a distância entre outro vetor, mudar sua magnitude, escalonar ou qualquer outra operação matemática referente a cálculo vetorial. Este objeto é armazenado na variável *pos*.

Além da posição, um ponto também possui seu tamanho, que é uma variável que representa o diâmetro do círculo a ser desenhado na tela pelo *framework*. Ele é utilizado no método *show*, onde é definido a cor do contorno, a cor de seu preenchimento, ambos no padrão

[7] https://www.youtube.com/channel/UCvjgXvBlbQiydffZU7m1_aw - data de acesso: 08/05/17
[8] https://p5js.org/libraries/ - data de acesso: 13/05/17
[9] https://github.com/processing/p5.js - data de acesso: 13/05/17

RGB, e por fim a função que de fato desenha o círculo (*ellipse*), que recebe como parâmetro a posição do eixo horizontal, vertical, largura e altura, respectivamente.

Por se tratar de um círculo, os valores de largura e altura são os mesmos, ou seja, o atributo *size* do objeto ponto. Como pode ser observado no código da classe abaixo:

```
class Point {
  constructor(posX, posY) {
    this.pos  = createVector(posX, posY)
    this.size = 3
  }

  show() {
    stroke(175)
    fill(255)
    ellipse(this.pos.x, this.pos.y, this.size, this.size)
  }
}
```

Código 9: Classe Ponto do código
Fonte: autor

4.3.2 Sketch

O arquivo *sketch.js* é por onde é iniciado o programa, ele é constituído por 4 funções e a definição de uma variável global, que é um *array* dos pontos. Apesar de não ser uma boa prática desenvolver código que utilize variáveis globais, neste caso não interfere nem prejudica a funcionalidade da heurística. Seu propósito é para facilitar a depuração, pois dessa forma é possível consultar o status da variável no console do navegador, podendo assim visualizar as propriedades de cada ponto. A declaração da variável é feita no topo do arquivo e possui a seguinte sintaxe:

```
let points = []
```

Código 10: Instância da variável global de Pontos
Fonte: autor

4.3.2.1 Setup

A função *setup* é padrão do *framework*, é a primeira função a ser chamada automaticamente quando a página *web* é carregada. A partir dela é possível dar início a outras funcionalidades, e neste caso foram atribuídas 5 etapas dentro de seu escopo, a serem executadas em ordem.

A primeira é para coletar a quantidade de pontos que o usuário deseja visualizar a rota, isto é feito através de um *prompt*, cujo é um *pop-up* exibido ao carregar a página, onde nele é perguntado quantos pontos devem ser criados. Após o usuário inserir o valor, este é armazenado na variável *numPoints*, e a partir daí o algoritmo pode seguir para as outras etapas.

Das 4 etapas restantes, 3 são funções contidas no mesmo arquivo (*sketch.js*) que a função *setup*, e tem como finalidade a criação, preparação e representação dos pontos e trajeto na tela, para que possam ser visualizados pelo usuário. Ou seja, são funções próprias para este ambiente ao qual foi implementado, podendo ser completamente diferente a depender de cada necessidade.

Apenas a quarta etapa está localizada em um arquivo próprio (*reciprocity.js*), que é justamente onde encontra-se o código para a implementação da heurística em si, onde este sim, deve ser implementado da forma como descrito, adaptando apenas aos requisitos de cada linguagem.

A função *setup* com suas 5 etapas, onde cada uma será detalhada, pode ser visualizada no código abaixo:

```
function setup() {
  let numPoints = prompt('How many points?') // Step 1
                  init(numPoints)            // Step 2
  let distances = setDist()                  // Step 3
  let route     = reciprocity(distances)     // Step 4
                  drawRoute(route)           // Step 5
}
```

Código 11: Função inicial do código
Fonte: autor

4.3.2.2 Inicializar os pontos

Essa função na verdade tem dois propósitos, mas que não compensaria deixa-los em funções separadas. A primeira cria o *canvas*, onde é renderizado todas o conteúdo visual pelo *framework*, seria como uma tela a ser pintada, onde na função *createCanvas* é definido

respectivamente a largura e altura do mesmo, e neste caso, é preenche a tela inteira. Em seguida é definida a cor de fundo desta tela, que neste caso é (51, 51, 51) no padrão RGB, ou simplesmente (51).

O segundo propósito é para a criação das cidades, onde é gerado um *loop* com a quantidade de repetições definida através do parâmetro dessa função, que no caso vem ser o número de cidades informado pelo usuário na primeira etapa. Para cada ciclo da repetição é criado uma nova instância da classe *Point*, e este objeto é então exibido e por fim adicionado no *array* global de pontos, com pode ser visto no código abaixo:

```
function init(n) {
  createCanvas(windowWidth, windowHeight)
  background(51)

  for (let i = 0; i < n; i++) {
    let newPoint = new Point(
      random(20, width - 20),
      random(20, height - 20))
    newPoint.show()
    points.push(newPoint)
  }
}
```

Código 12: Função para inicializar o cenário e criar os pontos
Fonte: autor

4.3.2.3 Calcular as distâncias

Nesta função é calculado as distâncias entre cada um dos pontos, e já que todos estão armazenados na variável global, ela não precisa receber nenhum parâmetro, apenas retorna um outro *array* contendo as distâncias. Sua estrutura visa economizar dados, afinal, a distância da cidade A para a B é a mesma da B para a A, não sendo necessário portanto, armazenar duas vezes.

Para que isto seja possível, o *array dist*, onde é armazenado as distâncias, possui um aspecto triangular, isto é, apenas a metade de uma tabela quadricular a ser cortada em sua diagonal principal (canto superior esquerdo até o canto inferior direito), bastando pegar qualquer uma das duas partes, já que seus valores são os mesmos.

Esta tabela quadricular possui em suas linhas e colunas, cada um dos pontos, e a posição onde elas se cruzam, como exemplo, o encontro da linha 4 (4º ponto) com a 5º coluna (5º ponto),

contém o valor da distância entre elas, que é o mesmo no encontro entre a 5ª linha com a 4ª coluna.

E para evitar a redundância, é utilizado apenas uma metade, onde neste caso é o da esquerda inferior, pois neste há um crescimento ascendente nas linhas, isto é, a primeira linha só possui um valor, a segunda dois, a terceira três, até que a última tenha as distâncias entre todos os outros pontos. Isso significa que sempre a cidade de maior número será a posição da linha, e a de menor a da coluna, exemplo, para saber a distância entre a cidade 3 e 7, esta informação está na linha 7, coluna 3.

Para que a função seja capaz de armazenar nesta estrutura são necessários dois *loops*, um dentro do outro, ou seja, para cada ciclo do primeiro *loop*, é completado todos os ciclos do segundo *loop*. Onde neste caso o primeiro *loop* percorre a quantidade de pontos que existem, e o segundo até a quantidade de ciclos já realizados pelo primeiro, ou seja, se o primeiro *loop* está em seu 5º ciclo, o segundo *loop* fará 5 repetições, onde para cada uma dessas será calculado a distância entre o 5º ponto e cada um dos 5 primeiros pontos.

O primeiro *loop* inicia pela segunda cidade, afinal, não há o interesse em calcular a distância entre o primeiro ponto com ele mesmo.

```
function setDist() {
  let dist = []
  for (var i = 1; i < points.length; i++) {
    let currentRow = []
    for (var j = 0; j < i; j++)
      currentRow.push(
        points[i].pos.dist(points[j].pos))
    dist.push(currentRow)
  }
  return dist
}
```

Código 13: Função para calcular as distâncias entre os pontos
Fonte: autor

4.3.2.4 Desenhar a rota

Já que nesta sessão é tratado apenas as funções presentes no arquivo *sketch.js*, a quarta etapa do código, que é onde encontra-se a heurística, só será abordada na próxima sessão, dando vez para a quinta e última etapa do processo, que é desenhar a rota calculada pelo algoritmo.

Essa função recebe como parâmetro uma rota, que nada mais é que um *array* contendo a ordem em que os pontos devam ser conectados, ou seja, o que a heurística retorna é justamente uma lista ordenada desses pontos.

A primeira coisa que a função de desenhar a rota faz é definir a cor da linha, onde nesse caso é roxa já que utiliza o vermelho e azul em sua totalidade (255, 0, 255) no padrão RGB (vermelho, verde e azul). A seguir é definido que não haverá preenchimento, afinal, é uma linha e não uma forma geométrica.

Após configurado essas questões visuais, é iniciado o processo de ligar os pontos. O *framework* possibilita desenhar qualquer forma geométrica apenas passando seus vértices, onde de um para o outro é desenhado a linha que os conecta, o que é exatamente o que o PCV precisa, porém nesse caso, cada vértice será um ponto e a ordem na qual eles vão sendo ligados está definida pelo parâmetro recebido.

Para que seja realizado tal desenho, é preciso inicializar o contorno com a função *beginShape* e finalizar com a função *endShape*, e entre estas duas, são postos os vértices. Como cada ponto será um vértice, e estes estão armazenados em ordem na lista passada como parâmetro, é feito um ciclo *for* percorrendo todo o *array*, e para cada iteração é definido um vértice para este ponto.

Para que seja definido um vértice é preciso indicar sua posição no eixo horizontal, e vertical, onde neste caso é obtido através da consulta a variável global de pontos, pois esta é que armazena os objetos pontos, enquanto que a variável recebida pelo parâmetro apenas aponta a posição daquele ponto na lista global dos pontos.

Ou seja, se o primeiro elemento da variável *route*, que é a recebida por parâmetro com a ordem dos pontos a serem conectados, for 3, isso significa que o primeiro ponto da rota será o objeto ponto que está localizado na quarta posição da lista de pontos (já que a lista começa pelo 0, o de número 3 será o quarto elemento). E o que deseja-se é a posição horizontal e vertical deste objeto, para isso é acessado sua variável *pos* (posição), cujo é um vetor contendo as propriedade *x* (horizontal) e *y* (vertical), como pode ser visualizado no código abaixo:

```
function drawRoute(route) {
  stroke(255, 0, 255)
  noFill()

  beginShape()
  for (let i = 0; i < route.length; i++)
    vertex(points[route[i]].pos.x, points[route[i]].pos.y)
  endShape()
}
```

Código 14: Função para desenhar a rota determinada pela heurística
Fonte: autor

4.3.3 O código da heurística

A quarta etapa do código é a heurística em si, responsável por receber certa informação e a partir desta devolver a ordem a qual os pontos devam ser conectados. Esta etapa não é constituída apenas por uma função, mas sim por 6, e já que todas são referentes a heurística em si, estão localizadas em um arquivo próprio (*reciprocity.js*).

Mas dentre estas 6, há a função inicial, cujo é chamada no arquivo *sketch.js*, que foi sinalizado como a etapa 4 (*step 4*). É através desta função que são chamadas as outras pertencentes ao arquivo da heurística, e ela mesma após passar por todas as etapas retorna a rota, que é o *array* contendo a ordem a qual os pontos devam ser ligados.

Mas ao invés de começar introduzindo esta função principal do arquivo, é válido primeiro abordar duas outras simples, pois serão utilizadas como auxílio ao decorrer da heurística. São elas a função que consulta as distâncias entre os pontos, e a função que verifica se um determinado elemento está contido em um *array*. Suas implementações serão abordadas nas duas próximas sessões, porém finalidades ficarão mais claras a medida que o restante da heurística for introduzido.

4.3.3.1 Consultar a distância

Sabendo que a heurística recebe como parâmetro as distâncias entre os pontos, e estas estão em uma estrutura triangular para que economize dados, uma vez que se fosse uma tabela teria distâncias repetidas. É preciso haver uma função que dado dois pontos, seja capaz de consultar nesta estrutura triangular, onde está a célula que indica a distância entre estas.

Já que foi utilizado a metade inferior esquerda da tabela, sendo cortada pela diagonal principal, conforme vai se aproximando do último ponto, maior é a quantidade de pontos

comparados a cada linha, isto é, um determinada ponto possui em sua linha a distância entre ele e todas os outros pontos que estão antes dele na lista global de pontos, então quanto mais próximo do final da lista, maior será a quantidade que este ponto terá de distâncias com os outros.

Isso significa que ao comparar dois pontos, o que está mais posterior sempre será a linha da tabela, enquanto que o anterior sempre será a coluna. Ou seja, ao comparar o ponto 5 com o 7, a distância entre eles estará na 8ª linha (posição 7) da tabela das distâncias, e na 6ª coluna (posição 5), já que ambas iniciam por 0.

Portanto, a função de consultar a distância recebe como parâmetro o primeiro ponto, aqui denominado *x*, o segundo, denominado *y* e a tabela das distâncias. É criado uma variável *d* cujo servirá para armazenar a distância entre estes pontos, e, se *x* for posterior a *y*, então *d* será igual a célula que se encontra na linha *x*, coluna *y* da tabela das distâncias. Caso contrário, isto é, *y* for posterior a *x*, então *d* será o valor encontrado na linha *y*, coluna *x* da tabela. E por fim, a função retorna a variável *d* que contém a distância entre os dois pontos, como observado no código abaixo:

```
function getDist(x, y, dist) {
  let d
  x > y ?
    d = dist[x][y] :
    d = dist[y][x]
  return d
}
```

Código 15: Função para consultar as distâncias entre os pontos
Fonte: autor

4.3.3.2 Verificar se pertence a uma lista

O objetivo desta função é servir como auxílio na hora de agrupar os pontos recíprocos, pois para cada novo ciclo a heurística descobre as conexões desejadas de cada grupo, verifica as que são recíprocas, e as une gerando um novo conjunto dos pontos. Esse processo é repetido até que só reste apenas um grupo de pontos, e portanto, para cada nova vez que um ciclo é completado, um novo conjunto de grupos é formado, constituindo pelos grupos que agora foram unidos e os que permaneceram intactos durante este ciclo, ou seja, os que não tiveram seu desejo de conexão correspondido.

Para que o novo conjunto de dados seja capaz de conter não somente as novas uniões, mas também estes que não foram unidos, é preciso distinguir esses os grupos para que durante a criação do novo conjunto para o próximo ciclo, inclua os grupos corretamente. E justamente esse é o objetivo desta função, e durante a sessão 0, onde é abordado a união dos grupos, seu uso torna-se mais claro, mas seu código pode ser conferido agora:

```
function contains(elem, list) {
  for (let i = 0; i < list.length; i++)
    if (elem == list[i])
        return true
  return false
}
```

Código 16: Função para verificar se determinado elemento está contido em uma lista
Fonte: autor

4.3.3.3 Função principal

Por mais que a heurística seja composta por 6 funções, é preciso haver a principal entre elas, cujo tem como responsabilidade ser a chamada inicial para que o algoritmo seja efetuado, e também, a responsável por retornar a ordem dos pontos a serem ligados. Ou seja, é através dela que a heurística é iniciada, da procedência as demais funções, e por fim retorna o resultado.

Como já descrito anteriormente, a única informação que a mesma precisa para realizar seus procedimentos, é a distância entre todos os vértices, e a partir disso, consegue devolver a ordem aos quais estes devem ser ligados. Portanto, o único parâmetro que a mesma recebe, é justamente uma variável que contém todas as distâncias, onde esta estrutura, como também já explicado nas sessões anteriores, segue o modelo da metade triangular de uma tabela, para economizar memória ao não ter distâncias duplicadas. Porém, é necessário realizar uma pequena alteração nesta estrutura das distâncias, como já explicado na sessão 4.2.2, e isto é feito logo na primeira linha da função.

Uma vez que a estrutura das distâncias possui em suas linhas todos os pontos, e corrigindo o problema de que o primeiro ponto poderá querer conectar-se consigo mesmo ao inserir o valor de infinito. A função pode dar seguimento a heurística, e a próxima tarefa que é realizada, é a criação do conjunto de grupos, onde aqui é um *array* que no primeiro momento é inicializado com a mesma quantidade de pontos. Isto é, já que ao iniciar a heurística, todos os pontos estão isolados, a quantidade de grupos existentes será justamente a quantidade de pontos, onde cada grupo possui cada ponto.

Em seguida é criado um ciclo *while*, onde efetuará todas as tarefas inseridas dentro dele enquanto não atingir o critério de parada, cujo aqui é quando a quantidade de grupos for igual a 1. Ou seja, enquanto a quantidade de grupos for maior que 1, significa que ainda há conexões a serem realizadas, e portanto, as tarefas dentro dele devem continuar a serem efetuadas.

Estas tarefas dentro do ciclo *while*, constituem de 3 etapas. Na primeira, é criado uma variável chamada *reciprocity*, cujo armazena os desejos de conexão de cada grupo, e para cada nova vez que um ciclo é iniciado, ela recomeça do zero, pois os desejos de conexão dos grupos de um estado não será o mesmo que os dos estados anteriores, já que conexões foram efetuadas no ciclo anterior e os grupos são então diferentes.

Na segunda etapa, é justamente preenchido a lista *reciprocity*, pois uma vez que ela é instanciada, é preciso preenche-la de acordo com os desejos de conexão dos grupos deste atual estado, e inclusive respeitando o índice de cada linha, ou seja, a primeira linha da variável *reciprocity* indica o desejo de conexão do grupo que também está na primeira linha do conjunto de grupos, e assim sucessivamente.

Para isso, é realizado um ciclo *for*, cujo itera por todos os grupos pertencentes aos conjuntos de grupos, onde para cada um deles, é chamado a função *findClosest*, cujo tem por objetivo retornar justamente o grupo ao qual este atual grupo deseja conectar-se, e assim, essa informação é inserida na variável *reciprocity* em seu respectivo índice. A estrutura de como os dados são armazenados é representada na Figura 24.

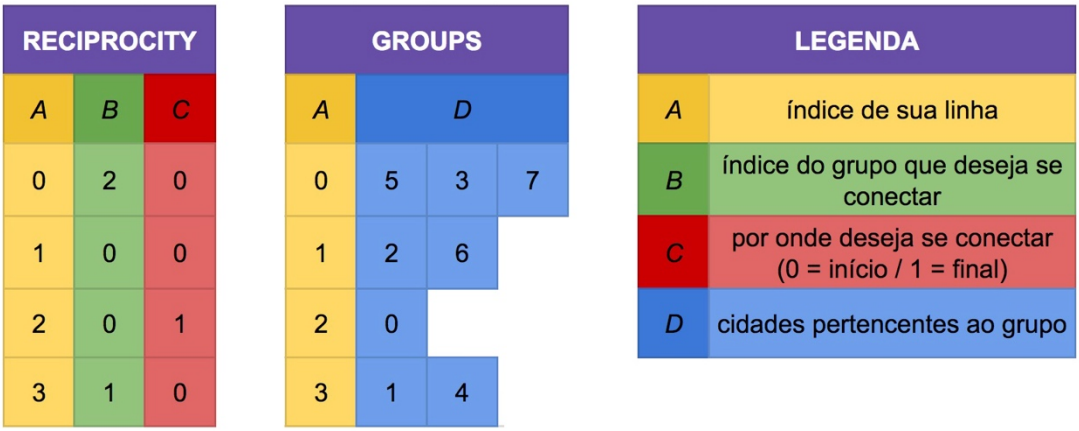

Figura 24: Estrutura de dados da heurística
Fonte: autor

Na ocasião representada pela Figura 24, há 4 grupos sendo referenciados por seus índices iniciando em 0 e indo até 3, tanto na variável *groups*, como na variável *reciprocity*. Os valores armazenados em cada linha de *groups* representa o índice do ponto, onde neste caso há 8 pontos, começando do 0 até 7. Em *reciprocity*, o primeiro valor armazenado representa o índice do grupo ao qual este deseja se conectar, enquanto que o segundo valor indica qual das extremidades receberá a conexão, sendo 0 para uma conexão pelo início e 1 pelo final.

Podemos notar que o primeiro grupo (com os pontos 5, 3 e 7), de índice 0 (representado em amarelo) deseja se conectar com o grupo do índice 2 (que possui apenas o ponto 0), como indicado em seu respectivo índice, mas na tabela *reciprocity*, em verde. E por seu desejo de conexão estar indicado como 0 (em vermelho), significa que o ponto que deseja se conectar com o 0 (do grupo de índice 2) é o ponto de número 5 (primeiro ponto do grupo de índice 0).

O grupo do índice 2 também deseja se conectar com o de índice 0, como pode ser visto em verde na tabela *reciprocity*, ou seja, a conexão entre eles é recíproca, então pode haver a união. O grupo de índice 2 deseja se conectar pelo seu final (pelo indicativo 1), mas como só há um elemento em seu grupo, o final e início representam o mesmo ponto, portanto, o resultado do novo grupo que surgirá após a união destes dois, terá os pontos 0, 5, 3 e 7.

Repare que o grupo de índice 1 (com os pontos 2 e 6) também deseja se conectar com o primeiro ponto, porém como o primeiro não corresponde ao seu desejo, a conexão entre eles não é estabelecida. Então nesse caso, houve apenas uma união, entre o grupo 0 e 2, resultando agora em 3 grupos na variável *groups*. Esse ciclo é repetido até que reste apenas um grupo contendo todos os pontos dentro da variável *groups*.

Uma vez que o ciclo *for* é terminado, ou seja, já foi descoberto o desejo de conexão de cada um dos grupos, e estas informações estão armazenadas em *reciprocity*, a próxima e última etapa do ciclo *while*, é atualizar o conjunto de grupos, unindo aqueles que possuem reciprocidade e mantendo os que não. Isto é feito através da função *joinMutual*, cujo é passado a variável *reciprocity* e o conjunto atual de grupo, e o que ela retorna o novo conjunto de grupos, onde desta vez estão unidos os que foram recíprocos e os restantes que não.

Este processo, descobrir o desejo de conexão de cada grupo e ir unindo os que são recíprocos, até que reste apenas um grupo, é a tarefa principal da heurística, e é realizada neste ciclo *while*. Ou seja, após atingir o critério de parada, no conjunto de grupos há apenas um inserido nele, cujo é sequência ao qual a heurística sugeriu conectar os pontos.

A partir deste estágio, a heurística então pode ser considerada como completa, e assim retornar seu resultado, porém, antes disso, esta solução é enviada para a função *improveConnections*, cujo seu objetivo é aperfeiçoar um pouco mais a rota, e este procedimento

é melhor descrito em sua respectiva seção (*4.3.6.6*). Vale observar que, o que é enviado para a função *improveConnections* é o primeiro grupo (posição 0) do conjunto de grupos (*grups[0]*), pois nele há somente um, que é justamente o que possui todos os pontos conectados, além da variável que contem todas as distâncias.

A função principal, cujo possui todos estes passos descritos, pode ser observada a partir deu código a seguir:

```
function reciprocity(dist) {
  dist.splice(0, 0, [Infinity])

  let groups = []
  for (let i = 0; i < dist.length; i++)
    groups.push([i])

  while (groups.length > 1) {
    let reciprocity = []
    for (let i = 0; i < groups.length; i++)
      reciprocity.push(findClosest(i, groups, dist))
    groups = joinMutual(reciprocity, groups)
  }

  return improveConnections(groups[0], dist)
}
```

Código 17: Função principal da heurística
Fonte: autor

4.3.3.4 Encontrar o mais próximo

Esta é a função responsável por localizar o ponto mais próximo relativo a cada grupo, o procedimento é bem simples quando há apenas pontos isolados, pois, basta verificar todas as distâncias entre estes, e encontrar aquelas que possuem o menor valor referente a cada um. Porém, quando é adotado o conceito de grupos, onde apenas é verificado, tanto para encontrar a melhor conexão a partir de um, quanto para comparar com os outros grupos, é preciso ter a preocupação de apenas verificar os que estejam nas extremidades dos grupos, pelas razões já descritas previamente.

Por mais que isto dificulte um pouco a legibilidade do código, a economia de não precisar considerar os outros pontos que já possuam duas conexões efetuadas, é de suma importância para garantir um bom desempenho computacional do algoritmo. Para isso, a função

recebe como parâmetro, o índice do grupo ao qual deseja-se encontrar sua melhor conexão, o conjunto contendo todos os grupos, e as distâncias entre todos os pontos.

A primeira tarefa a ser realizada é a criação da variável *shorter*, que tem por finalidade armazenar a menor distância encontrada para este respectivo grupo, e seu valor é inicializado como infinito, pois se fosse como 0, nenhuma distância seria menor que esta, e logo, não encontraria o melhor grupo.

A seguir, é criada a variável *comparisons*, e é inicializada contendo os dois pontos das extremidades do atual grupo, o que está na primeira e na última posição, ou seja, caso este grupo tenha apenas um ponto, este estará duplicado nesta variável, mas que não é um problema para a execução do algoritmo.

Depois é criada a variável *desiredIndex*, que armazenará o índice do grupo ao qual este gostaria de se conectar, bem como por onde ele gostaria de se conectar, sendo 0 caso ele deseja conectar-se através de seu ponto que está na extremidade inicial, e 1 caso seja pelo seu fim.

Agora que foi configurado as variáveis, o algoritmo segue para finalmente procurar a melhor conexão respectiva ao grupo. Para isso é realizado um ciclo *for* iterando por todos os grupos existentes, e para cada um deles, que não seja o mesmo que este atual, é inserido as suas duas extremidades, início e fim, na variável *comparisons*, ou seja, esta agora possui as duas extremidades do grupo atual, e as duas extremidades do grupo que está na iteração do ciclo *for*.

O que é feito a seguir, é uma comparação das distâncias entre cada um desses 4 pontos, sendo a primeira extremidade do grupo atual com as outras duas do outro grupo, e a segunda extremidade desse grupo, também com essas outras duas do outro grupo. Onde para cada verificação, e comparada se esta é menor que a variável *shorter* que no começo foi inicializada com infinito, mas que caso encontre alguma distância menor que essa, seu valor é atualizado.

Quando o ciclo *for* é terminado, isto é, as duas extremidades do atual grupo foram comparadas com as duas extremidades de todos os outros grupos, a variável *desiredIndex* está agora atualizada contendo o índice do grupo ao qual este deseja conecta-se, baseado na menor distância obtida através da variável *shorter*, bem como por qual extremidade este grupo gostaria de se conectar com este desejado.

Assim, a função retorna a variável *desiredIndex*, e esta é então armazenada na lista de *reciprocity* da função principal, cujo o índice ao qual o *desiredIndex* deste grupo é armazenado, é o mesmo índice que este grupo está na lista de todos os grupos, ou seja, o índice da linha onde o grupo está localizado na lista de *groups*, é o mesmo índice da linha que contém seu desejo de conexão em *reciprocity*.

O código que encontra a melhor conexão de um determinado grupo, pode ser verificado abaixo:

```
function findClosest(i, groups, dist) {
  let shorter = Infinity
  let comparisons = [
    groups[i][0],
    groups[i][groups[i].length - 1]
  ]
  let desiredIndex = [
    // index of the desired point
    // from where to connect (0 = beginning / 1 = end)
  ]

  for (let j = 0; j < groups.length; j++) {
    if (i != j) {
      comparisons[2] = groups[j][0]
      comparisons[3] = groups[j][groups[j].length - 1]

      for (let k = 0; k < 2; k++) {
        for (let w = 2; w < 4; w++) {
          let currentDist =
            getDist(comparisons[k], comparisons[w], dist)
          if (currentDist < shorter) {
            desiredIndex = [j, k]
            shorter = currentDist
  } } } } }

  return desiredIndex
}
```

Código 18: Função para determinar a preferência de conexão de um grupo
Fonte: autor

4.3.3.5 Unir os recíprocos

Esta função é chamada a cada nova iteração do ciclo *while* localizado na função principal, sua responsabilidade está em unir os grupos que são recíprocos, e devolver o novo conjunto de grupos contendo os unidos e também os que não foram. Ou seja, a cada nova vez que é chamada, retorna uma quantidade menor de grupos, até que por fim, haverá apenas um, que é justamente o que contém todos os pontos a serem conectados e em ordem.

Para que a função una os grupos recíprocos, é preciso possuir uma referência que contém o desejo de conexão de cada grupo, e esta informação está contida na variável *reciprocity* da função principal, onde aqui nesta função, é chamada simplesmente de *array*, cujo é o primeiro parâmetro da função. O segundo é o próprio conjunto de grupos, para que assim, seja possível manipula-lo a fim de conectar os grupos recíprocos.

Outra preocupação que a função precisa possuir, é quanto a maneira que será realizada a união entre dois grupos, como já explicado na Figura 21. A informação de por onde um grupo gostaria de se conectar, também esta contida na lista *reciprocity* que aqui é chamada de *array*.

A função começa criando a variável *newGroup*, que é quem armazenará o novo conjunto de grupos após a união dos recíprocos. Em seguida, é criada a variável *connectedList*, onde armazena os índices dos grupos que já foram conectados, para que evite uma união redundante, pois se após a conexão do primeiro grupo com o terceiro, não é desejável que ocorra novamente uma união, só que agora do terceiro com o primeiro, pois estes já foram conectados.

Após a criação destas duas variáveis, o algoritmo inicia um ciclo *for*, iterando por todos os desejos de conexão, que estão armazenados na variável *array*. Para cada iteração, é criada a variável *want*, que armazena o índice do grupo ao qual o atual grupo da iteração deseja conectar-se. Além disso é criado também a variável *newArray* que é onde guardará o resultado a união de dois grupos.

Em seguida é verificado se o grupo ao qual o atual deseja conecta-se, também gostaria de conectar-se com ele, além disso, se este não pertence ao grupo de já conectados, para que evite a união redundante. Se, ambos os grupos são recíprocos e nenhum deles ainda já foi conectado, eles então podem dar seguimento para a união.

Como ilustrado na Figura 21, há 4 formas que dois grupos podem se unir, e para cada um dos casos, há um código específico para realizar esta união, sem que comprometa a ordem de conexão dos pontos. Este procedimento é feito em um *switch*, e dependendo de qual dos 4 casos a união entre os dois grupos será realizada, a variável *newArray* armazenará o resultado desta respectiva conexão.

Após o *switch* e consequentemente a união entre os dois grupos, os índices de cada um destes, são inseridos na lista de *connectedList*, pois agora estes já foram unidos e, portanto, não devem ser novamente, o que poderia ocorrer em casos como, se o o grupo que está na posição 1 for recíproco como o que está na posição 3, quando chegar na vez de verificar se o 3 possui algum recíproco, que no caso é 1, haveria novamente uma união entre estes, o que seria redundante, pois a união já ocorreu quando foi verificado que o 1 é recíproco.

Pode haver o caso de o grupo da vez não ser correspondido com o grupo ao qual deseja conecta-se, e, portanto, a variável *newArray* ficará vazia. Por isso, ao término de cada iteração do ciclo *for*, é verificado se *newArray* não está vazio, onde somente nesta ocasião, ele é inserido na variável *newGroup*, pois possui o resultado da união de dois grupos.

Quando todas as iterações do ciclo *for* terminarem, ou seja, todos os grupos recíprocos já foram unidos e inseridos na variável de *newGroup*, resta agora inserir os grupos que não tiveram sua união realizada. Para tal, é feito novamente uma iteração entre os grupos, e aqueles que não estejam localizados na lista de *connectedList* são inseridos em *newGroup*.

Após isso a variável *newGroup* possui agora os novos grupos unidos e os que não foram, podendo agora retornar esta variável para a função principal, e assim atualizando a variável de grupos, para que dê seguimento as novas iterações do ciclo *while* até que reste apenas um grupo contendo todos os pontos. O código que une os grupos recíprocos e retorna o novo de conjuntos de grupos, pode ser observado a seguir:

```
function joinMutual(array, groups) {
  let newGroup = []
  let connectedList = []

  for (let i = 0; i < array.length; i++) {
    let want = array[i][0]
    let newArray = []

    if (array[want][0] == I &&
        !contains(want, connectedList)
    {
      switch (true) {
        case (array[i][1] == 0 && array[want][1] == 0):
          groups[i].reverse()
          newArray = groups[i].concat(groups[want])
          break;

        case (array[i][1] == 0 && array[want][1] == 1):
          newArray = groups[want].concat(groups[i])
          break;

        case (array[i][1] == 1 && array[want][1] == 0):
          newArray = groups[i].concat(groups[want])
          break;

        case (array[i][1] == 1 && array[want][1] == 1):
          groups[i].reverse()
          newArray = groups[want].concat(groups[i])
          break;
      }
      connectedList.push(i, want)
    }

    if (newArray.length > 0)
      newGroup.push(newArray)
  }

  for (let i = 0; i < groups.length; i++)
    if (!contains(i, connectedList))
      newGroup.push(groups[i])

  return newGroup
}
```

Código 19: Função para unir os grupos recíprocos
Fonte: autor

4.3.3.6 Refinar a rota

Uma vez que todos os pontos já pertencem a um grupo, ou seja, a ordem de suas conexões já foi definida pela heurística, o algoritmo pode dar-se por encerrado. Porém durante seu processo ocorre um problema de cruzamento entre as conexões, o que acaba não deixando a rota tão eficiente assim.

Felizmente esse problema pode ser facilmente resolvido, pois uma vez identificado duas retas que se cruzam, basta trocar a ligação entre seus pontos e o trajeto está aperfeiçoado. Para que a troca das conexões seja feita de forma correta, é preciso identificar o sentido ao qual o trajeto está caminhando, pois dessa forma evita que após realizar a troca, a rota deixa de estar conectada.

Dentre 4 pontos, só existe uma forma de conexão possível de modo que o sentido do trajeto não seja interrompido e que as duas retas não se cruzem. Quaisquer outras combinações violam algum desses dois fatores, porém, pode haver casos onde mesmo tendo o cruzamento entre as duas retas, este de fato seja o caminho mais eficiente, portanto, é preciso verificar se a opção que não cruza é melhor que a atual.

Para obter a configuração onde as duas retas não se cruzam e ainda mantendo o trajeto intacto, basta seguir um fluxo, independente de qual o sentido, ao chegar no primeiro ponto que pertence aos 4 cujo suas conexões se cruzam, o segundo após ele deve ser substituído pelo terceiro, independente de haver outros pontos entre o segundo o terceiro. E consequentemente, o segundo deve ser conectado com o quarto, como pode ser observado na Figura 25.

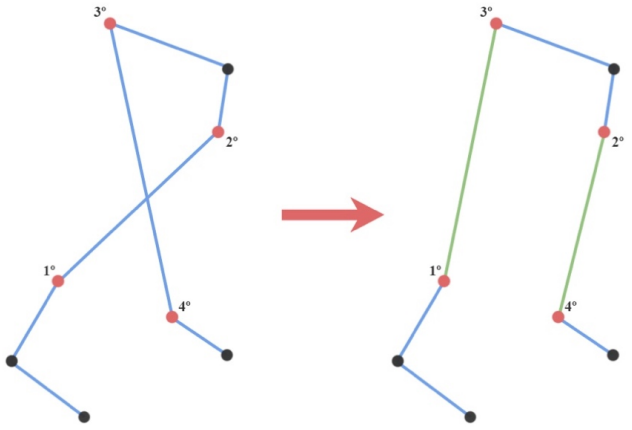

Figura 25: Troca entre as retas para evitar o cruzamento
Fonte: autor

Se após realizar este procedimento, o comprimento do trajeto total aumentar, é porque a configuração anterior, em que havia o cruzamento, é a melhor, e caso contrário, esta deve ser a nova ordem do trajeto.

Realizar o procedimento de troca é simples, o problema está em identificar onde está ocorrendo um cruzamento. Para um humano, é simples observar que duas retas estão se cruzando, mas para um computador, é preciso adotar outra forma. Uma maneira seria encontrar a equação da reta de cada uma das duas, e verificar onde elas se tocam, porém, o problema é que com isso não obtemos a informação se as duas retas traçadas estão se cruzando, e sim em que ponto elas se tocariam, independente do comprimento de cada uma.

A melhor forma, portanto, de verificar se duas retas se cruzam é observar se a maior distância entre os 4 pontos, é uma dessas duas retas, e claro, sabendo que cada uma dessas duas retas existentes, conectam dois pontos distintos. Mas há um problema com essa técnica, pois para verificar todos as 4 possíveis combinações entre pontos, e saber quais destes se encaixam neste critério, é algo que exige bastante processamento computacional, especialmente quando há um grande número de pontos.

Uma forma de amenizar o custo, seria ir verificando a cada vez que uma nova conexão é efetuada, e comparar se esta cruza com alguma das conexões já efetuadas até então. Porém, neste trabalho não foi implementado tal recurso, pois iria comprometer o desempenho da heurística de forma significativa.

O que foi realizado, entretanto, foi uma comparação entre os 4 pontos que estejam em sequência no trajeto, isto é, entre o segundo e terceiro ponto, não há outros intermediários. Isso foca apenas no refinamento entre pontos que estejam próximos um do outro na ordem, ou seja, a maior troca entre um ponto e outro seria em uma distância de apenas um ponto entre eles, como ilustrado na Figura 26, e que não prejudica o rendimento computacional da heurística.

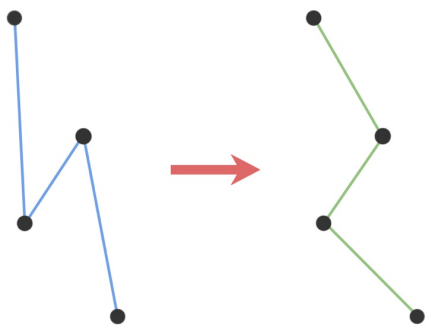

Figura 26: Refinamento entre 4 pontos subsequentes
Fonte: autor

Para que isso seja possível é realizado um ciclo *for* iniciando no segundo ponto do trajeto e indo até antepenúltimo, e para cada iteração, é pego o ponto anterior ao atual, e os dois posteriores a ele. É feita uma comparação, se após realizar a troca, o trajeto irá diminuir ou não seu comprimento, caso negativo, avança para o próximo ponto e realiza o procedimento de novo, mas caso positivo, e realizado a troca, conectando o primeiro com o terceiro e o segundo com o quarto.

A função recebe como parâmetro o trajeto já completo de todos os pontos, e a variável que possui todas as distâncias entre os pontos, ao término de toda a iteração do ciclo *for*, e efetuado os procedimentos, a função devolve o trajeto corrigido, como pode ser observado no código abaixo:

```
function improveConnections(path, dist) {
  for (let i = 1; i < path.length - 2; i++) {
    let pointAback  = path[i - 1]
    let pointA      = path[i]
    let pointB      = path[i + 1]
    let pointBfront = path[i + 2]

    if ((getDist(pointAback, pointA,      dist) +
         getDist(pointA,     pointB,      dist) +
         getDist(pointB,     pointBfront, dist))
         >
        (getDist(pointAback, pointB,      dist) +
         getDist(pointB,     pointA,      dist) +
         getDist(pointA,     pointBfront, dist)))
    [path[i], path[i + 1]] = [path[i + 1], path[i]]
  }
  return path
}
```

Código 20: Função para melhorar as conexões entre pontos adjacentes
Fonte: autor

4.4 RESULTADOS

Com o objetivo de verificar a eficiência da heurística, foram realizados testes em diferentes dispositivos, cujo é possível graças ao código ter sido implementado em *JavaScript* e assim poder ser executado em qualquer dispositivo que possua um navegador à internet. Para tal, o código foi hospedado em um servidor e qualquer equipamento que acessar o endereço do *site*, poderá executar o algoritmo.

Vale ressaltar que apesar de os testes terem sido realizados através do *site* em que o código encontra-se hospedado, não há interferência no tempo de execução, pois uma vez que a página é carregada, todos os arquivos são enviados pelo servidor e executados no cliente.

Por outro lado, o que há influência no tempo é a implementação do cenário, cujo inclui toda a parte visual do algoritmo, ou seja, a criação de todos os pontos, o cálculo da distância entre eles e também o desenho que representa a trajeto final, tudo isso a cargo do *framework p5.js*. Ou seja, o tempo mostrado nestes testes, não representa somente o tempo que a heurística leva para solucionar o problema, mas também está incluso o tempo gasto para a representação visual dos elementos.

O método utilizado para calcular o tempo foi obtido através de um aplicativo de cronômetro fornecido pelo celular, e o intervalo medido foi do momento em que o usuário, após inserir a quantidade de pontos desejados, confirma o valor, seja apertando a tecla *enter* ou clicando em *ok* no *pop-up* exibido, até o momento em que o navegador exibe a rota finalizada.

Por ter sido desenvolvido com uma linguagem de programação *web*, que é o *JavaScript*, a gama de dispositivos testados pode ser diversificada, já que apenas depende de um navegador para interpreta-la. O que facilita enxergar o desempenho da heurística em diferentes plataformas.

Para apresentação dos resultados, será primeiro indicado os dispositivos aos quais foram utilizados para o teste, conforme demonstrado na Tabela 2, e em seguida o tempo gasto por cada um deles para resolver uma quantidade pré-determinada de pontos.

Tabela 2: Dispositivos utilizados para realização dos testes
Fonte: autor

Dispositivo	Processador	Memória RAM	Sistema Operacional	Navegador
Computador D	i5 2500 3,3GHz	4GB	Windows 7	Firefox
Computador F	i7 6700 3,4GHz	8GB	Windows 10	Chrome
Computador M	i5 4430 3GHz	8GB	Windows 10	Chrome
MacBook	i5 2,5GHz	8GB	macOS High Sierra	Chrome
Netbook	AMD C-50 1GHz	2GB	Windows 7 32bits	Chrome
Notebook	i5 2410M 2,3GHz	4GB	Windows 10	Edge
Celular 1	Cortex-A72 1,8GHz	3GB	Android 7.0	Chrome
Celular 2	Cortex-A7 1,2GHz	512MB	Android 4.4.2	Chrome

As informações mais relevantes para a resolução do algoritmo são o modelo e velocidade do processador, pois é ele que executará as tarefas do algoritmo, a quantidade de memória RAM disponível, pois é nele onde o código armazena seus dados, o sistema operacional, que possibilita a comunicação entre o *hardware* e o *software* e por fim o navegor, que é responsável por interpretar o código *JavaScript*.

O objetivo era diversificar as configurações dos equipamentos ao máximo possível, porém ainda assim, não foi como o desejado. Foram utilizados 3 computadores, 3 notebooks e 2 celulares, tentando seguir um padrão de equipamento fraco, mediano e potente. E os resultados podem ser conferidos na Figura 27.

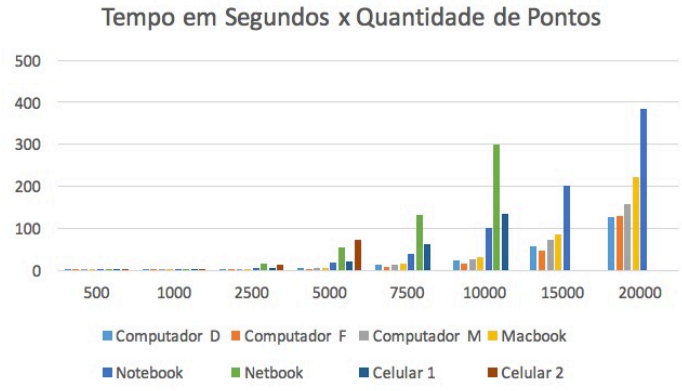

Figura 27: Tempo gasto e quantidade de pontos resolvidos de cada dispositivo
Fonte: autor

Cabe ressaltar que os celulares 1 e 2, e o Netbook, não foram capazes de completar os testes até 20000 pontos. O celular 1 e o Netbook fizeram até 10000, enquanto que o celular 2 completou até 5000, e, portanto, os valores posteriores a este estão omitidos o gráfico.

Como é possível ver na Figura 27, a comparação de desempenho entre os dispositivos, apesar dos diferentes potenciais de cada equipamento, não há grandes distinções até 2500 pontos, o que já é uma quantidade extraordinária para um problema de crescimento fatorial como o PCV.

A partir de 5000, podemos notar mais facilmente as distinções entre eles, mas que ainda assim, não apresentam o resultado em tempo inviável. O teste segue até 20000 pontos, um número muito acima do que a temática do problema condiz, onde cada ponto seria uma cidade a ser visitada. Mas esse valor serve para ver como a heurística se comporta mesmo em condições complexas.

Nota-se também que, dentre os dispositivos utilizados nos testes, é possível separa-los em dois grupos quanto ao seu desempenho, onde o primeiro, que são os que tiveram melhores resultados, estão melhor detalhados na Tabela 3. Enquanto o segundo grupo, que contém os outros 4 dispositivos restantes, onde 3 deles não conseguiram chegar ao limite de 20000 pontos dos testes, pode ser visto na Tabela 4.

Tabela 3: Resultados dos melhores equipamentos
Fonte: autor

	Computador D	Computador F	Computador M	Macbook
500	0,24	0,1	0,23	0,4
1000	0,27	0,2	0,52	0,9
2500	1,6	1,27	1,5	1,7
5000	6	3,8	5,7	6,37
7500	12,7	9,13	13	15,39
10000	24,65	16,77	25	31,82
15000	58,4	45,72	72	85,86
20000	127	130,2	158	221,1

Tabela 4: Resultados dos piores equipamentos
Fonte: autor

	Notebook	Netbook	Celular 1	Celular 2
500	0,88	1,21	0,9	1,82
1000	1,28	2,5	1,4	3,49
2500	6,22	16,1	5,2	14,55
5000	17,55	54,9	22,35	72,68
7500	39,64	132	63,2	
10000	99,8	300	135,7	
15000	201,84			
20000	383,74			

Devido os resultados serem apresentados em segundos, mesmo para consideráveis quantidades de pontos, e ter sua implementação em *JavaScript*, a heurística permite sua utilização para aplicações onde possam ser executadas no lado do cliente, como é o caso justamente deste teste.

Isso significa que no caso de uma aplicação, onde é preciso calcular a trajetória, o resultado pode ser obtido e calculado no próprio dispositivo que está executando a aplicação, não necessitando por exemplo, fazer uma requisição a um servidor, para que este sim solucione e retorne a rota calculada.

Ou seja, por a heurística apresentar resultados eficientes, mesmo em dispositivos com baixo desempenho computacional, aplicações que a utilizem podem ser executadas de forma independente. Não dependendo da centralização de todas as requisões à um servidor com alto poder computacional.

5 CONCLUSÃO

O PCV continua sem uma solução exata obtida em tempo polinomial, e, portanto, o uso de heurísticas faz-se necessário. Afinal, obter uma resposta razoável, mas dentro do tempo necessário, é preferível, na maioria dos casos, que obter a resposta com exatidão em um tempo intangível. O foco então passa a conciliar a qualidade da resposta com o custo computacional que a mesma requere, o que justifica o contínuo estudo nas heurísticas até que talvez seja encontrada uma solução determinística.

Neste trabalho foi explorado uma alternativa de resolução, cujo apresentou resultados bem satisfatórios, especialmente quanto ao requisito de tempo e custo computacional. A solução implementada, utilizando a linguagem de programação *JavaScript*, possibilitou verificar que mesmo executando em dispositivos de baixo desempenho, apresentou a mesma qualidade de solução que dispositivos com alto, divergindo apenas em alguns segundos, a depender da quantidade dos valores de entrada utilizado.

Demonstrou que caso seja implementada para ser utilizada em ambientes de produção, onde normalmente os valores de entrada não são tão altos como os empregados nos testes, os resultados obtidos contemplam as necessidades, como por exemplo, para 5000 cidades, os melhores dispositivos usados nos testes, concluíram em cerca de 5 segundos.

O algoritmo preocupa-se também em ser genérico o suficiente para que possa ser utilizado em diversos cenários, já que o único dado que necessita para executar seu método, são as distâncias entre cada vértice numa mesma unidade de medida. O que possibilita sua utilização para diferentes cenários, seja para entrega de mercadorias ou movimento de um braço robótico. Além disso, é possível que seja implementado em outras linguagens de programação, e por ser um código enxuto, facilita a adoção nas mais variadas necessidades de aplicação.

Há ainda questões que podem continuar sendo exploradas, e que resultar em trabalho s futuros, como por exemplo a resolução quanto ao cruzamento de rotas. Também há o desejo de explorar em outros cenários, como em um espaço tridimensional, não restringindo seu uso à grafos planares. Ou em rotas assimétricas, em que o deslocamento de ida entre duas cidades não é o mesmo que o de volta.

O método de embrulho, descrito na sessão 4.1.1.1, e o de melhores conexões, da sessão 4.1.1.2, também são uma possibilidades de trabalhos futuros, já que o algoritmo implementado não descreve o devido funcionamento que a estratégia propõe. Inclusive, durante a pesquisa, ideias de novos métodos de resolução surgiram, afinal, quanto mais conhecimento é adquirido, melhor torna-se a capacidade de solucionar problemas.

6 REFERÊNCIAS

ANDRETTA, M. Algoritmos de aproximação – Problema do Caixeiro Viajante. ICMC-USP, São Paulo, 2015.

BARBIERI, C. C. e BONASSER, U. O. *Experimentos Computacionais com Heurísticas de Melhorias para o Problema do Caixeiro Viajante*. XVI Congresso da Anpet – Associação Nacional de Pesquisa e Ensino em Transportes. Natal, 2002.

CHAVES, A. A. et al. *Metaheurísticas híbridas para resolução do problema do caixeiro viajante com coleta de prêmios.* Produção, v. 17, n. 2, p. 263-272, 2007

DIJKSTRA, E. *A note on two problems in connection with graph*, 1959.

FORISEK, M. *How would you explain an algorithm that generates permutations using lexicographic ordering?* Disponível em < http://bit.ly/2toGEas> Acessado em: 02/06/2017

GIGERENZER, G. e GAISSMAIER, W. *Heuristic Decision Making*. Annual Review of Psychology, 2011

GOLDBARG, M. e GOLDBARG, E. *Grafos – Conceitos, algoritmos e aplicações*. 1.ed. Rio de Janeiro: Elsevier, 2012, 640p.

HAYKIN, S. S. Neural Networks: A Comprehensive Foundation, Editora Prentice-Hall, 2. ed, 1999.

HIDDEN FIGURES. Direção: Theodore Melfi. Atlanta, Georgia (EUA). 20th Century Fox, 2016

HOLLAND, J. H. Adaptation in Natural and Artificial Systems, 1992, 211p.

KRUSKAL, J. B. *On the shortest spanning subtree of a graph and the traveling salesman problem*. Proceedings of the American Mathematical Society, 1956.

MACGREGOR, J. N. e ORMEROD, T. Human Performance on the Traveling Salesman Problem, Perception & Psychophysics, Loughborough ,1996, 58 (4), 527-539, 1996

MATHEMATICAL PROGRAMMING GLOSSARY. *The Nature of Mathematical Programming Archived*. INFORMS Computing Society.

RESENDE, C. B. e SCARPEL, R. A. *Aplicação de um Método de Geração de Agrupamentos, Baseado no Problema do Caixeiro Viajante, para Segmentação de Mercado*. SOBRAPO, PODes – Revista Eletrônica Pesquisa Operacional para o Desenvolvimento. Rio de Janeiro, v.3, n.3, p.186-200, 2011.

SHAPLEY, L. S. e ROTH, A. E. Stable Allocations and the Practice of Market Design, Economic Sciences Prize Committee of the Royal Swedish Academy of Sciences 2012

TALBI, E.-G. Metaheuristics: From Design to Implementation, volume 74. John Wiley & Sons.