# Relatório do Trabalho de Inteligência Artificial

Guilherme Dantas

Abril 2020

# Conteúdo

# Capítulo 1

# Relatório

## 1.1 Introdução

O trabalho foi implementado em C++17, e o código foi estruturado em função das seguintes categorias.

- bibliotecas (.cpp em **src**, .h em **include**)
- bibliotecas gráficas (.cpp em **vis/src**, .h em **vis/include**)
- aplicações (.cpp e .h em **app**)
- aplicações gráficas (.cpp e .h em **vis/app**)

O sistema de build escolhido foi o **CMake** pela sua portabilidade e facilidade de uso. Foi desenvolvido usando o Visual Studio 2019 Community, no ambiente Windows. Quanto à portabilidade, é bastante seguro dizer que as bibliotecas "normais"devem ser independentes de plataforma, enquanto as gráficas podem apresentar alguma incompatibilidade, pois usam da biblioteca **freeglut/glut**. Contudo, caso o CMake não ache esta ou o **OpenGL**, as bibliotecas e aplicações não serão construídas. Clique **aqui** para baixar o CMake.

Foram implementadas as heurísticas de **ILS** (*Iterated Local Search*), **VNS** (*Variable Neighbourhood Search*) e **GA** (*Genetic Algorithm*). O ILS e GA são completamente independentes, mas ambos usam do **VNS** para encontrarem o mínimo local de uma solução e para perturbar uma solução (que seria a "mutação", no contexto do algoritmo genético).

## 1.2 Definição do Problema

O problema é minimizar a função custo dada pela equação A.4 (ver Apêndice A para definição formal do problema).

## 1.3 Metodologia

### 1.3.1 Algoritmos

Aqui estarão ilustrados os principais algoritmos que compõem a inteligência do "resolvedor", em especial, as heurísticas utilizadas para "resolver"soluções.

#### Variable Neighbourhood Search (Local Search)

O algoritmo segue o seguinte pseudo-código.

```
t <- 0
repeat:
    improved_once <- false
    for p in Clients(S):
        for q in GammaSet(p):
            feasible, delta <- Test(S, N(t), p, q)
            if not feasible or
               (must_improve and delta >= 0):
                continue
            S <- Apply(S, N(t), p, q)
            improved_once <- true
            t <- 0
    if not improved_once:
        t <- t + 1
while t < |N|
```

O pseudo-código comentado com explicações sobre como cada linha foi interpretada na implementação segue.

```
for p in Clients(S):
```

A variável "p"é apenas o índice de um cliente da solução "S". A ordem é randomizada para não ter nenhum *bias*.

```
for q in GammaSet(p):
```

A variável "q"também é apenas o índice de um dos k-vizinhos de "p". O "GammaSet"é o conjunto dos k vizinhos mais próximos de um dado nó. É uma estrutura de dados criada juntamente com a instância, então esta consulta tem complexidade temporal constante. A ordem dos vizinhos também é randomizada.

```
feasible, delta <- Test(S, N(t), p, q)
```

É testado se uma dada vizinhança (denotada por "N(t)") é válida (valor booleano armazenado em "feasible") e é avaliada a diferença no custo da solução se o movimento fosse efetuado (armazenado em "delta"). Os cálculos por trás deste valor delta são profundamente explicados no apêndice A.

```
S <- Apply(S, N(t), p, q)
```

É aplicado o movimento à solução. Na prática, é passado um parâmetro "improve"indicando que, somente se o movimento for válido e o delta for negativo, o movimento pode ser aplicado. Caso falso, o movimento, se válido, é sempre aplicado forma.

```
while t < |N|
```

Encare "N"como um conjunto de vizinhanças ordenados por prioridade. Na prática, N(0) seria shift, N(1) seria 2-opt, etc... Como estão especificados apenas 4 vizinhanças, —N— seria 4.

### Variable Neighbourhood Search (Perturbation)

```
t <- 0
s <- perturbation_size
repeat:
    for p in Clients(S):
        for q in GammaSet(p):
            feasible, size <- Test(S, N(t), p, q)
            if (not feasible) or
               (size > s):
                continue
            s <- s - size
            S <- Apply(S, N(t), p, q)
            t <- (t + 1) % N
            if s == 0:
                goto end
:end
while s > 0
```

O algoritmo tem suas similaridades com o Local Search, portanto serão comentadas aqui apenas as linhas que diferem desse.

```
s <- perturbation_size
```

O parâmetro **perturbation_size** é um número inteiro entre 1 e **n**, o tamanho da instância. Equivale ao número de nós que são "perturbados", ou seja, movidos de sua posição original.

```
feasible, size <- Test(S, N(t), p, q)
```

Aqui, a função **Test** retorna mais um valor, **size**, que indica o tamanho da perturbação que o movimento de vizinhança "N(t)"causaria entre os nós p e q. O valor **delta** não é mais usado pois não estamos interessados em minimizar a função custo.

```
if (not feasible) or
   (size > s):
    continue
```

Aqui não é testado se o delta é negativo pois também não estamos interessados em melhorar a solução, só perturbá-la. Além disso, caso o movimento cause uma perturbação maior do que "s", ele não será aplicado.

```
s <- s - size
```

Subtraímos "size"de "s". Significa que aplicamos um movimento de tamanho "size".

```
t <- (t + 1) % N
```

Após todo movimento, passamos para a próxima vizinhança, para promover uma riqueza na perturbação.

```
if s == 0:
   goto end
```

Aqui escapamos dos loops quando "s"se torna 0. É um efeito cosmético, pois de qualquer forma nenhum movimento seria aplicado.

### Iterated Local Search

O algoritmo segue o seguinte pseudo-código, inspirado do paper *Mestria M.; Ochi L. S.; Martins L. S.; "Iterated Local Search para o problema do Caixeiro Viajante com Grupamentos"*.

```
S0 <- InitialSolution;
S <- LocalSearch(S0);
repeat:
  S' <- Perturbation(S, history);
  S'' <- LocalSearch(S');
  S <- AcceptanceCriterion(S, S'', history);
until stopping criterion is not satisfied anymore
return S;
```

Vai ser aqui então comentada cada linha do pseudo-código, explicando como de fato foi implementada.

```
S0 <- InitialSolution;
```

Aqui há simplesmente uma cópia da solução inicial, salvando em S0. A solução inicial é construída por uma heurística construtiva gulosa.

```
S <- LocalSearch(S0);
S'' <- LocalSearch(S');
```

É feita uma busca local, procurando o mínimo local através de um *Variable Neighbourhood Search*, em que só são aceitos movimentos vantajosos, salvando em S e S". (Na prática, a função opera a própria solução passada como parâmetro)

```
S' <- Perturbation(S, history);
```

Aqui é aplicada uma perturbação na solução cuja magnitude segue uma exponencial decrescente. De certa forma, esta heurística pode ser classificada como um *Simulated Annealing*. O tamanho da perturbação é a quantidade de movimentos que serão aplicados à solução. É dado por $floor(p \cdot n)$, aonde $p$ é a magnitude da perturbação e $n$ o tamanho da instância. A magnitude é dada pela equação 1.1.

$$p = p_0 \cdot e^{-i/I} \tag{1.1}$$

Aonde $p_0$ é a magnitude de perturbação inicial (parâmetro), $i$ é o número da iteração atual e $I$ é um fator de decaimento (parâmetro). Os movimentos também alteram de vizinhança a cada movimento bem-sucedido (também um *Variable Neighbourhood Search*, mas aceitando qualquer movimento válido).

```
S <- AcceptanceCriterion(S, S'', history);
```

O critério de aceitação é completamente elitista. Só são aceitas as melhores soluções. No caso, o parâmetro "history"nem sequer é considerado.

```
until stopping criterion is not satisfied anymore
```

O critério de aceitação é também um parâmetro ajustável. São passados para o critério (que é na prática uma função *lambda*, uma *callback* para o "resolvedor") as seguintes informações do status atual da ILS:

- Melhor solução
- Tamanho da perturbação

- Tempo desde a última melhora
- Tempo desde o início do algoritmo
- Número de iterações deste a última melhora

Tipicamente, é um critério de parada o tamanho da perturbação ser 1, pois indica que a perturbação efetuará um movimento, que será facilmente revertido pela busca local. É usado por padrão no algoritmo também como critério de parada o gap da melhor solução ser igual a 0.0%, já que todas as instâncias apresentadas possuem BKS comprovadamente ótimo.

### Genetic Algorithm (Iterativo)

O algoritmo iterativo segue o seguinte pseudo-código.

```
P <- InitialPopulation
repeat:
    P <- NextGeneration(P)
until stopping criterion is not satisfied anymore
return P
```

Vai ser aqui então comentada cada linha do pseudo-código, explicando como de fato foi implementada.

```
P <- InitialPopulation
```

É criada uma população inicial com um tamanho mínimo (parâmetro) e máximo (parâmetro). As soluções são criadas por uma heurística construtiva gulosa randomizada - isto é, é escolhido o próximo vizinho aleatoriamente dentre os $w$ (parâmetro) nós mais próximos.

```
until stopping criterion is not satisfied anymore
```

O critério de aceitação é também um parâmetro ajustável. São passados para o critério (que é na prática uma função *lambda*, uma *callback* para o "resolvedor") as seguintes informações do status atual do GA:

- Melhor solução
- Número da geração
- Tempo desde a última melhora
- Tempo desde o início do algoritmo
- Número de gerações desde a última melhora

```
P <- NextGeneration(P)
```

É efetuada uma geração. Este passo será esclarecido com mais detalhes a seguir. Na prática, a operação ocorre na população em si, e não é criada outra a partir da primeira.

### Genetic Algorithm (Próxima Geração)

O algoritmo que gera novas proles pode ser sintetizada pelo seguinte pseudo-código.

```
mating_pool <- {}
for i = 1..mating_pool_size:
    Sa, Sb <- P.sample(2)
    if cost(Sa) < cost(Sb):
        mating_pool <- mating_pool U {Sa}
    else:
        mating_pool <- mating_pool U {Sb}
for wife, husband in mating_pool:
    if wife == husband:
        continue
    offspring <- crossover(wife, husband)
    if apply_mutation(mutation_chance):
        p <- unif(pmin, pmax)
        offspring <- perturb(offspring, p)
        offspring <- local_minima(offspring)
    P <- P U {offspring}
if |P| > maxsize:
    P <- P - clones(P)
    repeat:
        P <- P - worst(P)
    while |P| > minsize
```

Aonde as seguintes linhas referem-se a explicações sobre suas implementações.

```
Sa, Sb <- P.sample(2)
```

Isso se traduz a obter duas soluções aleatórias de uma população. Obs: A variável "mating_pool_size"é um parâmetro.

```
for wife, husband in mating_pool:
```

Aqui, "wife"e "husband"são nomes-fantasia para pares de soluções adjacentes únicos. Na prática, os pares são "casados"pelos índices, exemplo: (0,1), (2,3), (4,5), ...

```
if wife == husband:
    continue
```

Aqui nos certificamos que os pais não são coincidentes, para evitar clones na população. Isto pode ser enxergado como uma otimização, pois ao fim do algoritmo, todos os clones serão descartados de igual forma.

```
offspring <- crossover(wife, husband)
```

Aqui o crossover aplicado é o *Cycle Crossover (CX)*. Como a sua real implementação (em *src/tspsol/solution.cpp*) é muito pouco abstrata, o seu pseudo-código não estará presente aqui neste documento. Contudo, é um algoritmo muito difundido na literatura e seu pseudo-código pode ser facilmente encontrados na Internet. Um vídeo que ilustra bem este crossover está disponível no Youtube **(clique aqui)**.

```
if apply_mutation(mutation_chance):
```

Aqui é feito um ensaio de Bernoulli com probabilidade de sucesso dado pelo número real "mutation_chance"(parâmetro).

```
p <- unif(pmin, pmax)
```

Aqui é feito um ensaio de uma distribuição uniforme entre $pmin$ e $pmax$, representando as perturbações mínimas e máximas a serem aplicadas à prole - correspondente a uma "mutação", na literatura.

```
offspring <- perturb(offspring, p)
```

Aqui é feito uma perturbação de magnitude $p$ sobre a solução "offspring". Na prática, é realizado um *Variable Neighbourhood Search*, em que são efetuados $p \cdot n$ movimentos válidos, aonde $n$ é o número de nós.

```
offspring <- local_minima(offspring)
```

Aqui é feita uma busca local nas vizinhanças da solução "offspring". Na prática, o mínimo local também é obtido usando um *Variable Neighbourhood Search*, mas aplicando apenas movimentos válidos e vantajosos.

A busca local é realizada apenas quando há uma mutação, assim favorecendo o crossover como o operador principal do algoritmo genético, e não a busca local. Por ser uma operação mais custosa, a busca local deve ser realizada raramente, e seus "genes"serão propagados para novas proles por meio do crossover.

```
P <- P - clones(P)
```

A função "clones"obtém as soluções idênticas da população. Quando removidas, a população continua com um único "exemplar"destes clones, para não prejudicar a diversidade da população.

```
P <- P - worst(P)
```

A função "worst"obtém a pior solução da população, meramente em função do custo.

### 1.3.2 Como executar o programa

Todos os executáveis exibem uma mensagem de ajuda quando o parâmetro *–help* é passado na linha de comando. No caso do "resolvedor", alguns parâmetros importantes estão expostos na tabela 1.1.

| Comando | Significado |
|---|---|
| –help | Ajuda |
| –ifile | Caminho do arquivo de instância |
| –ifolder | Caminho do diretório com instâncias (arquivos ".tsp") |
| –heuristic | Identificador da heurística ("ils"ou "gen"). Padrão: "ils" |
| –verbose | Saída padrão em modo verboso |
| –seed | Semente do gerador de números aleatórios. Padrão: 2020 |

Tabela 1.1: Alguns parâmetros de linha de comandos do **solverapp**.

Todos os caminhos são relativos à pasta **data**. Alguns exemplos de uso:

```
$ solverapp --ifolder=. --heuristic=gen
```

O comando acima roda a heurtística genética para todas as instâncias.

```
$ solverapp --ifolder=. --heuristic=ils
```

O comando acima roda a heurtística ILS para todas as instâncias.

### 1.3.3 Bibliotecas

Para modularizar o projeto, foram criadas bibliotecas para cada seção independente.

#### iparserlib

Primeiramente, foram formalizadas as estruturas de dados que representariam uma instância de problema de mínima latência, e de uma solução para uma dada instância. Para tal, foi preciso criar a biblioteca **iparserlib** para ler os arquivos de instância e produzir uma matriz de distâncias quadrada de dimensão n, em que n é o número de nós, dentre outros campos não essenciais para a resolução do problema (como nome, descrição, matriz de posições dos nós...).

#### tspsollib

Assim feito, foi criada outra biblioteca, chamada **tspsollib** que formalizava a representação das soluções como uma **lista encadeada de nós**, em que o primeiro e último nós eram o depósito e os outros eram clientes distintos. No começo, a solução inicial era aquela que perpassava os nós em ordem (0,1,2,...,n-1,n,0). Foi escolhida esta estrutura de dados pois as operações de inserção e remoção de um nó - muito frequentes em buscas locais - têm complexidade O(1), descontando o tempo de obter o n-ésimo nó arbitrário de uma lista.

#### bksparserlib

Para se poder comparar o custo de uma solução com a melhor conhecida na literatura (ou *BKS*, de *Best Known Solution*), foi implementada outra biblioteca, chamada de **bksparserlib**, que lê o arquivo **data/bks.txt**, associando cada nome de instância a um número inteiro, o menor custo possível para uma solução sua. Assim, já foi possível incorporar o método **Solution::GetCostGap**, que retorna o gap de uma solução.

#### argparserlib

Para auxiliar no reconhecimento léxico de parâmetros na linha de comando, foi implementada a biblioteca **argparserlib**. Graça aos argumentos de linha de comando, é possível alterar parâmetros de execução sem mudar o código, o que apresenta inúmeras vantagens.

#### csvlib

Para auxiliar na geração de arquivos CSV, foi implementada a biblioteca **csvlib**. Estes arquivos são tipicamente relatórios com informações a respeito das instâncias resolvidas, a heurística empregada, os parâmetros do algoritmo, tempo de execução e gap da função custo.

#### tspvislib

Para auxiliar na visualização das instâncias que possuem informações a respeito das disposição dos nós no espaço bidimensional, foi implementada a biblioteca **tspvislib**, que visualiza não somente instâncias mas também soluções e populações (ver tspgenlib).

### 1.3.4 Aplicações

#### solverapp

Resolve uma instância ou múltiplas instâncias em um diretório, aplicando uma heurística. Ou, ainda, carrega uma solução pré-existente e serializada

em um arquivo, e aplica uma heurística. É capaz também de gerar um relatório em formato CSV com tempo de execução e gap de cada solução final obtida. É a aplicação chamada de "resolvedor"neste arquivo, informalmente.

#### readerapp

Lê um arquivo de solução e informa o valor da função custo e o gap em função da melhor solução conhecida. É possível, ainda, exportar esta informação em formato CSV.

#### tspvisapp

Permite visualizar uma instância, uma solução, ou, ainda, uma população. Possui ainda uma funcionalidade interativa que permite avançar um dado número de gerações, visualizar cada solução de uma população e seus "gaps".

## 1.4 Resultados

O "resolvedor"foi executado para cada instância com 5 seeds diferentes, para que a média fosse obtida e os ruídos gerados pelo aleatoriedade amenizados. Caso deseje replicar os resultados, as seeds utilizadas foram os números 1 ao 5, com os mesmos parâmetros que os exemplos mostrados anteriormente.

| Instância | Gap (%) | Tempo (s) |
|---|---|---|
| brazil58 | **0,00%** | $< 1$ |
| dantzig42 | **0,00%** | $< 1$ |
| gr120 | -0,60% | 27 |
| gr48 | **0,00%** | $< 1$ |
| pa561 | -3,88% | 1325 |
| MÉDIA | -0,90% | 270 |

Tabela 1.2: Resultados médios do algoritmo **ILS**

| Instância | Gap (%) | Tempo (s) |
|---|---|---|
| brazil58 | **0,00%** | $< 1$ |
| dantzig42 | **0,00%** | $< 1$ |
| gr120 | -0,66% | 35 |
| gr48 | **0,00%** | $< 1$ |
| pa561 | -1,48% | 10900 |
| MÉDIA | -0,43% | 2187 |

Tabela 1.3: Resultados médios do algoritmo **genético**

Pode-se perceber que o algoritmo genético, em média, para as instâncias testadas, obteve soluções de melhor qualidade, apesar de em um tempo muito maior (até de 8 horas).

Contudo, o algoritmo ILS conseguiu atingir, em média, soluções de qualidade semelhante em muito menos tempo.

Para instâncias pequenas (com menos de 120 nós, por exemplo), os algoritmos se comportaram de forma idêntica - atingindo soluções ótimas em um curto espaço de tempo.

## 1.5 Conclusão

Conclui-se que é ambos os algoritmos obtiveram soluções de qualidade similar, contudo, com uma grande diferença em questão de performance. Para instâncias grandes, o algoritmo ILS se comporta melhor, construindo soluções de boa qualidade e em menor período de tempo.

# Apêndice A

# Estudo de vizinhanças

## A.1 Introdução

Este documento tem como interesse analisar como que a função custo de um problema de mínima latência se comporta após algum movimento, e tentar quantificar com fórmulas fechadas quando este movimento é vantajoso. Primeiro serão estabelecidas algumas definições a respeito da natureza do problema e, então, serão feitas análises dos principais movimentos.

## A.2 Definições

### A.2.1 Instância

Definimos uma instância de um problema de mínima latência por uma matriz quadrada $D$ de dimensão $n$, aonde o elemento $D(i, j)$ desta matriz é distância entre os nós $i$ e $j$. Definimos o nó 0 como sendo o **depósito**. Definimos os nós $1...n-1$ como sendo os **clientes**. Dizemos que a instância tem tamanho $n$ quando sua matriz tem dimensão $n$. Trabalharemos apenas com instâncias de dimensão finita.

### A.2.2 Solução

Definimos uma solução como uma permutação dos $n-1$ clientes de uma instância de tamanho $n$. Dizemos que a solução tem tamanho $n$ se a instância associada tem tamanho $n$. Trabalharemos apenas com soluções cujas instâncias têm dimensão maior que 2, ou seja, com ao menos dois clientes. Para razões práticas, representamos esta sequência começando e terminando com o depósito, como na equação A.1.

$$S =< s_0^*, s_1, ..., s_{n-1}, s_n^* > \tag{A.1}$$

Aonde ambos os nós com * indicam o **depósito**. Para remover a ambiguidade, chamaremos $s_0$ de **depósito inicial** e $s_n$ de **depósito final**. Mas

lembre-se que ambos os depósitos representam o mesmo nó, o nó de índice 0 na matriz.

### A.2.3 Custo de uma solução

Definimos o custo de uma solução como sendo a soma das latências dos nós $s_1, ..., s_n$. A latência do i-ésimo nó da sequência $S$, $l(S, i)$ é dado pela sua distância até o depósito inicial, vide a equação A.2.

$$l(S, i) = \sum_{j=1}^{i} D(s_{j-1},\ s_j),\ i \leq n \tag{A.2}$$

Obtemos a equação recursiva A.3 para o i-ésimo nó, que não o depósito inicial.

$$l(S, i) = l(S, i-1) + D(s_{i-1},\ s_i),\ 0 < i \leq n \tag{A.3}$$

Portanto, o custo de uma solução $S$, $f(S)$ é dado pela equação A.4.

$$f(S) = \sum_{i=1}^{n} l(S, i) = \sum_{i=1}^{n} \sum_{j=1}^{i} D(s_{j-1},\ s_j) \tag{A.4}$$

É possível perceber que uma parcela $D(s_{j-1},\ s_j)$, para $1 \leq j \leq n$ ocorre $(n - j + 1)$ vezes na função $f(S)$ de uma solução $S$ de tamanho $n$, o que nos permite reescrever a equação A.4 com apenas um somatório na equação A.5.

$$f(S) = \sum_{j=1}^{n} (n - j + 1) \cdot D(s_{j-1},\ s_j) = \sum_{j=1}^{n} H(S, j) \tag{A.5}$$

Uma interpretação da equação A.5 é que uma aresta mais perto do depósito inicial tem um maior peso no custo da solução. É importante destacar que $H(S, j)$ depende de $s_{j-1}$ e de $s_j$.

### A.2.4 Movimento

Chamamos de movimento uma função $m$ bijetiva com domínio nos clientes de uma solução $S$ e com contra-domínio nos clientes de uma solução $S'$.

### A.2.5 Custo de um movimento

Definimos o custo de um movimento $m$ para uma dada solução $S$ como $\delta(m, S)$, dada pela equação A.6.

$$\delta(m, S) = f(m(S)) - f(S) \tag{A.6}$$

### A.2.6 Movimento vantajoso

Dizemos que um movimento $m$ é vantajoso para uma dada solução $S$ quando $\delta(m, S) < 0$, isto é, quando o custo da solução diminui com o movimento.

### A.2.7 Movimento indiferente

Dizemos que um movimento $m$ é indiferente para uma dada solução $S$ quando $\delta(m, S) = 0$, isto é, quando o custo da solução não se altera com o movimento.

### A.2.8 Movimento desvantajoso

Dizemos que um movimento $m$ é desvantajoso para uma dada solução $S$ quando $\delta(m, S) > 0$, isto é, quando o custo da solução aumenta com o movimento.

## A.3 Metodologia

Será definido cada movimento $m$ formalmente em função da posição dos clientes da solução $S$ para a solução $m(S)$. Então, será feito o cálculo do custo do movimento, procurando eliminar o maior número de parcelas possíveis da soma.

Além disso, se possível, serão feitas substituições triviais como a equação A.7, pois em casos em que a latência de todos os nós estejam pré-calculados, este cálculo terá complexidade temporal $O(1)$, e não $O(n)$.

$$\sum_{j=a}^{b} D(s_{j-1},\ s_j) = l(S, b) - l(S, a-1) \tag{A.7}$$

## A.4 Movimentos

### A.4.1 shift(p,q)

Um cliente na posição $p$ é deslocado para outra posição $q$.

#### rightshift ($p < q$)

Movimento definido para um par de índices $(p,\ q)$, tal que $0 < p < q < n$. Dado uma solução $S$, rightshift$(S, p, q) = S^*$, e o i-ésimo cliente da solução $S^*$ é definido pela equação A.8.

$$s_i^* = \begin{cases} s_i, \ i < p \\ s_{i+1}, \ p \leq i < q \\ s_p, \ i = q \\ s_i, \ i > q \end{cases} \tag{A.8}$$

O custo da solução com o movimento é dado pela equação a seguir. Serão riscados de vermelhos as parcelas que não dependem mais de $S^*$, mas somente de $S$, para deixar o papel mais limpo. Mas imagine que elas ainda estão lá.

$$\begin{aligned}
f(S^*) &= \sum_{j=1}^{n} H(S^*, j) \\
&= \sum_{j=1}^{p-1} H(S^*, j) + \sum_{j=p}^{n} H(S^*, j) \\
&= \cancel{\sum_{j=1}^{p-1} H(S, j) +} \sum_{j=p}^{n} H(S^*, j) \\
&= \sum_{j=p}^{q+1} H(S^*, j) + \sum_{j=q+2}^{n} H(S^*, j) \\
&= \sum_{j=p}^{q+1} H(S^*, j) + \cancel{\sum_{j=q+2}^{n} H(S, j)} \\
&= H(S^*, p) + \sum_{j=p+1}^{q-1} H(S, j) + \sum_{j=q}^{q+1} H(S^*, j) \\
&= H(S^*, p) + \sum_{j=p+2}^{q} D(s_{j-1}, s_j) \cdot (n - (j-1) + 1) + \sum_{j=q}^{q+1} H(S^*, j) \\
&= H(S^*, p) + \cancel{\sum_{j=p+2}^{q} H(S, j) +} \sum_{j=p+2}^{q} D(s_{j-1}, s_j) + \sum_{j=q}^{q+1} H(S^*, j) \\
&= H(S^*, p) + l(S, q) - l(S, p+1) + \sum_{j=q}^{q+1} H(S^*, j) \\
&= D(s_{p-1}, s_{p+1}) \cdot (n - p + 1) \\
&+ D(s_q, s_p) \cdot (n - q + 1) \\
&+ D(s_p, s_{q+1}) \cdot (n - q) \\
&+ l(S, q) - l(S, p+1)
\end{aligned}$$

O custo da solução $S$ terá os membros anteriormente riscados removidos. Já que nosso interesse é calcular a diferenças dos dois, teremos o mesmo resultado.

$$
\begin{aligned}
f(S) &= \sum_{j=1}^{n} H(S,j) \\
&= \cancel{\sum_{j=1}^{p-1} H(S,j) +} H(s,p) + H(s,p+1) \\
&+ \cancel{\sum_{j=p+2}^{q} H(S,j) +} H(s,q+1) + \cancel{\sum_{j=q+2}^{n} H(S,j)} \\
&= D(s_{p-1}, s_p) \cdot (n-p+1) \\
&+ D(s_p, s_{p+1}) \cdot (n-p) \\
&+ D(s_q, s_{q+1}) \cdot (n-q)
\end{aligned}
$$

Assim, temos que $\delta(\text{rightshift}, S)$ é dado pela seguinte equação.

$$
\begin{aligned}
\delta(\text{rightshift}, S) = &(n-p+1) \cdot [D(s_{p-1}, s_{p+1}) - D(s_{p-1}, s_p)] \\
&+ (n-q) \cdot [D(s_p, s_{q+1}) - D(s_q, s_{q+1})] \\
&+ (n-q+1) \cdot D(s_q, s_p) \\
&+ l(S,q) \\
&- l(S,p+1) \\
&- (n-p) \cdot D(s_p, s_{p+1})
\end{aligned}
$$

**leftshift** $(p > q)$

Movimento definido para um par de índices $(p,\ q)$, tal que $0 < q < p < n$. Dado uma solução $S$, lefttshift$(S,p,q) = S^*$, e o i-ésimo cliente da solução $S^*$ é definido pela equação A.9.

$$
s_i^* = \begin{cases} s_i, & i < q \\ s_p, & i = q \\ s_{i-1}, & q < i \leq p \\ s_i, & i > p \end{cases} \tag{A.9}
$$

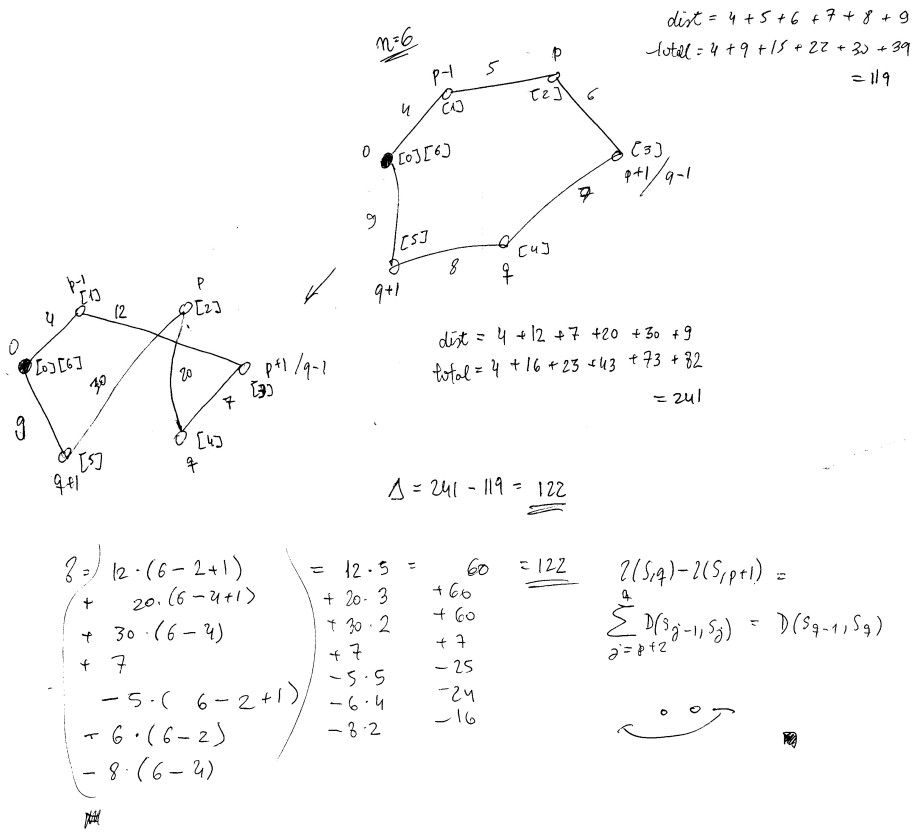


Figura A.1: Exemplo de right shift para $n = 6$ com cálculos

$$
\begin{aligned}
f(S^*) &= \sum_{j=1}^{n} H(S^*, j) \\
&= \sum_{j=1}^{q-1} H(S^*, j) + \sum_{j=q}^{n} H(S^*, j) \\
&= \cancel{\sum_{j=1}^{q-1} H(S, j)} + \sum_{j=q}^{n} H(S^*, j) \\
&= \sum_{j=q}^{p+1} H(S^*, j) + \sum_{j=p+2}^{n} H(S^*, j) \\
&= \sum_{j=q}^{p+1} H(S^*, j) + \cancel{\sum_{j=p+2}^{n} H(S, j)} \\
&= \sum_{j=q}^{q+1} H(S^*, j) + \sum_{j=q+2}^{p} H(S^*, j) + H(S^*, p+1) \\
&= \sum_{j=q}^{q+1} H(S^*, j) + \sum_{j=q+2}^{p} \left[D(s^*_{j-1}, s^*_j) \cdot (n-j+1)\right] + H(S^*, p+1) \\
&= \sum_{j=q}^{q+1} H(S^*, j) + \sum_{j=q+2}^{p} \left[D(s_{j-2}, s_{j-1}) \cdot (n-j+1)\right] + H(S^*, p+1) \\
&= \sum_{j=q}^{q+1} H(S^*, j) + \sum_{j=q+1}^{p-1} \left[D(s_{j-1}, s_j) \cdot (n-(j+1)+1)\right] + H(S^*, p+1) \\
&= \sum_{j=q}^{q+1} H(S^*, j) + \cancel{\sum_{j=q+1}^{p-1} H(S, j)} - \sum_{j=q+1}^{p-1} \left[D(s_{j-1}, s_j)\right] + H(S^*, p+1) \\
&= H(S^*, q) + H(S^*, q+1) + H(S^*, p+1) + l(S, q) - l(S, p-1)
\end{aligned}
$$

$$
\begin{aligned}
f(S) &= \sum_{j=1}^{n} H(S, j) \\
&= \cancel{\sum_{j=1}^{q-1} H(S, j)} + H(s, q) + \cancel{\sum_{j=q+1}^{p-1} H(S, j)} \\
&\quad + H(s, p) + H(s, p+1) + \cancel{\sum_{j=p+2}^{n} H(S, j)} \\
&= H(s, q) + H(s, p) + H(s, p+1)
\end{aligned}
$$

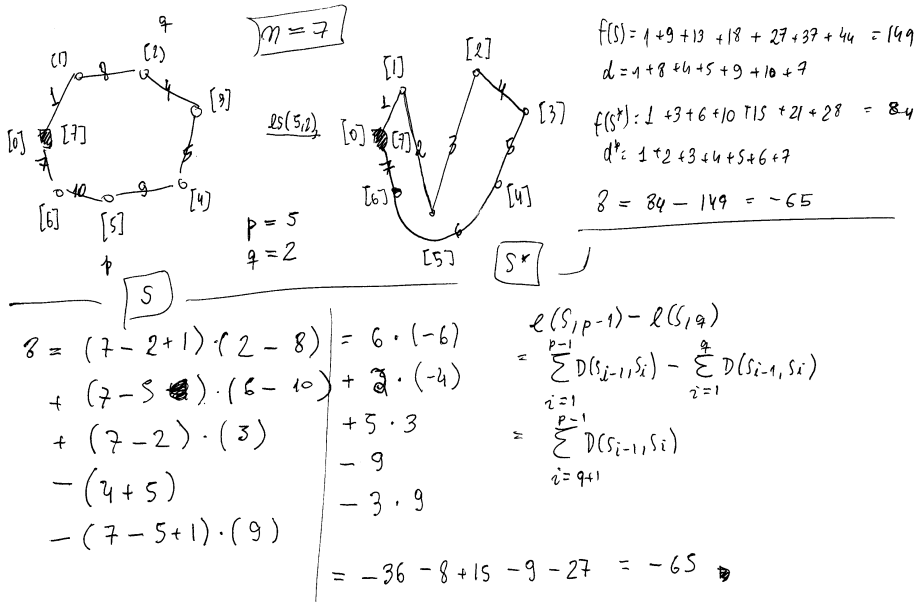


Figura A.2: Exemplo de left shift para $n = 7$ com cálculos

$$\begin{aligned}\delta(\text{leftshift}, S) = &(n-q+1)\cdot[D(s_{q-1}, s_p) - D(s_{q-1}, s_q)]\\ &+ (n-p)\cdot[D(s_{p-1}, s_{p+1}) - D(s_p, s_{p+1})]\\ &+ (n-q)\cdot D(s_p, s_q)\\ &+ l(S, q)\\ &- l(S, p-1)\\ &- (n-p+1)\cdot D(s_{p-1}, s_p)\end{aligned}$$

### A.4.2 swap(p,q)

Um cliente na posição $p$ é trocado com outro na posição $q$. Como $\text{swap}(S, p, q) = \text{swap}(S, q, p)$, então os cálculos aqui feitos levarão em conta que $0 < p < q < n$. Dado uma solução $S$, $\text{swap}(S, p, q) = S^*$, e o i-ésimo cliente da solução $S^*$ é definido pela equação A.10.

$$s_i^* = \begin{cases} s_i, & i < p \\ s_q, & i = p \\ s_i, & p < i < q \\ s_p, & i = q \\ s_i, & i > q \end{cases} \tag{A.10}$$

$$
\begin{aligned}
\delta(\text{swap}, S) &= f(S^*) - f(S) \\
&= \sum_{j=1}^{n} H(S^*, j) - \sum_{j=1}^{n} H(S, j) \\
&= \left[\sum_{j=1}^{p-1} H(S^*, j) + \sum_{j=p}^{n} H(S^*, j)\right] - \left[\sum_{j=1}^{p-1} H(S, j) + \sum_{j=p}^{n} H(S, j)\right] \\
&= \left[\cancel{\sum_{j=1}^{p-1} H(S, j) +} \sum_{j=p}^{n} H(S^*, j)\right] - \left[\cancel{\sum_{j=1}^{p-1} H(S, j) +} \sum_{j=p}^{n} H(S, j)\right] \\
&= \sum_{j=p}^{n} H(S^*, j) - \sum_{j=p}^{n} H(S, j) \\
&= \left[\sum_{j=p}^{q+1} H(S^*, j) + \sum_{j=q+2}^{n} H(S^*, j)\right] - \left[\sum_{j=p}^{q+1} H(S, j) + \sum_{j=q+2}^{n} H(S, j)\right] \\
&= \left[\sum_{j=p}^{q+1} H(S^*, j) + \cancel{\sum_{j=q+2}^{n} H(S, j)}\right] - \left[\sum_{j=p}^{q+1} H(S, j) + \cancel{\sum_{j=q+2}^{n} H(S, j)}\right] \\
&= \sum_{j=p}^{q+1} H(S^*, j) - \sum_{j=p}^{q+1} H(S, j) \\
&= \left[\sum_{j=p}^{p+1} H(S^*, j) + \sum_{j=p+2}^{q-1} H(S^*, j) + \sum_{j=q}^{q+1} H(S^*, j)\right] \\
&- \left[\sum_{j=p}^{p+1} H(S, j) + \sum_{j=p+2}^{q-1} H(S, j) + \sum_{j=q}^{q+1} H(S, j)\right] \\
&= \left[\sum_{j=p}^{p+1} H(S^*, j) + \cancel{\sum_{j=p+2}^{q-1} H(S, j) +} \sum_{j=q}^{q+1} H(S^*, j)\right] \\
&- \left[\sum_{j=p}^{p+1} H(S, j) + \cancel{\sum_{j=p+2}^{q-1} H(S, j) +} \sum_{j=q}^{q+1} H(S, j)\right] \\
&= [H(S^*, p) + H(S^*, p+1) + H(S^*, q) + H(S^*, q+1)] \\
&- [H(S, p) + H(S, p+1) + H(S, q) + H(S, q+1)] \\
&= (n - p + 1) \cdot [D(s_{p-1}, s_q) - D(s_{p-1}, s_p)] \\
&+ (n - p) \cdot [D(s_q, s_{p+1}) - D(s_p, s_{p+1})] \\
&+ (n - q + 1) \cdot [D(s_{q-1}, s_p) - D(s_{q-1}, s_q)] \\
&+ (n - q) \cdot [D(s_p, s_{q+1}) - D(s_q, s_{q+1})]
\end{aligned}
$$

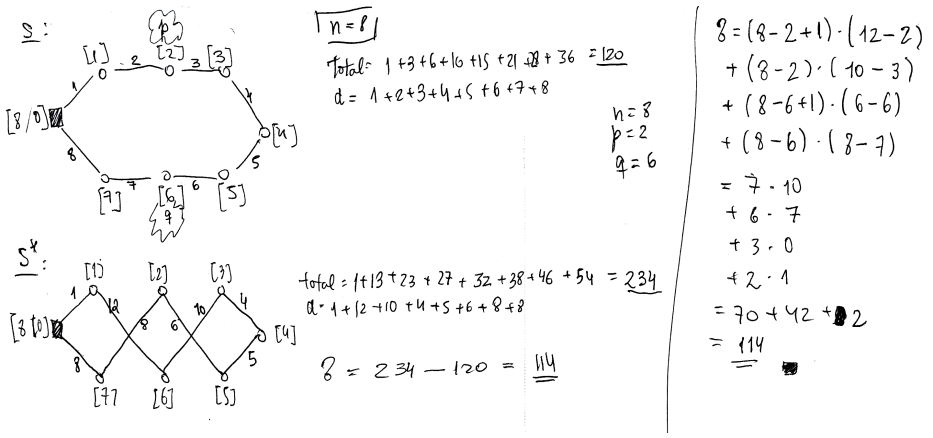

Figura A.3: Exemplo de swap para $n = 8$ com cálculos

É importante notar que a equação acima só é válida quando $p + 1 \neq q$, em outras palavras, somente quando $p$ e $q$ não são vizinhos. Contudo, não será coberto este caso pois ele é idêntico ao movimento de shift$(p, q)$.

### A.4.3 2-opt(p,q)

A rota de $p$ a $q$ é invertida. Como 2-opt$(S, p, q)$ = 2-opt$(S, q, p)$, então os cálculos aqui feitos levarão em conta que $0 < p < q < n$. Dado uma solução $S$, 2-opt$(S, p, q) = S^*$, e o i-ésimo cliente da solução $S^*$ é definido pela equação A.11.

$$s_i^* = \begin{cases} s_i, & i < p \\ s_{\phi(i)}, & p \leq i \leq q \\ s_i, & i > q \end{cases} \tag{A.11}$$

$$\phi(i) = p + q - i \tag{A.12}$$

$$\begin{aligned} f(S^*) &= \sum_{j=1}^{n} H(S^*, j) \\ &= \cancel{\sum_{j=1}^{p-1} H(S, j)} + \sum_{j=p}^{q+1} H(S^*, j) + \cancel{\sum_{j=q+2}^{n} H(S, j)} \\ &= D(s_{p-1}^*, s_p^*) \cdot (n - p + 1) \\ &+ D(s_q^*, s_{q+1}^*) \cdot (n - q) && \alpha^* \\ &+ \sum_{j=p+1}^{q} D(s_{j-1}^*, s_j^*) \cdot (n - j + 1) && \beta^* \end{aligned}$$

$$\begin{aligned}
\beta^* &= \sum_{j=p+1}^{q} D(s_{p+q-j+1}, s_{p+q-j}) \cdot (n-j+1) \\
&= \sum_{j=p}^{q-1} D(s_{j+1}, s_j) \cdot [n-(p+q-j)+1] \\
&= \sum_{j=p+1}^{q} D(s_j, s_{j-1}) \cdot [n-(p+q-j)] \\
&= \sum_{j=p+1}^{q} D(s_{j-1}, s_j) \cdot [n-(p+q-j)]
\end{aligned}$$

$$\begin{aligned}
f(S) &= \sum_{j=1}^{n} H(S,j) \\
&= \cancel{\sum_{j=1}^{p-1} H(S,j)} + \sum_{j=p}^{q+1} H(S,j) + \cancel{\sum_{j=q+2}^{n} H(S,j)} \\
&= D(s_{p-1}, s_p) \cdot (n-p+1) \\
&+ D(s_q, s_{q+1}) \cdot (n-q) && \alpha \\
&+ \sum_{j=p+1}^{q} D(s_{j-1}, s_j) \cdot (n-j+1) && \beta
\end{aligned}$$

$$\begin{aligned}
\delta(\text{2-opt}, S) = &(n-p+1) \cdot (D(s_{p-1}, s_q) - D(s_{p-1}, s_p)) \\
&+ (n-q) \cdot (D(s_p, s_{q+1}) - D(s_q, s_{q+1})) \\
&+ \sum_{j=p+1}^{q} D(s_{j-1}, s_j) \cdot (2j-p-q-1)
\end{aligned}$$

Esta equação só serve para soluções em que $p+1 \neq q$, pois presume que $p$ e $q$ não são vizinhos, ou seja, que possuem algum cliente entre eles. Mas caso $p$ e $q$ sejam vizinhos, o efeito será o mesmo de fazer um shift$(p,q)$. E ainda, pode-se ainda desconsiderar o caso de $p+2=q$, pois o efeito será o mesmo de um swap$(p,q)$.

### A.4.4 shift2(p,q,r)

Os clientes enter $p$ e $q$ são movidos para depois do cliente $r$.

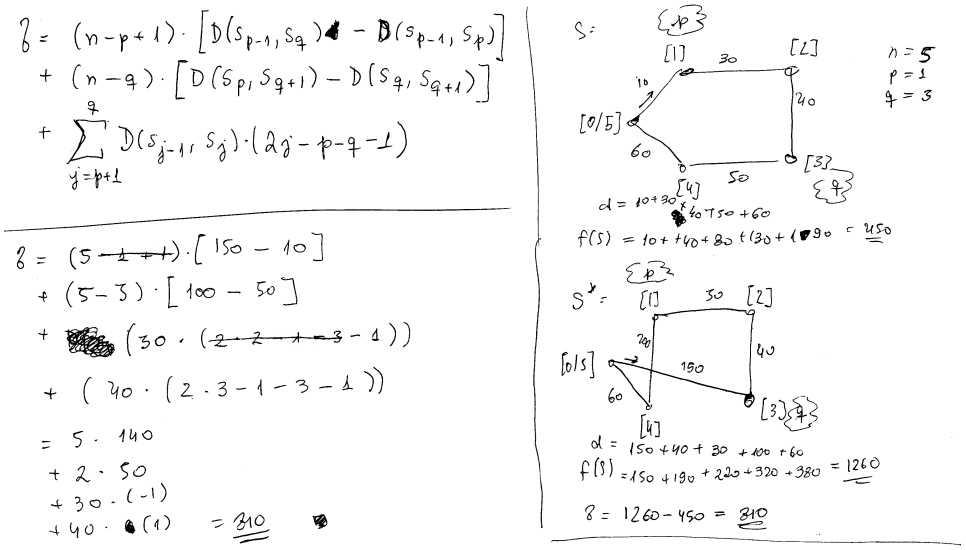

Figura A.4: Exemplo de 2-opt para $n = 5$ com cálculos

**rightshift2** $(p < q < r)$

Movimento definido para uma tupla de índices $(p,\ q,\ r)$, tal que $0 < p < q < q + 1 < r < n$ (os clientes $q$ e $r$ não podem ser vizinhos). Dado uma solução $S$, rightshift2$(S, p, q, r) = S^*$, e o i-ésimo cliente da solução $S^*$ é definido pela equação A.13.

$$s_i^* = \begin{cases} s_i,\ i < p \\ s_{\phi(i)},\ p \leq i < p + r - q,\ \phi(i) = i - p + q + 1 \\ s_{\psi(i)},\ p + r - q \leq i \leq r,\ \psi(i) = i - r + q \\ s_i,\ i > r \end{cases} \tag{A.13}$$

$$
\begin{aligned}
f(S^*) &= \sum_{j=1}^{n} H(S^*, j) \\
&= \cancel{\sum_{j=1}^{p-1} H(S, j)} + \sum_{j=p}^{r+1} H(S^*, j) + \cancel{\sum_{j=r+2}^{n} H(S, j)} \\
&= [H(S^*, p) + H(S^*, p + r - q) + H(S^*, r + 1)] \leftarrow \alpha^* \\
&+ \left[ \sum_{j=p+1}^{p+r-q-1} H(S^*, j) \right] \leftarrow \beta_1^* \\
&+ \left[ \sum_{j=p+r-q+1}^{r} H(S^*, j) \right] \leftarrow \beta_2^*
\end{aligned}
$$

$$
\begin{aligned}
f(S) &= \sum_{j=1}^{n} H(S, j) \\
&= \cancel{\sum_{j=1}^{p-1} H(S, j)} + \sum_{j=p}^{r+1} H(S, j) + \cancel{\sum_{j=r+2}^{n} H(S, j)} \\
&= [H(S, p) + H(S, q + 1) + H(S, r + 1)] \leftarrow \alpha \\
&+ \left[ \sum_{j=p+1}^{q} H(S, j) \right] \leftarrow \beta_2 \\
&+ \left[ \sum_{j=q+2}^{r} H(S, j) \right] \leftarrow \beta_1
\end{aligned}
$$

$$
\begin{aligned}
\alpha = & D(s_{p-1}, s_p) \cdot (n - p + 1) \\
& + D(s_q, s_{q+1}) \cdot (n - q) \\
& + D(s_r, s_{r+1}) \cdot (n - r)
\end{aligned}
$$

$$\begin{aligned}
\alpha^* &= D(s^*_{p-1}, s^*_p) \cdot (n-p+1) \\
&+ D(s^*_{p+r-q-1}, s^*_{p+r-q}) \cdot [n-(p+r-q)+1] \\
&+ D(s^*_r, s^*_{r+1}) \cdot (n-r) \\
&= D(s_{p-1}, s_{q+1}) \cdot (n-p+1) \\
&+ D(s_r, s_p) \cdot [n-(p+r-q)+1] \\
&+ D(s_q, s_{r+1}) \cdot (n-r)
\end{aligned}$$

$$\begin{aligned}
\beta^*_1 &= \sum_{j=p+1}^{p+r-q-1} H(S^*, j) \\
&= \sum_{j=p+1}^{p+r-q-1} D(s^*_{j-1}, s^*_j) \cdot (n-j+1) \\
&= \sum_{j=p+1}^{p+r-q-1} D(s_{j-p+q}, s_{j-p+q+1}) \cdot (n-j+1) \\
&= \sum_{j=q+2}^{r} D(s_{j-1}, s_j) \cdot [n-(j+p-q-1)+1]
\end{aligned}$$

$$\begin{aligned}
\beta^*_2 &= \sum_{j=p+r-q+1}^{r} H(S^*, j) \\
&= \sum_{j=p+r-q+1}^{r} D(s^*_{j-1}, s^*_j) \cdot (n-j+1) \\
&= \sum_{j=p+r-q+1}^{r} D(s_{j-r+q-1}, s_{j-r+q}) \cdot (n-j+1) \\
&= \sum_{j=p+1}^{q} D(s_{j-1}, s_j) \cdot [n-(j+r-q)+1]
\end{aligned}$$

$$\begin{aligned}\beta_1^* - \beta_1 &= \sum_{j=q+2}^{r} D(s_{j-1}, s_j) \cdot [n - (j + p - q - 1) + 1] \\ &\quad - \sum_{j=q+2}^{r} D(s_{j-1}, s_j) \cdot (n - j + 1) \\ &= \sum_{j=q+2}^{r} D(s_{j-1}, s_j) \cdot (q - p + 1) \\ &= [l(S, r) - l(S, q + 1)] \cdot (q - p + 1)\end{aligned}$$

$$\begin{aligned}\beta_2^* - \beta_2 &= \sum_{j=p+1}^{q} D(s_{j-1}, s_j) \cdot [n - (j + r - q) + 1] \\ &\quad - \sum_{j=p+1}^{q} D(s_{j-1}, s_j) \cdot (n - j + 1) \\ &= \sum_{j=p+1}^{q} D(s_{j-1}, s_j) \cdot (q - r) \\ &= -[l(S, q) - l(S, p)] \cdot (r - q)\end{aligned}$$

$$\begin{aligned}\delta(\text{rightshift2}, S) &= \alpha^* - \alpha + \beta_1^* - \beta_1 + \beta_2^* - \beta_2 \\ &= (n - p + 1) \cdot [D(s_{p-1}, s_{q+1}) - D(s_{p-1}, s_p)] \\ &\quad + (n - r) \cdot [D(s_q, s_{r+1}) - D(s_r, s_{r+1})] \\ &\quad + (n - p - r + q + 1) \cdot D(s_r, s_p) \\ &\quad + (q - p + 1) \cdot [l(S, r) - l(S, q + 1)] \\ &\quad - (n - q) \cdot D(s_q, s_{q+1}) \\ &\quad - (r - q) \cdot [l(S, q) - l(S, p)]\end{aligned}$$

**leftshift2** $(r < p < q)$

Movimento definido para uma tupla de índices $(p,\ q,\ r)$, tal que $0 < r < p - 1 < p < q < n$ (os clientes $r$ e $p$ não podem ser vizinhos). Dado uma solução $S$, leftshift2$(S, p, q, r) = S^*$, e o i-ésimo cliente da solução $S^*$ é definido pela equação A.14.

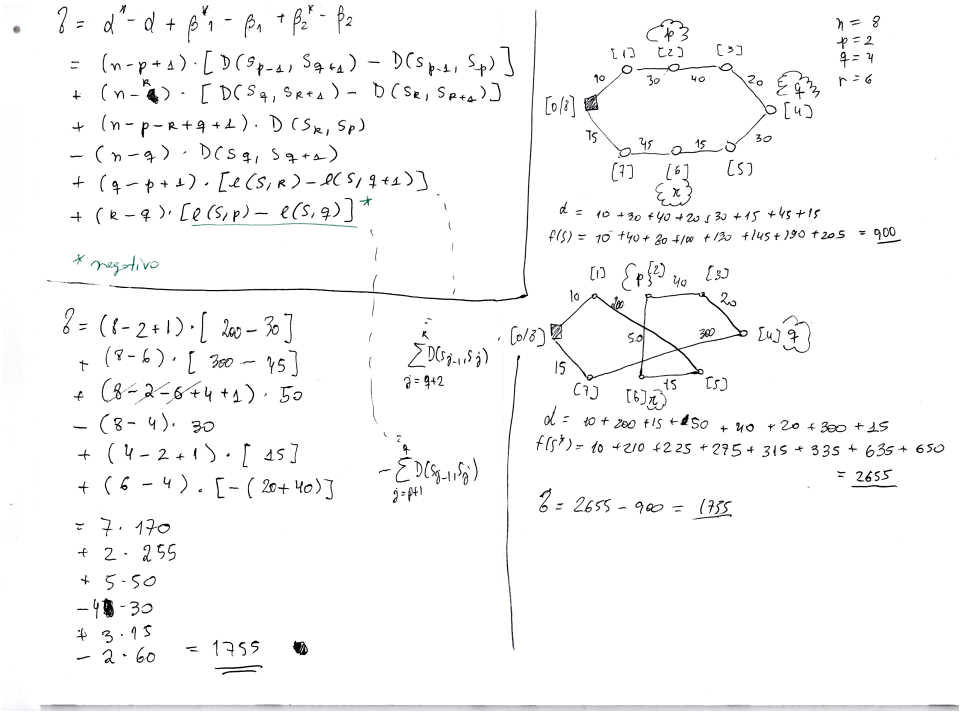

Figura A.5: Exemplo de rightshift2 para $n = 8$ com cálculos

$$s_i^* = \begin{cases} s_i, \ i < r \\ s_{\phi(i)}, \ r \leq i \leq r + q - p, \ \phi(i) = i + p - r \\ s_{\psi(i)}, \ r + q - p < i \leq q, \ \psi(i) = i + p - q - 1 \\ s_i, \ i > q \end{cases} \tag{A.14}$$

$$
\begin{aligned}
f(S^*) &= \sum_{j=1}^{n} H(S^*, j) \\
&= \cancel{\sum_{j=1}^{r-1} H(S, j)} + \sum_{j=r}^{q+1} H(S^*, j) + \cancel{\sum_{q=r+2}^{n} H(S, j)} \\
&= [H(S^*, r) + H(S^*, r+q-p+1) + H(S^*, q+1)] \leftarrow \alpha^* \\
&+ \left[ \sum_{j=r+1}^{r+q-p} H(S^*, j) \right] \leftarrow \beta_1^* \\
&+ \left[ \sum_{j=r+q-p+2}^{q} H(S^*, j) \right] \leftarrow \beta_2^*
\end{aligned}
$$

$$
\begin{aligned}
f(S) &= \sum_{j=1}^{n} H(S, j) \\
&= \cancel{\sum_{j=1}^{r-1} H(S, j)} + \sum_{j=r}^{q+1} H(S, j) + \cancel{\sum_{q=r+2}^{n} H(S, j)} \\
&= [H(S, r) + H(S, p) + H(S, q+1)] \leftarrow \alpha \\
&+ \left[ \sum_{j=r+1}^{p-1} H(S, j) \right] \leftarrow \beta_1 \\
&+ \left[ \sum_{j=p+1}^{q} H(S, j) \right] \leftarrow \beta_2
\end{aligned}
$$

$$
\begin{aligned}
\alpha = & D(s_{r-1}, s_r) \cdot (n - r + 1) \\
& + D(s_{p-1}, s_p) \cdot (n - p + 1) \\
& + D(s_q, s_{q+1}) \cdot (n - q)
\end{aligned}
$$

$$\begin{aligned}
\alpha^* &= D(s^*_{r-1}, s^*_r) \cdot (n - r + 1) \\
&+ D(s^*_{r+q-p}, s^*_{r+q-p+1}) \cdot [n - (r + q - p + 1) + 1] \\
&+ D(s^*_q, s^*_{q+1}) \cdot (n - q) \\
&= D(s_{r-1}, s_p) \cdot (n - r + 1) \\
&+ D(s_q, s_r) \cdot [n - (r + q - p + 1) + 1] \\
&+ D(s_{p-1}, s_{q+1}) \cdot (n - q)
\end{aligned}$$

$$\begin{aligned}
\beta^*_1 &= \sum_{j=r+1}^{r+q-p} H(S^*, j) \\
&= \sum_{j=r+1}^{r+q-p} D(s^*_{j-1}, s^*_j) \cdot (n - j + 1) \\
&= \sum_{j=r+1}^{r+q-p} D(s_{j+p-r-1}, s_{j+p-r}) \cdot (n - j + 1) \\
&= \sum_{j=p+1}^{q} D(s_{j-1}, s_j) \cdot [n - (j - p + r) + 1]
\end{aligned}$$

$$\begin{aligned}
\beta^*_2 &= \sum_{j=r+q-p+2}^{q} H(S^*, j) \\
&= \sum_{j=r+q-p+2}^{q} D(s^*_{j-1}, s^*_j) \cdot (n - j + 1) \\
&= \sum_{j=r+q-p+2}^{q} D(s_{j+p-q-2}, s_{j+p-q-1}) \cdot (n - j + 1) \\
&= \sum_{j=r+1}^{p-1} D(s_{j-1}, s_j) \cdot [n - (j - p + q + 1) + 1]
\end{aligned}$$

$$\begin{aligned}
\beta_1^* - \beta_1 &= \sum_{j=p+1}^{q} D(s_{j-1}, s_j) \cdot (n - (j - p + r) + 1) \\
&\quad - \sum_{j=p+1}^{q} D(s_{j-1}, s_j) \cdot (n - j + 1) \\
&= \sum_{j=p+1}^{q} D(s_{j-1}, s_j) \cdot (p - r) \\
&= [l(S, q) - l(S, p)] \cdot (p - r)
\end{aligned}$$

$$\begin{aligned}
\beta_2^* - \beta_2 &= \sum_{j=r+1}^{p-1} D(s_{j-1}, s_j) \cdot (n - (j - p + q + 1) + 1) \\
&\quad - \sum_{j=r+1}^{p-1} D(s_{j-1}, s_j) \cdot (n - j + 1) \\
&= \sum_{j=r+1}^{p-1} D(s_{j-1}, s_j) \cdot (p - q - 1) \\
&= -[l(S, p-1) - l(S, r)] \cdot (q - p + 1)
\end{aligned}$$

$$\begin{aligned}
\delta(\text{leftshift2}, S) &= \alpha^* - \alpha + \beta_1^* - \beta_1 + \beta_2^* - \beta_2 \\
&= (n - r + 1) \cdot [D(s_{r-1}, s_p) - D(s_{r-1}, s_r)] \\
&\quad + (n - q) \cdot [D(s_{p-1}, s_{q+1}) - D(s_q, s_{q+1})] \\
&\quad + (n + p - q - r) \cdot D(s_q, s_r) \\
&\quad + (p - r) \cdot [l(S, q) - l(S, p)] \\
&\quad - (q - p + 1) \cdot [l(S, p-1) - l(S, r)] \\
&\quad - (n - p + 1) \cdot D(s_{p-1}, s_p)
\end{aligned}$$

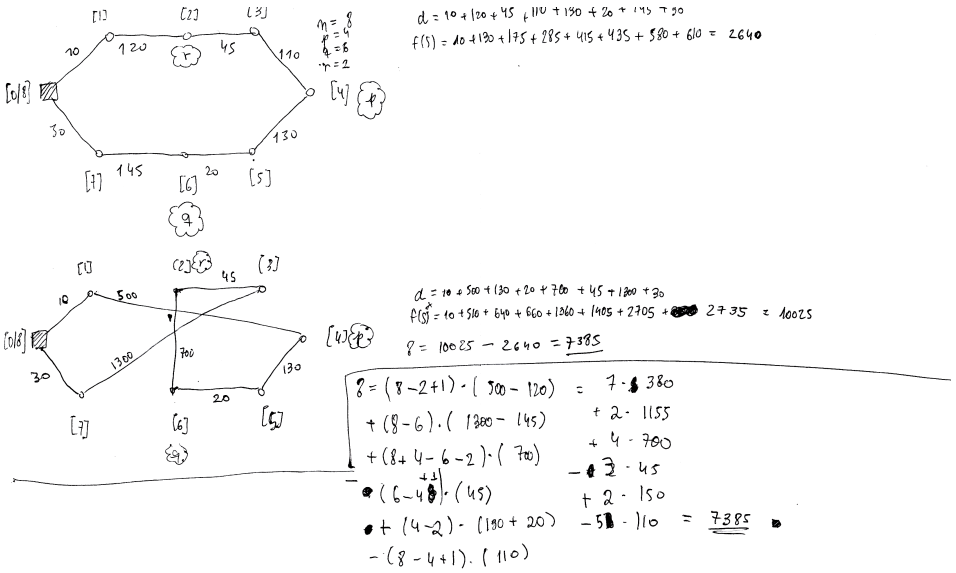


Figura A.6: Exemplo de leftshift2 para $n = 8$ com cálculos

# Apêndice B

# Otimizações

Ao implementar um algoritmo de busca local, aplicando tais movimentos de vizinhança numa solução qualquer, podemos otimizar nosso programa de algumas formas:

## B.1 Cálculo de custo

De fato, uma das operações que mais custa na busca local são os movimentos nas estruturas de dados que representam uma solução. Para que movimentos desnecessários como esse não precisem ser feitos para averiguar se um movimento é vantajoso ou não, podemos usar a metodologia de calcular o custo do movimento e averiguar apenas a partir de cálculos se o movimento será vantajoso ou não.

## B.2 Limites superiores e inferiores

Quando aplicamos um movimento $m$, se saberá de antemão se será vantajoso ou não pela fórmula do custo do movimento $\delta(m, S)$, que depende de um **conjunto contínuo de clientes**. Assim, podemos dizer que depende dos clientes no conjunto Dep, aonde $lb$ representa o limite inferior (ou *lower bound*) e $ub$ representa o limite superior (ou *upper bound*).

Se sabemos também que um movimento altera apenas um conjunto contínuo de clientes dado por Alt. Assim, quando aplicamos um movimento a uma solução, sabemos que todos os movimentos antes que não eram vantajosos continuarão a não ser vantajosos se $Dep \cap Alt = \emptyset$.

$$Dep = \{s_i \mid lb \leq i \leq ub\} \tag{B.1}$$

$$Alt = \{s_i \mid lb - 1 \leq i \leq ub + 1\} \tag{B.2}$$

## B.3 *Gamma Set*

É uma estrutura de dados que armazena os k vizinhos mais próximos de um dado nó. Ajuda a otimizar a busca local pois reduz casos esdrúxulos entre nós muito distantes, que **provavelmente** não devem ficar juntos. O parâmetro a ser ajustado é somente o k, que, se muito pequeno, pode resultar em buscas locais muito "cegas", e, se muito grandes, podem tirar o propósito desta estrutura de melhorar a performance de operações de $O(n^2)$ para $O(kn)$. O valor padrão usado no "resolvedor"é $k = 50$, mas pode ser ajustado facilmente pelo parâmetro –**gamma-k**.