BACHARELADO EM INFORMÁTICA

9904/32 - MODELAGEM E OTIMIZAÇÃO ALGORÍTMICA

PROFESSOR DANIEL KIKUTI

DISCENTES: John William Vicente RA: 118237

Nayane Batista RA: 117189

Yuri Pires Alves RA: 118792

SUMÁRIO

PARTE I

1. PROBLEMA DA MOCHILA BINÁRIA

1.1. Instâncias utilizadas

Foi utilizado as instâncias de teste para o problema da mochila encontradas no seguinte *site*: http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/. Neste endereço *web* foi encontrado diversas instâncias e os valores ótimos esperados para os algoritmos que resolvem esse problema. Essas instâncias foram divididas em duas categorias: a de pequena dimensão, *Low-dimensional*, com a quantidade de itens menores e a de larga dimensão, *Large scale*, com problemas com uma quantidade de itens mais elevada. Segue na figura 1 um exemplo da relação de arquivos e da resposta ótima encontrada no respectivo *site*.

10 test problems

file	Optimum
f1_l-d_kp_10_269	295
f2_l-d_kp_20_878	1024
f3_l-d_kp_4_20	35
f4_l-d_kp_4_11	23
f5_l-d_kp_15_375	481,0694
f6_l-d_kp_10_60	52
f7_l-d_kp_7_50	107
f8_l-d_kp_23_10000	9767
f9_l-d_kp_5_80	130
f10_l-d_kp_20_879	1025

Figura 1 - Relação de arquivos e resposta desejada

O nome da instância já traz por si mesmo algumas informações relevantes a serem consideradas, por exemplo, os últimos dois valores separados pelo '_' simbolizam respectivamente o número de itens e a capacidade da mochila naquele caso.

O formato das instâncias encontradas não eram compatíveis com a entrada exigida para o programa escrito em *ampl*, para resolver esse problema foi escrito um programa na linguagem C que lê os arquivos e gera um segundo arquivo com os padrões de entrada exigidos pelo *ampl*.

1.2. Modelagem por PLI

$$Maximizar: z = \sum_{i=1}^{n} vi\ xi$$

Sujeito a: $\sum_{i=1}^{n} wi\, xi \leq W,\ xi\ \in \{o, 1\}$ sendo wi o peso de cada item i, vi o valor associado a um item i e W a capacidade da mochila.

1.3. Comportamento do código em AMPL

Para concretizar em forma de código a modelagem proposta pela Programação Linear Inteira, foi aplicado o seguinte código em AMPL.

```
set ITEM;
param peso {ITEM};
param valor {ITEM};
param capacidade > 0;

var item {i in ITEM} binary;
maximize z: sum {i in ITEM} item[i] * valor[i];

subject to Capacidade: sum {i in ITEM} item[i] * peso[i] <= capacidade;
```

1.4. Formulação Recursiva

A formulação recursiva do problema da mochila consiste em simular, cada vez que a função *'opt'* é chamada, dois acontecimentos, a inserção caso possível de um objeto na mochila, levando em conta seu peso e a capacidade restante da mesma, ou não inserção do item na mochila e buscando realizar a inserção de algum outro item. Como a função realiza diversas chamadas recursivas ela busca priorizar as soluções que possuem maior valor em seu retorno. Dessa maneira, com a execução de todas as chamadas de *'opt'*, no final se tem o resultado ótimo do problema da mochila para uma mochila com determinada capacidade limitada levando em conta os itens que se deseja inserir.

1.5. Formulação recursiva com memoização

O problema recursivo da mochila que utiliza do recurso de memoização possui a mesma lógica de funcionamento da formulação do tópico 1.4, porém, neste outro algoritmo é armazenado os resultados dos retornos da função *'opt'* em uma matriz *'memo'*, a fim de proporcionar uma forma eficiente de recuperar valores ótimos e evitar retrabalho do algoritmo.

1.6. Implementação

Este trabalho apresenta a resolução do problema da mochila recursiva nas seguintes linguagens: C, Java, JavaScript e Python. Isso foi feito a fim de se obter uma análise do

comportamento em relação a eficiência de um mesmo algoritmo em mais de uma linguagem de programação. Nos tópicos a seguir se encontram as implementações e suas respectivas linguagens.

1.6.1. C

FORMULAÇÃO RECURSIVA:

```
int mochila(int pos, int w, int* pesos,int* valores){
    if(pos < 0 || w == 0){
        return 0;
    }

    int valor1 = mochila(pos-1, w, pesos, valores);
    if(pesos[pos] > w){
        return valor1;
    }
    int valor2 = mochila(pos-1, w-pesos[pos], pesos, valores) +
valores[pos];

    return valor1>valor2 ? valor1 : valor2;
}
```

FORMULAÇÃO RECURSIVA COM RECURSO DE MEMOIZAÇÃO:

```
int mochila(int pos, int w, int* pesos,int* valores){
    if(pos < 0 || w == 0){
        return 0;
    }

    if(memo[pos][w]!=-1){
        return memo[pos][w];
    }

    int valor1 = mochila(pos-1, w, pesos, valores);
    if(pesos[pos] > w){
        return valor1;
    }

    int valor2 = mochila(pos-1, w-pesos[pos], pesos, valores) +
valores[pos];

    memo[pos][w] = valor1>valor2 ? valor1 : valor2;
    return memo[pos][w];
}
```

1.6.2. Java

FORMULAÇÃO RECURSIVA:

```
    private int opt(int pos, int capacidade){
        if(pos < 0 || capacidade == 0){
            return 0;
        }

        numeroDeChamadas += 1;

        int optOne = opt(pos-1, capacidade);

        if(pesos[pos]>capacidade){
            return optOne;
        }

        return Math.max(optOne, opt(pos-1, capacidade-pesos[pos]) +
valores[pos]);
    }
```

FORMULAÇÃO RECURSIVA COM RECURSO DE memoização:

```
    private int opt(int pos, int capacidade){
        if(pos < 0 || capacidade == 0){
            return 0;
        }

        if(memo[pos][capacidade] != -1)
            return memo[pos][capacidade];

        numeroDeChamadas += 1;

        int optOne = opt(pos-1, capacidade);
        int ans;
        if(pesos[pos]>capacidade){
            ans = optOne;
        }else{
            ans = Math.max(optOne, opt(pos-1, capacidade-pesos[pos]) +
valores[pos]);
        }

        memo[pos][capacidade] = ans;
        return ans;
    }
```

1.6.3. JavaScript

FORMULAÇÃO RECURSIVA:

```
function opt(i, W) {
  if (i < 0 || W == 0) {
      return 0;
  }
  const optOne= opt(i - 1, W);

  if (pesos[i] > W) {
      return optOne;
  }

  return Math.max(optOne, opt(i - 1, W - pesos[i]) + valores[i]);
}
```

FORMULAÇÃO RECURSIVA COM RECURSO DE MEMOIZAÇÃO:

```
function opt(i, W) {
  if (i < 0 || W == 0) {
      return 0;
  }

  if (memo[i][W] != null) {
      return memo[i][W];
  }

  let optOne = opt(i - 1, W);

  if (pesos[i] > W) {
      return optOne;
  } else {
      optOne = Math.max(optOne, opt(i - 1, W - pesos[i]) + valores[i]);
  }

  memo[i][W] = optOne ;
  return optOne ;
}
```

1.6.4. Python

FORMULAÇÃO RECURSIVA:

```
def opt(i, w):
  if i < 0 or w == 0:
      return 0

  optOne = opt(i-1, w)
  if pesos[i] > w:
      return optOne

  return max(optOne, opt(i -1, w-pesos[i]) + valores[i])
```

FORMULAÇÃO RECURSIVA COM RECURSO DE memoização:

```
def opt(i, w):
  if i < 0 or w == 0:
      return 0

  if memo[i][w] != None:
      return memo[i][w]

  optOne = opt(i-1, w)
  if pesos[i] > w:
      ans = optOne
  else:
      ans = max(optOne, opt(i -1, w-pesos[i]) + valores[i])

  memo[i][w] = ans
  return ans
```

1.7. Comportamento do Algoritmo por Programação Dinâmica

Ao testar todas as implementações desenvolvidas para resolver o problema da mochila, o mais perceptível foi que as soluções que não usavam o recurso de memoização levaram muito mais tempo para finalizar o seu processamento.

Isso se deve ao fato de que sem uma estrutura de registros de soluções, o algoritmo tem a necessidade de ficar realizando um retrabalho para computar as soluções mesmo que essas já tenham sido calculadas. Em um caso de testes realizado com uma instância de 200 itens, o problema da mochila levou mais de 10 horas para ser computado, enquanto que com um recurso de memoização a mesma instância foi resolvida em menos de 2 minutos. Devido a isso, instâncias que possuíam uma grande quantidade de itens não foram executadas por completo nos algoritmos sem memoização, pois o tempo gasto por esses casos se tornaria inviável para os alunos responsáveis por esse trabalho.

Um outro fator notado foi que as implementações desenvolvidas sempre retornavam o valor ótimo esperado. A técnica de programação dinâmica busca analisar todas as combinações, desde que respeite algumas restrições, a fim de selecionar a mais otimizada, seja o maior ou menor valor de determinado conjunto de soluções possíveis.

1.8. Testes e Resultados

Foi realizada uma bateria de testes com algumas instâncias em todas as implementações feitas. Para a análise dos resultados foi registrado o tempo, em milisegundos,

gasto pelas máquinas para execução de cada teste. Para isso foi utilizado as seguintes máquinas:

- Máquina A: Intel® Core™ i7-1165G7 CPU @ 2.80GHz com 8 GB de RAM DDR4 que executa Windows 11;
- Máquina B: Intel® Core™ i5-4210U CPU @ 1.70GHz × 4 com 8 GB de RAM DDR3 que executa um sistema operacional baseado em Linux;
- Máquina C: Intel® Core™ i7 de 3ª geração CPU @ 2.90GHz com 8 GB de RAM DDR3 que executa macOS Catalina.

Foi feito a utilização da biblioteca de tempo das respectivas linguagens de programação usadas, deve-se destacar que por ser um valor custoso de ser medido e ainda estiver sendo executado em uma máquina com outros processos em paralelo ele pode não refletir com total confiança e exatidão o tempo de processamento dos algoritmos.

Na parte de testes envolvendo o *ampl* não foi encontrado pelos alunos uma maneira de extrair o tempo gasto nas interações de cada caso de testes. Porém, foi colocado em seu lugar o número de interações que o *solver* CPLEX utilizado levou para encontrar a solução do problema.

Dados coletados da implementação que utiliza da linguagem JavaScript:
Sem recurso de memoização*:*

Instância	Máquina A	Máquina B	Valor
f1_l-d_kp_10_269	2	1	295
f2_l-d_kp_20_878	15	25.99	1024
f4_l-d_kp_4_11	1	1	23
f8_l-d_kp_23_10000	56	135	9767
f10_l-d_kp_20_879	13	44	1025

Com recurso de memoização:

Instância	Máquina A	Máquina B	Valor
f1_l-d_kp_10_269	1	1	295
f2_l-d_kp_20_878	2	6	1024
f8_l-d_kp_23_10000	2	5	9767
f10_l-d_kp_20_879	2	4	1025
knapPI_1_200_1000_1	16	33	9147

Instância	Máquina A	Máquina B	Valor
knapPI_1_500_1000_1	96	251	28857
knapPI_1_1000_1000_1	300	868	54503

Dados coletados da implementação que utiliza da linguagem Python:

Sem recurso de memoização*:*

Instância	Máquina A	Máquina B	Valor
f1_l-d_kp_10_269	0.00099	0.00077	295
f2_l-d_kp_20_878	0.294	0.7833	1024
f4_l-d_kp_4_11	-	-	23
f8_l-d_kp_23_10000	1.5102	3.7031	9767
f10_l-d_kp_20_879	0.3060	0.8063	1025

Com recurso de memoização:

Instância	Máquina A	Máquina B	Valor
f1_l-d_kp_10_269	0.0010	0.0007	295
f2_l-d_kp_20_878	0.0029	0.0140	1024
f8_l-d_kp_23_10000	0.0050	0.0166	9767
f10_l-d_kp_20_879	0.0029	0.0098	1025

Dados coletados da implementação que utiliza da linguagem C:

Sem recurso de memoização*:*

Instância	Máquina A	Máquina B	Valor
f1_l-d_kp_10_269	0	0	295
f2_l-d_kp_20_878	6	15	1024
f4_l-d_kp_4_11	0	0	23
f8_l-d_kp_23_10000	29	105	9767
f10_l-d_kp_20_879	5	18	1025

Com recurso de memoização:

Instância	Máquina A	Máquina B	Valor
f1_l-d_kp_10_269	0	0	295
f2_l-d_kp_20_878	0	0	1024
f8_l-d_kp_23_10000	0	0	9767
f10_l-d_kp_20_879	0	0	1025
knapPI_1_500_1000_1	32	78	28857
knapPI_1_1000_1000_1	104	260	54503

Dados coletados da implementação que utiliza da linguagem Java:

Sem recurso de memoização*:*

Instância	Máquina B	Máquina C	Valor
f1_l-d_kp_10_269	1	0	295
f2_l-d_kp_20_878	25	10	1024
f4_l-d_kp_4_11	0.01	0.007	23
f8_l-d_kp_23_10000	51	38	9767
f10_l-d_kp_20_879	37	13	1025

Com recurso de memoização:

Instância	Máquina B	Máquina C	Valor
f1_l-d_kp_10_269	0.20	0.10	295
f2_l-d_kp_20_878	6	1	1024
f8_l-d_kp_23_10000	2	1	9767
f10_l-d_kp_20_879	2	0.87	1025
knapPI_1_200_1000_1	8	4	9147
knapPI_1_500_1000_1	72	24	28857
knapPI_1_1000_1000_1	213	100	54503

AMPL

Instância	Número de iterações do algoritmo	Valor
f1_l-d_kp_10_269	4	295

Instância	Número de iterações do algoritmo	Valor
f2_l-d_kp_20_878	2	1024
knapPI_1_200_1000_1	6	9147
knapPI_1_500_1000_1	7	28857
knapPI_3_200_1000_1	3	2697

1.9. Conclusão

Pode-se concluir através dos resultados, obtidos com os testes, que os códigos que buscam resolver o problema da mochila, implementando métodos recursivos para isso, são muito mais performáticos, por utilizarem de meios para evitar o retrabalho. Também pode se inferir que dado as configurações da máquina e a linguagem utilizada pode influenciar no tempo de processamento do problema.

Um outro detalhe importante que foi notado ao realizar os testes é a falta de exatidão na medida do tempo do processamento dos casos de testes. O algoritmo feito na linguagem C, por exemplo, retorna o valor 0 para muitas instâncias do problema, isso se deve tanto pela velocidade da linguagem, que é muito eficiente, quanto pela falta de uma ferramenta de medição de tempo mais eficiente. Em linguagens como o Java é possível medir a quantidade de nanosegundos, porém como essa medida de tempo é complicada de ser adquirida ela não reflete com exatidão o tempo gasto pelos algoritmos.

No *ampl*, embora não foi possível coletar as informações de tempo dos casos de testes foi perceptível pelo número de iterações do *solver* que o processo foi extremamente eficiente, pois o número de interações se manteve pequeno até para casos onde o número de itens foram maiores. Isso demonstra que a solução através é muito eficiente, sem levar em conta o tempo para realizar o tempo de uma interação.

2. PROBLEMA DO CAIXEIRO VIAJANTE

2.1. Instâncias utilizadas

Foi utilizado as instâncias de teste para o problema do caixeiro viajante encontradas no seguinte *site*: http://elib.zib.de/pub/mp-testdata/tsp/tsplib/atsp/index.html. Neste endereço *web* foi encontrado diversas instâncias para o problema. E os valores ótimos esperados para cada instância foram encontrados em outro endereço: http://elib.zib.de/pub/mp-testdata/tsp/tsplib/atsp-sol.html. Essas instâncias estão relacionadas

com o *Symmetric traveling salesman problem* que é uma variação do problema do caixeiro viajante, na qual um vértice A que possui uma aresta *r1* para B pode possuir uma aresta *r2* sendo *r1* igual a *r2* ou *r1* diferente de *r2*. Segue na figura 2 um exemplo de instância tirada do *site*:

```
NAME:  br17
TYPE: ATSP
COMMENT: 17 city problem (Repetto)
DIMENSION:  17
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
 9999    3    5   48   48    8    8    5    5    3    3    0    3    5    8    8
    5
    3 9999    3   48   48    8    8    5    5    0    0    3    0    3    8    8
    5
    5    3 9999   72   72   48   48   24   24    3    3    5    3    0   48   48
   24
   48   48   74 9999    0    6    6   12   12   48   48   48   48   74    6    6
   12
   48   48   74    0 9999    6    6   12   12   48   48   48   48   74    6    6
   12
    8    8   50    6    6 9999    0    8    8    8    8    8    8   50    0    0
    8
    8    8   50    6    6    0 9999    8    8    8    8    8    8   50    0    0
    8
    5    5   26   12   12    8    8 9999    0    5    5    5    5   26    8    8
    0
    5    5   26   12   12    8    8    0 9999    5    5    5    5   26    8    8
    0
    3    0    3   48   48    8    8    5    5 9999    0    3    0    3    8    8
    5
    3    0    3   48   48    8    8    5    5    0 9999    3    0    3    8    8
    5
    0    3    5   48   48    8    8    5    5    3    3 9999    3    5    8    8
    5
    3    0    3   48   48    8    8    5    5    0    0    3 9999    3    8    8
    5
    5    3    0   72   72   48   48   24   24    3    3    5    3 9999   48   48
   24
    8    8   50    6    6    0    0    8    8    8    8    8    8   50 9999    0
    8
    8    8   50    6    6    0    0    8    8    8    8    8    8   50    0 9999
    8
    5    5   26   12   12    8    8    0    0    5    5    5    5   26    8    8
 9999
EOF
```

Figura 2 - Exemplo de instância

Ao realizar testes com algumas instâncias foi notado que o algoritmo desenvolvido levava muito tempo para processar uma quantidade até mesmo pequena de vértices. Por isso, os membros do grupo desenvolveram algumas instâncias próprias para testes, segue logo abaixo as instâncias desenvolvidas:

```
NAME:  t66
TYPE: ATSP
COMMENT: 6 city problem
DIMENSION:  6
```

```
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
    9999       1   9999    9999    9999    9999
       5    9999   9999    9999    9999       2
    9999      90   9999       1      25    9999
       1    9999   9999    9999    9999     200
    9999    9999      1    9999    9999    9999
    9999     500   9999    9999       3    9999
EOF
```

```
NAME:  t4
TYPE: ATSP
COMMENT:  4
DIMENSION:  4
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
    9999   175   9999      2
       1  9999      5   9999
    9999     3   9999     11
      15    22      5   9999

EOF
```

```
NAME:  t2-4
TYPE: ATSP
COMMENT:  4
DIMENSION:  4
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
   9999     10     15     20
     10   9999     25     25
     15     25   9999     30
     20     25     30   9999

EOF
```

```
NAME:  t5
TYPE: ATSP
COMMENT:
DIMENSION:  5
```

```
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
   9999      90    10     30    9999
     40    9999  9999     10    9999
     50      70   9999     60     10
     10    9999   9999   9999   9999
   9999      10   9999     70   9999

EOF
```

```
NAME : a8
COMMENT :
TYPE : TSP
DIMENSION: 8
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
1 1 1
2 2 1
3 3 1
4 1 2
5 3 2
6 1 3
7 2 3
8 3 3
EOF
```

```
NAME : a17
COMMENT :
TYPE : TSP
DIMENSION: 17
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
    1  0 0
    2  0 1
    3  0 2
    4  0 4
    5  1 2
    6  1 3
    7  2 0
    8  2 4
    9  3 4
    10  4 4
```

```
    11  5 0
    12  6 1
    13  7 2
    14  7 4
    15  4 2
    16  8 3
    17  8 4 ;
EOF
```

2.2. Modelagem por PLI

Minimizar: $z = \sum_{i=1}^{n} \sum_{j=1}^{n} xij\ cij$

Sujeito a: $\sum_{i \epsilon S} \sum_{j \epsilon S} xij \leq |S| - 1, \forall S \subset V,$

$S \subset V, |S| \geq 2$

Sendo S um conjunto de arestas e V um conjunto de vértices.

2.3. Comportamento do código em AMPL

Para concretizar em forma de código a modelagem proposta pela Programação Linear Inteira, foi aplicado o seguinte código em AMPL. O código foi desenvolvido através de pesquisas e de exemplos de outros códigos escritos para a ferramenta *ampl*. Foi necessário uma instância separada de testes.

```
set VERTICES ordered;

param posicao_y {VERTICES};
param posicao_x {VERTICES};

set PARES := {i in VERTICES, j in VERTICES: ord(i) < ord(j)};

param dist {(i,j) in PARES} := sqrt((posicao_y[i]-posicao_y[j])**2+(posicao_x[i]-posicao_x[j])**2);

var X {PARES} binary;

minimize z: sum {(i, j) in PARES} dist[i, j] * X[i, j];

subject to varre_todos_vertices {i in VERTICES}:
    sum{(i, j) in PARES} X[i, j] + sum {(j, i) in PARES} X[j, i] = 2;
```

2.4. Formulação Recursiva

O algoritmo utiliza-se de uma classe auxiliar chamada *BitMask*, o objetivo dela é servir como uma lista de vértices onde é possível realizar a adição, remoção e consulta de vértices.

A ideia do algoritmo é iniciar a chamada a partir de um vértice *s* e, através de recursão, chamar a mesma função para os vértices presentes no conjunto *mask*, com exceção do vértice *s* original, para evitar ciclos, o caso base consiste quando sobrar apenas dois vértices no conjunto *mask*, onde é retornada a distância entre os dois vértices. O algoritmo retorna o menor valor possível do custo de um ciclo hamiltoniano formado pelos vértices.

Para auxiliar no desempenho do algoritmo foi criado uma estrutura de registros dos valores retornados pela função, chamada *'memo'*, essa estrutura de dados atribui a resposta encontrada pelo algoritmo com uma relação de vértice com um conjunto de vértices. Foi verificado a partir de testes empíricos que muito do tempo gasto pelo algoritmo está relacionado com a necessidade de recuperar e comparar os valores presentes dentro dos registros de *'memo'*.

2.5. Implementação

A implementação do algoritmo que soluciona o caixeiro viajante foi desenvolvido unicamente na linguagem Java. Segue o código da função principal:

FUNÇÃO CAIXEIRO VIAJANTE RECURSIVA COM MEMOIZAÇÃO:

```
    private int opt(int k, Mask mask ){

        if(mask.size() == 2&&k!=0){
            return this.dist[0][k];
        }

        Pair<Integer, Mask> p = new ImmutablePair<>(k, mask);

        if(memo.get(p)!=null){
            return memo.get(p);
        }

        int ans = Integer.MAX_VALUE;

        for(int j=0; j < this.dimension; j++ ){
            if(mask.contain(j)&&j!=k&&j!=0){

                ans = Math.min(ans,
                        opt(j, mask.remove(k))+dist[j][k]);
            }
        }
```

```
        memo.put(p, ans);

        return ans;
    }
```

```
public class BitMask implements Mask {

    private int mask;

    public BitMask(int mask) {
        this.mask = mask;
    }

    @Override
    public int size() {
        int count = 0;
        int copy = mask;
        while (copy > 0) {
            count += copy & 1;
            copy >>= 1;
        }
        return count;
    }

    @Override
    public boolean contain(int n) {
        if(n==0&&mask==0){
            return true;
        }
        int b = mask & (1 << n);
        return b>0;
    }

    @Override
    public Mask remove(int n) {
        return new BitMask(mask & (~(1 << n)));
    }

}
```

2.5.1. Comportamento do Algoritmo por Programação Dinâmica

O algoritmo do caixeiro viajante verificou-se como sendo muito custoso para o processamento de algumas instâncias, mesmo com um número de vértices não muito alto. Isso se deve pelo número de chamadas que cada interação na função *'opt'* realiza, que é n - 1 sendo n o número de elementos do conjunto.

Um outro fator que dificultou o desempenho do algoritmo foi o funcionamento do recurso de memoização que se utilizava de um mapa onde associava um *index* com um conjunto. Possuir tal associação é muito cara para se manter devido ao número de interações que o algoritmo realiza e o número de instâncias de conjuntos que são criadas e que tem que ser comparadas com outras instâncias. Por exemplo, ao implementar a classe *Mask* utilizamos em uma primeira versão as coleções disponibilizadas pela própria linguagem. Porém, isso logo se viu inviável, pois além do processo se tornar mais lento também ocupou mais espaço, pois muitos objetos são instanciados no decorrer das chamadas recursivas. Dessa maneira, trocamos a utilização das coleções do Java por uma técnica chamada *BitMask*, que consegue representar um conjunto através dos *bits* de um número inteiro, por exemplo os *bits* 1011 representam um conjunto {0, 1, 3}. A utilização do *BitMask* limitou o número de vértices que o algoritmo consegue gerenciar, porém facilitou o teste e a confirmação que o algoritmo estava correto.

2.6. Testes e Resultados

Segue a tabela com algumas instâncias usadas para o teste do algoritmo recursivo:

Instância	Máquina B	Máquina C	Valor
t66	2	1	9
t4	0.87	0.75	11
t2-4	2	1	80
t5	1	1	50
b17	2352	1328	39

Segue as tabela com algumas instâncias usadas para o teste do algoritmo em AMPL:

Instância	Número de iterações do algoritmo	Máquina C
a8	2	1
a17	21	0.75

2.7. Conclusão

Por meio dos testes realizados e das dificuldades ao implementar o algoritmo, o que fica mais evidente é que o algoritmo do caixeiro viajante utilizando programação dinâmica se configura como sendo bem custoso, consumindo bastante espaço para a técnica de memoização e realizando muitas operações recursivas. Em grafos com poucos vértices esses detalhes podem passar despercebidos, porém em grafos com um número um pouco maior isso já é visível, como no teste realizado com um grafo de 17 vértices já levou um tempo considerável para a execução.

PARTE II

3. HEURÍSTICAS COM O PROBLEMA DA MOCHILA BINÁRIA

3.1. Heurística de Construção

Na heurística de construção foi utilizado o valor relativo dos itens que podem ser colocados dentro da mochila. Os itens foram ordenados em ordem decrescente em relação a esse valor, depois foram adicionados um a um respeitando a capacidade da mochila, para cada item adicionado o valor da capacidade disponível da mochila é atualizado. Quando a mochila não consegue adicionar mais itens se tem uma solução válida que pode ser melhorada por outro algoritmo.

3.2. Heurística de Melhoramento

Para melhorar a solução encontrada pela heurística de melhoramento foi utilizado a seguinte lógica: retira um elemento de uma posição da mochila; depois tenta inserir os elementos com menor peso que estão fora da mochila; se não for possível inserir nenhum elemento se retira outro item da mochila; quando é possível inserir novos itens na mochila é calculado o valor da solução e comparado com o valor da solução anterior; no final prevalece a solução com o maior valor possível. Dessa maneira é possível encontrar uma espécie de solução máxima local a partir da solução do algoritmo construtivo.

3.3. Implementação

Segue a implementação do algoritmo heurístico de construção e melhoramento para o problema da mochila:

```
private int opt(){

        ValorObjetoRelativo[] valorRelativo = new
ValorObjetoRelativo[valores.length];

        for(int i=0 ;i<valores.length; i++){
            valorRelativo[i] = new ValorObjetoRelativo(pesos[i],
valores[i]);
        }

        Arrays.sort(valorRelativo, Collections.reverseOrder());

        List<ValorObjetoRelativo> objetosSelecionados = new
ArrayList<>();

        int pesoObjetos = 0, cont = 0;

        while (pesoObjetos<this.capacidade&&cont<this.qtd){

if(valorRelativo[cont].getPeso()+pesoObjetos<=this.capacidade){
                pesoObjetos += valorRelativo[cont].getPeso();
                objetosSelecionados.add(valorRelativo[cont]);
            }
            cont++;
        }

        Set<ValorObjetoRelativo> collect =
Arrays.stream(valorRelativo)
                .collect(Collectors.toCollection(
                        () -> new
TreeSet<>(Comparator.comparing(ValorObjetoRelativo::getPeso))
                ));

        collect.removeAll(objetosSelecionados);

        objetosSelecionados = melhoramento(collect,
objetosSelecionados);


        return getAns(objetosSelecionados);
    }

    private List<ValorObjetoRelativo> melhoramento(
Set<ValorObjetoRelativo> objetosForaDaMochila,
```

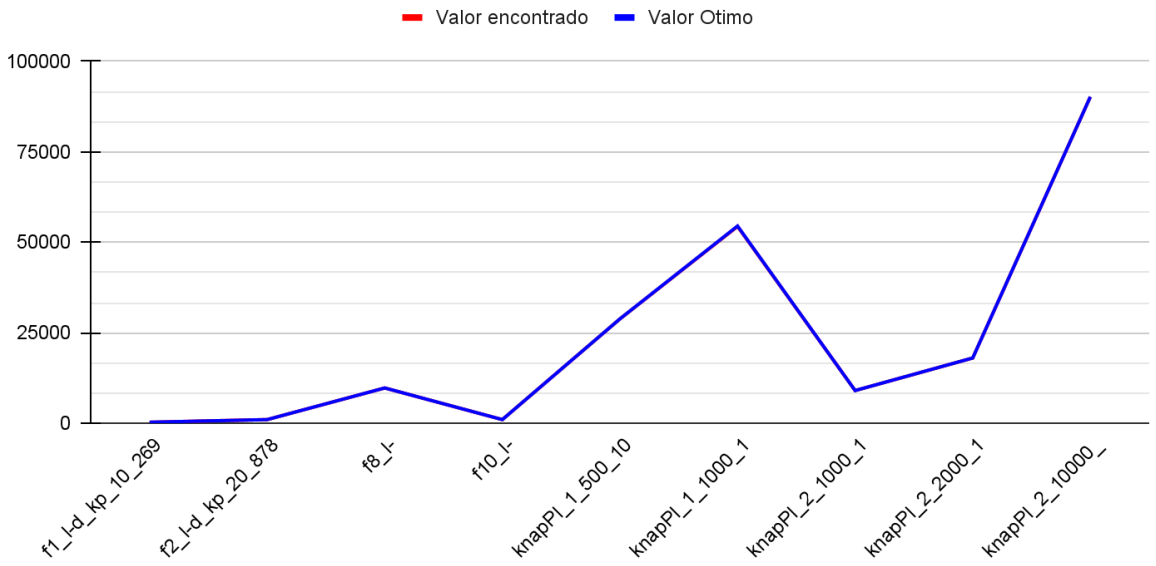
```
List<ValorObjetoRelativo> objetosSelecionados){
        int totalItens =
objetosForaDaMochila.size()+objetosSelecionados.size();

        int total = getAns(objetosSelecionados);
        List<ValorObjetoRelativo> novaResposta = new
LinkedList<>(objetosSelecionados);

        int pesoMochila = getSomaPeso(novaResposta);

        Collections.sort(novaResposta,
Collections.reverseOrder());
        objetosForaDaMochila.removeAll(novaResposta);

        List<ValorObjetoRelativo> itensExcluidosForaDaMochila =
new ArrayList<>();

        int cont = 0, i = 0;

        while (cont<novaResposta.size()){

            ValorObjetoRelativo valorObjetoRelativo =
novaResposta.get(novaResposta.size() - (cont+1));
            pesoMochila -= valorObjetoRelativo.getPeso();
            novaResposta.remove(valorObjetoRelativo);

            while (!objetosForaDaMochila.isEmpty()){
                ValorObjetoRelativo objeto =
objetosForaDaMochila.stream().findFirst().get();

                if( objeto.getPeso()+pesoMochila>this.capacidade){
                    break;
                }

                novaResposta.add(objeto);
                pesoMochila += objeto.getPeso();
                objetosForaDaMochila.remove(objeto);
                itensExcluidosForaDaMochila.add(objeto);
            }


if(getAns(novaResposta)>total&&getSomaPeso(novaResposta)<=this.cap
acidade){
                total = getAns(novaResposta);
```

```
                objetosSelecionados = new
ArrayList<>(novaResposta);
                objetosForaDaMochila.add(valorObjetoRelativo);
                cont = 0;
            }else{
                if(itensExcluidosForaDaMochila.isEmpty()){
                    continue;
                }
                novaResposta = new
LinkedList<>(objetosSelecionados);
                Collections.sort(novaResposta,
Collections.reverseOrder());

objetosForaDaMochila.addAll(itensExcluidosForaDaMochila);
                cont++;
            }
        }
        return objetosSelecionados;
    }
    private int getAns(List<ValorObjetoRelativo>
objetosSelecionados){
        return objetosSelecionados.stream()
                .map(ValorObjetoRelativo::getValor)
                .reduce(Integer::sum)
                .get();
    }

    private int getSomaPeso(List<ValorObjetoRelativo>
objetosSelecionados){
        return objetosSelecionados.stream()
                .map(ValorObjetoRelativo::getPeso)
                .reduce(Integer::sum)
                .get();
    }
```

3.4. Testes a partir das instâncias selecionadas anteriormente

Instância	Máquina A	Máquina B	Valor encontrado	Valor
f1_l-d_kp_10_269	15	17	295	295
f2_l-d_kp_20_878	17	17	1024	1024

Instância	Máquina A	Máquina B	Valor encontrado	Valor
f8_l-d_kp_23_10000	16	36	9759	9767
f10_l-d_kp_20_879	16	19	1025	1025
knapPI_1_500_1000_1	22	103	28834	28857
knapPI_1_1000_1000_1	31	71	54386	54503
knapPI_2_1000_1000_1	31	55	9046	9052
knapPI_2_2000_1000_1	51	73	18038	18051
knapPI_2_10000_1000_1	156	ERRO	90200	90204



3.5. Resultados e Conclusão

Ao observar os testes é possível concluir que a soma do algoritmo de construção e de melhoramento trouxe boas respostas para instâncias submetidas, pois ficaram bem próximas das respostas ótimas. Como pode ser observado no gráfico acima, os valores encontrados pelo algoritmo ficaram muito perto dos valores ótimos, quase que não é possível ver a outra linha, que é uma ótima solução, isso diz que o método construtivo montou boas soluções para o algoritmo de melhoramento.

Uma instância em particular causou uma exceção no programa, porém apenas em uma máquina, isso deve ter sido causado por alguma configuração particular.

4. HEURÍSTICAS COM O PROBLEMA DO CAIXEIRO VIAJANTE

4.1. Heurísticas de Construção e Melhoramento

O algoritmo selecionado para a construção de uma solução inicial para o problema do caixeiro viajante foi a da inserção do mais barato. Este algoritmo parte de um início com três vértices, que foram selecionados a partir da sua distância com o vértice de origem s. A partir desse momento o algoritmo busca inserir o elemento com menor custo no conjunto solução, para isso ele faz o cálculo de ligar dois vértices no novo, que irá entrar, e de desligar o vértice já existente. O algoritmo sempre insere um vértice entre outros dois vértices que já foram selecionados. No final é apenas necessário inserir o vértice s, o de origem, no final da solução, para formar um ciclo hamiltoniano.

Para o algoritmo de melhoramento foi utilizado o algoritmo *2-opt*, que consiste em desligar dois vértices de um conjunto e realizar a troca de posições destes e depois calcular se houve melhora ou não. O algoritmo sempre começa a partir de uma posição *i* de um vértice que será trocado por outro em seguida, se não houver melhoria o algoritmo seleciona um próximo vértice para ser trocado, os vértices que estavam entre o vértice i e o vértice candidato para troca tem sua ordem invertida. A partir do momento que ao realizar trocas dos vértices nenhuma solução melhor é encontrada, esta última é retornada como resposta do problema.

4.2. Implementação

Segue a implementação do algoritmo de construção e melhoramento:

```
private  List<Integer> opt(int s){
        int pontoVerificar = s;
        List<Integer> rotaMinimizada = new ArrayList<>();
        rotaMinimizada.add(s);

        int menor = Integer.MAX_VALUE;
        int pontoMenor;

        pontoMenor = pontoMaisProximo(s, rotaMinimizada);
        rotaMinimizada.add(pontoMenor);

        pontoMenor = pontoMaisProximo(s, rotaMinimizada);
```

```
        rotaMinimizada.add(pontoMenor);

        int tamanhoRotaMinimizada = this.dimension -2;

        while(tamanhoRotaMinimizada>0){
            int custo = Integer.MAX_VALUE;
            int pontoInserir = 0;

            for (int i=0;i<this.dimension;i++){
                if(rotaMinimizada.contains(i)){
                    continue;
                }

                for(int j = 0; j<rotaMinimizada.size()-1; j++ ){
                    int k1 = rotaMinimizada.get(j);
                    int k2 = rotaMinimizada.get(j+1);
                    int custoNovaRota = dist[k1][i] + dist[k2][i] -
dist[k1][k2];

                    if(custoNovaRota<custo){
                        custo = custoNovaRota;
                        pontoInserir = i;
                        pontoMenor = k1;
                    }
                }

            }

            if(custo!=Integer.MAX_VALUE){
                inserirPontoApos(pontoInserir, rotaMinimizada,
pontoMenor);
            }

            tamanhoRotaMinimizada-=1;
        }

        rotaMinimizada.add(s);

        System.out.println(rotaMinimizada);

        rotaMinimizada = melhoramento2Opt(rotaMinimizada);


        return rotaMinimizada;
    }

    private int pontoMaisProximo(int s, List<Integer> listaPontos){
        int menor = Integer.MAX_VALUE;
        int pontoMaisProximo = 0;
```

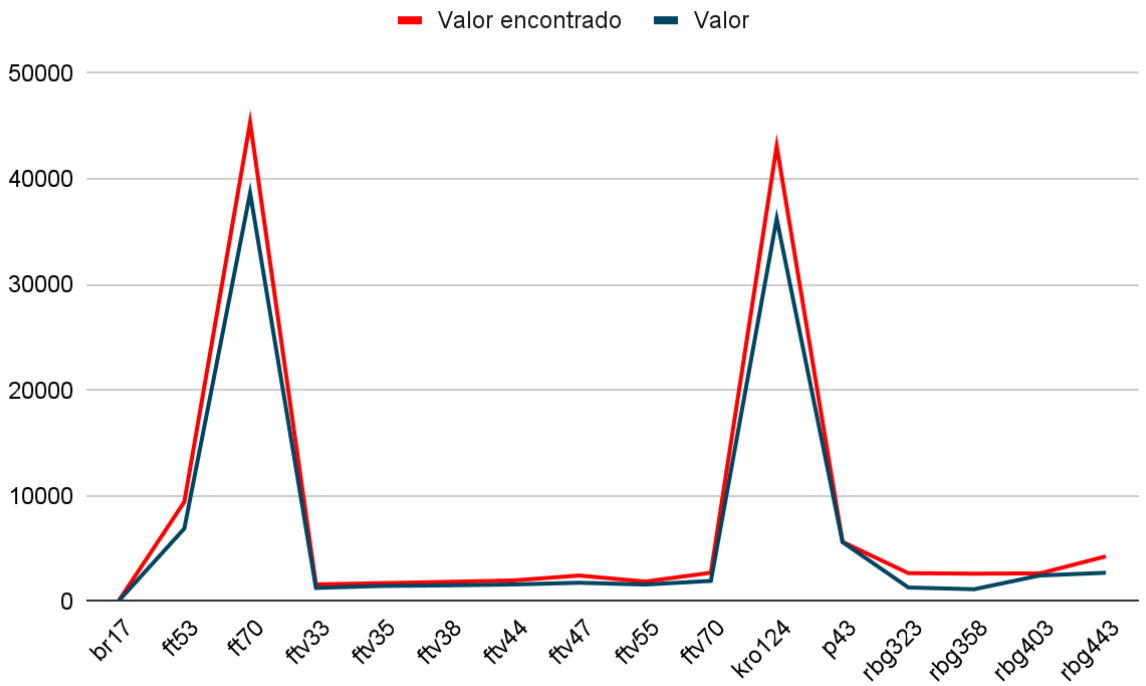
```
        for (int i=0;i<this.dimension;i++){
            if(listaPontos.contains(i)){
                continue;
            }
            if(dist[s][i]<menor) {
                menor = dist[s][i];
                pontoMaisProximo = i;
            }
        }


        return pontoMaisProximo;
    }

    private void inserirPontoApos(int pontoInserir, List<Integer>
rotaMinimizada, int pontoMenor){
        int lastIndexOf = rotaMinimizada.lastIndexOf(pontoMenor);
        lastIndexOf++;
        rotaMinimizada.add(lastIndexOf, pontoInserir);
    }

    private List<Integer> melhoramento2Opt(List<Integer> rota){

        int custoMelhorRota = custoRota(rota);
        boolean encontrouMelhorRota = true;
        int tamanhoRota = rota.size();

        while(encontrouMelhorRota){
            encontrouMelhorRota = false;

            for(int i=1; i< tamanhoRota - 1;i++ ){
                for(int j = i+2; j<rota.size()-1;j++){
                    List<Integer> novaRota = opt2Swap(i, j, rota);
                    int custNovaRota = custoRota(novaRota);
                    if(custNovaRota<custoMelhorRota){
                        rota = novaRota;
                        custoMelhorRota = custNovaRota;
                        encontrouMelhorRota = true;
                    }

                }
            }
        }

        return rota;
    }

    private List<Integer> opt2Swap(int i, int j, List<Integer> rota){
```

```
        List<Integer> parteInicial = rota.subList(0, i+1);

        List<Integer> subRota = new ArrayList<>(rota.subList(i + 1, j +
1));
        Collections.reverse(subRota);

        List<Integer> rotaFinal = rota.subList(j+1, rota.size());

        List<Integer> novaRota =  new ArrayList<>();
        novaRota.addAll(parteInicial);
        novaRota.addAll(subRota);
        novaRota.addAll(rotaFinal);

        return novaRota;
    }

    private int custoRota(List<Integer> rota){
        int custo = 0;

        for(int i = 0; i<rota.size()-1;i++){
            int v1 = rota.get(i);
            int v2 = rota.get(i+1);
            custo += dist[v1][v2];
        }

        return custo;
    }
```

4.3. Testes a partir das instâncias selecionadas anteriormente

Instância	Máquina C	Máquina B	Valor encontrado	Valor
br17	6	10	39	39
ft53	ERRO	35	9426	6905
ft70	48	93	45358	38673
ftv33	14	18	1619	1286
ftv35	18	16	1736	1473
ftv38	21	64	1862	1530
ftv44	20	23	1995	1613
ftv47	19	26	2458	1776
ftv55	25	38	1889	1608

ftv70	41	77	2723	1950
kro124	81	133	43106	36230
p43	20	36	5641	5620
rbg323	ERRO	747	2684	1326
rbg358	701	944	2635	1163
rbg403	ERRO	3710	2666	2465
rbg443	ERRO	1982	4271	2720



4.4. Resultados e Conclusão

Pode-se concluir que através das heurísticas usadas é possível ampliar as instâncias passíveis de testes, podemos inserir instâncias com muitos vértices e ter uma resposta em um tempo muito menor do que no algoritmo com programação dinâmica, que era inviável. Observando os valores obtidos e o gráfico gerado é observável que os algoritmos respeitam o seu propósito, eles retornam valores maiores porém perto da solução ótima. Quando comparamos com os resultados das heurísticas feitas no problema da mochila, vemos que o do caixeiro viajante não demonstrou tamanha eficiência em chegar perto da solução ótima, quanto aquele outro.

Algumas instâncias geraram erros ao ser executadas, porém apenas em uma máquina, isso deve ter sido causado por alguma configuração particular.