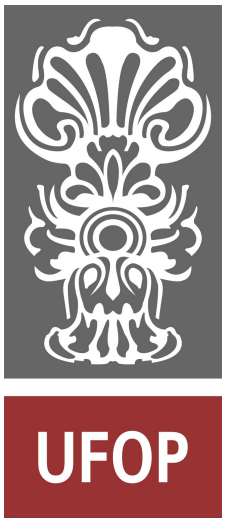

UNIVERSIDADE FEDERAL DE OURO PRETO

Mestrado em Ciência da Computação

Projeto e Análise de Algoritmos – Problema Problema da Alocação Generalizada (GAP) com uso de Algoritmos Heurísticos

Autor:
Rodolfo Labiapari Mansur Guimarães - rodolfolabiapari@gmail.com

Professor Orientador:
Haroldo Gambini Santos - haroldo@iceb.ufop.br

Ouro Preto - MG
7 de agosto de 2016

Sumário

1 Introdução

1.1 Problema da Alocação Generalizada

Neste capítulo, faz-se descrições do Problema de Alocação Generalizada (do inglês, *Generalizated Assingment Problem*, GAP) em sua forma clássica.

GAP é um problema de alocação de recursos, denominado tarefas, para determinados agentes, alocadores, com propósito de custo mínimo.

$$\min \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij}$$

Os elementos básicos da fórmula pra este problema são:

- Um conjunto I de agentes ($i = 1, 2, 3, \ldots, m$);
- Um conjunto J de agentes ($j = 1, 2, 3, \ldots, n$);

Cada tarefa $T_j \in T$ consome uma quantidade de recursos a_{ij} do agente $i \in I$, ou seja, consome uma parte da capacidade do agente, a um diferente custo c_{ij}.

Este problema deve atender a três restrições:

1. Cada agente tem uma capacidade limitada;
2. Cada tarefa só pode ser alocada a um único agente;
3. Todas as tarefas devem ser alocadas;

A solução para o GAP é um vetor de n elementos, onde a k-ésima posição do vetor guarda o agente ao qual a k-ésima tarefa foi associada.

Dasgupta explica [1]:

> *"Existe um número de agentes e um número de tarefas. Podemos alocar qualquer agente a qualquer uma das tarefas, porém, cada alocação tem um custo que pode variar dependendo da tarefa e agente específicos. É necessário que todas as tarefas sejam feitas, designando exatamente um agente para cada tarefa de modo que o custo total (a soma) de todas as alocações seja minimizado."*

1.2 Heurística em Algoritmos

As heurísticas (ou modelos heurísticos) não garantem que seja encontrado o ótimo. Geralmente encontram soluções próximas à ótima, mas algumas vezes, dependendo das circunstâncias, é possível que sejam obtidos resultados arbitrariamente ruins (ou também o resultado ótimo). Isso denota uma relação de custo benefício. Ao utilizarmos este tipo de método, não precisamos da melhor solução possível, mas geralmente obtemos uma solução viável com um tempo computacional aceitável.

Dentro da categoria das heurísticas estão alguns métodos caracterizados como metaheurísticas. São definidos como uma metaheurística os métodos que otimizam um problema ao iterativamente buscar a melhora de uma canditada à solução, utilizando alguma medida de qualidade. São métodos de otimização estocástica, ou seja,

algoritmos e técnicas que utilizam alguma forma de aleatoriedade para encontrar uma solução ótima ou próxima à ótima em problemas difíceis de serem resolvidos.

Metaheurísticas são utilizadas para encontrar soluções para problemas para os quais não se tem muitas informações. Não se sabe qual deve ser a solução ótima e o espaço de busca é vasto. Mas, se existe uma candidata à solução do problema, é possível testá-la e definir o quão boa ela é. Isto é verdade para muitos problemas de Otimização Combinatória, o que leva uma metaheurística a ser bastante utilizada para resolver problemas da área.

2 Decisões Iniciais de Projeto

2.1 Modularização de Funções em Arquivos

Primeiramente, como se trata de um único problema abordado em vários algoritmos, percebeu-se que poderia desenvolver uma única estrutura de dados para para todos os tipos de algoritmos a serem implementados além de funções genéricas do problema que seriam comuns entre alguns aloritoms. Com isso, decidiu-se inicialmente serparar a estrutura de dados e algumas funções comumente utilizadas em um arquivo de funções incluído à cada algoritmo.

A modularização deste trabalho trouxe várias melhorias como portabilidade do problema, além de boa refatoração do código de modo que, realizando uma alteração no arquivo de funções, todos os algoritmos serão alterados automaticamente economizando tempo e reduzindo a quantidade de linhas de código por algoritmo.

2.2 Estrutura de Dados

Utilizou-se de uma única estrutura de dados que representaria o problema no qual permitiu-se ser aplicada em todos os quartos algoritmos implementados situada no arquivo de cabeçalho. Como é uma estrutura geral, será descrita previamente.

A estrutura segue abaixo:

```
1 typedef struct Struct_Solucao {
2    int * excesso;
3    int * tarefas;
4    double avaliacao;
5    double custo;
6 } Solucao;
```

A estrutura é formada por duas variáveis e dois vetores que representam as informações de cada solução. O vetor *tarefas* armazena o agente responsável por realizar a tarefa de índice i e consequentemente, o vetor excesso armazena a quantidade de recursos utilizada pelo agente i. Este vetor de excessos é comparado com o vetor de capacidade de cada agente verificando se alguma posição ultrapassou a quantidade máxima de recursos que determinado agente pode exercer.

Os dois vetores são necessários para calcular as duas outras variáveis que são avaliação e custo. Custo é a soma de todos os custos de cada agente em cada tarefa

e avaliação é o quão boa é esta solução em relação à suas características. A avaliação destes serão descritos no decorrer do documento.

2.3 Geração de Vizinhos Otimizada

Após entender como funciona a estrutura de dados do problema, deve-se entender como é feita a geração de vizinhos já que este procedimento é utilizado em três dos quatros algoritmos implementados e todos eles utilizam o mesmo procedimento sem alterações.

Deve-se atentar ao detalhe que esta função é otimizada ao ponto de avaliar a solução gerada sem invocar o procedimento `Avalia_Solucao(Solucao)`. O procedimento é descrito à seguir.

```
1  void Gera_Vizinho(Solucao * atual, Solucao ** proxima) {
2     int i = 0, j = 0, agente_atual = 0, agente_novo = 0,
3     tarefa_escolhida1 = 0, sum_recursos = 0,
4     quant_alteracoes = 0;
5     char solucao_invalida = 0;
6
7     // Copia os valores do atual
8     for (i = 0; i < QUANT_TAREFAS; i++) {
9        (*proxima)->tarefas[i] = atual->tarefas[i];
10       if (i < QUANT_AGENTES)
11          (*proxima)->excesso[i] = atual->excesso[i];
12    }
13
14    (*proxima)->custo = atual->custo;
15
16
17    // Define uma quantidade de alterações pra gerar o vizinho
18    quant_alteracoes = 2;
19
20    // Altera o indivíduo
21    for (i = 0; i < quant_alteracoes; i++) {
22
23       // Escolhe a tarefa que será alterada
24       tarefa_escolhida1 = random() % QUANT_TAREFAS;
25       agente_atual = (*proxima)->tarefas[tarefa_escolhida1];
26
27       // Gera um novo agente pra ela e certifica que ele é diferente do
         ↪  anterior.
28       do {
29          agente_novo = random() % QUANT_AGENTES;
30       } while (agente_novo == agente_atual);
31
32       // Atribui o novo agente à tarefa
33       (*proxima)->tarefas[tarefa_escolhida1] = agente_novo;
```

```
34
35          // Procedimento Otimizado:
36             // A cada alteração, realiza-se a alteração dos valores da nova
                ↪  geração.
37             // Como este novo vizinho não é feito do zero e sim sobre cópia de um
                ↪  anterior,
38             // basta alterar os valores herdados do seu anterior, atualiando em
                ↪  O(1)
39
40          // Sendo assim, atualiza o excesso de cada agente
41             // Retira recurso do agente que ficou livre
42          (*proxima)->excesso[agente_atual] -=
             ↪  RECURSOS_A_T[agente_atual][tarefa_escolhida1];
43             // Acrescenta recurso do agente que recebeu a tarefa atual
44          (*proxima)->excesso[agente_novo ] += RECURSOS_A_T[agente_novo
             ↪  ][tarefa_escolhida1];
45
46          // Calcula o custo atual desta solução
47          (*proxima)->custo += CUSTO_A_T[agente_novo ][tarefa_escolhida1] -
             ↪  CUSTO_A_T[agente_atual][tarefa_escolhida1];
48       }
49
50       // Verifica se a solução gerada é válida
51       for (j = 0; j < QUANT_AGENTES; j++) {
52          sum_recursos += (*proxima)->excesso[j];
53
54          if ((*proxima)->excesso[j] > CAPAC_AGENTES[j]) {
55             solucao_invalida = 1;
56          }
57       }
58
59       // Calcula o fator avaliação
60       if (solucao_invalida)
61          (*proxima)->avaliacao = ((double) sum_recursos ) * 1000000;
62       else {
63          (*proxima)->avaliacao = ((double)  (*proxima)->custo);
64       }
65    }
```

A geração é um procedimento simples que realiza a alteração de apenas dois itens dentro da solução gerando uma nova solução com valores diferentes da original e sem afastar do local do espaço de solução. Assim, primeiro é copiado todos os dados da solução original e depois realizado duas alterações em qualquer uma das tarefas da solução alterando o agente delas.

Recapitulando, a cada alteração da tarefa todos suas propriedades são recalculadas em tempo constante ao contrário da invocação do método `Avalia_Solucao(Solucao)` que executaria em O(n). A vantagem de executar o método de forma mais veloz

proporciona que o algoritmo ao todo possa executar mais tarefas sobre o mesmo intervalo de tempo e assim, aumentando a probabilidade de encontrar uma solução melhor ainda.

A forma de avaliação propriamente dita é descrita a seguir.

2.4 Função Avaliação

```
1  double Avalia_Solucao(Solucao * sol) {
2     int i = 0, capacidade_agentes[QUANT_AGENTES], sum_recursos = 0;
3     double custo = 0; char solucao_invalida = 0;
4
5     // Define a capacidade inicial utilizada de cada agente com 0
6     for (i = 0; i < QUANT_AGENTES; i++)
7        capacidade_agentes[i] = 0;
8
9     // Realiza os cálculos de custo e capacidade
10    for (i = 0; i < QUANT_TAREFAS; i++) {
11       custo += CUSTO_A_T[sol->tarefas[i]][i];
12       capacidade_agentes[sol->tarefas[i]] += RECURSOS_A_T[sol->tarefas[i]][i];
13    }
14
15    // Verifica se algum agente passou sua capacidade máxima
16    for (i = 0; i < QUANT_AGENTES; i++) {
17       sol->excesso[i] = capacidade_agentes[i];
18       sum_recursos += capacidade_agentes[i];
19
20       // Se sim define esta solução como inválida
21       if (capacidade_agentes[i] > CAPAC_AGENTES[i])
22          solucao_invalida = 1;
23    }
24
25    sol->custo = custo;
26
27    // Caso a solução foi excedida, altera a avaliação do indivíduo tornando-o
28       // pior.
29    if (solucao_invalida)
30       sol->avaliacao = ((double) sum_recursos ) * 1000000;
31    else {
32       sol->avaliacao = ((double) custo);
33    }
34
35    return sol->avaliacao;
36 }
```

O procedimento inicia calculando a quantidade de recursos utilizado por cada agente além do custo total da solução independente dela ser uma solução válida ou não.

Após calculado, é feito uma verificação (linha 16-23) onde analisa se pelo menos um agente ultrapassou sua quantidade de recursos limite, tornando a solução inválida. Após analisado, é realizado duas tomadas de decisão já suponto o problema de minimização, sendo elas quando a solução for:

- **Inválida:** O valor da avalização será calculado de forma diferente da solução válida. Foi desenvolvido um novo método para conseguir gerar soluções iniciais válidas de forma mais rápida no algoritmo a fim de permitir que ele trabalhe na maior parte do tempo com soluções válidas do que simplesmente procurando soluções. A fórmula desenvolvida para tal é $quantidade\_recurso\_utilizado * 1000000$.

  É possível perceber que o fator custo não entra neste cálculo. Isso deve ao fato de soluções inválidas serem dependente somente do fator de *recurso* já que, diferente do fator *custo* no qual queremos otimizar, o fator *recurso* é uma restrição. Se esta restrição não for satisfeita, então todo o resto é inválido, justificando assim o uso desta fórmula de avalização.

  Por fim, existe a penalização de soluções inválidas. Como o procedimento é de minimização, multiplicou-se esse valor por um milhão tornando esta solução totalmente inválida;

- **Válida:** O valor da avaliação será obtido pela fórmula *custo*. Uma vez obtido uma solução válida (que satisfaz todas as restrições), não queremos minimizar as restrições mas sim o seu fator *custo*. Sendo assim, a fórmula de avaliação de soluções válidas é somente o valor de seu custo válido, com intuito da minimização ser mais clara e objetiva possível.

3 Algoritmo Genético

3.1 Execução

O algoritmo implementado necessita somente de duas populações instanciadas pra realizar suas operações. Isso pois elas comutação entre si na realização do papel principal de *população atual* sendo uma servindo de *buffer* da outra a cada iteração.

Assim, é instanciadas duas populações. Os novos filhos gerados e mutados serão postos na segunda população e em seguida será completada com indivíduos da primeira de forma aleatória. Esta estratégia fio desenvolvida para reduzir chamadas de sistema para alocação e liberação de memória.

Abaixo serão descrito alguns procedimentos fundamentais do Algoritmo Genético.

3.1.1 Criando Uma População Inicial

O processo de geração de população inicial funciona de duas formas. A primeira utiliza escolha aleatória de agentes e atribuindo a tarefa i. Já a segunda é uma forma gulosa de preenchimento de tarefas. Para a tarefa i é vasculhado todos os agentes procurando o que possui o que gasta o menor recurso pra executar esta tarefa e assim o é escolhido.

O primeiro indivíduo escolhido é gerado totalmente pelo procedimento guloso. Os demais são uma mescla de guloso com aleatoriedade onde o algoritmo aleatório possui 66% de chance de ser escolhido para determinar a tarefa i, restando 33% para o guloso.

O algoritmo é descrito a seguir.

```
1  void Cria_Nova_Populacao(Individuo *** P) {
2     int i = 0, j = 0, k = 0, menor = 0;
3     Individuo ** p_local = 0;
4
5     p_local = *P;
6
7     // Para cada item a ser criado
8     for (i = 0; i < TAM_POP; i++) {
9
10       // Aloca suas variáveis que armazenarão suas informações
11       p_local[i]           = calloc(1, sizeof(Individuo));
12       p_local[i]->excesso = calloc(QUANT_AGENTES, sizeof(int));
13       p_local[i]->tarefas = calloc(QUANT_TAREFAS, sizeof(int));
14
15       // Gera valores pra este indivíduo
16       for (j = 0; j < QUANT_TAREFAS; j++) {
17          // O primeiro indivíduo será gerado de forma gulosa e os outros
18             // Serão uma mistura de Guloso com Aleatoriedade
19
20          // Se não for o primeiro indivíduo, possui 66% de gerar valores
21             // por meio de função randomica
22          if (i > 0 && random() % 3 != 0) {
23             p_local[i]->tarefas[j] = random() % QUANT_AGENTES;
24
25          // Caso contrário, utiliza uma geração gulosa pra esta tarefa.
26          } else {
27             // O método guloso escolhe o recurso mais leve desta tarefa
28             p_local[i]->tarefas[j] = 0;
29             menor = 0;
30
31             // Seleciona o agente que utiliza o menor recurso desta
32                // tarefa
33             for (k = 1; k < QUANT_AGENTES; k++) {
34                if (RECURSOS_A_T[k][j] < RECURSOS_A_T[menor][j]) {
35                   menor = k;
36                   p_local[i]->tarefas[j] = k;
37                }
38             }
39          }
40       }
41
```

```
42        // Avalia o novo indivíduo gerado
43        Avalia_Individuo(p_local[i]);
44    }
```

3.1.2 Geração de Novos Filhos e Mutação

A geração de filhos é dividida em duas partes sendo a primeira a geração de filhos derivando de dois pais distintos e a segunda sua mutação.

A geração de novos filhos acontece selecionando dois pais por torneio e realizando o cruzamento uniforme entre eles. A quantidade de filhos gerados é calculada pela fórmula $TAM\_POP * TAX\_CRUZAM + 1$ sendo a taxa de cruzamento a porcentagem em relação à população, e o valor $+1$ representando pelo menos 1 cruzamento por iteração. Os novos filhos gerados (ainda não mutados) serão salvos na futura população para que o processamento possa continuar.

Gerado os filhos, inicia-se o processo de mutação. Como a próxima população só contem indivíduos gerados nesta iteração, é realizado a mutação de todos os eles. A quantidade de mutação que um indivíduo receberá é calculado por $QUANT\_TAREFAS * TAX\_MUT$, sendo a taxa de mutação a porcentagem da quantidade de tarefas do indivíduo. A mutação é feita alterando os agentes aleatoriamente das tarefas escolhidas ao acaso. Ao final é feito a avaliação do indivíduo gerado.

3.1.3 Seleção

Ao final destes processos descritos, a nova população possuirá apenas filhos novos mutados e por isso deve ser preenchida com indivíduos aleatórios da população anterior.

Este processo acontece com dois passos. O primeiro é escolhendo o indivíduo com melhor avaliação (elite) e adicionando-o na nova população. Adicionado este indivíduo, realiza-se o preenchimento dela com indivíduos aleatórios.

Ao final, terá uma população pronta para iterar novamente no algoritmo.

3.2 Arquivos de Configuração de Execução

O arquivo de configuração deste algoritmo é composto dos seguintes itens:

1. **Quantidade de Indivíduos na População:** Quantidade de possíveis soluções que representarão uma População no algoritmo; e
2. **Valor da Taxa de Geração de Novos Filhos em Relação à População:** Valor real representando qual a porcentagem de filhos a serem gerados em relação ao tamanho da população; e
3. **Valor da Taxa de Mutação em Relação ao Tamanho da Solução a ser Mutada:** Valor real que representa qual a porcentagem de mutação que determinado filho receberá ao compor à nova População.
4. **Tempo em Segundos**.

A configuração utilizada no problema foi:

```
10000
0.2
0.01
60
```

4 Algoritmo de Recozimento Simulado

Aqui exibido detalhes do Algoritmo de Recozimento Simulado (do inglês *Simulated Annealing*, SA) implementado para este problema.

O algoritmo implementado foi baseado no livro sobre Meta-heurísticas [2], disponibilizado gratuitamente ao público pela editora Omnipax.

4.1 Execução

4.1.1 Atualização de Temperatura

Utilizou-se de um método que utiliza cálculo geométrico para a atualização da temperatura. Assim, a temperatura é iniciada com um valor definido pelo arquivo de configuração do algoritmo e este valor é decaído até chegar em uma temperatura igual à 0,2.

```
1 void Atualiza_Temperatura(double * t) {
2    *t = 0.995 * *t;
3 }
```

Com uma temperatura decaindo de forma lenta, é possível percorrer mais busca locais a procura de soluções vizinhas melhores. Da mesma forma, o fator *temperatura* decairá de forma lenta quando estiver próximo do valor 1, aperfeiçoando ainda mais a solução encontrada.

4.2 Arquivos de Configuração de Execução

O arquivo de configuração deste algoritmo é composto dos seguintes itens:

1. **Valor Inteiro da Temperatura Inicial:** Configuração do valor inicial da temperatura ao iniciar o processamento do problema; e
2. **Quantidade de Iterações:** Número de iterações de busca realizados em cada alteração da temperatura. Este valor representa uma busca em soluções melhores a fim de aprimorar a solução da temperatura atual.

A configuração utilizada no problema foi:

```
50
200000
```

5 Método Reinício

Aqui exibido detalhes do Método Reinício implementado para este problema.

O algoritmo implementado foi baseado no pseudocódigo disponibilizado pelo professor-orientador da disciplina.

5.1 Execução

O algoritmo baseia-se na simples ideia de gerar uma solução inicial e realizar i iterações sobre ela a procura de vizinhos que possuam uma avaliação melhor. Isso repete enquanto houver tempo necessário para processamento.

5.2 Arquivos de Configuração de Execução

O arquivo de configuração deste algoritmo é composto dos seguintes itens:

1. **Valor Inteiro de Tempo de Processamento:** Quantidade de segundos de processamento; e
2. **Quantidade de Iterações:** Número de iterações de busca realizados na solução gerada.

A configuração utilizada no problema foi:

```
60
10000000
```

6 Greedy Randomized Adaptive Search Procedure (GRASP)

Utilizou-se também uma versão do algotitmo GRASP para a procura de soluções boas para o GAP.

6.1 Execução

6.1.1 Construção Randomicamente Gulosa com Otimização em Avalização de Solução

Para este procedimento de geração de solução inicial, definiu-se que a lista RCL os possíveis agentes j para a tarefa i.

Assim, como é possível ver no algoritmo abaixo, a lista RCL compões de todos os agentes para a tarefa i. Entretanto, é realizado o cálculo do fator $fator = agente_{min} + \alpha(agente_{mx} - agente_{min})$, onde $agente_{min}$ e $agente_{max}$ são obitidos pelo menor e maior valor de $recurso + custo$ dos agentes j naquela tarefa i, respectivamente, e α é a porcentagem de quão guloso ou aleatório o algoritmo irá se comportar.

Ao final, o agente escolhido para a tarefa i é o agente j com valor mais próximo ao fator calculado.

```
1  void GreedyRandomizedConstruction(Solucao ** s, float alfa) {
2     int i = 0, j = 0, min = 0, max = 0, fator = 0, sum_recursos = 0;
3     char solucao_invalida = 0;
4
5     for (i = 0; i < QUANT_AGENTES; i++)
6        (*s)->excesso[i] = 0;
7
8     (*s)->custo = 0;
9
10    // Para cada tarefa
11    for (i = 0; i < QUANT_TAREFAS; i++) {
12       min = max = 0;
13
14       // Encontra os valores máximos e mínimos dos agentes
15       for (j = 1; j < QUANT_AGENTES; j++) {
16          if (RECURSOS_A_T[j][i] + CUSTO_A_T[j][i] < RECURSOS_A_T[min][i] +
             ↪  CUSTO_A_T[min][i])
17             min = j;
18
19          if (RECURSOS_A_T[j][i] + CUSTO_A_T[j][i] > RECURSOS_A_T[max][i] +
             ↪  CUSTO_A_T[max][i])
20             max = j;
21       }
22
23       // Calcula um fator de acordo com o valor alfa
24       fator = RECURSOS_A_T[min][i] + alfa * (RECURSOS_A_T[max][i] -
          ↪  RECURSOS_A_T[min][i]);
25
26       // procura o agente que tem maior proximidade com o fator
27       min = 0;
28       for (j = 1; j < QUANT_AGENTES; j++) {
29          if (abs(RECURSOS_A_T[j][i] - fator) < abs(RECURSOS_A_T[min][i] - fator))
30             min = j;
31       }
32
33       // Atribui a esta tarefa
34       (*s)->tarefas[i] = min;
35
36       // Calcula a quantidade de recusto utilizado ao atribuir a nova tarefa ao
          ↪  agente.
37       (*s)->excesso[min] += RECURSOS_A_T[min][i];
38
39       // Calcula o custo daquela tarefa
40       (*s)->custo += CUSTO_A_T[min][i];
41    }
42
43    // Verifica se a solução gerada é válida
```

```
44      for (j = 0; j < QUANT_AGENTES; j++) {
45         sum_recursos += (*s)->excesso[j];
46
47         if ((*s)->excesso[j] > CAPAC_AGENTES[j]) {
48            solucao_invalida = 1;
49         }
50      }
51
52      // Calcula o fator avaliação
53      if (solucao_invalida)
54         (*s)->avaliacao = ((double) sum_recursos ) * 1000000;
55      else {
56         (*s)->avaliacao = ((double) (*s)->custo);
57      }
58   }
```

Como já descrita uma vez na Seção 2.3, o procedimento *Construção Randomicamente Gulosa* também ganhou o processamento de avaliação otimizada sem a necessidade de solicitar o procedimento `Avalia_Solucao(Solucao)`.

6.2 Arquivos de Configuração de Execução

O arquivo de configuração deste algoritmo é composto dos seguintes itens:

1. **Valor Inteiro de Tempo de Processamento:** Quantidade de segundos de processamento; e
2. **Quantidade de Iterações:** Número de iterações de busca realizados na solução gerada.

A configuração utilizada no problema foi:

```
60
5000000
```

7 Entrada de Argumentos por Linha de Comando

Como argumentos de entrada para a execução do programa, definiu-se os seguintes parâmetros:

1. Nome do Programa;
2. Arquivo de Configuração do Algoritmo;
3. Aquivo com a instância;
4. *Seed* para o gerador aleatório e reprodução de experimentos.

Estes podem ser acessados pelo comando:
`./nome_do_programa arq_conf arq_instancia_OR seed`.

8 Experimentação

Para a coleta de resultados, cada algoritmo foi executado no mesmo intervalo de tempo e seu resultado foi comparado com os demais incluindo os valores de ótimo.

Cada algoritmo realizou-se 10 (dez) iterações em cada uma das 18 instâncias selecionadas para testes com temo médio de um minuto de execução. Elas serão descritas a seguir.

Utilizou-se também da opção `-Ofast` para a compilação de forma otimizada em todos os algoritmos.

8.1 Ambiente de *Hardware* e *Software* Utilizado para Compilação

A descrição do ambiente de testes é descrito na Tabela 1.

Tabela 1: Tabela com as informações de ambiente de execução do trabalho realizado.

Item	**Descrição**
Processador	1 Processador Intel Core i7 - 2,9 GHz
Núcleos	4 Núcleos
Cache L2 (por Núcleo)	256 KB
Cache L3	4 MB
Memória RAM	10 GB DDR3
Arquitetura	Arquitetura de von Neumann
Sistema Operacional	OS X 10.11.4 (15E65)
Versão do Kernel	Darwin 15.4.0
Compilador	Apple LLVM version 7.3.0 (clang-703.0.31)

8.2 Análise de Código

Como o computador utilizado para exeuções possui como compilador nativo o *clang-703.0.31* da LLVM, então pôde-se executar os analizadores de código em tempo de compilação já corrigindo os respectivos erros por meio de ferramentas como o comando `--analyze` e a ferramenta *Clang Static Analyser*.

8.3 Instâncias

As instâncias disponibilizadas para testes são descritas na Tabela 2.

Tabela 2: Tabela com as informações das Instâncias.

Tipo	Nome do Arquivo	Quant. de Agentes	Quant. de Tarefas
A	*a05100*	5	100
A	*a10100*	10	100
A	*a20100*	20	100
A	*a05200*	5	200
A	*a10200*	10	200
A	*a20200*	20	200
C	*c05100*	5	100
C	*c10100*	10	100
C	*c20100*	20	100
C	*c05200*	5	200
C	*c10200*	10	200
C	*c20200*	20	200
E	*e05100*	5	100
E	*e10100*	10	100
E	*e20100*	20	100
E	*e05200*	5	200
E	*e10200*	10	200
E	*e20200*	20	200

9 Resultados

Cada algoritmo executará dez vezes cada instância no intervalo de tempo de um minuto alterando os valores de *seed*. Após a execução, obteve os seguintes resultados exibidos na Tabela 3 de cada algoritmo em cada instância.

Tabela 3: Tabela com resultados estatísticos obtidos.

Algor.	Arquivo	Min	Média	Max	Desvio Padrão	Ótimo Literatura	% Acerto
GA	*a05100*	1698	1698	1698	0	1698	100 %
GA	*a10100*	1360	1360	1360	0	1360	100 %
GA	*a20100*	1158	1158	1158	0	1158	100 %
GA	*a05200*	3235	3235	3235	0	3235	100 %
GA	*a10200*	2623	2623	2623	0	2623	100 %
GA	*a20200*	2339	2339	2339	0	2339	100 %
GA	*c05100*	1982	1982	1982	0	1931	97.42684 %
GA	*c10100*	1439	1439	1439	0	1402	97.42877 %
GA	*c20100*	1281	1281	1282	0.421637	1243	97.03357 %
GA	*c05200*	3552	3553	3554	0.9660918	3456	97.2973 %
GA	*c10200*	3006	3006	3006	0	2806	93.34664 %
GA	*c20200*	2586	2587	2590	1.414214	2391	92.4594 %
GA	*e05100*	13900	13900	13900	0	12673	91.20547 %
GA	*e10100*	13920	13920	13920	0	11568	83.11539 %
GA	*e20100*	10170	10170	10170	0	8431	82.91699 %
GA	*e05200*	26820	26820	26820	0	24927	92.9453 %
GA	*e10200*	27190	27190	27190	0	23302	85.70693 %
GA	*e20200*	27580	27580	27580	0	22377	81.12606 %
SA	*a05100*	1698	1698	1698	0	1698	100 %
SA	*a10100*	1360	1360	1360	0	1360	100 %
SA	*a20100*	1158	1158	1158	0	1158	100 %
SA	*a05200*	3235	3235	3235	0	3235	100 %
SA	*a10200*	2623	2623	2623	0	2623	100 %
SA	*a20200*	2339	2339	2339	0	2339	100 %
SA	*c05100*	1937	1937	1937	0	1931	99.69024 %
SA	*c10100*	1415	1415	1415	0	1402	99.08127 %

SA	*c20100*	1264	1264	1264	0	1243	98.33861 %
SA	*c05200*	3460	3460	3460	0	3456	99.88439 %
SA	*c10200*	2838	2838	2838	0	2806	98.87245 %
SA	*c20200*	2413	2413	2413	0	2391	99.08827 %
SA	*e05100*	12760	12760	12760	0	12673	99.30262 %
SA	*e10100*	11830	11830	11830	0	11568	97.7605 %
SA	*e20100*	8837	8837	8837	0	8431	95.40568 %
SA	*e05200*	25110	25110	25110	0	24927	99.26725 %
SA	*e10200*	23820	23820	23820	0	23302	97.80483 %
SA	*e20200*	23570	23570	23570	0	22377	94.93445 %
Reinício	*a05100*	1698	1698	1698	0	1698	100 %
Reinício	*a10100*	1360	1360	1360	0	1360	100 %
Reinício	*a20100*	1158	1158	1158	0	1158	100 %
Reinício	*a05200*	3235	3235	3235	0	3235	100 %
Reinício	*a10200*	2623	2623	2623	0	2623	100 %
Reinício	*a20200*	2339	2339	2339	0	2339	100 %
Reinício	*c05100*	1953	1953	1953	0	1931	98.87353 %
Reinício	*c10100*	1433	1433	1433	0	1402	97.83671 %
Reinício	*c20100*	1264	1264	1264	0	1243	98.33861 %
Reinício	*c05200*	3503	3503	3503	0	3456	98.65829 %
Reinício	*c10200*	2852	2852	2852	0	2806	98.3871 %
Reinício	*c20200*	2445	2445	2445	0	2391	97.79141 %
Reinício	*e05100*	13950	13950	13950	0	12673	90.8589 %
Reinício	*e10100*	13200	13200	13200	0	11568	87.643 %
Reinício	*e20100*	9534	9534	9534	0	8431	88.43088 %
Reinício	*e05200*	28690	28690	28690	0	24927	86.87485 %
Reinício	*e10200*	27230	27230	27230	0	23302	85.58416 %
Reinício	*e20200*	25990	25990	25990	0	22377	86.09187 %
GRASP	*a05100*	1698	1698	1698	0	1698	100 %
GRASP	*a10100*	1360	1360	1360	0	1360	100 %
GRASP	*a20100*	1158	1158	1158	0	1158	100 %
GRASP	*a05200*	3235	3235	3235	0	3235	100 %
GRASP	*a10200*	2623	2623	2623	0	2623	100 %
GRASP	*a20200*	2339	2339	2339	0	2339	100 %
GRASP	*c05100*	1955	1955	1955	0	1931	98.77238 %
GRASP	*c10100*	1419	1419	1419	0	1402	98.80197 %
GRASP	*c20100*	1268	1268	1268	0	1243	98.02839 %
GRASP	*c05200*	3490	3490	3490	0	3456	99.02579 %
GRASP	*c10200*	2860	2860	2860	0	2806	98.11189 %
GRASP	*c20200*	2457	2457	2457	0	2391	97.3138 %
GRASP	*e05100*	13840	13840	13840	0	12673	91.58777 %
GRASP	*e10100*	12760	12760	12760	0	11568	90.6228 %
GRASP	*e20100*	9164	9164	9164	0	8431	92.00131 %
GRASP	*e05200*	29050	29050	29050	0	24927	85.81314 %
GRASP	*e10200*	26660	26660	26660	0	23302	87.39124 %
GRASP	*e20200*	25580	25580	25580	0	22377	87.47508 %

9.1 Resultados em Gráficos

Abaixo serão exibidos as comparações de cada algoritmo em cada instância. Para melhor visualização, foi adicionado uma linha horizontal azulada representando o ótimo da literatura.

9.1.1 Instância *a05100*

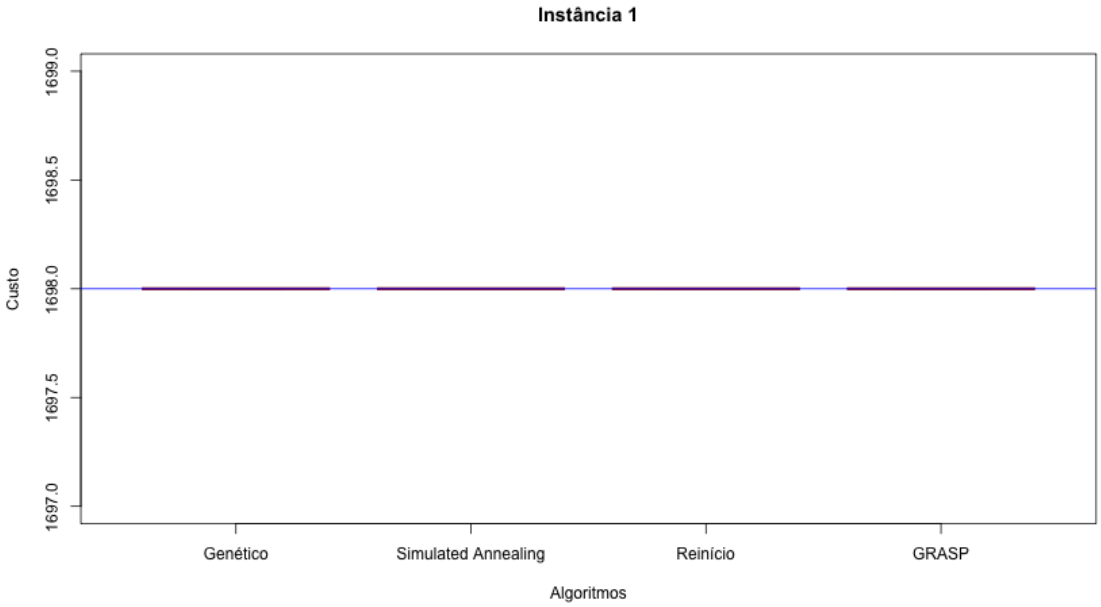

Figura 1: BoxPlot da Instância *a05100*.

9.1.2 Instância *a05200*

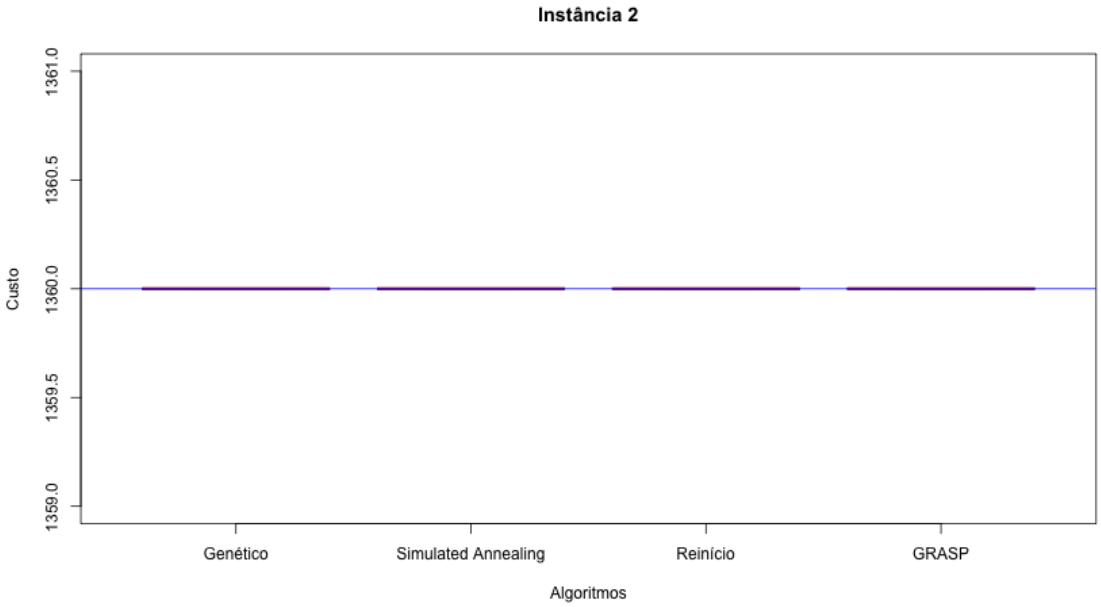

Figura 2: BoxPlot da Instância *a05200*.

9.1.3 Instância *a10100*

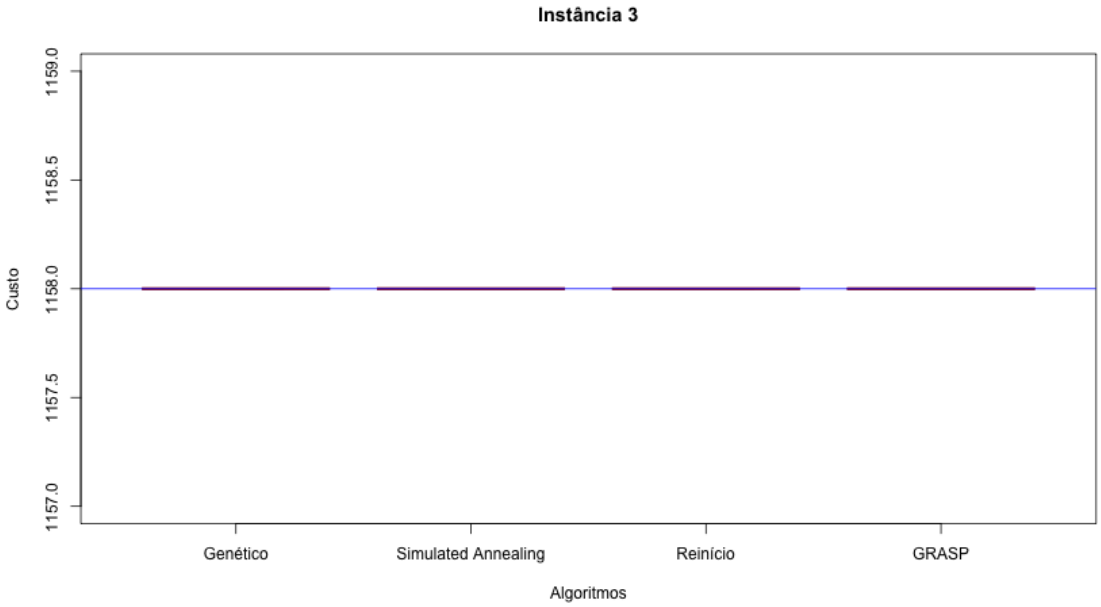

Figura 3: BoxPlot da Instância *a10100*.

9.1.4 Instância *a10200*

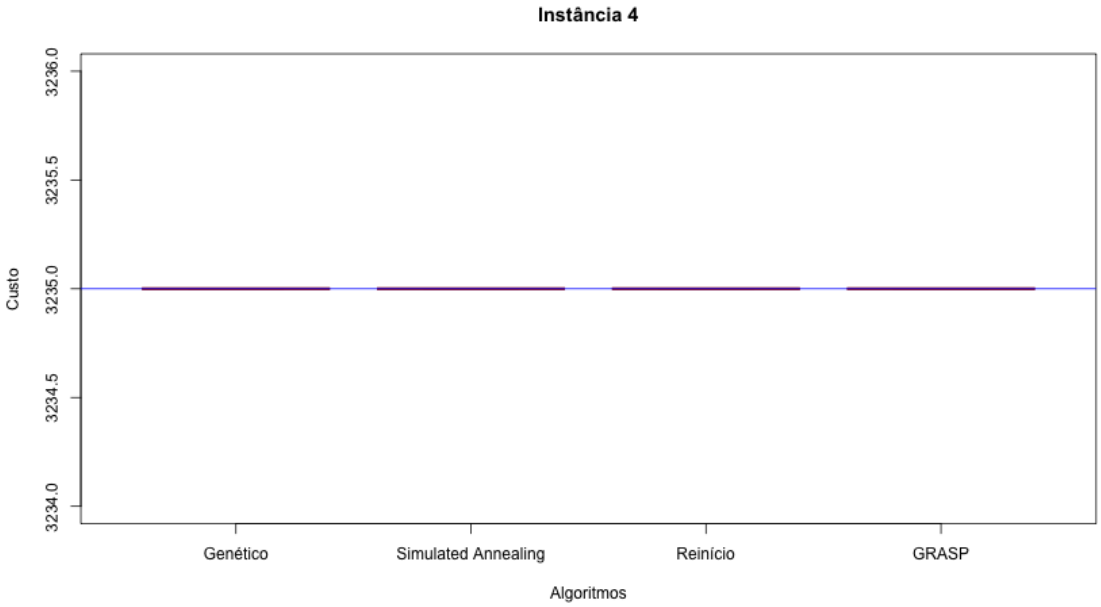

Figura 4: BoxPlot da Instância *a10200*.

9.1.5 Instância *a20100*

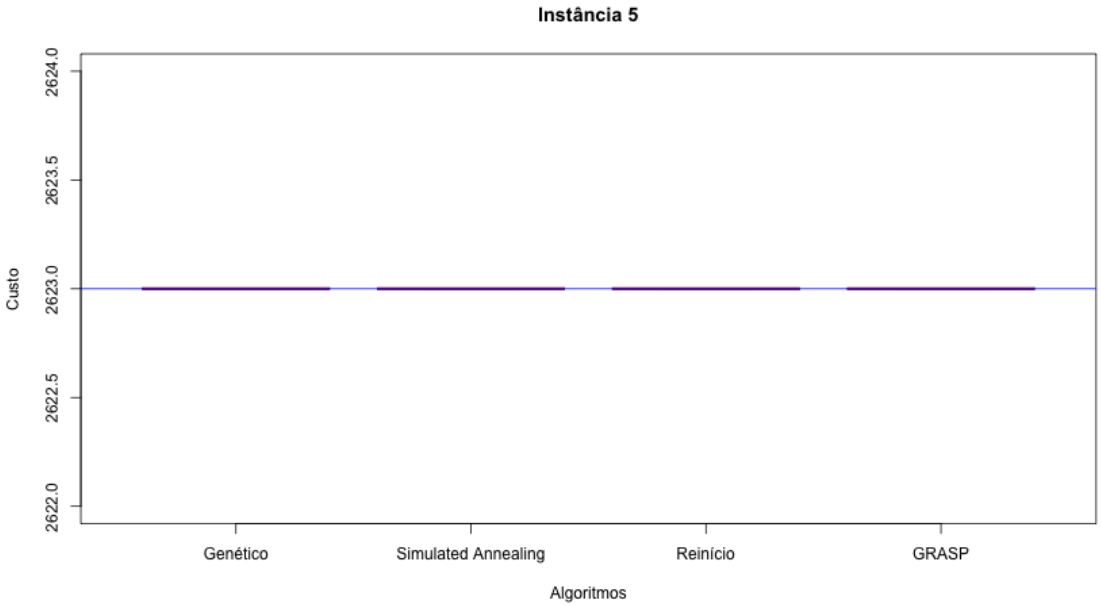

Figura 5: BoxPlot da Instância *a20100*.

9.1.6 Instância *a20200*

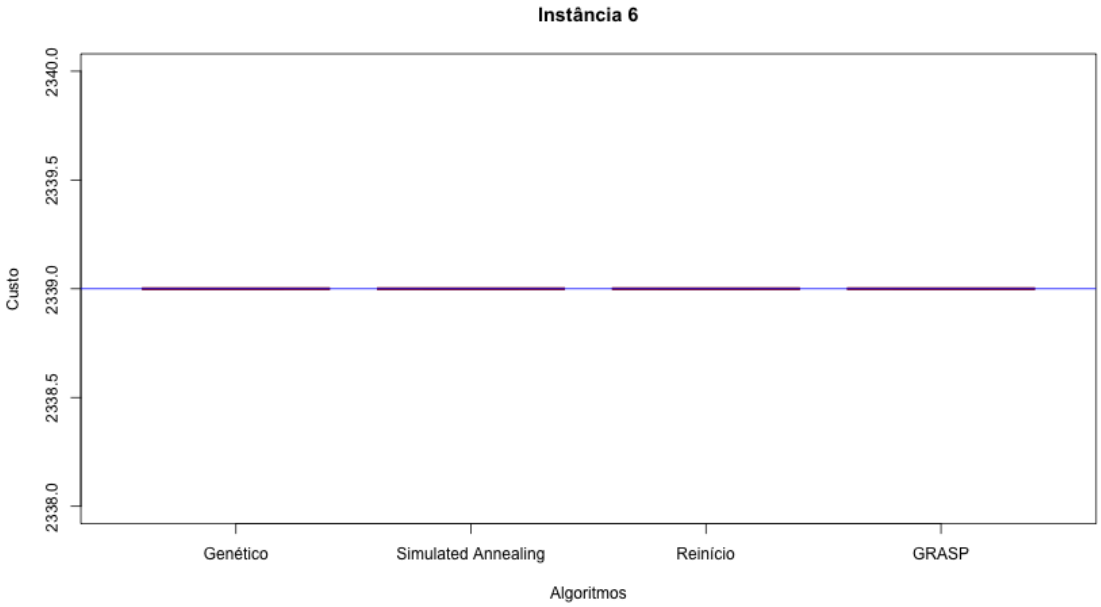

Figura 6: BoxPlot da Instância *a20200*.

9.1.7 Instância *c05100*

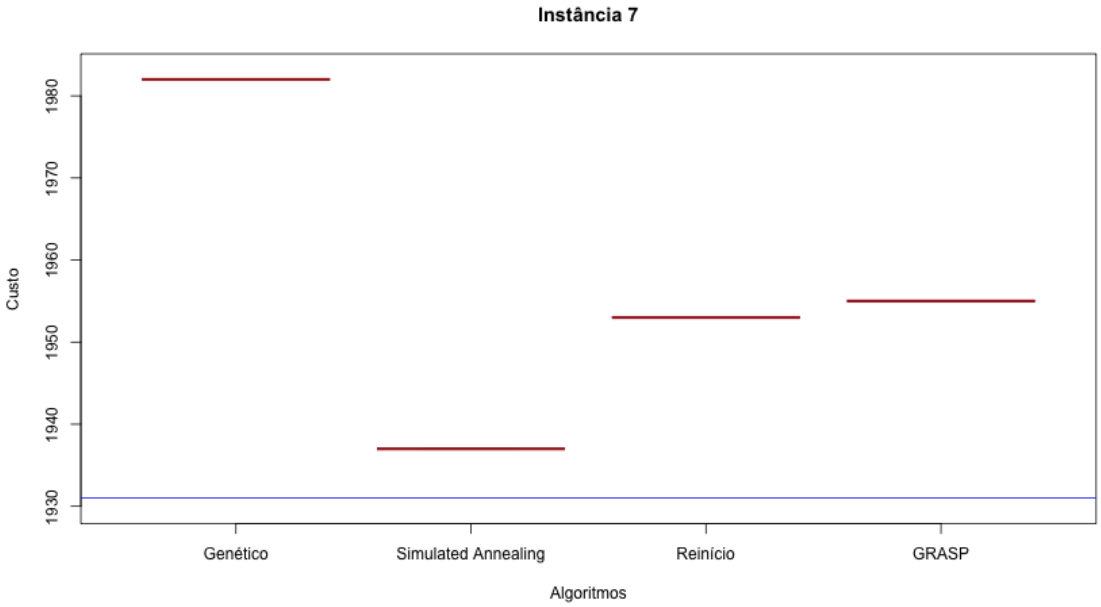

Figura 7: BoxPlot da Instância *c05100*.

9.1.8 Instância *c05200*

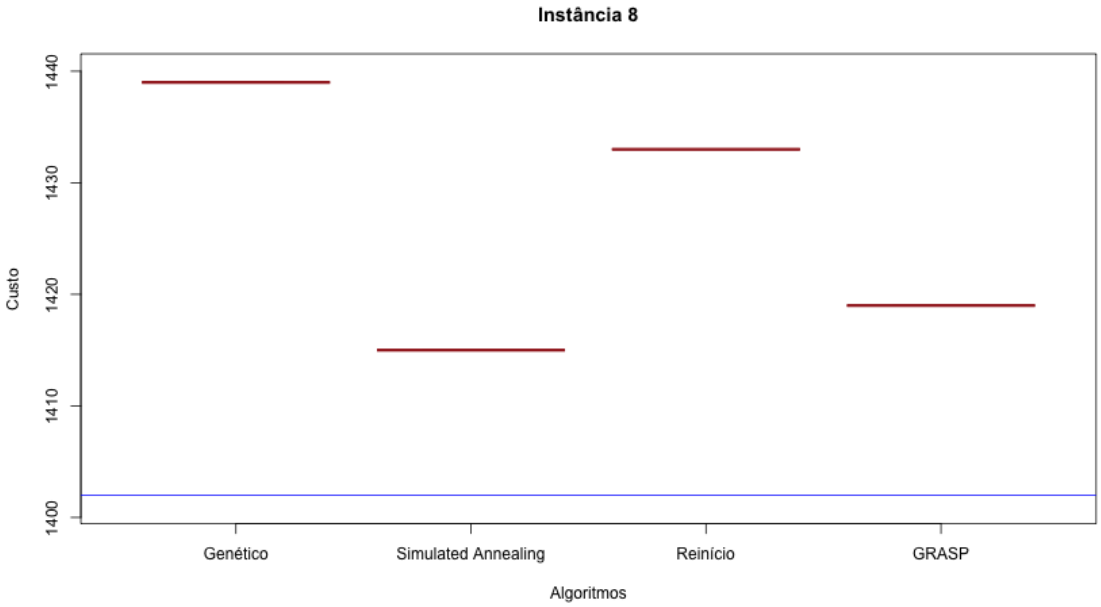

Figura 8: BoxPlot da Instância *c05200*.

9.1.9 Instância *c10100*

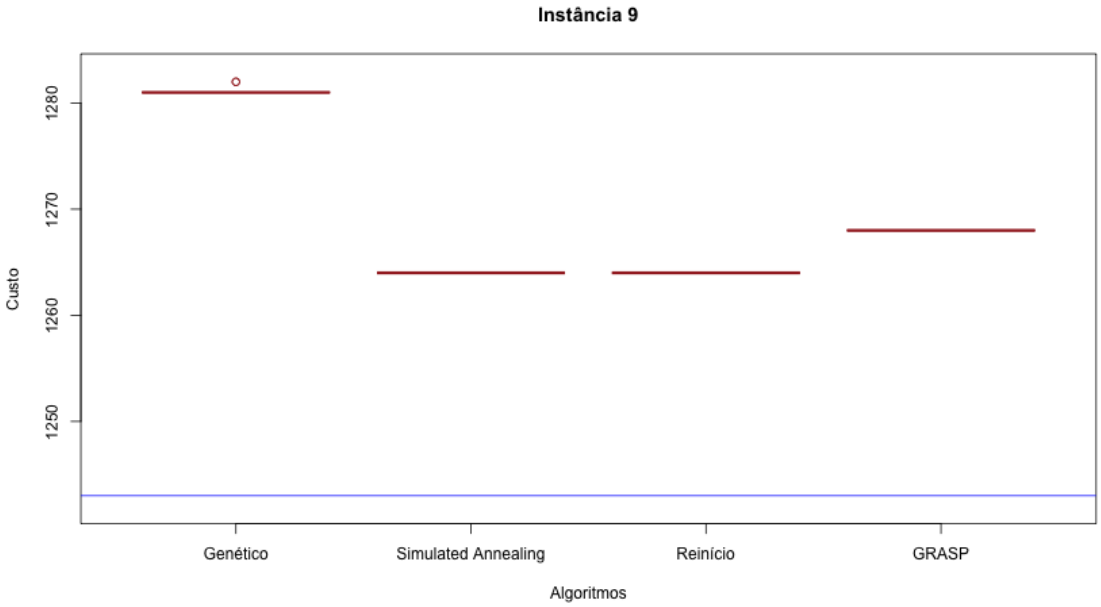

Figura 9: BoxPlot da Instância *c10100*.

9.1.10 Instância *c10200*

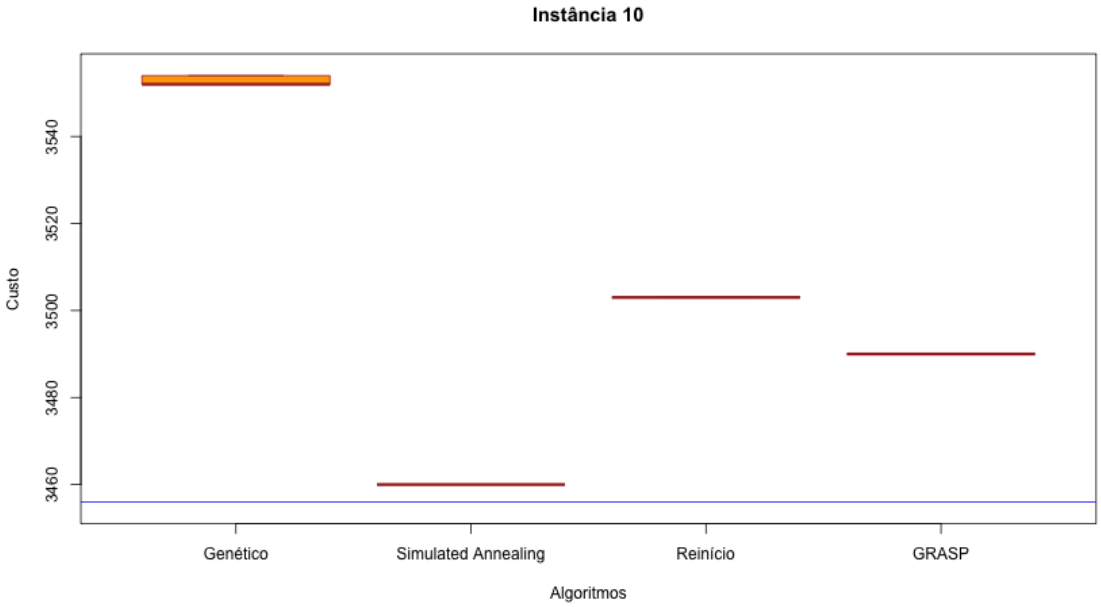

Figura 10: BoxPlot da Instância *c10200*.

9.1.11 Instância *c20100*

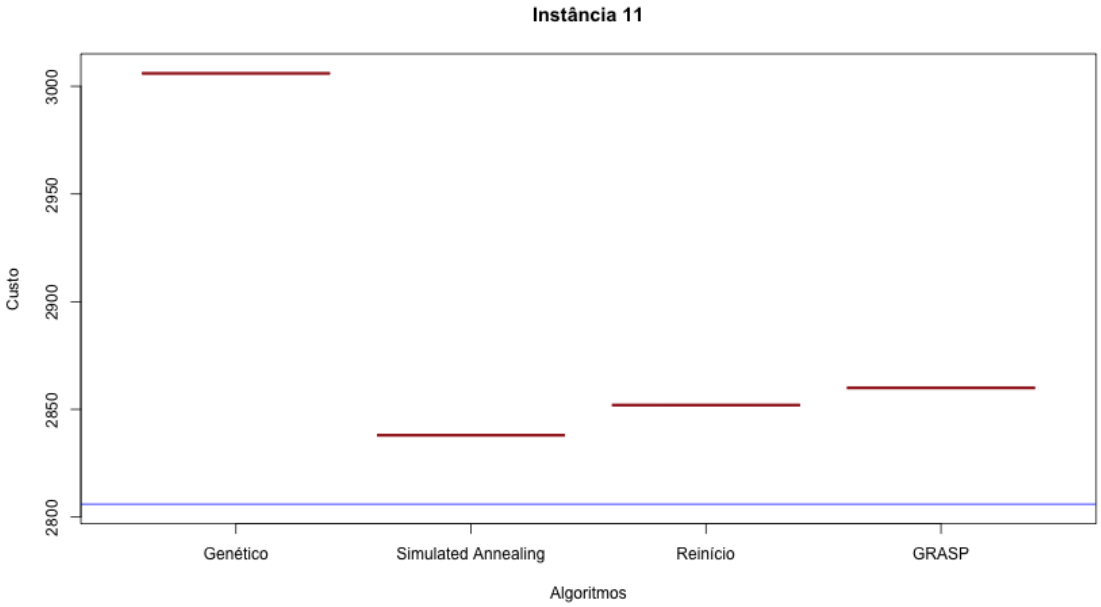

Figura 11: BoxPlot da Instância *c20100*.

9.1.12 Instância *c20200*

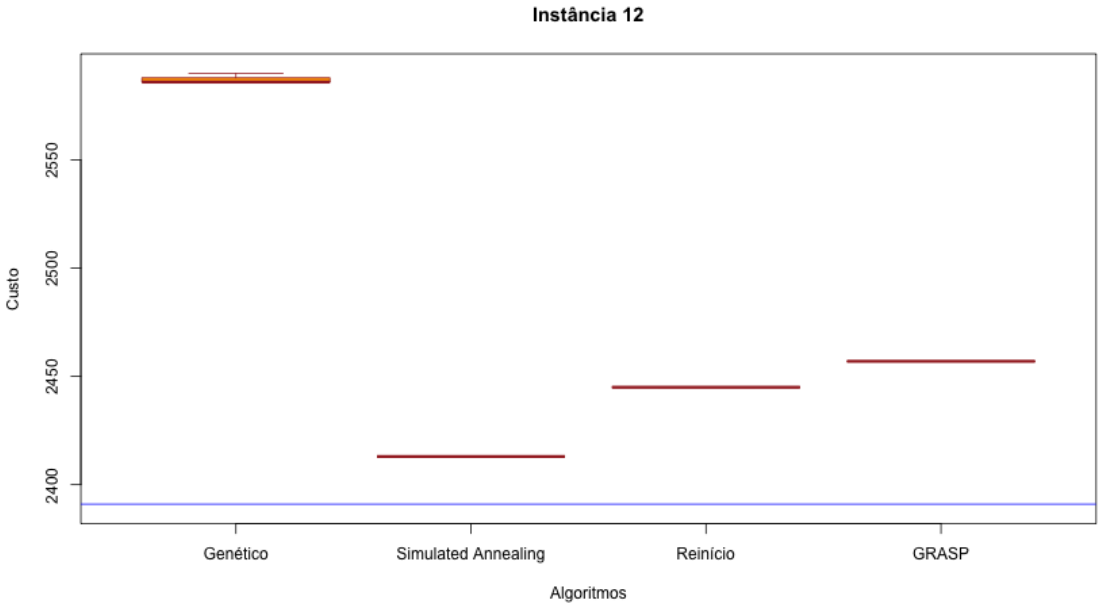

Figura 12: BoxPlot da Instância *c20200*.

9.1.13 Instância *e05100*

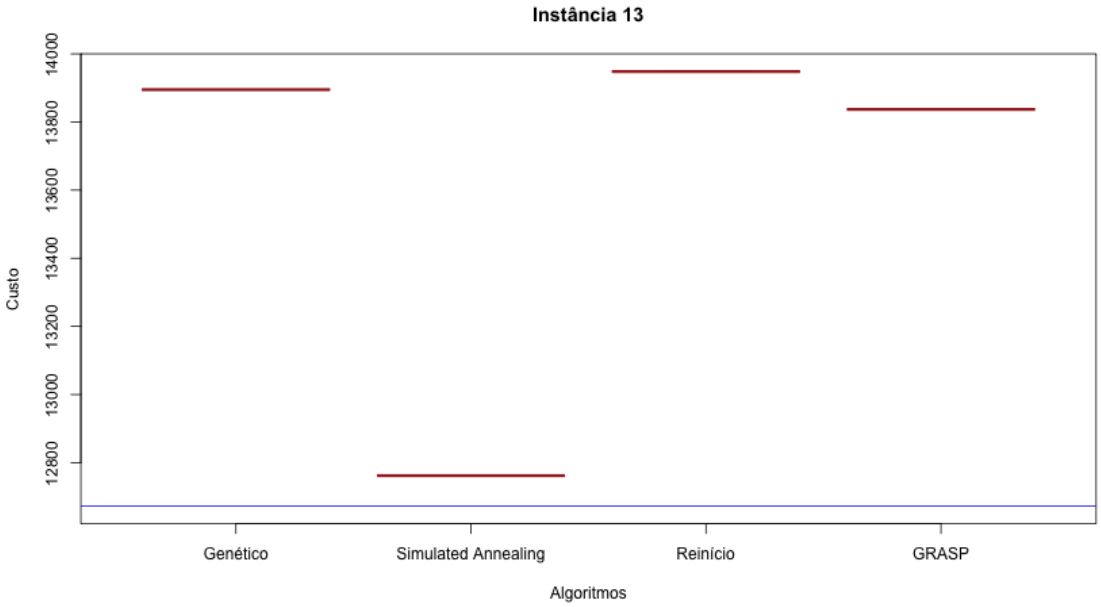

Figura 13: BoxPlot da Instância *e05100*.

9.1.14 Instância *e05200*

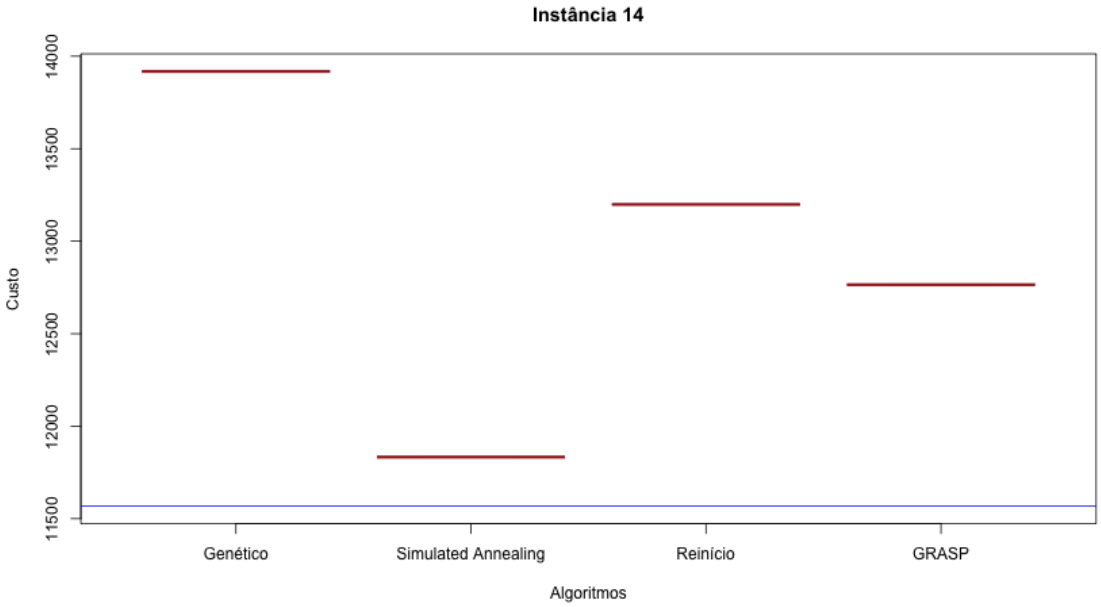

Figura 14: BoxPlot da Instância *e05200*.

9.1.15 Instância *e10100*

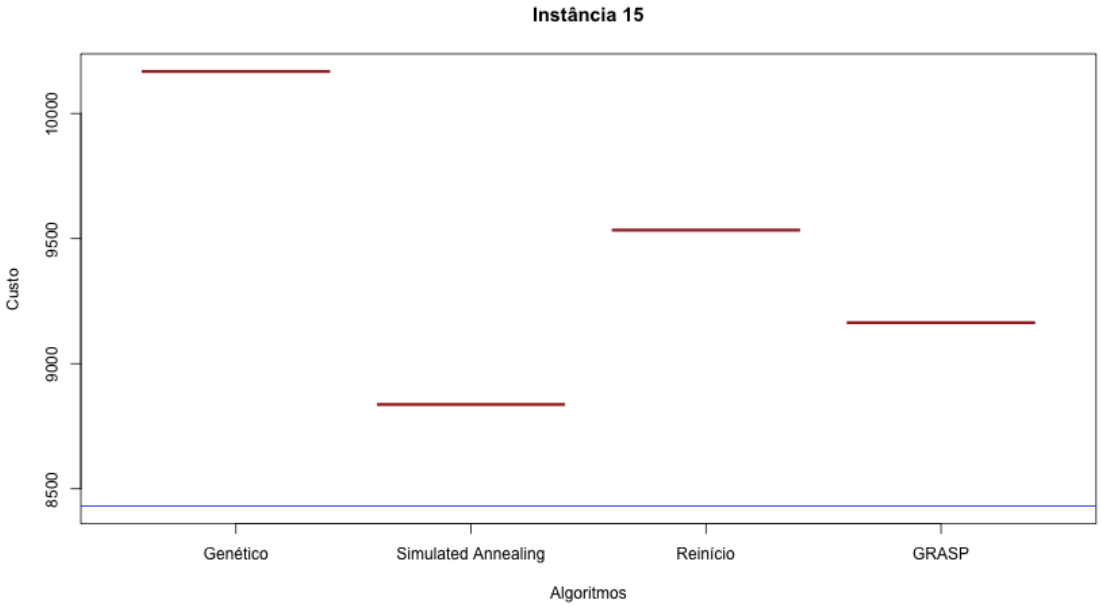

Figura 15: BoxPlot da Instância *e10100*.

9.1.16 Instância *e10200*

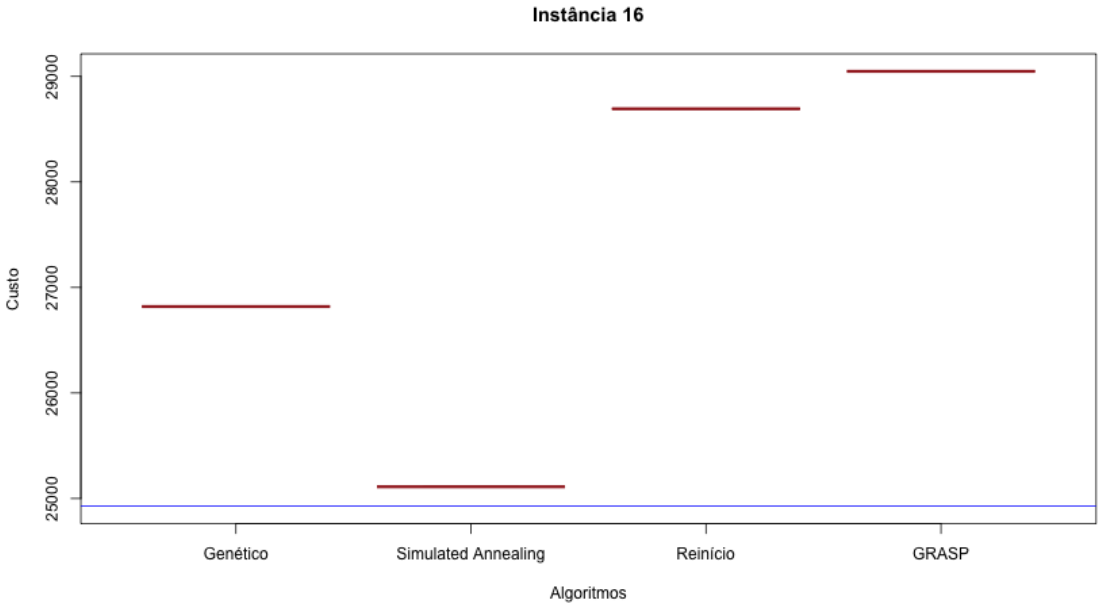

Figura 16: BoxPlot da Instância *e10200*.

9.1.17 Instância *e20100*

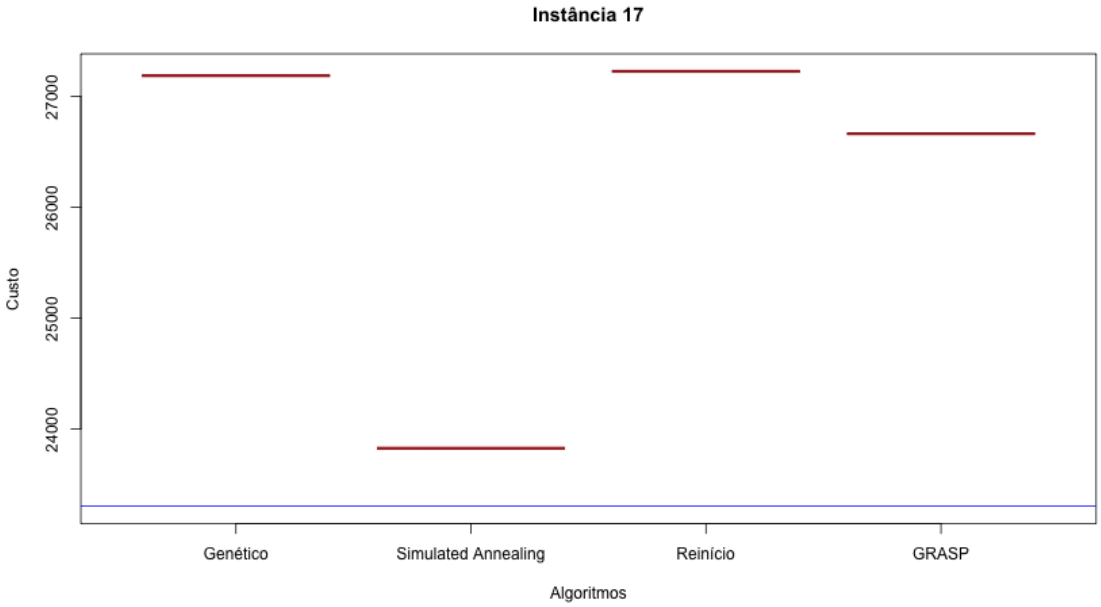

Figura 17: BoxPlot da Instância *e20100*.

9.1.18 Instância *e20200*

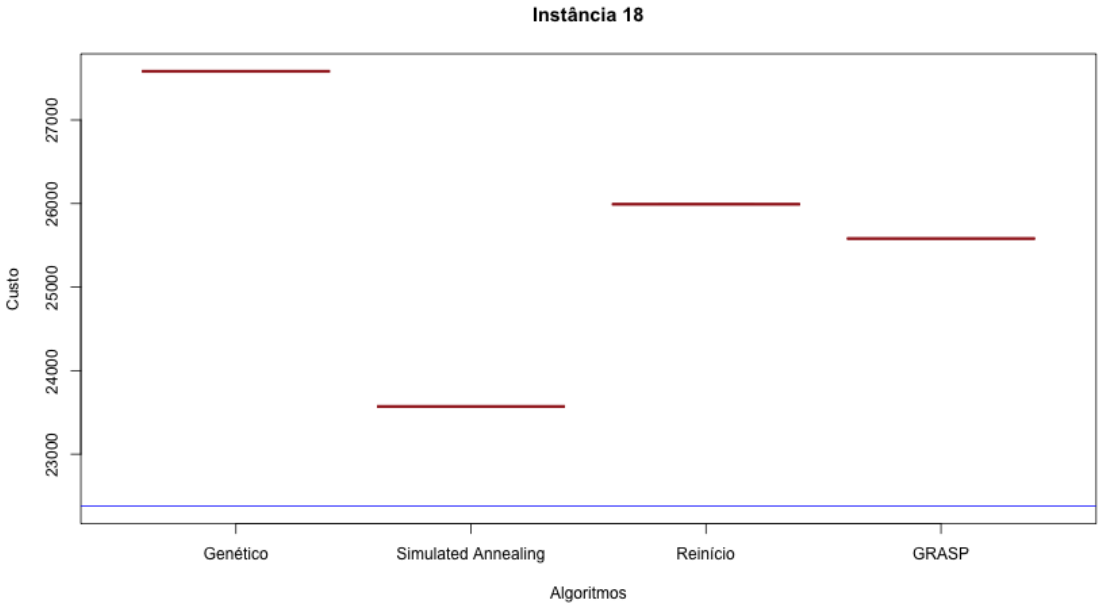

Figura 18: BoxPlot da Instância *e20200*.

9.2 Estudo Estatísticos

Como análise comparativa final, foi confeccionado uma tabela comparando a média das distância de todos os algoritmos em todas as instâncias executadas. O resultado é exibido na Tabela 4.

Tabela 4: Análise final de proximidade de cada algoritmo em relação ao ótimo.

Algoritmos	**Média Geral em Relação ao Ótimo**
Algoritmo Genético	94.00048 %
Recozimento Simulado	98.85725 %
Método Reinício	95.29829 %
GRASP	95.83031 %

É possível concluir que todos os algoritmos teve um resultado médio acima de 94% em todas as instâncias, sendo uma boa porcentagem quando necessita-se de um valor próximo ao ótimo em problemas combinatórios.

Um item que deve-se atentar é no algoritmo de Recozimento Simulado que obteve uma média geral de quase 99% de proximidade ao ótimo global encontrado pela literatura.

10 Comentários Finais

A primeira conclusão a ser feita é a diferença do Algoritmo Genético em relação aos outros. Diferentemente do Genético, os três algoritmos compartilham da mesma natureza de procura que é baseada na geração de indivíduos aleatórios e busca local em seus vizinhos o que implica exatamente nos dados amostrados. Os três outros algoritmos (Recozimento Simulado, Método de Reinício e GRASP) utilizam de um processamento simples. Com não é necessário realizar muitas operações para completar uma iteração de seu processamento, eles permitem que sejam realizados mais cálculos por intervalo de tempo, proporcionando uma vantagem maior quando o processador de testes consegue realizar muito processamento por segundo. Estes algoritmos, nesta vantagem, permitem que vasculhem uma gama maior de locais a procura de um ótimo/solução boa favorável, o que realmente acontece. Ao contrário, o Algoritmo Genético é um algoritmo lento e com curva de otimização lenta já que é preciso realizar várias operações em seu processamento.

Um item importante a ser abordado é o método de avaliação presente nos métodos `Avalia_Solucao()` e `Gera_Vizinho()`. Como o método possui duas funções diferentes (uma pra soluções válidas e outra para inválidas), é possível perceber que resultados válidos/factíveis são gerados de forma mais rápida tornando o algoritmo mais eficaz. O método `Avalia_Solucao()` é invocado somente na construção de uma solução incial aleatória nos algoritmos Recozimento Simulado, Método de Reinício e GRASP melhorando ainda mais sua performance ao utilizar o procedimento de geração de vizinho otimizada (no qual não necessita de invocar o método de avaliação).

Os parâmetros de todos os algoritmos foram mantidos os mesmos para todas as instâncias para generalizar seu desempenho e eficiência na procura de boas soluções sem depender de determinadas características específicas de determinada instância.

Também foi possível concluir que a literatura utiliza um tempo maior de execução para encontrar soluções ótimas, o que difere deste trabalho. Neste, foi determinado um tempo de um minuto por execução o que restringe ainda mais a proximidade do ótimo global. Entretanto, todos os algoritmos obtiveram excelente resultados mesmo em tempo de execução curto.

11 Código dos Algoritmos

Os algoritmos *shell*, Algoritmo Genético, Recozimento Simulado, Método Reinício e GRASP, utilizados no trabalho, estão situados nas páginas 27, 41, 52, 56, 59, respectivamente.

11.1 *Shell Script*

```
#!/bin/bash

eval "gcc -Ofast gap.c GAP-Genetic.c -o GAP-Genetic.o"
eval "gcc -Ofast gap.c GAP-GRASP.c   -o GAP-GRASP.o"
eval "gcc -Ofast gap.c GAP-Greedy.c  -o GAP-Greedy.o"
eval "gcc -Ofast gap.c GAP-SA.c      -o GAP-SA.o"

instancias=(
  ../Instancias/gap_a/a05100
  ../Instancias/gap_a/a05200
  ../Instancias/gap_a/a10100
  ../Instancias/gap_a/a10200
  ../Instancias/gap_a/a20100
  ../Instancias/gap_a/a20200

  ../Instancias/gap_c/c05100
  ../Instancias/gap_c/c05200
  ../Instancias/gap_c/c10100
  ../Instancias/gap_c/c10200
  ../Instancias/gap_c/c20100
  ../Instancias/gap_c/c20200

  ../Instancias/gap_e/e05100
  ../Instancias/gap_e/e05200
  ../Instancias/gap_e/e10100
  ../Instancias/gap_e/e10200
  ../Instancias/gap_e/e20100
  ../Instancias/gap_e/e20200
  )

algoritmos=( GAP-Genetic.o GAP-GRASP.o GAP-Greedy.o GAP-SA.o )

configuracoes=( p_ga.txt p_grasp.txt p_greedy.txt p_sa.txt )

echo "Quantas iteracoes?"
read quantidade_iteracoes;
echo


```

```
for indice_algoritmo in "${!algoritmos[@]}"
do
   echo $indice_algoritmo

  for instancia in "${instancias[@]}"
  do
     echo $instancia

    for (( i = 0; i < "$quantidade_iteracoes"; i++ )); do
      echo "$i"

      cmd="./${algoritmos[$indice_algoritmo]} ${configuracoes[$indice_algoritmo]}
       ↪  $instancia $i"
      date
      echo $cmd
      $cmd
      echo
      echo "------------------------------------------------------------"
      echo
    done
    echo

  done
  echo

done
```

11.2 Códigos em *R*

11.2.1 Procedimento Estatístico

```
GeraDados <- function(diretorio) {
  arq_gen <- read.table(diretorio)
  vet_gen <- rep(arq_gen$V1);

  desloca <- 1;
  cat("\tMin. 1st Qu.  Median    Mean 3rd Qu.    Max.\n")



  otimo <- c(1698   ,
             1360   ,
             1158   ,
             3235   ,
             2623   ,
             2339   ,
             1931  ,
```

```
                1402  ,
                1243  ,
                3456  ,
                2806  ,
                2391  ,
                12673,
                11568,
                8431  ,
                24927,
                23302,
                22377)

  # 3 * 6 arquivos (a, c, d)
  for (j in 1:18) {
    cat(j, "\t");
    vetor_buffer <- vet_gen[desloca:(desloca + 9)];
    cat(summary(vetor_buffer), "\t\t")

    cat(sd(vetor_buffer))
    cat("\n")

    desloca = desloca + 10;
  }
  desloca = 1;
  for (j in 1:18) {
    #cat(j, "\t");
    vetor_buffer <- vet_gen[desloca:(desloca + 9)];
    cat(100 * (otimo[j] / min(vetor_buffer)))
    cat("\n")

    desloca = desloca + 10;
  }

}

GeraDados("out_genetico.txt")
GeraDados("out_simulated_annealing.txt")
GeraDados("out_reinicio.txt")
GeraDados("out_grasp.txt")
```

11.2.2 Gráficos

```
otimo <- c(1698  ,
           1360  ,
           1158  ,
           3235  ,
```

```
5             2623   ,
6             2339   ,
7             1931   ,
8             1402   ,
9             1243   ,
10            3456   ,
11            2806   ,
12            2391   ,
13            12673,
14            11568,
15            8431 ,
16            24927,
17            23302,
18            22377)
19 GeraDados <- function(d_ga, d_sa, d_r, d_gr) {
20   arq_ga <- read.table(d_ga)
21   arq_sa <- read.table(d_sa)
22   arq_r <- read.table(d_r )
23   arq_gr <- read.table(d_gr)
24
25   ga <- rep(arq_ga$V1);
26   sa <- rep(arq_sa$V1);
27   r  <- rep(arq_r$V1);
28   gr <- rep(arq_gr$V1);
29
30   desloca <- 1;
31
32
33   for (j in 1:18) {
34     #png(paste(d_ga, "_", j, ".png", sep=""), width = 840, height = 480, units =
       ↪  "px");
35     png(paste(j, ".png", sep=""), width = 840, height = 480, units = "px");
36
37     boxplot(ga[desloca:(desloca + 9)],
38             sa[desloca:(desloca + 9)],
39             r[desloca:(desloca + 9)],
40             gr[desloca:(desloca + 9)],
41             ylim = c(otimo[j] - 1,
42                      max(max(ga[desloca:(desloca + 9)]), max(sa[desloca:(desloca
                         ↪  + 9)]), max(r[desloca:(desloca + 9)]),
                         ↪  max(gr[desloca:(desloca + 9)])) + 1),
43             main = paste("Instância ", j, sep = ""),
44             xlab = "Algoritmos",
45             ylab = "Custo",
46             col = "orange",
47             border = "brown",
48             names = c("Genético", "Simulated Annealing", "Reinício", "GRASP"));
49     abline(h = otimo[j], col = "blue");
```

```
50
51      dev.off()
52
53      desloca = desloca + 10;
54    }
55  }
56
57  GeraDados("out_genetico.txt", "out_simulated_annealing.txt", "out_reinicio.txt",
    ↪  "out_grasp.txt")
```

11.3 Códigos em *C*

11.3.1 *gap.h*

```
1   /*
2    * Trabalho de Projeto e Análise de Algoritmo
3    * Período 16.1
4    *
5    * Desenvolver Metaheurísticas para o Problema de Alocação Generalizada
6    *
7    * Funções Genéricas do Problam GAP.
8    * Data: 01/08/2016.
9    * Distribuição Livre, desde que referenciando o autor.
10   *
11   * Professor: Haroldo Santos
12   *
13   * Autor do Trabalho: Rodolfo Labiapari Mansur Guimarães
14   */
15
16  #include <time.h>
17
18  /*
19   * Estrutura de dados para armazenamento das informações de conjunto de
20   *   soluções.
21   *
22   * excesso - valor com a capacidade atual de cada agente desta solução
23   * tarefas - Vetor com tamanho Tarefas onde cada endereço é indicado o agente
24   *   responsável por tal
25   * avaliação - Valor fitness da solução (sum_excesso + custo) * penalidade
26   * custo - custo total desta solução
27   */
28  typedef struct Struct_Solucao {
29     int * excesso;
30     int * tarefas;
31     double avaliacao;
32     double custo;
33  } Solucao;
```

```
34
35 int QUANT_TAREFAS   ; // Quantidade de tarefas
36 int QUANT_AGENTES   ; // Quantidade de agentes
37 int * CAPAC_AGENTES ; // Vetor com capacidade máxima de cada gente.
38 int ** RECURSOS_A_T ; // Matriz de recursos[a,t]
39 int ** CUSTO_A_T    ; // Matriz de custo [a,t]
40
41
42 int    TAM_POP    ; // Tamanho da população do algoritmo
43 int    ITERACOES  ; // Quantidade de iterações
44 float  TAX_CRUZAM ; // Porcentagem de cruzamentos a serem realizados
45 float  TAX_MUT    ; // Porcentagem dos dados do filho que serão mutados
46
47 char IMPRIMIR ; // Variável que permite impressão na tela.
48 int SECONDS   ; // Tempo lido pelo arquivo de configuração
49 int MAXIteracoes ; // Quantidade de iterações daquela temperatura
50 FILE * out    ; // Arquivo para gravação de dados permanente
51 time_t endwait, start; // Variáveis de tempo para cálculo do intervalo de
52    //tempo de execução
53
54 double Avalia_Solucao(Solucao * sol) ;
55 void Gera_Vizinho(Solucao * atual, Solucao ** proxima) ;
56 Solucao * Instancia_Solucao() ;
57 void Gera_Solucao_Aleatoria(Solucao ** s) ;
58 char Teste_Aceita_Nova_Solucao(double temp, Solucao ** atual, Solucao * proxima)
    ↪  ;
59 void Aceita_Nova_Solucao(Solucao ** atual, Solucao * proxima) ;
60 void Le_Instancia(char * path) ;
61 void Imprime_Solucao(Solucao * ind) ;
62 void Imprime_Status(double t, Solucao * m) ;
63 void Desaloca_Solucao(Solucao ** p) ;
64 Solucao * Instancia_Solucao_Aleatoria() ;
65 void Desaloca_Populacao(Solucao *** p) ;
66 void Imprime_Instancia() ;
```

11.3.2 *gap.c*

```
1 /*
2  * Trabalho de Projeto e Análise de Algoritmo
3  * Período 16.1
4  *
5  * Desenvolver Metaheurísticas para o Problema de Alocação Generalizada
6  *
7  * Funções Genéricas do Problam GAP.
8  * Data: 01/08/2016.
9  * Distribuição Livre, desde que referenciando o autor.
```

```
10   *
11   * Professor: Haroldo Santos
12   *
13   * Autor do Trabalho: Rodolfo Labiapari Mansur Guimarães
14   */
15
16  #include <stdio.h>
17  #include <stdlib.h>
18  #include <math.h>
19  #include <limits.h>
20  #include "gap.h"
21
22
23  /*
24   * Procedimento que avalia a solução atual.
25   *
26   * Se a solução é factível, retorna seu custo real.
27   * Se infactível retorna o custo encontrado * 1000000
28   */
29  double Avalia_Solucao(Solucao * sol) {
30     int i = 0, capacidade_agentes[QUANT_AGENTES], sum_recursos = 0;
31     double custo = 0; char solucao_invalida = 0;
32
33     // Define a capacidade inicial utilizada de cada agente com 0
34     for (i = 0; i < QUANT_AGENTES; i++)
35        capacidade_agentes[i] = 0;
36
37     // Realiza os cálculos de custo e capacidade
38     for (i = 0; i < QUANT_TAREFAS; i++) {
39        custo += CUSTO_A_T[sol->tarefas[i]][i];
40        capacidade_agentes[sol->tarefas[i]] += RECURSOS_A_T[sol->tarefas[i]][i];
41     }
42
43     // Verifica se algum agente passou sua capacidade máxima
44     for (i = 0; i < QUANT_AGENTES; i++) {
45        sol->excesso[i] = capacidade_agentes[i];
46        sum_recursos += capacidade_agentes[i];
47
48        // Se sim define esta solução como inválida
49        if (capacidade_agentes[i] > CAPAC_AGENTES[i])
50           solucao_invalida = 1;
51     }
52
53     sol->custo = custo;
54
55     // Caso a solução foi excedida, altera a avaliação do indivíduo tornando-o
56        // pior.
57     if (solucao_invalida)
```

```
58          sol->avaliacao = ((double) sum_recursos ) * 1000000;
59       else {
60          sol->avaliacao = ((double) custo);
61       }
62
63       return sol->avaliacao;
64    }
65
66
67
68    /*
69     * Procedimento de geração de indivíduos por meio do procedimento shift.
70     *
71     * Um detalhe a se atentar é que é realizado shift em somente 1 tarefa da
72     *   solução.
73     */
74    void Gera_Vizinho(Solucao * atual, Solucao ** proxima) {
75       int i = 0, j = 0, agente_atual = 0, agente_novo = 0,
76       tarefa_escolhida1 = 0, sum_recursos = 0,
77       quant_alteracoes = 0;
78       char solucao_invalida = 0;
79
80       // Copia os valores do atual
81       for (i = 0; i < QUANT_TAREFAS; i++) {
82          (*proxima)->tarefas[i] = atual->tarefas[i];
83          if (i < QUANT_AGENTES)
84             (*proxima)->excesso[i] = atual->excesso[i];
85       }
86
87       (*proxima)->custo = atual->custo;
88
89
90       // Define uma quantidade de alterações pra gerar o vizinho
91       quant_alteracoes = 2;
92
93       // Altera o indivíduo
94       for (i = 0; i < quant_alteracoes; i++) {
95
96          // Escolhe a tarefa que será alterada
97          tarefa_escolhida1 = random() % QUANT_TAREFAS;
98          agente_atual = (*proxima)->tarefas[tarefa_escolhida1];
99
100         // Gera um novo agente pra ela e certifica que ele é diferente do
            ↪  anterior.
101         do {
102            agente_novo = random() % QUANT_AGENTES;
103         } while (agente_novo == agente_atual);
104
```

```
105         // Atribui o novo agente à tarefa
106         (*proxima)->tarefas[tarefa_escolhida1] = agente_novo;
107 
108         // Procedimento Otimizado:
109            // A cada alteração, realiza-se a alteração dos valores da nova
               ↪  geração.
110            // Como este novo vizinho não é feito do zero e sim sobre cópia de um
               ↪  anterior,
111            // basta alterar os valores herdados do seu anterior, atualiando em
               ↪  O(1)
112 
113         // Sendo assim, atualiza o excesso de cada agente
114            // Retira recurso do agente que ficou livre
115         (*proxima)->excesso[agente_atual] -=
           ↪  RECURSOS_A_T[agente_atual][tarefa_escolhida1];
116            // Acrescenta recurso do agente que recebeu a tarefa atual
117         (*proxima)->excesso[agente_novo ] += RECURSOS_A_T[agente_novo
           ↪  ][tarefa_escolhida1];
118 
119         // Calcula o custo atual desta solução
120         (*proxima)->custo += CUSTO_A_T[agente_novo ][tarefa_escolhida1] -
           ↪  CUSTO_A_T[agente_atual][tarefa_escolhida1];
121      }
122 
123      // Verifica se a solução gerada é válida
124      for (j = 0; j < QUANT_AGENTES; j++) {
125         sum_recursos += (*proxima)->excesso[j];
126 
127         if ((*proxima)->excesso[j] > CAPAC_AGENTES[j]) {
128            solucao_invalida = 1;
129         }
130      }
131 
132      // Calcula o fator avaliação
133      if (solucao_invalida)
134         (*proxima)->avaliacao = ((double) sum_recursos ) * 1000000;
135      else {
136         (*proxima)->avaliacao = ((double)  (*proxima)->custo);
137      }
138   }
139 
140 
141 
142   /*
143    * Procedimento que instancia uma nova solução com seus valores totalmente
144    *   aleatórios.
145    */
146   Solucao * Instancia_Solucao() {
```

```
147     Solucao * s;
148
149     s = calloc(1, sizeof(Solucao));
150
151     s->tarefas = calloc(QUANT_TAREFAS, sizeof(int));
152     s->excesso = calloc(QUANT_AGENTES, sizeof(int));
153
154     // Defini-o como uma solução inicial ruim
155     s->avaliacao = INT_MAX;
156     s->avaliacao = INT_MAX;
157
158     // Retorna a solução
159     return s;
160 }
161
162
163 /*
164  * Procedimento que instancia uma nova solução com seus valores totalmente
165  *   aleatórios.
166  */
167 void Gera_Solucao_Aleatoria(Solucao ** s) {
168     int i;
169
170     for (i = 0; i < QUANT_TAREFAS; i++) {
171         (*s)->tarefas[i] = random() % QUANT_AGENTES;
172     }
173
174     Avalia_Solucao(*s);
175 }
176
177
178 /*
179  * Procedimento que copia as informações de uma solução para a outra de forma
180  *   a aceitar aquela solução.
181  */
182 char Teste_Aceita_Nova_Solucao(double temp, Solucao ** atual, Solucao * proxima)
    ↪  {
183     int i;
184
185     // Verifica se a solução atual é melhor que a atual.
186     if ((*atual)->avaliacao > 2 * 1000000 ||
187             ((proxima->avaliacao < 2 * 1000000) &&
188             (proxima->custo < (*atual)->custo))) {
189
190         //Imprime_Status(temp, *atual);
191
192         // Copia a solução
193         for (i = 0; i < QUANT_TAREFAS; i++) {
```

```
194             (*atual)->tarefas[i] = proxima->tarefas[i];
195             if (i < QUANT_AGENTES)
196                (*atual)->excesso[i] = proxima->excesso[i];
197          }
198
199          (*atual)->avaliacao = proxima->avaliacao;
200          (*atual)->custo     = proxima->custo;
201
202          // Retorna true se tiver aceitado.
203          return 1;
204       } else {
205          return 0;
206       }
207    }
208
209
210    /*
211     * Procedimento que copia as informações de uma solução para a outra de forma
212     *   a aceitar aquela solução.
213     */
214    void Aceita_Nova_Solucao(Solucao ** atual, Solucao * proxima) {
215       int i = 0;
216
217       // Copia a solução
218       for (i = 0; i < QUANT_TAREFAS; i++) {
219          (*atual)->tarefas[i] = proxima->tarefas[i];
220          if (i < QUANT_AGENTES)
221             (*atual)->excesso[i] = proxima->excesso[i];
222       }
223
224       (*atual)->avaliacao = proxima->avaliacao;
225       (*atual)->custo     = proxima->custo;
226    }
227
228
229
230
231    /*
232     * Procedimento que realiza a leitura da instância problema.
233     */
234    void Le_Instancia(char * path) {
235       int i, j;
236       FILE * f = fopen(path, "r");
237
238       if (f) {
239          fscanf(f, "%d", &QUANT_AGENTES);
240          fscanf(f, "%d", &QUANT_TAREFAS);
241
```

```
242         if (QUANT_AGENTES < 1 || QUANT_TAREFAS < 1) {
243            printf("Valores da Instância Negativos!");
244            exit(2);
245         }
246
247         CAPAC_AGENTES = calloc(QUANT_AGENTES, sizeof (int));
248
249         RECURSOS_A_T = calloc(QUANT_AGENTES, sizeof (int*));
250         CUSTO_A_T = calloc(QUANT_AGENTES, sizeof (int*));
251
252         for (i = 0; i < QUANT_AGENTES; i++) {
253            RECURSOS_A_T[i] = calloc(QUANT_TAREFAS, sizeof (int));
254            CUSTO_A_T[i] = calloc(QUANT_TAREFAS, sizeof (int));
255         }
256
257         for (i = 0; i < QUANT_AGENTES; i++) {
258            for (j = 0; j < QUANT_TAREFAS; j++) {
259               fscanf(f, "%d", &CUSTO_A_T[i][j]);
260            }
261         }
262
263         for (i = 0; i < QUANT_AGENTES; i++) {
264            for (j = 0; j < QUANT_TAREFAS; j++) {
265               fscanf(f, "%d", &RECURSOS_A_T[i][j]);
266            }
267         }
268
269         for (i = 0; i < QUANT_AGENTES; i++)
270            fscanf(f, "%d", &CAPAC_AGENTES[i]);
271
272         fclose(f);
273
274      } else {
275         printf("Erro ao ler Instância!\n");
276         exit(-2);
277      }
278
279   }
280
281
282
283
284   /*
285    * Procedimento que informa o resultado obtido.
286    */
287   void Imprime_Solucao(Solucao * ind) {
288
289      fprintf(out, "\n\t%10lf\t", ind->custo);
```

```
290     printf("\n\t%10lf\t", ind->custo);
291
292     /*for (i = 0; i < QUANT_TAREFAS; i++) {
293        fprintf(out, "%3d ", ind->tarefas[i]);
294        printf("%3d ", ind->tarefas[i]);
295     }*/
296  }
297
298
299
300  /*
301   * Procedimento de impressão do status atual da execução.
302   */
303  void Imprime_Status(double t, Solucao * m) {
304     int i, sum = 0;
305
306     for (i = 0; i < QUANT_AGENTES; i++)
307        sum += m->excesso[i];
308
309     //printf("I:%7d\t", it);
310     printf("T:%7.4lf\t", t);
311     printf("Best(c,a,sum):%9.0lf|%9.0lf|%d\t", m->custo, m->avaliacao, sum);
312
313     //printf("AVA(c,a):At:%9.0lf|%9.0lf\tProx:%9.0lf|%9.0lf", a->custo,
        ↪  a->avaliacao, p->custo, p->avaliacao);
314     printf("\n");
315  }
316
317
318
319  /*
320   * Procedimento que instancia uma nova solução com seus valores totalmente
321   *   aleatórios.
322   */
323  Solucao * Instancia_Solucao_Aleatoria() {
324     int i;
325     Solucao * s;
326
327     s = calloc(1, sizeof(Solucao));
328
329     s->tarefas = calloc(QUANT_TAREFAS, sizeof(int));
330     s->excesso = calloc(QUANT_AGENTES, sizeof(int));
331
332     for (i = 0; i < QUANT_TAREFAS; i++) {
333        s->tarefas[i] = random() % QUANT_AGENTES;
334     }
335
336     Avalia_Solucao(s);
```

```
337
338     return s;
339 }
340
341
342
343 /*
344  * Procedimento que libera a memória da solução utilizada.
345  */
346 void Desaloca_Solucao(Solucao ** p) {
347
348     free((*p)->tarefas);
349     free((*p)->excesso);
350
351     free(*p);
352 }
353
354
355
356 /*
357  * Procedimento que realiza liberação de memória completa da população P
358  */
359 void Desaloca_Populacao(Solucao *** p) {
360     int i = 0;
361
362     // Desaloca uma população inteira
363     for (i = 0; i < TAM_POP; i++) {
364         if ((*p)[i] != NULL) {
365             free((*p)[i]->tarefas);
366             free((*p)[i]->excesso);
367         }
368     }
369
370     free(*p);
371 }
372
373
374
375 /*
376  * Imprime a Instância lida
377  */
378 void Imprime_Instancia() {
379     int i = 0, j;
380
381     printf("%d %d\n", QUANT_AGENTES, QUANT_TAREFAS);
382
383     for (i = 0; i < QUANT_AGENTES; i++) {
384         for (j = 0; j < QUANT_TAREFAS; j++)
```

```
385             printf("%d ", CUSTO_A_T[i][j]);
386         printf("\n");
387     }
388     printf("\n");
389 
390     for (i = 0; i < QUANT_AGENTES; i++) {
391         for (j = 0; j < QUANT_TAREFAS; j++)
392             printf("%d ", RECURSOS_A_T[i][j]);
393         printf("\n");
394     }
395 
396     printf("\n");
397 
398     for (i = 0; i < QUANT_AGENTES; i++)
399         printf("%d ", CAPAC_AGENTES[i]);
400 
401 
402     printf("\n");
403 }
```

11.3.3 Algoritmo Genético

```
1  /*
2   * Trabalho de Projeto e Análise de Algoritmo
3   * Período 16.1
4   *
5   * Desenvolver Metaheurísticas para o Problema de Alocação Generalizada
6   *
7   * Algoritmo: Genético.
8   * Data: 01/08/2016.
9   * Distribuição Livre, desde que referenciando o autor.
10  *
11  * Professor: Haroldo Santos
12  *
13  * Autor do Trabalho: Rodolfo Labiapari Mansur Guimarães
14  */
15 
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <time.h>
19 #include "gap.h"
20 
21 
22 
23 /*
24  * Procedimento que realiza a alocação de memória para novas instâncias.
```

```
25   *
26   * O procedimento instancia a população, mas não os indivíduos.
27   */
28  void Instancia_Populacoes(Solucao *** pop) {
29     (*pop) = calloc(TAM_POP, sizeof (Solucao*));
30  }
31
32
33
34  /*
35   * Procedimento que imprime os dados de toda a população.
36   */
37  void Imprime_Populacao (Solucao ** p) {
38     int i = 0, j = 0;
39     if (IMPRIMIR) {
40
41        printf("\n\n");
42
43        for (i = 0; i < TAM_POP; i++) {
44           printf("[%3d] - ", i);
45           if (p[i] != NULL) {
46              for (j = 0; j < QUANT_TAREFAS; j++) {
47                 printf("%1d  ", p[i]->tarefas[j]);
48              }
49
50              printf("\t%10.1lf\n", p[i]->avaliacao);
51           } else
52              printf("\n");
53
54           fflush(stdout);
55        }
56
57        fprintf(out, "\n\n");
58
59        for (i = 0; i < TAM_POP; i++) {
60           fprintf(out, "[%3d] - ", i);
61           if (p[i] != NULL) {
62              for (j = 0; j < QUANT_TAREFAS; j++) {
63                 fprintf(out, "%1d  ", p[i]->tarefas[j]);
64              }
65
66              fprintf(out, "\t%10.1lf\n", p[i]->avaliacao);
67           } else
68              fprintf(out, "\n");
69
70           fflush(out);
71        }
72     }
```

```
73  }
74
75
76
77  /*
78   * Procedimento que imprime de forma inteligente, quais índices da população
79   *   estão preenchidos e quais estão vazios.
80   */
81  void Imprime_Dados_Populacao (Solucao ** p) {
82     int i = 0;
83
84     printf("\n\n");
85
86     for (i = 0; i < TAM_POP; i++) {
87        printf("[%5d] ", i);
88        printf("c%5.0lf / sum = a%12.6lf", p[i]->custo, p[i]->avaliacao);
89
90        if (p[i]->custo > 5 * 209) {
91           printf(" X\n");
92        } else {
93           printf("\n");
94        }
95     }
96  }
97
98
99
100 /*
101  * Procedimento que realiza a leitura dos parâmetros de configuração do
102  *   algoritmo.
103  */
104 void Le_Parametros(char * conf) {
105    FILE * f = 0;
106
107    f = fopen(conf, "r");
108
109    if (f) {
110       fscanf(f, "%d", &TAM_POP);
111       fscanf(f, "%f", &TAX_CRUZAM);
112       fscanf(f, "%f", &TAX_MUT);
113       fscanf(f, "%d", &SECONDS);
114
115       fclose(f);
116
117    } else {
118       printf("Erro ao ler Configuração!\n");
119       exit(-1);
120    }
```

```
121 }
122
123
124
125 /*
126  * Procedimento que seleciona indivíduos não repetidos aleatoriamente da
127  *   população anterior adicionando-os na nova e eliminando os indivíduos
128  *   rejeitados.
129  */
130 void Seleciona_Nova_Geracao(Solucao *** atual, Solucao *** proxima) {
131    int i = 0, buffer[TAM_POP], indice = 0;
132    Solucao ** ind_buffer = 0;
133
134    // Inicializa o buffer de usados como 'nenhum item utilizado'
135    for (i = 0; i < TAM_POP; i++)
136       buffer[i] = 0;
137
138    // Para cada vagas da próxima população ainda não preenchida
139    for (i = (int) TAM_POP * TAX_CRUZAM + 1; i < TAM_POP; i++) {
140
141       // Escolhe um indivíduos da antiga população que ainda não foram
142          // escolhidos (a fim de não gerar uma população com indivíduos
143          // idênticos.
144       do {
145          indice = random() % TAM_POP;
146       } while (buffer[indice] == 1);
147
148       // Define-o como utilizado.
149       buffer[indice] = 1;
150       // Referencia-o na nova população.
151       (*proxima)[i] = (*atual)[indice];
152    }
153
154    // Assim, alguns indivíduos não serão referenciados e por isso devem ser
155       // eliminados
156    // Para cada item da população antiga
157    for (i = 0; i < TAM_POP; i++) {
158       // Verifica se o item não foi escolhido.
159       if (buffer[i] == 0) {
160          // Se sim, elimina-o
161          free((*atual)[i]->tarefas);
162          (*atual)[i]->tarefas = 0;
163          (*atual)[i]->avaliacao = -1;
164       }
165
166       // Reseta a população antiga para se tornar um 'proxima população'
167       (*atual)[i] = NULL;
168    }
```

```
169
170     // Comuta as populações.
171     ind_buffer = (*atual);
172     (*atual) = (*proxima);
173     (*proxima) = ind_buffer;
174 }
175
176
177
178 /*
179  * Procedimento que copia os dados de um indivíduo para outro.
180  */
181 void Copia_Melhor_Solucao(Solucao ** p, Solucao ** best){
182     int i = 0;
183     Solucao * best_pop_local = 0;
184
185     // Inicia-se definido que o melhor é o primeiro indivíduo.
186     best_pop_local = p[0];
187
188     // Compara-se o primeiro com os outros de forma a escolher o melhor
189        // indivíduo de toda a população
190     for (i = 1; i < TAM_POP; i++) {
191        if (p[i]->avaliacao < best_pop_local->avaliacao)
192           best_pop_local = p[i];
193     }
194
195     // Se o indivíduo best ainda não foi criado, cria-o
196     if (*best == NULL) {
197        // Cria indivíduo
198        *best = calloc (1, sizeof(Solucao));
199        (*best)->tarefas = calloc(QUANT_TAREFAS, sizeof(int));
200        (*best)->excesso = calloc(QUANT_AGENTES, sizeof(int));
201
202        // Copia do melhor encontrado
203        (*best)->avaliacao = best_pop_local->avaliacao;
204        (*best)->custo = best_pop_local->custo;
205
206        // Copia os dados 'tarefa' e 'excesso'
207        for (i = 0; i < QUANT_TAREFAS; i++) {
208           (*best)->tarefas[i] = best_pop_local->tarefas[i];
209           if (i < QUANT_AGENTES)
210              (*best)->excesso[i] = best_pop_local->excesso[i];
211        }
212
213     } else {
214        // verifica se o melhor encontrado é melhor que o indivíduo atual
215        if ((*best)->avaliacao > 2 * 1000000 ||
216              ((best_pop_local->avaliacao < 2 * 1000000) &&
```

```
217             (best_pop_local->custo < (*best)->custo))) {
218
219          // Copia os dados 'tarefa' e 'excesso'//
220          for (i = 0; i < QUANT_TAREFAS; i++) {
221             (*best)->tarefas[i] = best_pop_local->tarefas[i];
222             if (i < QUANT_AGENTES)
223                (*best)->excesso[i] = best_pop_local->excesso[i];
224          }
225
226          // Copia do melhor indivíduo
227          (*best)->avaliacao = best_pop_local->avaliacao;
228          (*best)->custo      = best_pop_local->custo;
229       }
230    }
231 }
232
233
234
235 /*
236  * Procedimento que recombina dois indivíduos gerando um terceiro por meio de
237  *   recombinação uniforme.
238  */
239 int * Recombina(Solucao * i1, Solucao * i2) {
240    int i = 0, * tarefas = 0;
241
242    // Cria um novo vetor de tarefas
243    tarefas = calloc(QUANT_TAREFAS, sizeof(int));
244
245    // Recombina de forma uniforme
246    for (i = 0; i < QUANT_TAREFAS; i++) {
247
248       // Escolhe de forma uniforme sobre dois indivíduos
249       if (random() % 2)
250          tarefas[i] = i1->tarefas[i];
251       else
252          tarefas[i] = i2->tarefas[i];
253
254    }
255
256    return tarefas;
257 }
258
259
260
261 /*
262  * Procedimento que cria vários indivíduos aleatórios preenchendo a população.
263  *   Além disso, é realizado a avaliação de cada um destes.
264  */
```

```
265 void Cria_Nova_Populacao(Solucao *** P) {
266    int i = 0, j = 0, k = 0, menor = 0;
267    Solucao ** p_local = 0;
268 
269    p_local = *P;
270 
271    // Para cada item a ser criado
272    for (i = 0; i < TAM_POP; i++) {
273 
274       // Aloca suas variáveis que armazenarão suas informações
275       p_local[i]           = calloc(1, sizeof(Solucao));
276       p_local[i]->excesso = calloc(QUANT_AGENTES, sizeof(int));
277       p_local[i]->tarefas = calloc(QUANT_TAREFAS, sizeof(int));
278 
279       // Gera valores pra este indivíduo
280       for (j = 0; j < QUANT_TAREFAS; j++) {
281          // O primeiro indivíduo será gerado de forma gulosa e os outros
282             // Serão uma mistura de Guloso com Aleatoriedade
283 
284          // Se não for o primeiro indivíduo, possui 66% de gerar valores
285             // por meio de função randomica
286          if (i > 0 && random() % 3 != 0) {
287             p_local[i]->tarefas[j] = random() % QUANT_AGENTES;
288 
289          // Caso contrário, utiliza uma geração gulosa pra esta tarefa.
290          } else {
291             // O método guloso escolhe o recurso mais leve desta tarefa
292             p_local[i]->tarefas[j] = 0;
293             menor = 0;
294 
295             // Seleciona o agente que utiliza o menor recurso desta
296                // tarefa
297             for (k = 1; k < QUANT_AGENTES; k++) {
298                if (RECURSOS_A_T[k][j] < RECURSOS_A_T[menor][j]) {
299                   menor = k;
300                   p_local[i]->tarefas[j] = k;
301                }
302             }
303          }
304       }
305 
306       // Avalia o novo indivíduo gerado
307       Avalia_Solucao(p_local[i]);
308    }
309 }
310 
311 
312 
```

```
313 /*
314  * Procedimento que realiza a seleção de dois indivíduos para a geração de um
315  *   terceiro.
316  */
317 void Gera_Filhos(Solucao ** selecao, Solucao *** filhos) {
318    int i = 0, j = 0, numero_torneio = 0;
319    Solucao * i1 = 0, * i2 = 0, * buffer = 0;
320 
321    Solucao ** filhos_local = * filhos;
322 
323    // O primeiro indivíduo gerado é o indivíduo com melhor avaliação de toda
324       // população. Sendo assim, é realizado uma busca na população do
325       // melhor indivíduo e adiciona-o na próxima geração.
326    j = 0;
327    for (i = 1; i < TAM_POP; i++)
328       if (selecao[i]->avaliacao < selecao[j]->avaliacao)
329          j = i;
330 
331    // Aloca memória pra nova solução
332    filhos_local[0]            = calloc(1, sizeof(Solucao));
333    filhos_local[0]->excesso   = calloc(QUANT_AGENTES, sizeof(int));
334    filhos_local[0]->tarefas   = calloc(QUANT_TAREFAS, sizeof(int));
335 
336    filhos_local[0]->avaliacao = selecao[j]->avaliacao;
337 
338    // Copia as tarefas do elemento escolhido
339    for (i = 0; i < QUANT_TAREFAS; i++)
340       filhos_local[0]->tarefas[i] = selecao[j]->tarefas[i];
341 
342 
343    // Para os outros indivíduos que devem ser gerados, serão selecionados dois
344       // pais por torneio e realizados a sua recombinação
345    for (i = 1; i < (int) TAM_POP * TAX_CRUZAM + 1; i++) {
346 
347       // Quantidade de indivíduos que disputarão o torneio
348       numero_torneio = 1 + random() % ((TAM_POP / 2) + 1);
349 
350       // Aloca o novo filho ainda vazio.
351       filhos_local[i] = calloc(1, sizeof(Solucao));
352       filhos_local[i]->excesso = calloc(QUANT_AGENTES, sizeof(int));
353 
354       // Realiza o torneio 1
355       i1 = selecao[random() % TAM_POP];
356 
357       // Realiza torneio pra seleção do primeiro pai
358       j = 0;
359       while (j++ < numero_torneio) {
360          buffer = selecao[random() % TAM_POP];
```

```
361
362             if (buffer->avaliacao < i1->avaliacao)
363                 i1 = buffer;
364         }
365
366         // Realiza o torneio 2
367         i2 = selecao[random() % TAM_POP];
368
369         // Realiza torneio pra seleção do segundo pai
370         j = 0;
371         while (j++ < numero_torneio) {
372             buffer = selecao[random() % TAM_POP];
373
374             if (buffer->avaliacao < i2->avaliacao)
375                 i2 = buffer;
376         }
377
378         // Verifica se os dois pais são idênticos, caso sim, gera um outro
379             // pai aleatório sem torneio e seleciona-o
380         if (i1 == i2)
381             i2 = selecao[random() % TAM_POP];
382
383         // Recombina os dois indivíduos
384         filhos_local[i]->tarefas = Recombina(i1, i2);
385     }
386 }
387
388
389
390 /*
391  * Procedimento que realiza a mutação dos filhos por meio do processo Creep
392  *   Mutation.
393  *
394  * Não são todos os filhos que são mutados. Eles são escolhidos aleatoriamente
395  *   e TAX_MUT representa a porcentagem de mutação que cada indivíduo receberá.
396  * Ao final, este novo é avaliado.
397  */
398 void Creep_Mutation(Solucao *** pop) {
399     int i = 0, j = 0, indice_tarefa = 0, quant_filhos = 0, agente_atual = 0;
400
401     // Quantidade de filhos gerados nesta próxima geração
402     quant_filhos = (int) TAM_POP * TAX_CRUZAM + 1;
403
404     // Para cada filho
405     for (i = 0; i < quant_filhos; i++) {
406
407         // Realiza mutação em uma porcentagem de do indivíduo
408         for (j = 0; j < (int) QUANT_TAREFAS * TAX_MUT; j++) {
```

```
409
410             // Muta a tarefa do indivíduo
411             indice_tarefa = random() % QUANT_TAREFAS;
412
413             agente_atual = (*pop)[i]->tarefas[indice_tarefa];
414
415             do {
416                 (*pop)[i]->tarefas[indice_tarefa] = random() % QUANT_AGENTES;
417             } while ((*pop)[i]->tarefas[indice_tarefa] == agente_atual);
418         }
419
420         // Avalia o novo indivíduo gerado.
421         Avalia_Solucao((*pop)[i]);
422     }
423 }
424
425
426
427 /*
428  * Processo de geração de filhos e mutação.
429  */
430 void Recombinacao(Solucao ** atual_p, Solucao *** proxima_p) {
431
432     // Gera filhos da próxima geraçãp
433     Gera_Filhos(atual_p, proxima_p);
434
435     // Muta os filhos gerados
436     Creep_Mutation(proxima_p);
437 }
438
439
440 /*
441  * Algoritmo Genético segundo Lopes 2008.
442  */
443 Solucao * Algoritmo_Genetico() {
444     Solucao ** p_atual = 0, ** p_proxima = 0;
445     Solucao * melhor_Solucao = 0;
446
447     // Instancia as duas populações
448     Instancia_Populacoes(&p_atual);
449     Instancia_Populacoes(&p_proxima);
450
451     // Gera a população inicial para execução
452     Cria_Nova_Populacao(&p_atual);
453
454     // Copia o melhor indivíduo gerado aleatoriamnte
455     Copia_Melhor_Solucao(p_atual, &melhor_Solucao);
456
```

```
457     // Inicia o contador de tempo
458     start   = time(NULL);
459      endwait = start + SECONDS;
460 
461     do {
462        // Realiza a recombinação gerando novos filhos já mutados
463        Recombinacao(p_atual, &p_proxima);
464 
465        // Completa a população com indivíduos da população anterior
466        Seleciona_Nova_Geracao(&p_atual, &p_proxima);
467 
468        // Verifica a existência de um indivíduo melhor que o atual conhecido
469        Copia_Melhor_Solucao(p_atual, &melhor_Solucao);
470 
471        // Recebe o tempo atual.
472        start = time(NULL);
473 
474        // Verifica se o tempo excedeu o limite estabelecido por parâmentro.
475     } while (start < endwait);
476 
477     // Desaloca populações após o fim do algoritmo
478     Desaloca_Populacao(&p_atual);
479     Desaloca_Populacao(&p_proxima);
480 
481     // Retorna o melhor indivíduo encontrado
482     return melhor_Solucao;
483 }
484 
485 
486 
487 /*
488  * Procedimento principal
489  */
490 int main(int argc, char** argv) {
491     Solucao * solve;
492     printf("\n/*");
493     printf("\n * Trabalho de Projeto e Análise de Algoritmo");
494      printf("\n * Período 16.1");
495      printf("\n * ");
496      printf("\n * Desenvolver Metaheurísticas para o Problema de Alocação
       ↪  Generalizada");
497      printf("\n * ");
498      printf("\n * Algoritmo: Genético.");
499      printf("\n * Data: 01/08/2016.");
500      printf("\n * Distribuição Livre, desde que referenciando o autor.");
501      printf("\n * ");
502      printf("\n * Professor: Haroldo Santos");
503      printf("\n * ");
```

```
504      printf("\n * Autor do Trabalho: Rodolfo Labiapari Mansur Guimarães");
505      printf("\n */");
506
507     if (argc != 4) {
508        printf("\n\nErro nos parâmetros! Quantidade lida: %d\t Quantidade
          ↪  requerida: %d.", argc, 4);
509        printf("\nnome_programa arq_configuracao arq_instancia seed\n\n");
510        exit(-1);
511     }
512      //printf("\n\nExecutando...\n");
513
514     char * instancia = argv[2];
515     char * configuracao = argv[1];
516     srand(atoi(argv[3]));
517
518     Le_Instancia(instancia);
519     Le_Parametros(configuracao);
520
521     out = fopen ("out_genetico.txt", "a");
522
523     solve = Algoritmo_Genetico();
524
525     Imprime_Solucao(solve);
526
527     free(solve->tarefas);
528     free(solve);
529
530     fflush(out);
531     fclose(out);
532
533     fflush(stdout);
534
535     return (EXIT_SUCCESS);
536 }
```

11.3.4 Algoritmo Recozimento Simulado

```
1  /*
2   * Trabalho de Projeto e Análise de Algoritmo
3   * Período 16.1
4   *
5   * Desenvolver Metaheurísticas para o Problema de Alocação Generalizada
6   *
7   * Algoritmo: Simulated Annealing.
8   * Data: 01/08/2016.
9   * Distribuição Livre, desde que referenciando o autor.
```

```
10  *
11  * Professor: Haroldo Santos
12  *
13  * Autor do Trabalho: Rodolfo Labiapari Mansur Guimarães
14  */
15
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <math.h>
19 #include <time.h>
20 #include "gap.h"
21
22 int TEMPERATURA = 0;
23
24
25 /*
26  * Procedimento que realiza a leitura dos parâmetros de configuração do
27  *   algoritmo.
28  */
29 void Le_Parametros(char * conf) {
30    FILE * f;
31
32    f = fopen(conf, "r");
33
34    if (f) {
35       fscanf(f, "%d", &TEMPERATURA);
36       fscanf(f, "%d", &MAXIteracoes);
37
38       fclose(f);
39
40    } else {
41       printf("Erro ao ler Configuração!\n");
42       exit(-1);
43    }
44 }
45
46
47
48 /*
49  * Procedimento que realiza a alteração da temperatura por meio de método
50  *   logaritmo.
51  */
52 void Atualiza_Temperatura(double * t) {
53    *t = 0.995 * *t;
54 }
55
56
57
```

```
58 /*
59  * Procedimento de recozimento simulado, baseado no Lopes 2008.
60  */
61 Solucao * Simulated_Annealing() {
62    int iteracoes = 0;
63    double fator_Boltzmann = 0, temperatura = 0, delta = 0, condicao_parada = 0,
    ↪  aceitacao_aleatoria = 0;
64    Solucao * melhor_s = 0, * atual_s = 0, * possivel_s = 0;
65 
66    // Define valores iniciais
67    temperatura     = TEMPERATURA;
68    condicao_parada = 0.2;
69 
70    // Instancia soluções aleatórias para início de execução
71    melhor_s   = Instancia_Solucao_Aleatoria();
72    atual_s    = Instancia_Solucao_Aleatoria();
73    possivel_s = Instancia_Solucao_Aleatoria();
74 
75    // verifica se alguma solução aleatória gerada é boa
76    Teste_Aceita_Nova_Solucao(0, &melhor_s, atual_s);
77    Teste_Aceita_Nova_Solucao(0, &melhor_s, possivel_s);
78 
79    // Enquanto tiver temperatura suficiente
80    while (temperatura > condicao_parada) {
81 
82       // Aperfeiçoa a solução desta temperatura.
83       while(iteracoes++ < MAXIteracoes) {
84          // Gera um novo vizinho
85          Gera_Vizinho(atual_s, &possivel_s);
86 
87          delta = possivel_s->avaliacao - atual_s->avaliacao;
88 
89          // verifica se a solução atual é válida
90          if (delta < 0) {
91             // Se sim aceita.
92             Aceita_Nova_Solucao(&atual_s, possivel_s);
93 
94             // Verifica se é melhor que a melhor
95             Teste_Aceita_Nova_Solucao(temperatura, &melhor_s, possivel_s);
96 
97          } else {
98             // calcula fator Boltzmann
99             aceitacao_aleatoria = random() / ((double)(RAND_MAX));
100            fator_Boltzmann = exp(- (delta / (double) temperatura));
101 
102            if (aceitacao_aleatoria < fator_Boltzmann) {
103               Aceita_Nova_Solucao(&atual_s, possivel_s);
104            }
```

```
105             }
106         }
107 
108         // Atualiza temperatura
109         Atualiza_Temperatura(&temperatura);
110 
111         //Imprime_Status(temperatura, melhor_s);
112 
113         iteracoes = 0;
114     }
115 
116     // Desaloca soluções
117     Desaloca_Solucao(&atual_s);
118     Desaloca_Solucao(&possivel_s);
119 
120     return melhor_s;
121 }
122 
123 
124 /*
125  * Procedimento principal
126  */
127 int main(int argc, char** argv) {
128     Solucao * solve;
129 
130     printf("\n/*");
131     printf("\n * Trabalho de Projeto e Análise de Algoritmo");
132      printf("\n * Período 16.1");
133      printf("\n * ");
134      printf("\n * Desenvolver Metaheurísticas para o Problema de Alocação
         ↪  Generalizada");
135      printf("\n * ");
136      printf("\n * Algoritmo: Simulated Annealing.");
137      printf("\n * Data: 01/08/2016.");
138      printf("\n * Distribuição Livre, desde que referenciando o autor.");
139      printf("\n * ");
140      printf("\n * Professor: Haroldo Santos");
141      printf("\n * ");
142      printf("\n * Autor do Trabalho: Rodolfo Labiapari Mansur Guimarães");
143      printf("\n */");
144 
145     if (argc != 4) {
146         printf("\n\nErro nos parâmetros! Quantidade lida: %d\t Quantidade
          ↪  requerida: %d.", argc, 4);
147         printf("\nnome_programa arq_configuracao arq_instancia seed\n\n");
148         exit(-1);
149     }
150 
```

```
151      //printf("\n\nExecutando...\n");
152
153     char * instancia = argv[2];
154     char * configuracao = argv[1];
155     srand(atoi(argv[3]));
156
157     Le_Instancia(instancia);
158     Le_Parametros(configuracao);
159
160     out = fopen ("out_simulated_annealing.txt", "a");
161
162     solve = Simulated_Annealing();
163
164     Imprime_Solucao(solve);
165
166     Desaloca_Solucao(&solve);
167     fclose(out);
168
169     return (EXIT_SUCCESS);
170 }
```

11.3.5 Método Reinício

```
1  /*
2   * Trabalho de Projeto e Análise de Algoritmo
3   * Período 16.1
4   *
5   * Desenvolver Metaheurísticas para o Problema de Alocação Generalizada
6   *
7   * Algoritmo: Guloso.
8   * Data: 01/08/2016.
9   * Distribuição Livre, desde que referenciando o autor.
10  *
11  * Professor: Haroldo Santos
12  *
13  * Autor do Trabalho: Rodolfo Labiapari Mansur Guimarães
14  */
15
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <math.h>
19 #include <limits.h>
20 #include <time.h>
21 #include "gap.h"
22
23
```

```
24
25  /*
26   * Procedimento que realiza a leitura dos parâmetros de configuração do
27   *   algoritmo.
28   */
29  void Le_Parametros(char * config) {
30     FILE * f;
31
32     f = fopen(config, "r");
33
34     if (f) {
35        fscanf(f, "%d", &SECONDS);
36        fscanf(f, "%d", &MAXIteracoes);
37
38        fclose(f);
39
40     } else {
41        printf("Erro ao ler Configuração!\n");
42        exit(-1);
43     }
44  }
45
46
47  /*
48     Método de reinício:
49        1 - Gera uma solução aleatória S
50        2 - Pesquisa em uma vizinhança N(S) por uma solução melhor.
51           Se a melhor solução S' pertencente a N(S) é melhor do que S, então S =
    ↪  S', volte para passo 2.
52        3 - Atualize a melhor solução encontrada até o momento (Solução
    ↪  inculbente)
53        4 - Se houver tempo, volta para passo 1.
54   */
55  Solucao * Metodo_reinicio() {
56     int i = 0;
57     Solucao * melhor_global = 0, * atual_s = 0, * vizinha_s = 0;
58
59     melhor_global  = Instancia_Solucao();
60     atual_s        = Instancia_Solucao();
61     vizinha_s      = Instancia_Solucao();
62
63     // Inicia o contador de tempo
64     start   = time(NULL);
65      endwait = start + SECONDS;
66
67     do {
68        // Gera soluções aleatórias
69        Gera_Solucao_Aleatoria(&atual_s);
```

```
70          //Gera_Solucao_Aleatoria(&vizinha_s);
71
72          // Testa se a solução é melhor que a atual
73          Teste_Aceita_Nova_Solucao(endwait - start, &melhor_global, atual_s);
74          //Teste_Aceita_Nova_Solucao(endwait - start, &melhor_global, vizinha_s);
75
76          // Realiza MAXIteracoes de vizinhos a procura de soluções
77             // melhoras que a atual.
78          for (i = 0; i < MAXIteracoes; i++) {
79             Gera_Vizinho(atual_s, &vizinha_s);
80
81             if (vizinha_s->avaliacao < atual_s->avaliacao)
82                Aceita_Nova_Solucao(&atual_s, vizinha_s);
83          }
84
85          // Verifica se na procura de soluções vizinhas, algoma é boa
86          Teste_Aceita_Nova_Solucao(endwait - start, &melhor_global, atual_s);
87
88          // Atualiza o tempo
89          start = time(NULL);
90
91          //Imprime_Status((double) endwait - start, melhor_global);
92
93       } while (start < endwait);
94
95       // Desaloca solução
96       Desaloca_Solucao(&atual_s);
97       Desaloca_Solucao(&vizinha_s);
98
99       return melhor_global;
100   }
101
102   /*
103    * Procedimento principal
104    */
105   int main(int argc, char** argv) {
106      Solucao * solve;
107
108      printf("\n/*");
109      printf("\n * Trabalho de Projeto e Análise de Algoritmo");
110       printf("\n * Período 16.1");
111       printf("\n * ");
112       printf("\n * Desenvolver Metaheurísticas para o Problema de Alocação
          ↪  Generalizada");
113       printf("\n * ");
114       printf("\n * Algoritmo: Guloso.");
115       printf("\n * Data: 01/08/2016.");
116       printf("\n * Distribuição Livre, desde que referenciando o autor.");
```

```
117      printf("\n * ");
118      printf("\n * Professor: Haroldo Santos");
119      printf("\n * ");
120      printf("\n * Autor do Trabalho: Rodolfo Labiapari Mansur Guimarães");
121      printf("\n */");
122
123     if (argc != 4) {
124        printf("\n\nErro nos parâmetros! Quantidade lida: %d\t Quantidade
          ↪  requerida: %d.", argc, 4);
125        printf("\nnome_programa arq_configuracao arq_instancia seed\n\n");
126        exit(-1);
127     }
128
129      //printf("\n\nExecutando...\n");
130
131     char * instancia = argv[2];
132     char * configuracao = argv[1];
133     srand(atoi(argv[3]));
134
135     Le_Instancia(instancia);
136
137     Le_Parametros(configuracao);
138
139     out = fopen ("out_reinicio.txt", "a");
140
141     solve = Metodo_reinicio();
142
143     Imprime_Solucao(solve);
144     Desaloca_Solucao(&solve);
145     fclose(out);
146
147     return (EXIT_SUCCESS);
148 }
```

11.3.6 Algoritmo GRASP

```
1  /*
2   * Trabalho de Projeto e Análise de Algoritmo
3   * Período 16.1
4   *
5   * Desenvolver Metaheurísticas para o Problema de Alocação Generalizada
6   *
7   * Algoritmo: GRASP.
8   * Data: 01/08/2016.
9   * Distribuição Livre, desde que referenciando o autor.
10  *
```

```
11  * Professor: Haroldo Santos
12  *
13  * Autor do Trabalho: Rodolfo Labiapari Mansur Guimarães
14  */
15
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <math.h>
19 #include <limits.h>
20 #include <time.h>
21 #include "gap.h"
22
23
24 /*
25  * Procedimento que realiza a leitura dos parâmetros de configuração do
26  *   algoritmo.
27  */
28 void Le_Parametros(char * config) {
29    FILE * f;
30
31    f = fopen(config, "r");
32
33    if (f) {
34       fscanf(f, "%d", &SECONDS);
35       fscanf(f, "%d", &MAXIteracoes);
36
37       fclose(f);
38
39    } else {
40       printf("Erro ao ler Configuração!\n");
41       exit(-1);
42    }
43 }
44
45
46
47 /*
48  * Método Guloso Randômico
49  */
50 void GreedyRandomizedConstruction(Solucao ** s, float alfa) {
51    int i = 0, j = 0, min = 0, max = 0, fator = 0, sum_recursos = 0;
52    char solucao_invalida = 0;
53
54    for (i = 0; i < QUANT_AGENTES; i++)
55       (*s)->excesso[i] = 0;
56
57    (*s)->custo = 0;
58
```

```
59      // Para cada tarefa
60      for (i = 0; i < QUANT_TAREFAS; i++) {
61         min = max = 0;
62
63         // Encontra os valores máximos e mínimos dos agentes
64         for (j = 1; j < QUANT_AGENTES; j++) {
65            if (RECURSOS_A_T[j][i] + CUSTO_A_T[j][i] < RECURSOS_A_T[min][i] +
               ↪  CUSTO_A_T[min][i])
66               min = j;
67
68            if (RECURSOS_A_T[j][i] + CUSTO_A_T[j][i] > RECURSOS_A_T[max][i] +
               ↪  CUSTO_A_T[max][i])
69               max = j;
70         }
71
72         // Calcula um fator de acordo com o valor alfa
73         fator = RECURSOS_A_T[min][i] + alfa * (RECURSOS_A_T[max][i] -
            ↪  RECURSOS_A_T[min][i]);
74
75         // procura o agente que tem maior proximidade com o fator
76         min = 0;
77         for (j = 1; j < QUANT_AGENTES; j++) {
78            if (abs(RECURSOS_A_T[j][i] - fator) < abs(RECURSOS_A_T[min][i] - fator))
79               min = j;
80         }
81
82         // Atribui a esta tarefa
83         (*s)->tarefas[i] = min;
84
85         // Calcula a quantidade de recusto utilizado ao atribuir a nova tarefa ao
            ↪  agente.
86         (*s)->excesso[min] += RECURSOS_A_T[min][i];
87
88         // Calcula o custo daquela tarefa
89         (*s)->custo += CUSTO_A_T[min][i];
90      }
91
92      // Verifica se a solução gerada é válida
93      for (j = 0; j < QUANT_AGENTES; j++) {
94         sum_recursos += (*s)->excesso[j];
95
96         if ((*s)->excesso[j] > CAPAC_AGENTES[j]) {
97            solucao_invalida = 1;
98         }
99      }
100
101     // Calcula o fator avaliação
102     if (solucao_invalida)
```

```
103         (*s)->avaliacao = ((double) sum_recursos ) * 1000000;
104      else {
105         (*s)->avaliacao = ((double) (*s)->custo);
106      }
107   }
108
109
110   /*
111    * Método GRASP
112    */
113   Solucao * GRASP() {
114      int i = 0;
115      Solucao * melhor_global = 0, * atual_s = 0, * vizinha_s = 0;
116
117      // Instancia soluções
118      melhor_global  = Instancia_Solucao();
119      atual_s        = Instancia_Solucao();
120      vizinha_s      = Instancia_Solucao();
121
122      // Inicia o contador de tempo
123      start   = time(NULL);
124       endwait = start + SECONDS;
125
126      do {
127         // Gera um novo indivíduo
128         GreedyRandomizedConstruction(&atual_s, random() / RAND_MAX);
129
130         // Verifica sua avaliação
131         Teste_Aceita_Nova_Solucao(endwait - start, &melhor_global, atual_s);
132
133         // Gera N vizinhos e verifica suas avaliações
134         for (i = 0; i < MAXIteracoes; i++) {
135            Gera_Vizinho(atual_s, &vizinha_s);
136
137            if (vizinha_s->avaliacao < atual_s->avaliacao)
138               Aceita_Nova_Solucao(&atual_s, vizinha_s);
139         }
140
141         // Testa se é a melhor solução encontrada
142         Teste_Aceita_Nova_Solucao(endwait - start, &melhor_global, atual_s);
143
144         // Verifica se ainda possui tempo
145         start = time(NULL);
146      } while (start < endwait);
147
148      Desaloca_Solucao(&atual_s);
149      Desaloca_Solucao(&vizinha_s);
150
```

```
151     return melhor_global;
152 }
153 
154 
155 
156 
157 /*
158  *
159  */
160 int main(int argc, char** argv) {
161     Solucao * solve;
162 
163     printf("\n/*");
164     printf("\n * Trabalho de Projeto e Análise de Algoritmo");
165      printf("\n * Período 16.1");
166      printf("\n * ");
167      printf("\n * Desenvolver Metaheurísticas para o Problema de Alocação
         ↪  Generalizada");
168      printf("\n * ");
169      printf("\n * Algoritmo: GRASP.");
170      printf("\n * Data: 01/08/2016.");
171      printf("\n * Distribuição Livre, desde que referenciando o autor.");
172      printf("\n * ");
173      printf("\n * Professor: Haroldo Santos");
174      printf("\n * ");
175      printf("\n * Autor do Trabalho: Rodolfo Labiapari Mansur Guimarães");
176      printf("\n */");
177 
178     if (argc != 4) {
179         printf("\n\nErro nos parâmetros! Quantidade lida: %d\t Quantidade
          ↪  requerida: %d.", argc, 4);
180         printf("\nnome_programa arq_configuracao arq_instancia seed\n\n");
181         exit(-1);
182     }
183      //printf("\n\nExecutando...\n");
184 
185     char * instancia = argv[2];
186     char * configuracao = argv[1];
187     srand(atoi(argv[3]));
188 
189     Le_Instancia(instancia);
190 
191     Le_Parametros(configuracao);
192 
193     out = fopen ("out_grasp.txt", "a");
194 
195     solve = GRASP();
196 
```

```
197     Imprime_Solucao(solve);
198     Desaloca_Solucao(&solve);
199     fclose(out);
200 
201     return (EXIT_SUCCESS);
202 }
```

Referências

[1] D Dasgupta, G Hernandez, and D Garrett. A comparison of multiobjective evolutionary algorithms with informed initialization and kuhn-munkres algorithm for the sailor assignment problem. *Proceedings of the 10th*, 2008.

[2] Heitor Silvério Lopes, Luiz Carlos de Abreu Rodrigues, and Maria Teresinha Arns Steiner. *Meta-heurísticas em Pesquisa Operacional*. Omnipax, 1 edition, 2013.

12 Anexos

Tabela 5: Tabela com todos os valores obtidos.

Algoritmo	Arquivo	Iteração/*Seed*	Agentes	Tarefas	Valor Encontrado	Valor Ótimo Literatura
GA	*a05100*	0	05	100	1698.000000	1698
GA	*a05100*	1	05	100	1698.000000	1698
GA	*a05100*	2	05	100	1698.000000	1698
GA	*a05100*	3	05	100	1698.000000	1698
GA	*a05100*	4	05	100	1698.000000	1698
GA	*a05100*	5	05	100	1698.000000	1698
GA	*a05100*	6	05	100	1698.000000	1698
GA	*a05100*	7	05	100	1698.000000	1698
GA	*a05100*	8	05	100	1698.000000	1698
GA	*a05100*	9	05	100	1698.000000	1698
GA	*a10100*	0	10	100	1360.000000	1360
GA	*a10100*	1	10	100	1360.000000	1360
GA	*a10100*	2	10	100	1360.000000	1360
GA	*a10100*	3	10	100	1360.000000	1360
GA	*a10100*	4	10	100	1360.000000	1360
GA	*a10100*	5	10	100	1360.000000	1360
GA	*a10100*	6	10	100	1360.000000	1360
GA	*a10100*	7	10	100	1360.000000	1360
GA	*a10100*	8	10	100	1360.000000	1360
GA	*a10100*	9	10	100	1360.000000	1360
GA	*a20100*	0	20	100	1158.000000	1158
GA	*a20100*	1	20	100	1158.000000	1158
GA	*a20100*	2	20	100	1158.000000	1158
GA	*a20100*	3	20	100	1158.000000	1158
GA	*a20100*	4	20	100	1158.000000	1158
GA	*a20100*	5	20	100	1158.000000	1158
GA	*a20100*	6	20	100	1158.000000	1158
GA	*a20100*	7	20	100	1158.000000	1158
GA	*a20100*	8	20	100	1158.000000	1158
GA	*a20100*	9	20	100	1158.000000	1158
GA	*a05200*	0	05	200	3235.000000	3235
GA	*a05200*	1	05	200	3235.000000	3235
GA	*a05200*	2	05	200	3235.000000	3235
GA	*a05200*	3	05	200	3235.000000	3235
GA	*a05200*	4	05	200	3235.000000	3235
GA	*a05200*	5	05	200	3235.000000	3235
GA	*a05200*	6	05	200	3235.000000	3235
GA	*a05200*	7	05	200	3235.000000	3235
GA	*a05200*	8	05	200	3235.000000	3235
GA	*a05200*	9	05	200	3235.000000	3235
GA	*a10200*	0	10	200	2623.000000	2623
GA	*a10200*	1	10	200	2623.000000	2623
GA	*a10200*	2	10	200	2623.000000	2623
GA	*a10200*	3	10	200	2623.000000	2623
GA	*a10200*	4	10	200	2623.000000	2623
GA	*a10200*	5	10	200	2623.000000	2623
GA	*a10200*	6	10	200	2623.000000	2623
GA	*a10200*	7	10	200	2623.000000	2623
GA	*a10200*	8	10	200	2623.000000	2623
GA	*a10200*	9	10	200	2623.000000	2623
GA	*a20200*	0	20	200	2339.000000	2339
GA	*a20200*	1	20	200	2339.000000	2339
GA	*a20200*	2	20	200	2339.000000	2339
GA	*a20200*	3	20	200	2339.000000	2339
GA	*a20200*	4	20	200	2339.000000	2339
GA	*a20200*	5	20	200	2339.000000	2339
GA	*a20200*	6	20	200	2339.000000	2339
GA	*a20200*	7	20	200	2339.000000	2339
GA	*a20200*	8	20	200	2339.000000	2339
GA	*a20200*	9	20	200	2339.000000	2339
GA	*c05100*	0	05	100	1982.000000	1931
GA	*c05100*	1	05	100	1982.000000	1931
GA	*c05100*	2	05	100	1982.000000	1931
GA	*c05100*	3	05	100	1982.000000	1931
GA	*c05100*	4	05	100	1982.000000	1931
GA	*c05100*	5	05	100	1982.000000	1931

GA	*c05100*	6	05	100	1982.000000	1931
GA	*c05100*	7	05	100	1982.000000	1931
GA	*c05100*	8	05	100	1982.000000	1931
GA	*c05100*	9	05	100	1982.000000	1931
GA	*c10100*	0	10	100	1439.000000	1402
GA	*c10100*	1	10	100	1439.000000	1402
GA	*c10100*	2	10	100	1439.000000	1402
GA	*c10100*	3	10	100	1439.000000	1402
GA	*c10100*	4	10	100	1439.000000	1402
GA	*c10100*	5	10	100	1439.000000	1402
GA	*c10100*	6	10	100	1439.000000	1402
GA	*c10100*	7	10	100	1439.000000	1402
GA	*c10100*	8	10	100	1439.000000	1402
GA	*c10100*	9	10	100	1439.000000	1402
GA	*c20100*	0	20	100	1281.000000	1243
GA	*c20100*	1	20	100	1281.000000	1243
GA	*c20100*	2	20	100	1281.000000	1243
GA	*c20100*	3	20	100	1281.000000	1243
GA	*c20100*	4	20	100	1281.000000	1243
GA	*c20100*	5	20	100	1281.000000	1243
GA	*c20100*	6	20	100	1281.000000	1243
GA	*c20100*	7	20	100	1281.000000	1243
GA	*c20100*	8	20	100	1282.000000	1243
GA	*c20100*	9	20	100	1282.000000	1243
GA	*c05200*	0	05	200	3552.000000	3456
GA	*c05200*	1	05	200	3554.000000	3456
GA	*c05200*	2	05	200	3552.000000	3456
GA	*c05200*	3	05	200	3554.000000	3456
GA	*c05200*	4	05	200	3552.000000	3456
GA	*c05200*	5	05	200	3552.000000	3456
GA	*c05200*	6	05	200	3552.000000	3456
GA	*c05200*	7	05	200	3552.000000	3456
GA	*c05200*	8	05	200	3552.000000	3456
GA	*c05200*	9	05	200	3554.000000	3456
GA	*c10200*	0	10	200	3006.000000	2806
GA	*c10200*	1	10	200	3006.000000	2806
GA	*c10200*	2	10	200	3006.000000	2806
GA	*c10200*	3	10	200	3006.000000	2806
GA	*c10200*	4	10	200	3006.000000	2806
GA	*c10200*	5	10	200	3006.000000	2806
GA	*c10200*	6	10	200	3006.000000	2806
GA	*c10200*	7	10	200	3006.000000	2806
GA	*c10200*	8	10	200	3006.000000	2806
GA	*c10200*	9	10	200	3006.000000	2806
GA	*c20200*	0	20	200	2590.000000	2391
GA	*c20200*	1	20	200	2588.000000	2391
GA	*c20200*	2	20	200	2586.000000	2391
GA	*c20200*	3	20	200	2586.000000	2391
GA	*c20200*	4	20	200	2588.000000	2391
GA	*c20200*	5	20	200	2588.000000	2391
GA	*c20200*	6	20	200	2586.000000	2391
GA	*c20200*	7	20	200	2586.000000	2391
GA	*c20200*	8	20	200	2586.000000	2391
GA	*c20200*	9	20	200	2586.000000	2391
GA	*e05100*	0	05	100	13895.000000	1267
GA	*e05100*	1	05	100	13895.000000	1267
GA	*e05100*	2	05	100	13895.000000	1267
GA	*e05100*	3	05	100	13895.000000	1267
GA	*e05100*	4	05	100	13895.000000	1267
GA	*e05100*	5	05	100	13895.000000	1267
GA	*e05100*	6	05	100	13895.000000	1267
GA	*e05100*	7	05	100	13895.000000	1267
GA	*e05100*	8	05	100	13895.000000	1267
GA	*e05100*	9	05	100	13895.000000	1267
GA	*e10100*	0	10	100	13918.000000	1156
GA	*e10100*	1	10	100	13918.000000	1156
GA	*e10100*	2	10	100	13918.000000	1156
GA	*e10100*	3	10	100	13918.000000	1156
GA	*e10100*	4	10	100	13918.000000	1156
GA	*e10100*	5	10	100	13918.000000	1156
GA	*e10100*	6	10	100	13918.000000	1156
GA	*e10100*	7	10	100	13918.000000	1156

GA	*e10100*	8	10	100	13918.000000	1156
GA	*e10100*	9	10	100	13918.000000	1156
GA	*e20100*	0	20	100	10168.000000	8431
GA	*e20100*	1	20	100	10168.000000	8431
GA	*e20100*	2	20	100	10168.000000	8431
GA	*e20100*	3	20	100	10168.000000	8431
GA	*e20100*	4	20	100	10168.000000	8431
GA	*e20100*	5	20	100	10168.000000	8431
GA	*e20100*	6	20	100	10168.000000	8431
GA	*e20100*	7	20	100	10168.000000	8431
GA	*e20100*	8	20	100	10168.000000	8431
GA	*e20100*	9	20	100	10168.000000	8431
GA	*e05200*	0	05	200	26819.000000	2492
GA	*e05200*	1	05	200	26819.000000	2492
GA	*e05200*	2	05	200	26819.000000	2492
GA	*e05200*	3	05	200	26819.000000	2492
GA	*e05200*	4	05	200	26819.000000	2492
GA	*e05200*	5	05	200	26819.000000	2492
GA	*e05200*	6	05	200	26819.000000	2492
GA	*e05200*	7	05	200	26819.000000	2492
GA	*e05200*	8	05	200	26819.000000	2492
GA	*e05200*	9	05	200	26819.000000	2492
GA	*e10200*	0	10	200	27188.000000	2330
GA	*e10200*	1	10	200	27188.000000	2330
GA	*e10200*	2	10	200	27188.000000	2330
GA	*e10200*	3	10	200	27188.000000	2330
GA	*e10200*	4	10	200	27188.000000	2330
GA	*e10200*	5	10	200	27188.000000	2330
GA	*e10200*	6	10	200	27188.000000	2330
GA	*e10200*	7	10	200	27188.000000	2330
GA	*e10200*	8	10	200	27188.000000	2330
GA	*e10200*	9	10	200	27188.000000	2330
GA	*e20200*	0	20	200	27583.000000	2237
GA	*e20200*	1	20	200	27583.000000	2237
GA	*e20200*	2	20	200	27583.000000	2237
GA	*e20200*	3	20	200	27583.000000	2237
GA	*e20200*	4	20	200	27583.000000	2237
GA	*e20200*	5	20	200	27583.000000	2237
GA	*e20200*	6	20	200	27583.000000	2237
GA	*e20200*	7	20	200	27583.000000	2237
GA	*e20200*	8	20	200	27583.000000	2237
GA	*e20200*	9	20	200	27583.000000	2237

Algoritmo	**Arquivo**	**Iteração/*Seed***	**Agentes**	**Tarefas**	**Valor Encontrado**	**Valor Ótimo Literatura**
Recozimento Simulado	*a05100*	0	05	100	1698.000000	1698
Recozimento Simulado	*a05100*	1	05	100	1698.000000	1698
Recozimento Simulado	*a05100*	2	05	100	1698.000000	1698
Recozimento Simulado	*a05100*	3	05	100	1698.000000	1698
Recozimento Simulado	*a05100*	4	05	100	1698.000000	1698
Recozimento Simulado	*a05100*	5	05	100	1698.000000	1698
Recozimento Simulado	*a05100*	6	05	100	1698.000000	1698
Recozimento Simulado	*a05100*	7	05	100	1698.000000	1698
Recozimento Simulado	*a05100*	8	05	100	1698.000000	1698
Recozimento Simulado	*a05100*	9	05	100	1698.000000	1698
Recozimento Simulado	*a10100*	0	10	100	1360.000000	1360
Recozimento Simulado	*a10100*	1	10	100	1360.000000	1360
Recozimento Simulado	*a10100*	2	10	100	1360.000000	1360
Recozimento Simulado	*a10100*	3	10	100	1360.000000	1360
Recozimento Simulado	*a10100*	4	10	100	1360.000000	1360
Recozimento Simulado	*a10100*	5	10	100	1360.000000	1360
Recozimento Simulado	*a10100*	6	10	100	1360.000000	1360
Recozimento Simulado	*a10100*	7	10	100	1360.000000	1360
Recozimento Simulado	*a10100*	8	10	100	1360.000000	1360
Recozimento Simulado	*a10100*	9	10	100	1360.000000	1360
Recozimento Simulado	*a20100*	0	20	100	1158.000000	1158
Recozimento Simulado	*a20100*	1	20	100	1158.000000	1158
Recozimento Simulado	*a20100*	2	20	100	1158.000000	1158
Recozimento Simulado	*a20100*	3	20	100	1158.000000	1158
Recozimento Simulado	*a20100*	4	20	100	1158.000000	1158
Recozimento Simulado	*a20100*	5	20	100	1158.000000	1158
Recozimento Simulado	*a20100*	6	20	100	1158.000000	1158
Recozimento Simulado	*a20100*	7	20	100	1158.000000	1158

Recozimento Simulado	*a20100*	8	20	100	1158.000000	1158
Recozimento Simulado	*a20100*	9	20	100	1158.000000	1158
Recozimento Simulado	*a05200*	0	05	200	3235.000000	3235
Recozimento Simulado	*a05200*	1	05	200	3235.000000	3235
Recozimento Simulado	*a05200*	2	05	200	3235.000000	3235
Recozimento Simulado	*a05200*	3	05	200	3235.000000	3235
Recozimento Simulado	*a05200*	4	05	200	3235.000000	3235
Recozimento Simulado	*a05200*	5	05	200	3235.000000	3235
Recozimento Simulado	*a05200*	6	05	200	3235.000000	3235
Recozimento Simulado	*a05200*	7	05	200	3235.000000	3235
Recozimento Simulado	*a05200*	8	05	200	3235.000000	3235
Recozimento Simulado	*a05200*	9	05	200	3235.000000	3235
Recozimento Simulado	*a10200*	0	10	200	2623.000000	2623
Recozimento Simulado	*a10200*	1	10	200	2623.000000	2623
Recozimento Simulado	*a10200*	2	10	200	2623.000000	2623
Recozimento Simulado	*a10200*	3	10	200	2623.000000	2623
Recozimento Simulado	*a10200*	4	10	200	2623.000000	2623
Recozimento Simulado	*a10200*	5	10	200	2623.000000	2623
Recozimento Simulado	*a10200*	6	10	200	2623.000000	2623
Recozimento Simulado	*a10200*	7	10	200	2623.000000	2623
Recozimento Simulado	*a10200*	8	10	200	2623.000000	2623
Recozimento Simulado	*a10200*	9	10	200	2623.000000	2623
Recozimento Simulado	*a20200*	0	20	200	2339.000000	2339
Recozimento Simulado	*a20200*	1	20	200	2339.000000	2339
Recozimento Simulado	*a20200*	2	20	200	2339.000000	2339
Recozimento Simulado	*a20200*	3	20	200	2339.000000	2339
Recozimento Simulado	*a20200*	4	20	200	2339.000000	2339
Recozimento Simulado	*a20200*	5	20	200	2339.000000	2339
Recozimento Simulado	*a20200*	6	20	200	2339.000000	2339
Recozimento Simulado	*a20200*	7	20	200	2339.000000	2339
Recozimento Simulado	*a20200*	8	20	200	2339.000000	2339
Recozimento Simulado	*a20200*	9	20	200	2339.000000	2339
Recozimento Simulado	*c05100*	0	05	100	1937.000000	1931
Recozimento Simulado	*c05100*	1	05	100	1937.000000	1931
Recozimento Simulado	*c05100*	2	05	100	1937.000000	1931
Recozimento Simulado	*c05100*	3	05	100	1937.000000	1931
Recozimento Simulado	*c05100*	4	05	100	1937.000000	1931
Recozimento Simulado	*c05100*	5	05	100	1937.000000	1931
Recozimento Simulado	*c05100*	6	05	100	1937.000000	1931
Recozimento Simulado	*c05100*	7	05	100	1937.000000	1931
Recozimento Simulado	*c05100*	8	05	100	1937.000000	1931
Recozimento Simulado	*c05100*	9	05	100	1937.000000	1931
Recozimento Simulado	*c10100*	0	10	100	1415.000000	1402
Recozimento Simulado	*c10100*	1	10	100	1415.000000	1402
Recozimento Simulado	*c10100*	2	10	100	1415.000000	1402
Recozimento Simulado	*c10100*	3	10	100	1415.000000	1402
Recozimento Simulado	*c10100*	4	10	100	1415.000000	1402
Recozimento Simulado	*c10100*	5	10	100	1415.000000	1402
Recozimento Simulado	*c10100*	6	10	100	1415.000000	1402
Recozimento Simulado	*c10100*	7	10	100	1415.000000	1402
Recozimento Simulado	*c10100*	8	10	100	1415.000000	1402
Recozimento Simulado	*c10100*	9	10	100	1415.000000	1402
Recozimento Simulado	*c20100*	0	20	100	1264.000000	1243
Recozimento Simulado	*c20100*	1	20	100	1264.000000	1243
Recozimento Simulado	*c20100*	2	20	100	1264.000000	1243
Recozimento Simulado	*c20100*	3	20	100	1264.000000	1243
Recozimento Simulado	*c20100*	4	20	100	1264.000000	1243
Recozimento Simulado	*c20100*	5	20	100	1264.000000	1243
Recozimento Simulado	*c20100*	6	20	100	1264.000000	1243
Recozimento Simulado	*c20100*	7	20	100	1264.000000	1243
Recozimento Simulado	*c20100*	8	20	100	1264.000000	1243
Recozimento Simulado	*c20100*	9	20	100	1264.000000	1243
Recozimento Simulado	*c05200*	0	05	200	3460.000000	3456
Recozimento Simulado	*c05200*	1	05	200	3460.000000	3456
Recozimento Simulado	*c05200*	2	05	200	3460.000000	3456
Recozimento Simulado	*c05200*	3	05	200	3460.000000	3456
Recozimento Simulado	*c05200*	4	05	200	3460.000000	3456
Recozimento Simulado	*c05200*	5	05	200	3460.000000	3456
Recozimento Simulado	*c05200*	6	05	200	3460.000000	3456
Recozimento Simulado	*c05200*	7	05	200	3460.000000	3456
Recozimento Simulado	*c05200*	8	05	200	3460.000000	3456

Recozimento Simulado	*c05200*	9	05	200	3460.000000	3456
Recozimento Simulado	*c10200*	0	10	200	2838.000000	2806
Recozimento Simulado	*c10200*	1	10	200	2838.000000	2806
Recozimento Simulado	*c10200*	2	10	200	2838.000000	2806
Recozimento Simulado	*c10200*	3	10	200	2838.000000	2806
Recozimento Simulado	*c10200*	4	10	200	2838.000000	2806
Recozimento Simulado	*c10200*	5	10	200	2838.000000	2806
Recozimento Simulado	*c10200*	6	10	200	2838.000000	2806
Recozimento Simulado	*c10200*	7	10	200	2838.000000	2806
Recozimento Simulado	*c10200*	8	10	200	2838.000000	2806
Recozimento Simulado	*c10200*	9	10	200	2838.000000	2806
Recozimento Simulado	*c20200*	0	20	200	2413.000000	2391
Recozimento Simulado	*c20200*	1	20	200	2413.000000	2391
Recozimento Simulado	*c20200*	2	20	200	2413.000000	2391
Recozimento Simulado	*c20200*	3	20	200	2413.000000	2391
Recozimento Simulado	*c20200*	4	20	200	2413.000000	2391
Recozimento Simulado	*c20200*	5	20	200	2413.000000	2391
Recozimento Simulado	*c20200*	6	20	200	2413.000000	2391
Recozimento Simulado	*c20200*	7	20	200	2413.000000	2391
Recozimento Simulado	*c20200*	8	20	200	2413.000000	2391
Recozimento Simulado	*c20200*	9	20	200	2413.000000	2391
Recozimento Simulado	*e05100*	0	05	100	12762.000000	1267
Recozimento Simulado	*e05100*	1	05	100	12762.000000	1267
Recozimento Simulado	*e05100*	2	05	100	12762.000000	1267
Recozimento Simulado	*e05100*	3	05	100	12762.000000	1267
Recozimento Simulado	*e05100*	4	05	100	12762.000000	1267
Recozimento Simulado	*e05100*	5	05	100	12762.000000	1267
Recozimento Simulado	*e05100*	6	05	100	12762.000000	1267
Recozimento Simulado	*e05100*	7	05	100	12762.000000	1267
Recozimento Simulado	*e05100*	8	05	100	12762.000000	1267
Recozimento Simulado	*e05100*	9	05	100	12762.000000	1267
Recozimento Simulado	*e10100*	0	10	100	11833.000000	1156
Recozimento Simulado	*e10100*	1	10	100	11833.000000	1156
Recozimento Simulado	*e10100*	2	10	100	11833.000000	1156
Recozimento Simulado	*e10100*	3	10	100	11833.000000	1156
Recozimento Simulado	*e10100*	4	10	100	11833.000000	1156
Recozimento Simulado	*e10100*	5	10	100	11833.000000	1156
Recozimento Simulado	*e10100*	6	10	100	11833.000000	1156
Recozimento Simulado	*e10100*	7	10	100	11833.000000	1156
Recozimento Simulado	*e10100*	8	10	100	11833.000000	1156
Recozimento Simulado	*e10100*	9	10	100	11833.000000	1156
Recozimento Simulado	*e20100*	0	20	100	8837.000000	8431
Recozimento Simulado	*e20100*	1	20	100	8837.000000	8431
Recozimento Simulado	*e20100*	2	20	100	8837.000000	8431
Recozimento Simulado	*e20100*	3	20	100	8837.000000	8431
Recozimento Simulado	*e20100*	4	20	100	8837.000000	8431
Recozimento Simulado	*e20100*	5	20	100	8837.000000	8431
Recozimento Simulado	*e20100*	6	20	100	8837.000000	8431
Recozimento Simulado	*e20100*	7	20	100	8837.000000	8431
Recozimento Simulado	*e20100*	8	20	100	8837.000000	8431
Recozimento Simulado	*e20100*	9	20	100	8837.000000	8431
Recozimento Simulado	*e05200*	0	05	200	25111.000000	2492
Recozimento Simulado	*e05200*	1	05	200	25111.000000	2492
Recozimento Simulado	*e05200*	2	05	200	25111.000000	2492
Recozimento Simulado	*e05200*	3	05	200	25111.000000	2492
Recozimento Simulado	*e05200*	4	05	200	25111.000000	2492
Recozimento Simulado	*e05200*	5	05	200	25111.000000	2492
Recozimento Simulado	*e05200*	6	05	200	25111.000000	2492
Recozimento Simulado	*e05200*	7	05	200	25111.000000	2492
Recozimento Simulado	*e05200*	8	05	200	25111.000000	2492
Recozimento Simulado	*e05200*	9	05	200	25111.000000	2492
Recozimento Simulado	*e10200*	0	10	200	23825.000000	2330
Recozimento Simulado	*e10200*	1	10	200	23825.000000	2330
Recozimento Simulado	*e10200*	2	10	200	23825.000000	2330
Recozimento Simulado	*e10200*	3	10	200	23825.000000	2330
Recozimento Simulado	*e10200*	4	10	200	23825.000000	2330
Recozimento Simulado	*e10200*	5	10	200	23825.000000	2330
Recozimento Simulado	*e10200*	6	10	200	23825.000000	2330
Recozimento Simulado	*e10200*	7	10	200	23825.000000	2330
Recozimento Simulado	*e10200*	8	10	200	23825.000000	2330
Recozimento Simulado	*e10200*	9	10	200	23825.000000	2330
Recozimento Simulado	*e20200*	0	20	200	23571.000000	2237

Recozimento Simulado	*e20200*	1	20	200	23571.000000	2237
Recozimento Simulado	*e20200*	2	20	200	23571.000000	2237
Recozimento Simulado	*e20200*	3	20	200	23571.000000	2237
Recozimento Simulado	*e20200*	4	20	200	23571.000000	2237
Recozimento Simulado	*e20200*	5	20	200	23571.000000	2237
Recozimento Simulado	*e20200*	6	20	200	23571.000000	2237
Recozimento Simulado	*e20200*	7	20	200	23571.000000	2237
Recozimento Simulado	*e20200*	8	20	200	23571.000000	2237
Recozimento Simulado	*e20200*	9	20	200	23571.000000	2237

Algoritmo	**Arquivo**	**Iteração/*Seed***	**Agentes**	**Tarefas**	**Valor Encontrado**	**Valor Ótimo Literatura**
Método Reinício	*a05100*	0	05	100	1698.000000	1698
Método Reinício	*a05100*	1	05	100	1698.000000	1698
Método Reinício	*a05100*	2	05	100	1698.000000	1698
Método Reinício	*a05100*	3	05	100	1698.000000	1698
Método Reinício	*a05100*	4	05	100	1698.000000	1698
Método Reinício	*a05100*	5	05	100	1698.000000	1698
Método Reinício	*a05100*	6	05	100	1698.000000	1698
Método Reinício	*a05100*	7	05	100	1698.000000	1698
Método Reinício	*a05100*	8	05	100	1698.000000	1698
Método Reinício	*a05100*	9	05	100	1698.000000	1698
Método Reinício	*a10100*	0	10	100	1360.000000	1360
Método Reinício	*a10100*	1	10	100	1360.000000	1360
Método Reinício	*a10100*	2	10	100	1360.000000	1360
Método Reinício	*a10100*	3	10	100	1360.000000	1360
Método Reinício	*a10100*	4	10	100	1360.000000	1360
Método Reinício	*a10100*	5	10	100	1360.000000	1360
Método Reinício	*a10100*	6	10	100	1360.000000	1360
Método Reinício	*a10100*	7	10	100	1360.000000	1360
Método Reinício	*a10100*	8	10	100	1360.000000	1360
Método Reinício	*a10100*	9	10	100	1360.000000	1360
Método Reinício	*a20100*	0	20	100	1158.000000	1158
Método Reinício	*a20100*	1	20	100	1158.000000	1158
Método Reinício	*a20100*	2	20	100	1158.000000	1158
Método Reinício	*a20100*	3	20	100	1158.000000	1158
Método Reinício	*a20100*	4	20	100	1158.000000	1158
Método Reinício	*a20100*	5	20	100	1158.000000	1158
Método Reinício	*a20100*	6	20	100	1158.000000	1158
Método Reinício	*a20100*	7	20	100	1158.000000	1158
Método Reinício	*a20100*	8	20	100	1158.000000	1158
Método Reinício	*a20100*	9	20	100	1158.000000	1158
Método Reinício	*a05200*	0	05	200	3235.000000	3235
Método Reinício	*a05200*	1	05	200	3235.000000	3235
Método Reinício	*a05200*	2	05	200	3235.000000	3235
Método Reinício	*a05200*	3	05	200	3235.000000	3235
Método Reinício	*a05200*	4	05	200	3235.000000	3235
Método Reinício	*a05200*	5	05	200	3235.000000	3235
Método Reinício	*a05200*	6	05	200	3235.000000	3235
Método Reinício	*a05200*	7	05	200	3235.000000	3235
Método Reinício	*a05200*	8	05	200	3235.000000	3235
Método Reinício	*a05200*	9	05	200	3235.000000	3235
Método Reinício	*a10200*	0	10	200	2623.000000	2623
Método Reinício	*a10200*	1	10	200	2623.000000	2623
Método Reinício	*a10200*	2	10	200	2623.000000	2623
Método Reinício	*a10200*	3	10	200	2623.000000	2623
Método Reinício	*a10200*	4	10	200	2623.000000	2623
Método Reinício	*a10200*	5	10	200	2623.000000	2623
Método Reinício	*a10200*	6	10	200	2623.000000	2623
Método Reinício	*a10200*	7	10	200	2623.000000	2623
Método Reinício	*a10200*	8	10	200	2623.000000	2623
Método Reinício	*a10200*	9	10	200	2623.000000	2623
Método Reinício	*a20200*	0	20	200	2339.000000	2339
Método Reinício	*a20200*	1	20	200	2339.000000	2339
Método Reinício	*a20200*	2	20	200	2339.000000	2339
Método Reinício	*a20200*	3	20	200	2339.000000	2339
Método Reinício	*a20200*	4	20	200	2339.000000	2339
Método Reinício	*a20200*	5	20	200	2339.000000	2339
Método Reinício	*a20200*	6	20	200	2339.000000	2339
Método Reinício	*a20200*	7	20	200	2339.000000	2339
Método Reinício	*a20200*	8	20	200	2339.000000	2339
Método Reinício	*a20200*	9	20	200	2339.000000	2339
Método Reinício	*c05100*	0	05	100	1953.000000	1931

Método Reinício	*c05100*	1	05	100	1953.000000	1931
Método Reinício	*c05100*	2	05	100	1953.000000	1931
Método Reinício	*c05100*	3	05	100	1953.000000	1931
Método Reinício	*c05100*	4	05	100	1953.000000	1931
Método Reinício	*c05100*	5	05	100	1953.000000	1931
Método Reinício	*c05100*	6	05	100	1953.000000	1931
Método Reinício	*c05100*	7	05	100	1953.000000	1931
Método Reinício	*c05100*	8	05	100	1953.000000	1931
Método Reinício	*c05100*	9	05	100	1953.000000	1931
Método Reinício	*c10100*	0	10	100	1433.000000	1402
Método Reinício	*c10100*	1	10	100	1433.000000	1402
Método Reinício	*c10100*	2	10	100	1433.000000	1402
Método Reinício	*c10100*	3	10	100	1433.000000	1402
Método Reinício	*c10100*	4	10	100	1433.000000	1402
Método Reinício	*c10100*	5	10	100	1433.000000	1402
Método Reinício	*c10100*	6	10	100	1433.000000	1402
Método Reinício	*c10100*	7	10	100	1433.000000	1402
Método Reinício	*c10100*	8	10	100	1433.000000	1402
Método Reinício	*c10100*	9	10	100	1433.000000	1402
Método Reinício	*c20100*	0	20	100	1264.000000	1243
Método Reinício	*c20100*	1	20	100	1264.000000	1243
Método Reinício	*c20100*	2	20	100	1264.000000	1243
Método Reinício	*c20100*	3	20	100	1264.000000	1243
Método Reinício	*c20100*	4	20	100	1264.000000	1243
Método Reinício	*c20100*	5	20	100	1264.000000	1243
Método Reinício	*c20100*	6	20	100	1264.000000	1243
Método Reinício	*c20100*	7	20	100	1264.000000	1243
Método Reinício	*c20100*	8	20	100	1264.000000	1243
Método Reinício	*c20100*	9	20	100	1264.000000	1243
Método Reinício	*c05200*	0	05	200	3503.000000	3456
Método Reinício	*c05200*	1	05	200	3503.000000	3456
Método Reinício	*c05200*	2	05	200	3503.000000	3456
Método Reinício	*c05200*	3	05	200	3503.000000	3456
Método Reinício	*c05200*	4	05	200	3503.000000	3456
Método Reinício	*c05200*	5	05	200	3503.000000	3456
Método Reinício	*c05200*	6	05	200	3503.000000	3456
Método Reinício	*c05200*	7	05	200	3503.000000	3456
Método Reinício	*c05200*	8	05	200	3503.000000	3456
Método Reinício	*c05200*	9	05	200	3503.000000	3456
Método Reinício	*c10200*	0	10	200	2852.000000	2806
Método Reinício	*c10200*	1	10	200	2852.000000	2806
Método Reinício	*c10200*	2	10	200	2852.000000	2806
Método Reinício	*c10200*	3	10	200	2852.000000	2806
Método Reinício	*c10200*	4	10	200	2852.000000	2806
Método Reinício	*c10200*	5	10	200	2852.000000	2806
Método Reinício	*c10200*	6	10	200	2852.000000	2806
Método Reinício	*c10200*	7	10	200	2852.000000	2806
Método Reinício	*c10200*	8	10	200	2852.000000	2806
Método Reinício	*c10200*	9	10	200	2852.000000	2806
Método Reinício	*c20200*	0	20	200	2445.000000	2391
Método Reinício	*c20200*	1	20	200	2445.000000	2391
Método Reinício	*c20200*	2	20	200	2445.000000	2391
Método Reinício	*c20200*	3	20	200	2445.000000	2391
Método Reinício	*c20200*	4	20	200	2445.000000	2391
Método Reinício	*c20200*	5	20	200	2445.000000	2391
Método Reinício	*c20200*	6	20	200	2445.000000	2391
Método Reinício	*c20200*	7	20	200	2445.000000	2391
Método Reinício	*c20200*	8	20	200	2445.000000	2391
Método Reinício	*c20200*	9	20	200	2445.000000	2391
Método Reinício	*e05100*	0	05	100	13948.000000	1267
Método Reinício	*e05100*	1	05	100	13948.000000	1267
Método Reinício	*e05100*	2	05	100	13948.000000	1267
Método Reinício	*e05100*	3	05	100	13948.000000	1267
Método Reinício	*e05100*	4	05	100	13948.000000	1267
Método Reinício	*e05100*	5	05	100	13948.000000	1267
Método Reinício	*e05100*	6	05	100	13948.000000	1267
Método Reinício	*e05100*	7	05	100	13948.000000	1267
Método Reinício	*e05100*	8	05	100	13948.000000	1267
Método Reinício	*e05100*	9	05	100	13948.000000	1267
Método Reinício	*e10100*	0	10	100	13199.000000	1156
Método Reinício	*e10100*	1	10	100	13199.000000	1156
Método Reinício	*e10100*	2	10	100	13199.000000	1156

Método Reinício	*e10100*	3	10	100	13199.000000	1156
Método Reinício	*e10100*	4	10	100	13199.000000	1156
Método Reinício	*e10100*	5	10	100	13199.000000	1156
Método Reinício	*e10100*	6	10	100	13199.000000	1156
Método Reinício	*e10100*	7	10	100	13199.000000	1156
Método Reinício	*e10100*	8	10	100	13199.000000	1156
Método Reinício	*e10100*	9	10	100	13199.000000	1156
Método Reinício	*e20100*	0	20	100	9534.000000	8431
Método Reinício	*e20100*	1	20	100	9534.000000	8431
Método Reinício	*e20100*	2	20	100	9534.000000	8431
Método Reinício	*e20100*	3	20	100	9534.000000	8431
Método Reinício	*e20100*	4	20	100	9534.000000	8431
Método Reinício	*e20100*	5	20	100	9534.000000	8431
Método Reinício	*e20100*	6	20	100	9534.000000	8431
Método Reinício	*e20100*	7	20	100	9534.000000	8431
Método Reinício	*e20100*	8	20	100	9534.000000	8431
Método Reinício	*e20100*	9	20	100	9534.000000	8431
Método Reinício	*e05200*	0	05	200	28693.000000	2492
Método Reinício	*e05200*	1	05	200	28693.000000	2492
Método Reinício	*e05200*	2	05	200	28693.000000	2492
Método Reinício	*e05200*	3	05	200	28693.000000	2492
Método Reinício	*e05200*	4	05	200	28693.000000	2492
Método Reinício	*e05200*	5	05	200	28693.000000	2492
Método Reinício	*e05200*	6	05	200	28693.000000	2492
Método Reinício	*e05200*	7	05	200	28693.000000	2492
Método Reinício	*e05200*	8	05	200	28693.000000	2492
Método Reinício	*e05200*	9	05	200	28693.000000	2492
Método Reinício	*e10200*	0	10	200	27227.000000	2330
Método Reinício	*e10200*	1	10	200	27227.000000	2330
Método Reinício	*e10200*	2	10	200	27227.000000	2330
Método Reinício	*e10200*	3	10	200	27227.000000	2330
Método Reinício	*e10200*	4	10	200	27227.000000	2330
Método Reinício	*e10200*	5	10	200	27227.000000	2330
Método Reinício	*e10200*	6	10	200	27227.000000	2330
Método Reinício	*e10200*	7	10	200	27227.000000	2330
Método Reinício	*e10200*	8	10	200	27227.000000	2330
Método Reinício	*e10200*	9	10	200	27227.000000	2330
Método Reinício	*e20200*	0	20	200	25992.000000	2237
Método Reinício	*e20200*	1	20	200	25992.000000	2237
Método Reinício	*e20200*	2	20	200	25992.000000	2237
Método Reinício	*e20200*	3	20	200	25992.000000	2237
Método Reinício	*e20200*	4	20	200	25992.000000	2237
Método Reinício	*e20200*	5	20	200	25992.000000	2237
Método Reinício	*e20200*	6	20	200	25992.000000	2237
Método Reinício	*e20200*	7	20	200	25992.000000	2237
Método Reinício	*e20200*	8	20	200	25992.000000	2237
Método Reinício	*e20200*	9	20	200	25992.000000	2237

Algoritmo	**Arquivo**	**Iteração/*Seed***	**Agentes**	**Tarefas**	**Valor Encontrado**	**Valor Ótimo Literatura**
GRASP	*a05100*	0	05	100	1698.000000	1698
GRASP	*a05100*	1	05	100	1698.000000	1698
GRASP	*a05100*	2	05	100	1698.000000	1698
GRASP	*a05100*	3	05	100	1698.000000	1698
GRASP	*a05100*	4	05	100	1698.000000	1698
GRASP	*a05100*	5	05	100	1698.000000	1698
GRASP	*a05100*	6	05	100	1698.000000	1698
GRASP	*a05100*	7	05	100	1698.000000	1698
GRASP	*a05100*	8	05	100	1698.000000	1698
GRASP	*a05100*	9	05	100	1698.000000	1698
GRASP	*a10100*	0	10	100	1360.000000	1360
GRASP	*a10100*	1	10	100	1360.000000	1360
GRASP	*a10100*	2	10	100	1360.000000	1360
GRASP	*a10100*	3	10	100	1360.000000	1360
GRASP	*a10100*	4	10	100	1360.000000	1360
GRASP	*a10100*	5	10	100	1360.000000	1360
GRASP	*a10100*	6	10	100	1360.000000	1360
GRASP	*a10100*	7	10	100	1360.000000	1360
GRASP	*a10100*	8	10	100	1360.000000	1360
GRASP	*a10100*	9	10	100	1360.000000	1360
GRASP	*a20100*	0	20	100	1158.000000	1158
GRASP	*a20100*	1	20	100	1158.000000	1158
GRASP	*a20100*	2	20	100	1158.000000	1158

GRASP	*a20100*	3	20	100	1158.000000	1158
GRASP	*a20100*	4	20	100	1158.000000	1158
GRASP	*a20100*	5	20	100	1158.000000	1158
GRASP	*a20100*	6	20	100	1158.000000	1158
GRASP	*a20100*	7	20	100	1158.000000	1158
GRASP	*a20100*	8	20	100	1158.000000	1158
GRASP	*a20100*	9	20	100	1158.000000	1158
GRASP	*a05200*	0	05	200	3235.000000	3235
GRASP	*a05200*	1	05	200	3235.000000	3235
GRASP	*a05200*	2	05	200	3235.000000	3235
GRASP	*a05200*	3	05	200	3235.000000	3235
GRASP	*a05200*	4	05	200	3235.000000	3235
GRASP	*a05200*	5	05	200	3235.000000	3235
GRASP	*a05200*	6	05	200	3235.000000	3235
GRASP	*a05200*	7	05	200	3235.000000	3235
GRASP	*a05200*	8	05	200	3235.000000	3235
GRASP	*a05200*	9	05	200	3235.000000	3235
GRASP	*a10200*	0	10	200	2623.000000	2623
GRASP	*a10200*	1	10	200	2623.000000	2623
GRASP	*a10200*	2	10	200	2623.000000	2623
GRASP	*a10200*	3	10	200	2623.000000	2623
GRASP	*a10200*	4	10	200	2623.000000	2623
GRASP	*a10200*	5	10	200	2623.000000	2623
GRASP	*a10200*	6	10	200	2623.000000	2623
GRASP	*a10200*	7	10	200	2623.000000	2623
GRASP	*a10200*	8	10	200	2623.000000	2623
GRASP	*a10200*	9	10	200	2623.000000	2623
GRASP	*a20200*	0	20	200	2339.000000	2339
GRASP	*a20200*	1	20	200	2339.000000	2339
GRASP	*a20200*	2	20	200	2339.000000	2339
GRASP	*a20200*	3	20	200	2339.000000	2339
GRASP	*a20200*	4	20	200	2339.000000	2339
GRASP	*a20200*	5	20	200	2339.000000	2339
GRASP	*a20200*	6	20	200	2339.000000	2339
GRASP	*a20200*	7	20	200	2339.000000	2339
GRASP	*a20200*	8	20	200	2339.000000	2339
GRASP	*a20200*	9	20	200	2339.000000	2339
GRASP	*c05100*	0	05	100	1955.000000	1931
GRASP	*c05100*	1	05	100	1955.000000	1931
GRASP	*c05100*	2	05	100	1955.000000	1931
GRASP	*c05100*	3	05	100	1955.000000	1931
GRASP	*c05100*	4	05	100	1955.000000	1931
GRASP	*c05100*	5	05	100	1955.000000	1931
GRASP	*c05100*	6	05	100	1955.000000	1931
GRASP	*c05100*	7	05	100	1955.000000	1931
GRASP	*c05100*	8	05	100	1955.000000	1931
GRASP	*c05100*	9	05	100	1955.000000	1931
GRASP	*c10100*	0	10	100	1419.000000	1402
GRASP	*c10100*	1	10	100	1419.000000	1402
GRASP	*c10100*	2	10	100	1419.000000	1402
GRASP	*c10100*	3	10	100	1419.000000	1402
GRASP	*c10100*	4	10	100	1419.000000	1402
GRASP	*c10100*	5	10	100	1419.000000	1402
GRASP	*c10100*	6	10	100	1419.000000	1402
GRASP	*c10100*	7	10	100	1419.000000	1402
GRASP	*c10100*	8	10	100	1419.000000	1402
GRASP	*c10100*	9	10	100	1419.000000	1402
GRASP	*c20100*	0	20	100	1268.000000	1243
GRASP	*c20100*	1	20	100	1268.000000	1243
GRASP	*c20100*	2	20	100	1268.000000	1243
GRASP	*c20100*	3	20	100	1268.000000	1243
GRASP	*c20100*	4	20	100	1268.000000	1243
GRASP	*c20100*	5	20	100	1268.000000	1243
GRASP	*c20100*	6	20	100	1268.000000	1243
GRASP	*c20100*	7	20	100	1268.000000	1243
GRASP	*c20100*	8	20	100	1268.000000	1243
GRASP	*c20100*	9	20	100	1268.000000	1243
GRASP	*c05200*	0	05	200	3490.000000	3456
GRASP	*c05200*	1	05	200	3490.000000	3456
GRASP	*c05200*	2	05	200	3490.000000	3456
GRASP	*c05200*	3	05	200	3490.000000	3456
GRASP	*c05200*	4	05	200	3490.000000	3456

GRASP	*c05200*	5	05	200	3490.000000	3456
GRASP	*c05200*	6	05	200	3490.000000	3456
GRASP	*c05200*	7	05	200	3490.000000	3456
GRASP	*c05200*	8	05	200	3490.000000	3456
GRASP	*c05200*	9	05	200	3490.000000	3456
GRASP	*c10200*	0	10	200	2860.000000	2806
GRASP	*c10200*	1	10	200	2860.000000	2806
GRASP	*c10200*	2	10	200	2860.000000	2806
GRASP	*c10200*	3	10	200	2860.000000	2806
GRASP	*c10200*	4	10	200	2860.000000	2806
GRASP	*c10200*	5	10	200	2860.000000	2806
GRASP	*c10200*	6	10	200	2860.000000	2806
GRASP	*c10200*	7	10	200	2860.000000	2806
GRASP	*c10200*	8	10	200	2860.000000	2806
GRASP	*c10200*	9	10	200	2860.000000	2806
GRASP	*c20200*	0	20	200	2457.000000	2391
GRASP	*c20200*	1	20	200	2457.000000	2391
GRASP	*c20200*	2	20	200	2457.000000	2391
GRASP	*c20200*	3	20	200	2457.000000	2391
GRASP	*c20200*	4	20	200	2457.000000	2391
GRASP	*c20200*	5	20	200	2457.000000	2391
GRASP	*c20200*	6	20	200	2457.000000	2391
GRASP	*c20200*	7	20	200	2457.000000	2391
GRASP	*c20200*	8	20	200	2457.000000	2391
GRASP	*c20200*	9	20	200	2457.000000	2391
GAR SP	*e05100*	0	05	100	13837.000000	1267
GAR SP	*e05100*	1	05	100	13837.000000	1267
GAR SP	*e05100*	2	05	100	13837.000000	1267
GAR SP	*e05100*	3	05	100	13837.000000	1267
GAR SP	*e05100*	4	05	100	13837.000000	1267
GAR SP	*e05100*	5	05	100	13837.000000	1267
GAR SP	*e05100*	6	05	100	13837.000000	1267
GAR SP	*e05100*	7	05	100	13837.000000	1267
GAR SP	*e05100*	8	05	100	13837.000000	1267
GAR SP	*e05100*	9	05	100	13837.000000	1267
GAR SP	*e10100*	0	10	100	12765.000000	1156
GAR SP	*e10100*	1	10	100	12765.000000	1156
GAR SP	*e10100*	2	10	100	12765.000000	1156
GAR SP	*e10100*	3	10	100	12765.000000	1156
GAR SP	*e10100*	4	10	100	12765.000000	1156
GAR SP	*e10100*	5	10	100	12765.000000	1156
GAR SP	*e10100*	6	10	100	12765.000000	1156
GAR SP	*e10100*	7	10	100	12765.000000	1156
GAR SP	*e10100*	8	10	100	12765.000000	1156
GAR SP	*e10100*	9	10	100	12765.000000	1156
GRASP	*e20100*	0	20	100	9164.000000	8431
GRASP	*e20100*	1	20	100	9164.000000	8431
GRASP	*e20100*	2	20	100	9164.000000	8431
GRASP	*e20100*	3	20	100	9164.000000	8431
GRASP	*e20100*	4	20	100	9164.000000	8431
GRASP	*e20100*	5	20	100	9164.000000	8431
GRASP	*e20100*	6	20	100	9164.000000	8431
GRASP	*e20100*	7	20	100	9164.000000	8431
GRASP	*e20100*	8	20	100	9164.000000	8431
GRASP	*e20100*	9	20	100	9164.000000	8431
GAR SP	*e05200*	0	05	200	29048.000000	2492
GAR SP	*e05200*	1	05	200	29048.000000	2492
GAR SP	*e05200*	2	05	200	29048.000000	2492
GAR SP	*e05200*	3	05	200	29048.000000	2492
GAR SP	*e05200*	4	05	200	29048.000000	2492
GAR SP	*e05200*	5	05	200	29048.000000	2492
GAR SP	*e05200*	6	05	200	29048.000000	2492
GAR SP	*e05200*	7	05	200	29048.000000	2492
GAR SP	*e05200*	8	05	200	29048.000000	2492
GAR SP	*e05200*	9	05	200	29048.000000	2492
GAR SP	*e10200*	0	10	200	26664.000000	2330
GAR SP	*e10200*	1	10	200	26664.000000	2330
GAR SP	*e10200*	2	10	200	26664.000000	2330
GAR SP	*e10200*	3	10	200	26664.000000	2330
GAR SP	*e10200*	4	10	200	26664.000000	2330
GAR SP	*e10200*	5	10	200	26664.000000	2330
GAR SP	*e10200*	6	10	200	26664.000000	2330

GAR SP	*e10200*	7	10	200	26664.000000	2330
GAR SP	*e10200*	8	10	200	26664.000000	2330
GAR SP	*e10200*	9	10	200	26664.000000	2330
GAR SP	*e20200*	0	20	200	25581.000000	2237
GAR SP	*e20200*	1	20	200	25581.000000	2237
GAR SP	*e20200*	2	20	200	25581.000000	2237
GAR SP	*e20200*	3	20	200	25581.000000	2237
GAR SP	*e20200*	4	20	200	25581.000000	2237
GAR SP	*e20200*	5	20	200	25581.000000	2237
GAR SP	*e20200*	6	20	200	25581.000000	2237
GAR SP	*e20200*	7	20	200	25581.000000	2237
GAR SP	*e20200*	8	20	200	25581.000000	2237
GAR SP	*e20200*	9	20	200	25581.000000	2237