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

Rodolfo Labiapari Mansur Guimarães

[1]Departamento de Computação – Universidade Federal de Ouro Preto
35.400-000 – Ouro Preto - MG – Brasil

rodolfolabiapari@gmail.com

Abstract. *This report aims to present the strategy implemented to solve the problem of n-Queens Awards using Branch and Bound. Along with the algorithm, all settings will be displayed, characteristics, reflection on the decisions taken, results of experiments and, by the end, the final considerations of the project.*

Fazer

Resumo. *Este relatório tem como principal objetivo apresentar estratégia implementada para a resolução do problema das n-Rainhas com Prêmios utilizando Branch and Bound. Junto com o algoritmo, serão apresentadas todas as definições, características, as reflexão sobre as decisões tomadas, resultados obtidos dos experimentos realizados e, por final, as considerações finais de projeto.*

Este relatório estuda modelos e algoritmos para o Problema de Alocação Generalizada (GAP) bem como alguns métodos para sua resolução, em várias instâncias do problema. São apresentadas, também, as soluções obtidas e suas avaliações.

1. Problema da Alocação Generalizada

Neste capítulo, faz-se descrições do GAP em sua forma clássica.

1.1. Definição

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 sé 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 2008 cita que text

> *"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."*

2. Heurísticas

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.

3. Algoritmo Genético

Aqui exibido as principais partes do Algoritmo Genético implementado para este problema.

3.1. Estrutura de Dados

```
1 typedef struct Struct_Individuo {
2    int * tarefas;
3    double avaliacao;
4 } Individuo;
```

tarefas é um vetor de tamanho *tarefas* em que cada posição armazena o agente responsável pela tarefa. *avaliacao* é o valor de custo desta solução.

A estrutura é nomeada *Individuo* pelo fato do algoritmo forma populações de indivíduos sendo estes, possíveis soluções.

3.2. Execução

É instanciadas duas populações. Uma atual contendo os indivíduos da população atual e outra que será utilizada como *buffer* para armazenamento das operações realizadas ao longo do algoritmo. Assim, os novos filhos gerados e mutados serão postos na segunda população sendo esta completada com indivíduos da primeira de forma aleatória.

Ao final desta iteração, a segunda população passará a ser a população atual e a primeira servirá de *buffer* para a próxima geração.

Isso foi desenvolvido para evitar chamadas de sistema para alocação e liberação de memória.

Abaixo serão descrito alguns procedimentos fundamentais.

3.2.1. Função de Avaliação

```
1  double Avalia_Individuo(int * tarefas) {
2     int i, j, capacidade = 0;
3     double custo = 0;
4     char solucao_invalida = 0;
5
6     for (i = 0; i < QUANT_AGENTES; i++) {
7        capacidade = 0;
8
9        for (j = 0; j < QUANT_TAREFAS; j++) {
10          if (tarefas[j] == i) {
11             custo += CUSTO_A_T[i][j];
12             capacidade += RECURSOS_A_T[i][j];
13          }
14       }
15
16       if (capacidade > CAPAC_AGENTES[i])
17          solucao_invalida = 1;
18    }
19
20    if (solucao_invalida)
21       return custo * 10000;
22    else
23       return custo;
24 }
```

Calcula-se o valor de custo da solução. Caso a solução for infactível, o custo é multiplicado por uma constante que aumenta significamente seu valor, distanciando-se de qualquer valor de custo factível.

3.2.2. Geração de Novos Filhos

A geração de filhos é dividida em duas partes sendo a primeira a geração de filhos e a segunda, sua mutação.

A primeira é justamente o processo de escolher dois indivíduos da população por meio de torneios e realizar o cruzamento entre eles por meio de cruzamento uniforme. Este novo indivíduo é adicionado à segunda população já formando a nova geração de descendentes.

Gerado os novos filhos, realiza-se o processo de mutação destes. Realiza-se a mutação aleatória *Creep Mutation* em filhos aleatórios.

Ao final, a população é preenchida com indivíduos aleatórios da população anterior.

3.3. 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**.

Um exemplo é exibido a seguir:

```
100
0.15
0.02
60
```

4. Algoritmo de Recozimento Simulado

Aqui exibido o Algoritmo de Recozimento Simulado implementado para este problema.

4.1. Referência do Algoritmo Implementado

O algoritmo implementado foi baseado no livro do , disponibilizado gratuitamente ao público pela editora Omnipax.

Lopes

4.2. Estrutura de Dados

```
typedef struct Struct_Individuo {
   int * tarefas;
   double avaliacao;
} Solucao;
```

A estrutura de dados utilizada segue o mesmo padrão da estrutura utilizada no Algoritmo Genético, vide Seção 3.1.

4.2.1. Procedimento que Instancia Variáveis do Algoritmo

Com o pretexto de reduzir o número de chamadas de sistema para alocação e liberação de memória, utilizou-se de três variáveis *Solucao* para que o algoritmo pudesse realizar seu processamento.

As variáveis utilizadas são *melhor_s*, *atual_s* e *possivel_s*. Como o nome ja diz, a *melhor_s* armazena a melhor solução encontrada em todos os tempos, a *atual_s* é a solução que está sendo comparada com a *proxima_s*, no qual é uma solução vizinha.

4.3. Geração de Soluções Pré-Processamento

Para tornar o processamento ainda mais heurístico, determinou-se que a solução s_0 fosse gerada também aleatoriamente.

```
Solucao * Instancia_Solucao_Aleatoria() {
   int i;
   Solucao * s;

   s = calloc(1, sizeof(Solucao));

   s->tarefas = calloc(QUANT_TAREFAS, sizeof(int));

   for (i = 0; i < QUANT_TAREFAS; i++)
      s->tarefas[i] = rand() % QUANT_AGENTES;

   s->avaliacao = Avalia_Solucao(s->tarefas);

   return s;
}
```

4.3.1. Execução

4.3.2. Avalia Solução

Método idêntico ao utilizado no algoritmo anterior, vide Seção 3.2.1.

4.3.3. Gerando Novo Vizinho

Para a geração de novos vizinhos, utilizou-se da técnica de geração *shift* para obter soluções ainda não avaliadas, abrangendo ainda mais o espaço de soluções. A técnica é exibida a seguir.

```
1  void Gera_Vizinho(Solucao * atual, Solucao ** proxima) {
2     int i, agente_atual, tarefa_escolhida1, tarefa_escolhida2,
       ↪  quant_alteracoes;
3
4     for (i = 0; i < QUANT_TAREFAS; i++) {
5        (*proxima)->tarefas[i] = atual->tarefas[i];
6     }
7
8     quant_alteracoes = rand() % (QUANT_TAREFAS / 50);
9
10    for (i = 0; i < quant_alteracoes; i++) {
11       if (random() % 2) {
12          tarefa_escolhida1 = random() % QUANT_TAREFAS;
13
14          agente_atual = (*proxima)->tarefas[tarefa_escolhida1];
15
16          do {
17             (*proxima)->tarefas[tarefa_escolhida1] = random() %
                ↪  QUANT_AGENTES;
18          } while ((*proxima)->tarefas[tarefa_escolhida1] ==
             ↪  agente_atual);
19       } else {
20
21          do {
22             tarefa_escolhida1 = random() % QUANT_TAREFAS;
23             tarefa_escolhida2 = random() % QUANT_TAREFAS;
24          } while (tarefa_escolhida1 == tarefa_escolhida2 ||
             ↪  (*proxima)->tarefas[tarefa_escolhida1] ==
             ↪  (*proxima)->tarefas[tarefa_escolhida2]);
25
26          agente_atual = (*proxima)->tarefas[tarefa_escolhida1];
27          (*proxima)->tarefas[tarefa_escolhida1] =
             ↪  (*proxima)->tarefas[tarefa_escolhida2];
28          (*proxima)->tarefas[tarefa_escolhida2] = agente_atual;
29       }
30    }
31
32    (*proxima)->avaliacao = Avalia_Solucao((*proxima)->tarefas);
33 }
```

4.3.4. Atualização de Temperatura

Utilizou-se de um método geométrico para a atualização da temperatura na execução do algoritmo. Assim, a tem.

tempo

```
1 void Atualiza_Temperatura(double * t) {
2     *t = 0.95 * *t;
3 }
```

4.4. 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.

   Um exemplo é exibido a seguir:

```
120
25000
```

5. 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.

6. Experimentação

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

6.1. Instâncias

As instâncias disponibilizadas para testes são descritas na Tabela 1.

Tabela 1. Tabela com as informações das Instâncias.

Tipo	Nome do Arquivo	Quantidade de Agentes	Quantidade de Tarefas
A	*a05100*	5	100
A	*a05200*	5	200
A	*a10100*	10	100
A	*a10200*	10	200
A	*a20100*	20	100
A	*a20200*	20	200

6.2. Ambiente de *Hardware* e *Software* Utilizado para Compilação

A descrição do ambiente de testes é descrito na Tabela 2.

Tabela 2. 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)

6.3. Análise Estática e Dinâmica de Código

Nesta seção, será descrito os detalhes dos procedimentos de análise de código.

6.3.1. *Clang Static Analyser*

A execução da análise estática de código foi realizada com sucesso eliminando todos os erros e avisos. Abaixo é exibido o *log* de algoritmo implementado.

Relatório do *backtracking* : Fazer

```
Marooned:bin pripyat$ ./scan-build n-queens-prize-backtracking.c
scan-build: Using '/Users/pripyat/Downloads/checker-278/bin/clang' for static analysis
Can't exec "n-queens-prize-backtracking.c": No such file or directory at ./scan-build line 1094.
scan-build: Removing directory '/var/folders/mc/3p0xb099489gmjk9pfzgs5_m0000gn/T/scan-build-2016
-06-09-001400-61577-1' because it contains no reports.
scan-build: No bugs found.
Marooned:bin pripyat$
```

6.3.2. *Valgrind*

A execução do detector de erros de memória *Valgrind* não foi realizada pelo motivo do detector não executar sua verificação de forma correta no programa implementado. Isso ocorre pelo fato do *Valgrind* não conseguir ultrapassar determinado parte do código afirmando que o *software* finalizou forçadamente sendo que este erro é causado pelo próprio *Valgrind*. Fazer

Deixando um pouco mais claro, o procedimento `n_Rainhas_Prize()` possui um teste verificando se a fila de rainhas no qual o algoritmo trabalha está vazia. Quando a fila está vazia nesta parte do procedimento significa que houve algum *erro de processamento* e assim, o programa é finalizado sem gerar resultados. Este teste foi posto simplesmente por precausão evitando resultados errôneos, pois esta fila nunca estará vazia neste pedaço do código. O único pedaço que ela ficará totalmente vazia é quando o algoritmo

encontra uma solução válida e realiza os cálculos verificando se é a melhor encontrada até então. Feito os cálculos, a última rainha operada é reinserida na fila tornando-a *não vazia antes da continuação do algoritmo*. Entretanto, o *Valgrind* executa tal pedaço forçando a fila estar vazia ocasionando o fim do programa.

Assim, tentou-se analisar as mensagens de erro do *Valgrind*, mas seus *reports* mostravam avisos onde não havia nenhum erro. Como o detector não foi executado corretamente, não foi possível detectar outros possíveis vazamentos de memória. Entretanto, isso possibilitou que o código fosse revisado várias vezes a fim de aprimorá-lo.

7. Resultados

Como já mencionado na Seção 6, serão executados três algoritmos e comparados de acordo com cada prêmio encontrado em um intervalo de tempo. Cada algoritmo executará 12 vezes no intervalo de tempo de 1 minuto alterando os valores de *seed*.

O resultado de cada operação é exibido pela Tabela 4 na Seção 9. Para melhor visualização dos dados, a Tabela 3 mostra os valores médios de cada instância.

Tabela 3. Tabela com as médias e desvios padrões dos prêmios das respectivas execuções.

Arquivo	$\bar{x}$.*BT*	σ.	$\bar{x}$.*B&B* 1	σ.	$\bar{x}$.*B&B* 2	σ.	Ótimo
nqp005.txt	167.0	0.0	167.0	0.0	167.0	0.0	167
nqp008.txt	298.0	0.0	298.0	0.0	298.0	0.0	298

Os símbolos $\bar{x}$ e σ representam a média e o desvio padrão de cada algoritmo respectivamente.

7.1. Resultados em Gráficos

A Figura 1 exibe somente os maiores valores médios obtidos na instância nqp200.txt, para uma fácil visualização dos melhores valores obtidos.

Figura 1. BoxPlot da Instância nqp200.txt.

8. Comentários Finais

Os algoritmos implementados possuem uma técnica de *análise de soluções parciais* visando a comparação dos *melhores valores* com os *valores atuais* obtidos, cortando ramos que aparentam não ser uma boa estratégia percorrê-los.

9. Anexos

Tabela 4: Tabela com todos os valores obtidos.

Arquivo	*Seed*	Prêmio *BT*	Prêmio *B&B* 1	Prêmio *B&B* 2	Ótimo
nqp100.txt	0	2924	3071	3083	Desconhecido

Tabela 4: (continuação)

nqp100.txt	1234	2951	2939	3033	Desconhecido
nqp100.txt	2468	0	0	0	Desconhecido
nqp100.txt	3702	2652	2878	2862	Desconhecido
Desconhecido					
nqp200.txt	12340	0	0	0	Desconhecido
nqp200.txt	13574	5289	5289	5289	Desconhecido