Projeto e Análise de Algoritmos – Problema das n-Rainhas com Prêmios utilizando *Branch and Bound*

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.*

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.*

1. Problema das n-Rainhas

1.1. Definição

A solução deste problema consiste em encontrar uma combinação possível de n rainhas num tabuleiro de dimensão n por n tal que nenhuma das rainhas ataque qualquer outra. Duas rainhas atacam-se uma à outra quando estão:

- Na mesma linha;
- Na mesma coluna; e
- Na mesma diagonal.

É um problema combinatório exponencial, sendo inviável sua execução em computadores com instâncias de tamanho grande por causa do custo tempo.

É resolvido facilmente utilizando a técnica *Backtracking*. Entretanto, seu tempo ainda continua sendo um fator problemático para n com valor grande.

1.2. n-Rainhas com Prêmios

Com o propósito de dificultar ainda mais o problema, foi proposto para o aluno uma variante deste problema com o propósito de busca de prêmios sobre o problema n-Rainhas.

Neste problema, cada célula do tabuleiro possui um valor. A medida que é posicionado as rainhas nas células válidas, o valor desta nova posição é somado com os as posições válidas já selecionadas. e com isso, deseja-se encontrar a maior soma desses valores nos quais estão posicionadas as n-rainhas de tal forma que o tabuleiro seja uma solução válida e com o maior prêmio possível.

Assim, o problema anterior de n-rainhas simples passa a ser um problema de otimização inteira com ainda mais restrições, o que eleva seu tempo de processamento já que somente a combinação de rainhas que fornece a maior soma de prêmios é que determina a solução ótima deste problema.

Para isso, será implementado a técnica de *Branch and Boud* para a tentativa de resolução deste problema.

2. A Técnica *Backtracking*

2.1. Definição

A técnica *backtracking* é um refinamento do algoritmo de busca por força bruta que utiliza a enumeração exaustiva de todo o espaço solução retirando soluções inválidas. Nesta técnica, boa parte das soluções podem ser eliminadas sem serem explicitamente examinadas.

2.2. Variações. A Técnica *Branch and Bound*

Técnica que permite cancelar uma recursão quando se sabe que a melhor solução da subárvore é pior do que a melhor solução já encontrada.

3. Detalhes do Algoritmo Implementado pelo Autor

3.1. Considerações de Projeto

Aqui será abordado algumas considerações prévias para o algoritmo desenvolvido.

- Neste trabalho, foram implementados três algoritmos. Um utilizando a técnica *Backtracking* e outros dois utilizando *Branch and Bound*. Isso foi feito justamente por exibir ambas as ideias de projeto que o autor obteve ao desenvolver este trabalho.
- A ideia de projeto dos algoritmos que utilizam a técnica *Branch and Bound* foi totalmente desenvolvida pelo autor. Utilizou-se somente a natureza da técnica *backtraking* para implementação inicial e, em seguida, realizou-se modificações a fim de adaptá-los à lógica que proporciona limitação de ramificações. As técnicas desenvolvidas são simples e o seus resultados serão exibidos e comentados no decorrer do relatório.
- Um detalhe a se atentar é que o algoritmo implementado não possui nenhuma verificação de entrada inválida. Isso deve-se ao fato da implementação ser focada na execução do algoritmo, evitando testes/condicionais de validação de entrada e processamento. Entradas que estiverem de acordo com o padrão estabelecidos pelo professor, serão executadas sem nenhum erro.

3.2. Ordem de Escolha das Rainhas

Visando uma execução que não utilize formas convencionais da literatura disponíveis atualmente, decidiu-se então desordenar as rainhas de forma que ela não tenham uma ordem definitiva. Gerando uma sequência desordenada e sem repetição de rainhas, possibilita ao algoritmo percorrer ramos que estão distribuídos de forma randômica pela árvore, saindo de uma convencional busca regional.

Para conseguir tal procedimento de desordenamento, utilizou-se de uma técnica simples de embaralhamento de chaves. Tem-se a priori um vetor de rainhas ordenadas e é escolhido duas posições aleatórias dentro do vetor e as rainhas são trocados entre esses dois valores repetindo o processo várias vezes. Não existe restrições de posições quanto ao número gerado, proporcionando mais aleatoriedade à sequência final.

Após gerada a nova ordem de rainhas, serão copiadas do vetor para a estrutura Fila (descrita na Seção 3.3.1) que será utilizada pelo algoritmo ao logo de suas execuções.

3.2.1. *Seed* do Gerador de Número Aleatório

Como o algoritmo utiliza de procedimentos que geram número pseudo-randomicamente para a escolha das posições do vetor a ser desordenado, então forneceu-se a função que redefine a semente (*seed*), para a reprodução de resultados.

A definição da semente é feita por parâmetros via linha de comando, como será descrito na Seção 3.6

3.3. Estruturas de Dados

Para a construção do algoritmo, foi desenvolvido duas estruturas de dados para o armazenamento de acesso rápido e inteligente de informações.

3.3.1. Fila de Rainhas

A primeira estrutura de dados a ser mencionada neste relatório é a *Fila*, código exibido abaixo. Utilizou-se de uma fila para armazenar as rainhas que ainda não foram utilizadas no tabuleiro. Assim, como demanda a natureza da fila, elas são retiradas na por ordem de chegada e, caso alguma retirada seja inválida por causa de sua posição perante o tabuleiro atual, será posta no final da fila, continuando a execução do algoritmo para a próxima rainha da fila.

A estrutura então é constituída de uma variável chamada de `lugar` que armazenará a posição da rainha e uma variável que apontará para a próxima rainha da fila. A última rainha da fila apontará para `vazio`.

```
1 typedef struct Struct_Fila {
2    int            lugar;
3    struct Struct_Fila * proximo;
4 } Fila;
```

Um detalhe a se atentar é que, por mais que a ordem inicial das rainhas sejam processadas a fim de gerar uma aleatoriedade entre elas, o algoritmo em si não trabalha de forma aleatória. Ou seja, após definida a ordem das rainhas, o algoritmo opera as colunas de forma sequencial processando primeiro a coluna 1 (um) adicionando uma rainha nela, em seguida adicionando uma rainha na coluna 2 (dois) e assim por diante, trabalhando da forma direcionada da esquerda para a direita. Assim, por mais que as rainhas possam

ocupar qualquer lugar da coluna (por estarem desordenadas), as colunas serão trabalhadas sequencialmente.

3.3.2. Estrutura de Dado Abstrato *Tabuleiro*

Com a sequência de rainhas já determinada, tem-se uma estrutura para armazenar as soluções encontradas pelo algoritmo, chamada de *Tabuleiro*.

A estrutura possui um vetor de inteiro que representa a posição de cada rainha em relação à coluna representada pelo índice do vetor, onde o vetor possui tamanho n sendo n a quantidade de colunas da matriz. Assim, na posição `coluna[0] = 5` deste vetor temos que a rainha 6 está posicionada na colina 1, `coluna[1] = 17` implica que a rainha 18 estará na coluna 2 e assim segue.

Caso a posição ainda não tenha uma rainha definida, terá o valor -1.

Outra variável pertencente à estrutura *Tabuleiro* é a `premio`. Esta váriável informa a soma total de cada rainha já posicionada de forma válida. O tabuleiro não precisa estar completo para calcular-se o seu prêmio. Basta que as rainhas sejam posicionadas de forma válida para que este valor seja alterado com a soma dos valores de cada rainha posta na solução pacial. Isso é necessário para o cálculo da técnica de *bound*, descrita na Seção 3.4.

```
1 typedef struct Struct_Tabuleiro {
2    int * colunas;
3    int   premio;
4 } Tabuleiro;
```

3.4. *Bound*

A operação de *bound* é o quesito chave da diferenciação do algoritmo *Backtracking* comum. A técnica *Backtracking* não utiliza nenhum meio a fim de realizar podas nas suas ramificações, diferentemente da *Branch and Bound*.

Nos algoritmos de *Branch and Bound*, com o propósito de podar alguns ramos da árvore de soluções, desenvolve-se duas técnicas que a partir da melhor solução encontrada analisam quão bom é a solução parcial atual proibindo ou não a continuação da recursão. Elas serão descritas separadamente a seguir.

3.4.1. Técnica *Bound* 1

Descrição cronológica da técnica de *bound* 1.

Solução Inicial: O algoritmo não inicia com uma solução válida. Então, como passo inicial, é realizado uma busca com força bruta a procura de uma solução factível. Encontrado uma solução, é definida como Solução com Maior Prêmio para ser comparada com futuras soluções.

Qualquer solução parcial encontrada após a solução inicial, passará por uma análise antes da continuação do preenchimento do tabuleiro[1]. Esta análise é descrita a seguir.

Cálculo do *Bound*: Relembrando que as colunas da matriz são acessadas sequencialmente, foi desenvolvido uma estratégia que utiliza cálculos estatísticos sobre o melhor prêmio encontrado comparando-o com o premio atual da solução parcial.

Para realizar estes cálculos, foi adicionado ao código vários testes que realizam a comparação entre os valores de prêmio entre duas soluções. A solução de melhor prêmio e a solução parcial atual. Calcula-se o erro percentual entre os dois prêmios dando uma margem de erro para aceitação da solução atual. Essa ideia foi retirada da matéria de Teoria dos Erros utilizada em Simulação de Eventos para verificação e decisão de poda do ramo.

Descrevendo de forma mais clara, primeiramente, as podas acontecem em níveis de profundidade e com diferentes parâmetros sobre a árvore de recursão. Cada nível possui valores de poda diferente e por este motivo, deve-se primeiro identificar os níveis de profundidade da árvore.

O nível atual da árvore é calculado pela fórmula

$$profundidade = \frac{coluna\_atual}{total\_culunas}$$

obtendo a porcentagem de profundidade da árvore. Assim, obtendo o valor $0,5$ da fórmula conclui-se que a recursão está na metade da altura total da árvore. Obter $0,8$ significaria que 80% foi percorrida restando apenas 20% de nós em profundidade para chegar à sua folha.

Com a altura da árvore em mão, pode-se aplicar vários filtros de poda com valores de parâmetros diferentes sobre diferentes níveis da árvore de solução.

O filtro é realizando analisando o erro relativo entre as duas soluções como é exibido pela fórmula

$$fator\_continua\_recursao = \frac{premio\_parcial}{melhor\_premio}$$

Para o cálculo, utiliza-se o valor de prêmio da melhor solução e o prêmio da solução parcial atual. Realizando a divisão de ambos os valores, é possível descobrir a porcentagem de erro que existe entre os dois valores. Quanto maior for o número resultante da fórmula, melhor é a solução encontrada até o momento. Caso o número ultrapasse o valor de 1 (um), significa que a solução parcial já possui prêmio maior que a melhor encontrada antes mesmo de ter preenchido todo o tabuleiro. Entretanto, não significa que ela será selecionada já que essa solução parcial pode não ter uma solução final válida nos seus sub-ramos. E caso o valor seja próximo de 0 (zero), mais indesejada é esta solução.

Sabendo cada uma dessas fórmulas, cria-se então uma série de filtros a fim analisar os prêmios. Eles possuem o propósito de podar todas as soluções parciais que não forem

[1]Lembrando que o cálculo do prêmio é feito mesmo quando a solução é incompleta. Sempre que adicionar ou retirar uma rainha de uma posição válida, será realizado o cálculo do prêmio para o tabuleiro final.

suficiente boas ao ponto de não atender os requisitos de determinado limite, sendo assim, cortando todo seu ramo.

Implementou-se 3 *bounds* com valores e níveis diferentes a fim de realizar filtros com propósitos diferentes na árvore de soluções. São eles:

Filtro de *bound* 1: O primeiro filtro possui $profundidade \geq 0.6$ e < 0.7 e $fator\_continua\_recursao = 0.6$. Ele é um filtro fraco justamente para realizar podas que possuem valor de prêmio muito baixos. Quando chega em 60% da altura da recursão, verifica-se se as soluções possuem pelo menos 50% do valor da maior solução encontrada até o momento.

Filtro de *bound* 2: O segundo filtro possui $profundidade \geq 0.7$ e < 0.8 e $fator\_continua\_recursao > 0.7$. Ele é um filtro com força mediana podando ramificações que parecem ser não promissoras no momento. Em 70% de altura, verifica-se se as soluções possuem pelo menos 60% do valor da maior solução encontrada até o momento.

Filtro de *bound* 3: O último filtro é um filtro com vazão pequena. Possui $profundidade \geq 0.8$ e < 0.9 e $fator\_continua\_recursao > 0.8$ justamente para selecionar apenas soluções que tem grande índice de obterem valores iguais ou maiores que o valor atual. E quando chega em 80% da altura da recursão, verifica-se se as soluções possuem pelo menos 82% do valor da maior solução encontrada até o momento.

3.4.2. Técnica *Bound* 2

A segunda técnica utiliza todos os procedimentos da primeira (Seção 3.4.1) incrementando um novo parâmetro na fórmula de cálculo de erro relativo.

Para isso, deve-se antes realizar um cálculos sobre as colunas de prêmios. Quando os prêmios são lidos, é realizado uma operação adicionado fazendo um cálculo da média dos valores de cada coluna da matriz. Com isso, teremos um vetor onde cada índice representa uma coluna da matriz de prêmios e seu valor a média dos valores desta.

Tendo este vetor de médias, alteramos a fórmula de Erro relativo adicionado o valor da respectiva coluna. A fórmula passará a ser

$$fator\_continua\_recursao = \frac{premio\_parcial + media\_coluna\_premios[coluna\_atual]}{melhor\_premio}$$

A ideia desta nova técnica de *bound* é tentar encontrar uma nova maneira mais segura de realizar podas na árvore de recursões evitando podas ou recursões indesejadas. Esta fórmula resulta também na erro percentual entre as duas soluções.

Portanto, como esta técnica baseia na técnica da Seção 3.4.1, tem-se os seguintes filtros:

Filtro de *bound* 1: O primeiro filtro possui $profundidade \geq 0.6$ e < 0.7 e $fator\_continua\_recursao > 0.6$.

Filtro de *bound* 2: O segundo filtro possui $profundidade \geq 0.7$ e < 0.8 e $fator\_continua\_recursao > 0.7$.

Filtro de *bound* 3: O último filtro possui $profundidade \geq 0.8$ e < 0.9 e $fator\_continua\_recursao > 0.8$.

3.4.3. Analisando a Estratégia de *Bound*

Alguns pontos importantes a serem discutidos sobre esta técnica:

- Percebeu-se que gerar a solução inicial pode ser um problema. Problemas com instância grande como o *nqp100.txt* e *nqp200.txt* podem ter um certo trabalho para encontrar uma solução válida usando força bruta. Como o tempo do algoritmo está limitado a um intervalo fixo, muito tempo pode ser gasto com a força bruta e pouco com os *Branch and Bounds*.
- Utilizar-se de cortes para uma $profundidade$ pequena, teremos um algoritmo que fará cortes numa na árvore tendo apenas uma visão local, excluindo o resto dos níveis. Não colocou-se nenhum limite em profundidades rasas pelo motivo de que as soluções poderiam ainda não conter uma representação suficiente de seu prêmio para que seja feita uma análise de *bound* sobre elas. É possível ver que mesmo o Filtro 1 seja posto em 60% da árvore, sua taxa é fraca eliminando apenas árvores com valores muito baixo de prêmio.
- Os valores de parâmetros definidos nos filtros foram resultados de uma bateria de testes realizados em algumas instâncias a procura de parâmetros suficientes bons para obter bons resultados.

3.5. Marcação do Tempo Limite de Execução

O trabalho possui um requisito fundamental que é a execução com um limite de tempo definido previamente.

Para o cálculo deste valor interrompendo o algoritmo utilizou-se de uma estratégia simples de verificação de tempo. Ela é descrita a seguir:

1. Após a inicialização das variáveis, conta-se o tempo atual em horas;
2. Definido o tempo atual, define-se o tempo limite que é obtido pela fórmula

$$tempo\_limite = tempo\_inicial + intervalo\_limite$$

3. Inicia-se a execução do algoritmo normalmente.
   Como o algoritmo é recursivo e realiza várias chamadas recursivas em seguida, criou-se a estratégia de colocar um verificador de tempo no início de cada chamada. Assim, o primeiro passo realizado a cada chamada é verificar se o tempo atual excedeu o $tempo\_limite$. Ou seja,

$$tempo\_agora > tempo\_limite$$

4. Enquanto não tiver extrapolado, não realiza nenhuma intervenção e volta para o Passo 3;
   Na primeira chamada recursiva que o tempo estiver extrapolado o limite, a recursão será cancelada utilizando o comando `return ;`, impedindo a continuação de processamento das rainhas e assim finalizando o procedimento recursivo, direcionando-o para o fim do algorimo;

5. Com o fim da execução da recursão, exibe o maior prêmio encontrado e finaliza o programa.

Deve-se atentar que a contagem de tempo é iniciada somente quando o procedimento de *Branch and Bound* é iniciado. Procedimentos de inicialização e finalização não participal da contagem.

3.6. Entrada de Argumentos por Linha de Comando

Para a execução correta do programa, deve-se utilizar entrada de argumentos por linha de comando.

Os argumentos são respectivamente:

1. Caminho do arquivo de entrada com a quantidade de colunas e com os valores de prêmio;
2. Semente para o gerador de número aleatório;
3. Valor de intervalo de tempo de processamento em segundos;
4. Valor booleando para impressão de informações internas de *debug* na tela. Deve-se colocar o valo 0 (zero) para a execução dos testes de tempo.

4. Execução

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 de cada instância fornecido pelo professor.

4.1. Instâncias

As instâncias seguem um formato padrão onde a primeira linha informa a quantidade de colunas da matriz. Sendo n a quantidade de colunas, as n^2 linhas a seguir do arquivo representarão os prêmios de cada coluna respectivamente.

As instâncias disponibilizadas para testes são descritas na Tabela 1.

Tabela 1. Tabela com as informações das Instâncias.

Nome do Arquivo	Quantidade de Colunas	Maior Prêmio
nqp005.txt	5	167
nqp008.txt	8	298
nqp010.txt	10	381
nqp020.txt	20	883
nqp030.txt	30	1372
nqp040.txt	40	1883
nqp050.txt	50	2380
nqp060.txt	60	2874
nqp070.txt	70	Desconhecido
nqp080.txt	80	Desconhecido
nqp090.txt	90	Desconhecido
nqp100.txt	100	Desconhecido
nqp200.txt	200	Desconhecido

Os valores ótimos conhecidos foram calculados com programação inteira pelo próprio professor.

4.2. 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)

4.3. Análise Estática e Dinâmica de Código

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

4.3.1. *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*:

```
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$
```

Relatório do *Branch and Bound* 1:

```
Marooned:bin pripyat$ ./scan-build n-queens-prize-branchAndBound-1.c
scan-build: Using '/Users/pripyat/Downloads/checker-278/bin/clang' for static analysis
Can't exec "n-queens-prize-branchAndBound-1.c": No such file or directory at ./scan-build line 1094.
scan-build: Removing directory '/var/folders/mc/3p0xb099489gmjk9pfzgs5_m0000gn/T/scan-build-2016
-06-09-001501-61598-1' because it contains no reports.
scan-build: No bugs found.
Marooned:bin pripyat$
```

Relatório do *Branch and Bound* 2:

```
Marooned:bin pripyat$ ./scan-build n-queens-prize-branchAndBound-2.c
scan-build: Using '/Users/pripyat/Downloads/checker-278/bin/clang' for static analysis
Can't exec "n-queens-prize-branchAndBound-2.c": No such file or directory at ./scan-build line 1094.
scan-build: Removing directory '/var/folders/mc/3p0xb099489gmjk9pfzgs5_m0000gn/T/scan-build-2016
-06-09-001506-61603-1' because it contains no reports.
scan-build: No bugs found.
Marooned:bin pripyat$
```

4.3.2. *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*.

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.

5. Resultados

Como já mencionado na Seção 4, 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 8. 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
nqp010.txt	381.0	0.0	381.0	0.0	381.0	0.0	381
nqp020.txt	757.9167	19.17128	793.5	17.40167	789.0833	18.56418	883
nqp030.txt	995.3333	39.35695	1075.0	47.86724	1082.417	43.58369	1372
nqp040.txt	1263.417	81.3773	1391.083	93.53702	1395.917	72.40977	1883
nqp050.txt	1529.5	60.92544	1658.0	70.13753	1691.417	50.86963	2380
nqp060.txt	1768.917	82.7004	1883.25	119.031	1922.917	109.1142	2874
nqp070.txt	2019.25	136.8231	2168.917	206.2084	2169.0	215.2462	Desconhecido
nqp080.txt	2086.167	674.6505	2241.917	724.6419	2254.167	725.4341	Desconhecido
nqp090.txt	2553.5	111.8835	2727.0	126.961	2737.167	126.1073	Desconhecido
nqp100.txt	2534.083	805.3561	2648.0	848.3781	2651.583	843.5402	Desconhecido
nqp200.txt	3543.75	2621.176	3543.75	2621.176	3543.75	2621.176	Desconhecido

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

5.1. Resultados em Gráficos

A Figura 1 exibe os valores médios obtidos na instância nqp005.txt.

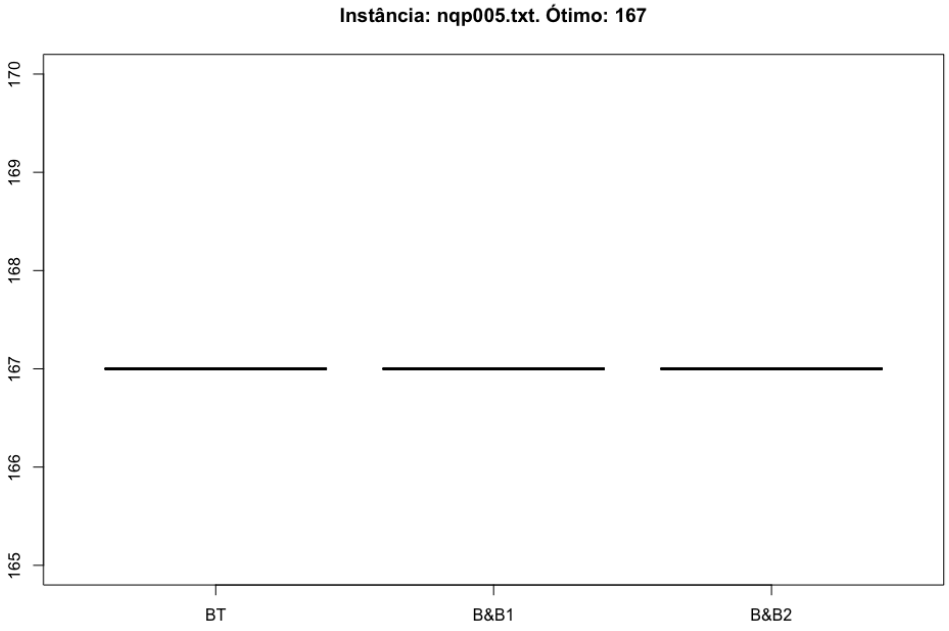

Figura 1. BoxPlot da Instância nqp005.txt.

A Figura 2 exibe os valores médios obtidos na instância nqp008.txt.

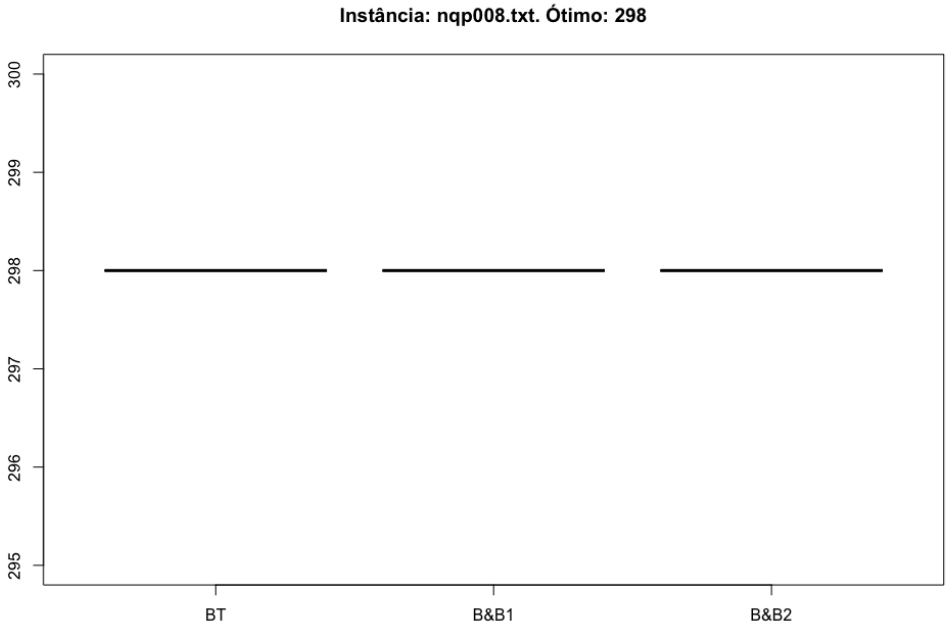

Figura 2. BoxPlot da Instância nqp008.txt.

A Figura 3 exibe os valores médios obtidos na instância nqp010.txt.

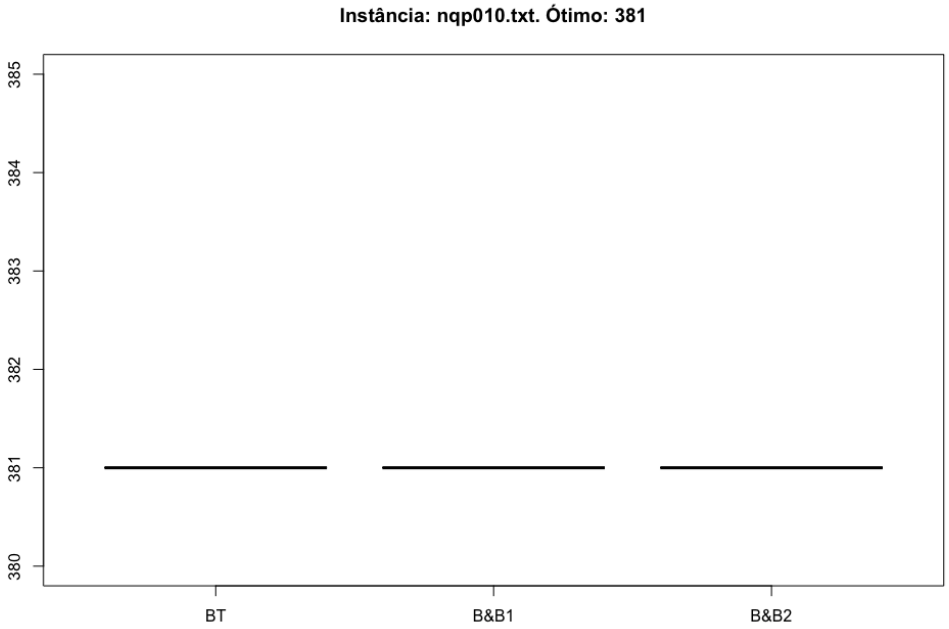

Figura 3. BoxPlot da Instância nqp010.txt.

A Figura 4 exibe os valores médios obtidos na instância nqp020.txt.

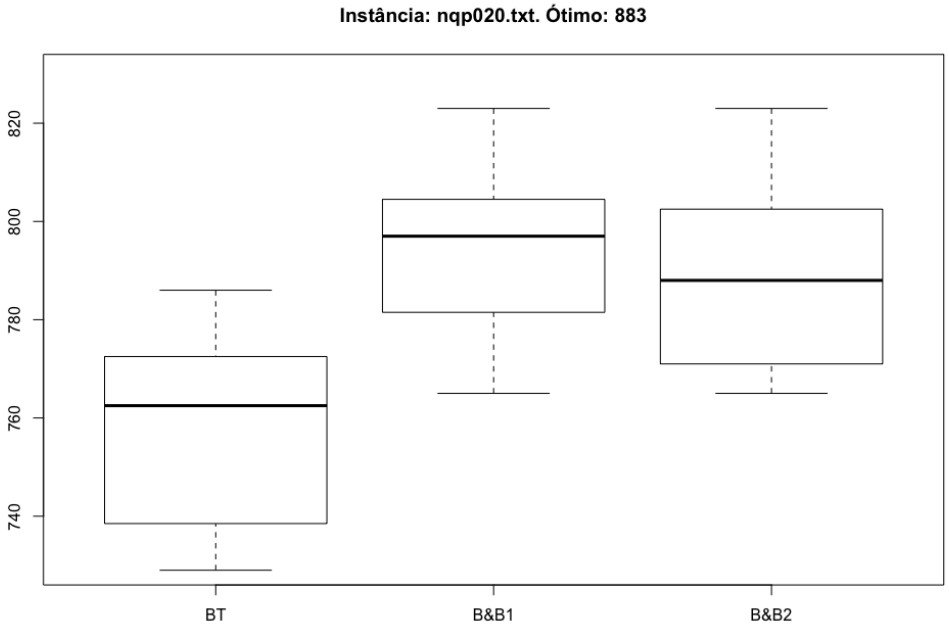

Figura 4. BoxPlot da Instância nqp020.txt.

A Figura 5 exibe os valores médios obtidos na instância nqp030.txt.

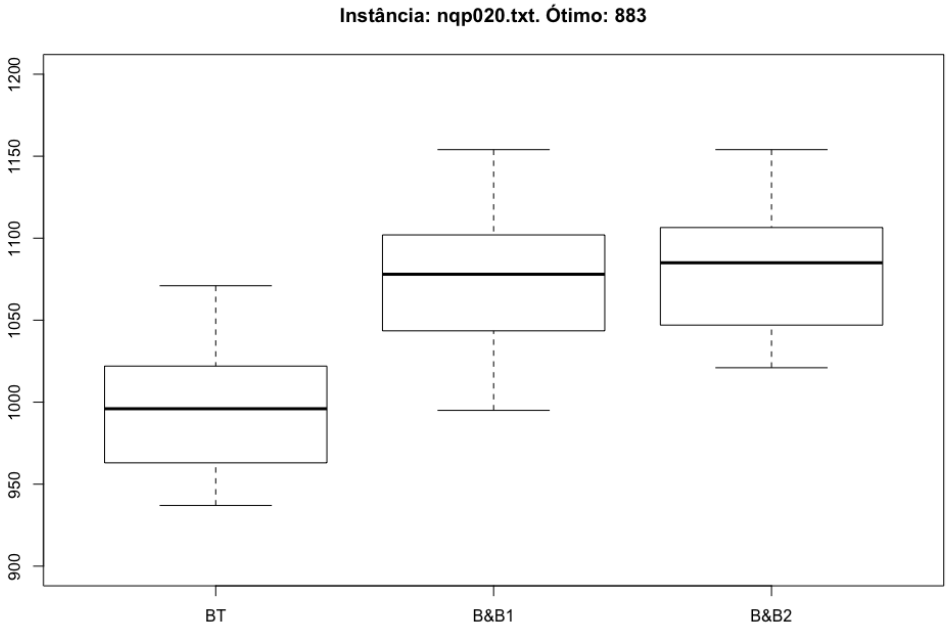

Figura 5. BoxPlot da Instância nqp030.txt.

A Figura 6 exibe os valores médios obtidos na instância nqp040.txt.

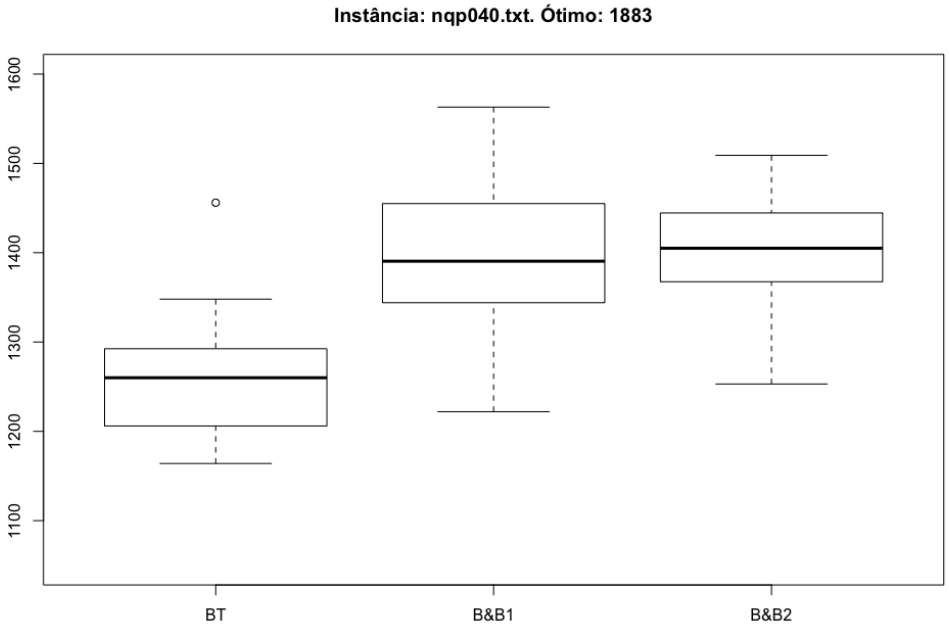

Figura 6. BoxPlot da Instância nqp040.txt.

A Figura 7 exibe os valores médios obtidos na instância nqp050.txt.

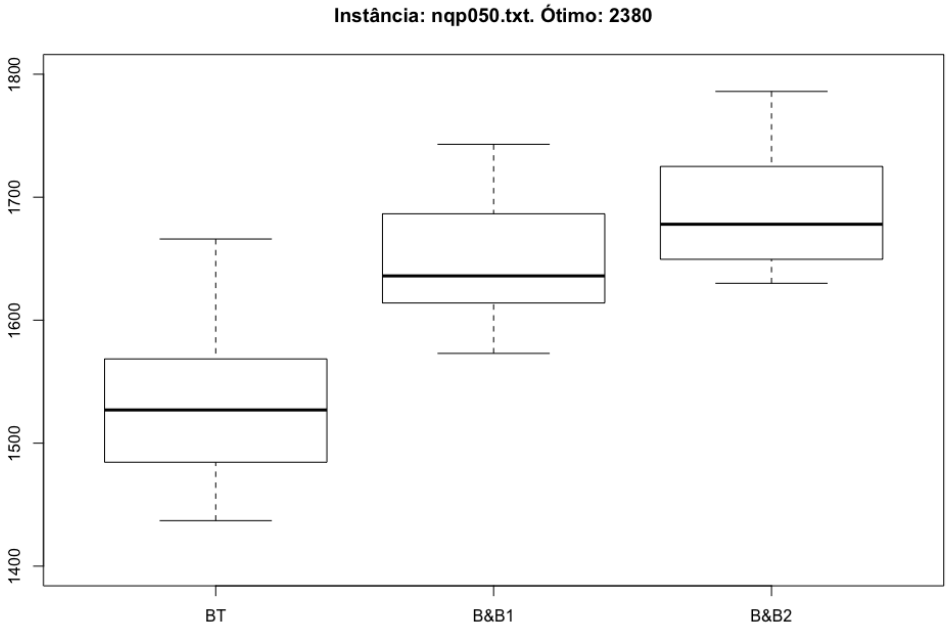

Figura 7. BoxPlot da Instância nqp050.txt.

A Figura 8 exibe os valores médios obtidos na instância nqp060.txt.

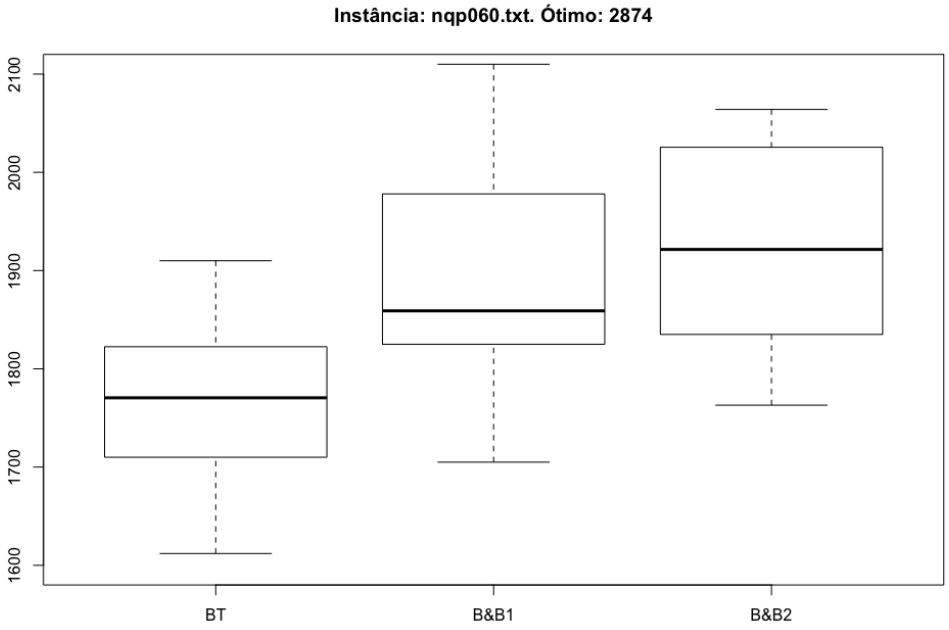

Figura 8. BoxPlot da Instância nqp060.txt.

A Figura 9 exibe os valores médios obtidos na instância nqp070.txt.

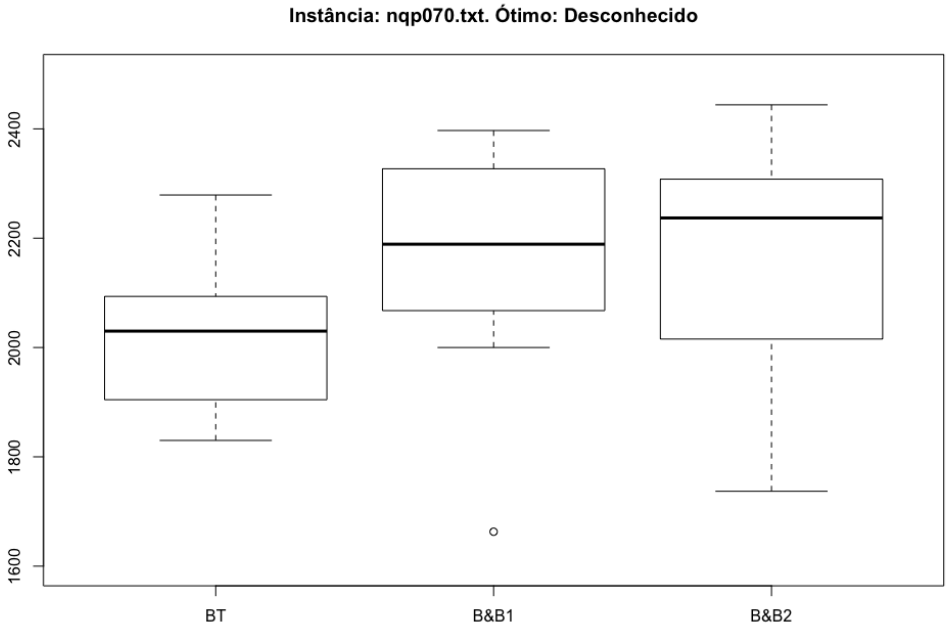

Figura 9. BoxPlot da Instância nqp070.txt.

A Figura 10 exibe os valores médios obtidos na instância nqp080.txt.

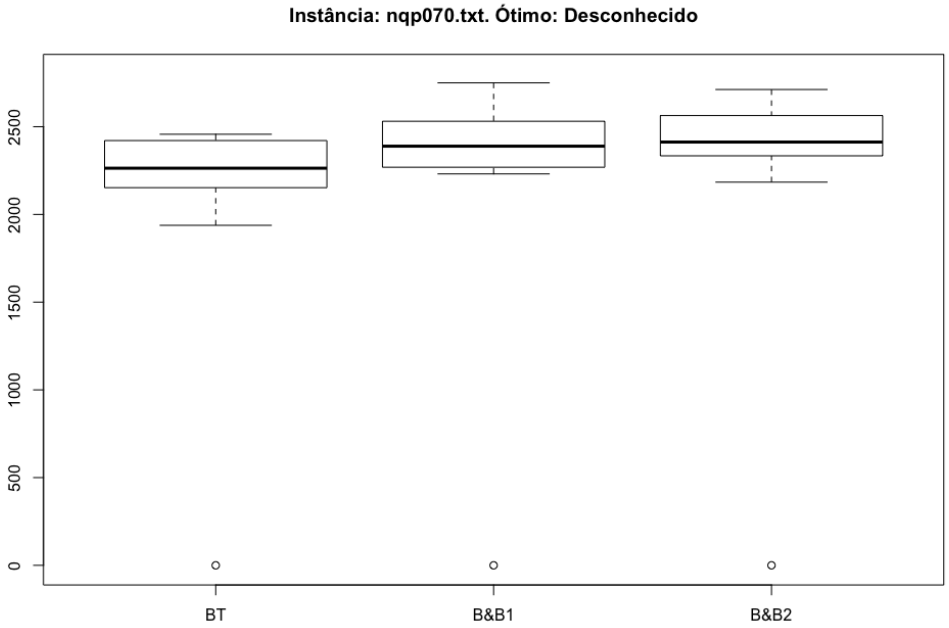

Figura 10. BoxPlot da Instância nqp080.txt.

A Figura 11 exibe somente os maiores valores médios obtidos na instância nqp080.txt, para uma fácil visualização dos melhores valores obtidos.

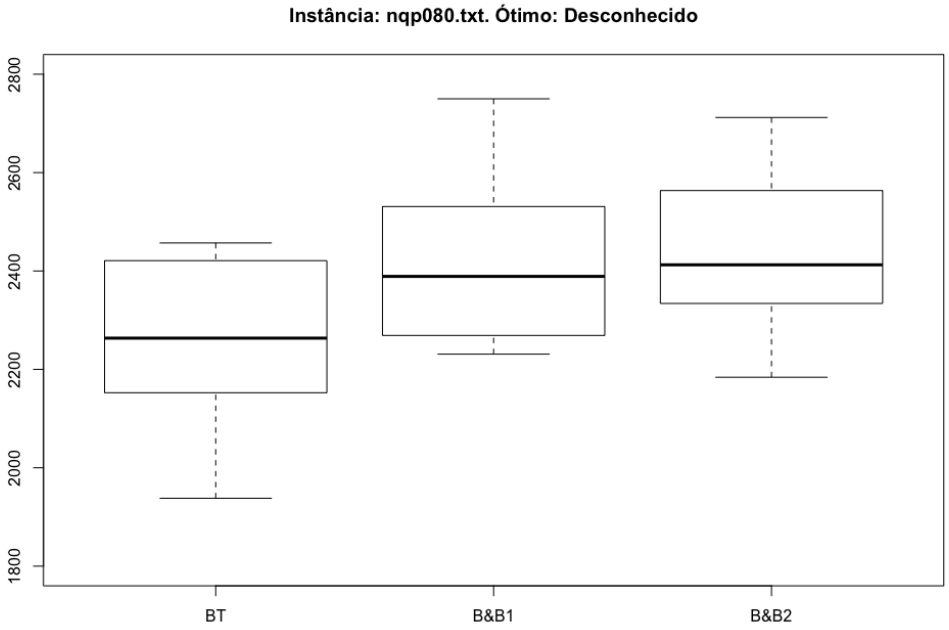

Figura 11. BoxPlot da Instância nqp080.txt.

A Figura 12 exibe os valores médios obtidos na instância nqp090.txt.

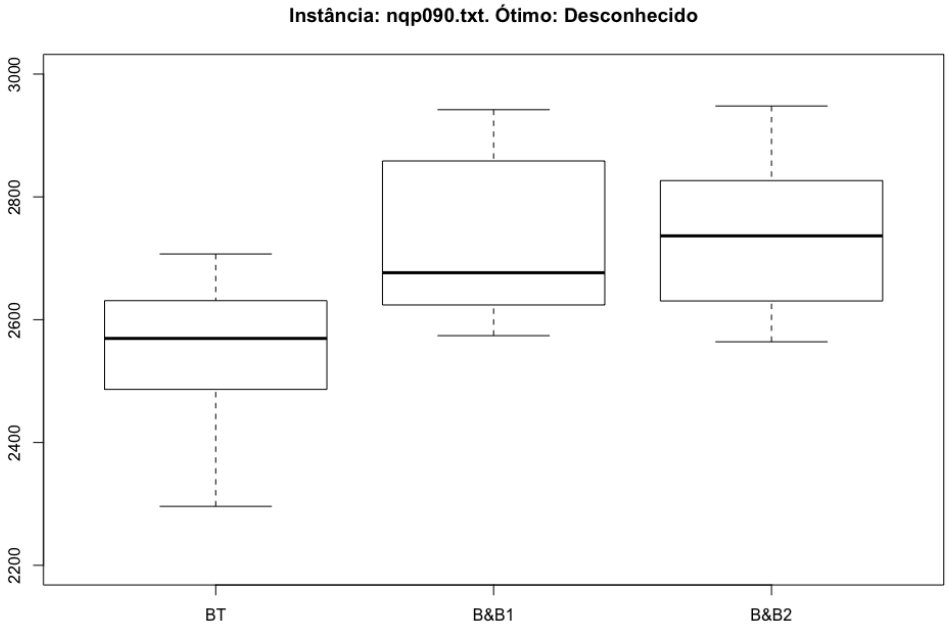

Figura 12. BoxPlot da Instância nqp090.txt.

A Figura 13 exibe os valores médios obtidos na instância nqp100.txt.

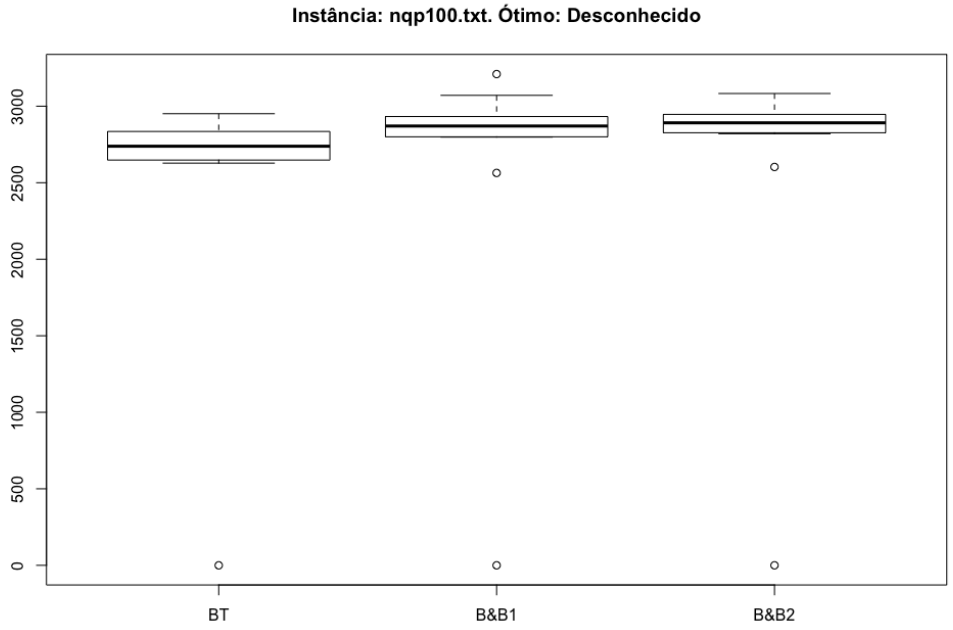

Figura 13. BoxPlot da Instância nqp100.txt.

A Figura 14 exibe somente os maiores valores médios obtidos na instância nqp100.txt, para uma fácil visualização dos melhores valores obtidos.

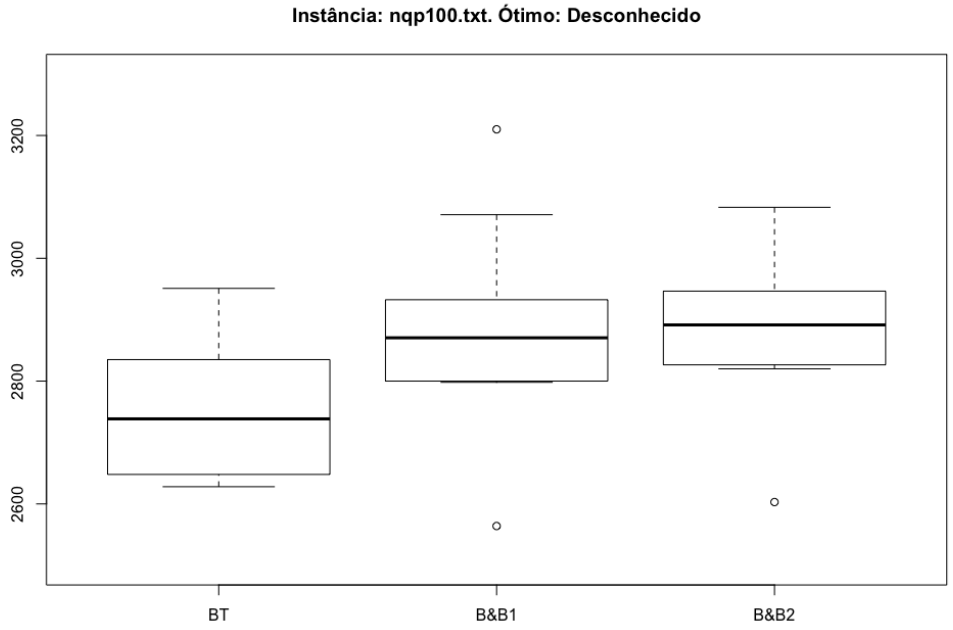

Figura 14. BoxPlot da Instância nqp100.txt.

A Figura 15 exibe os valores médios obtidos na instância nqp020.txt.

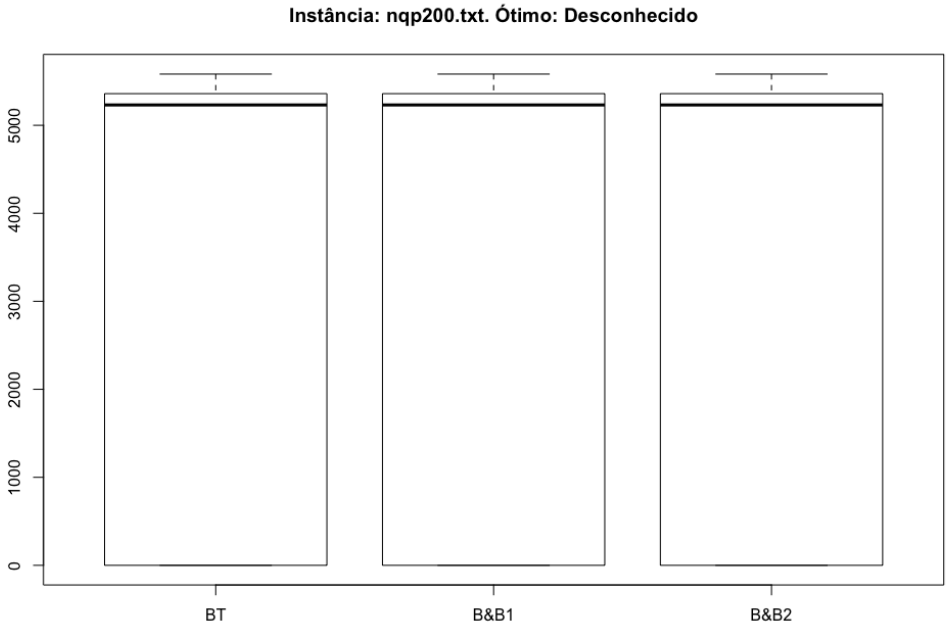

Figura 15. BoxPlot da Instância nqp200.txt.

A Figura 16 exibe somente os maiores valores médios obtidos na instância nqp200.txt, para uma fácil visualização dos melhores valores obtidos.

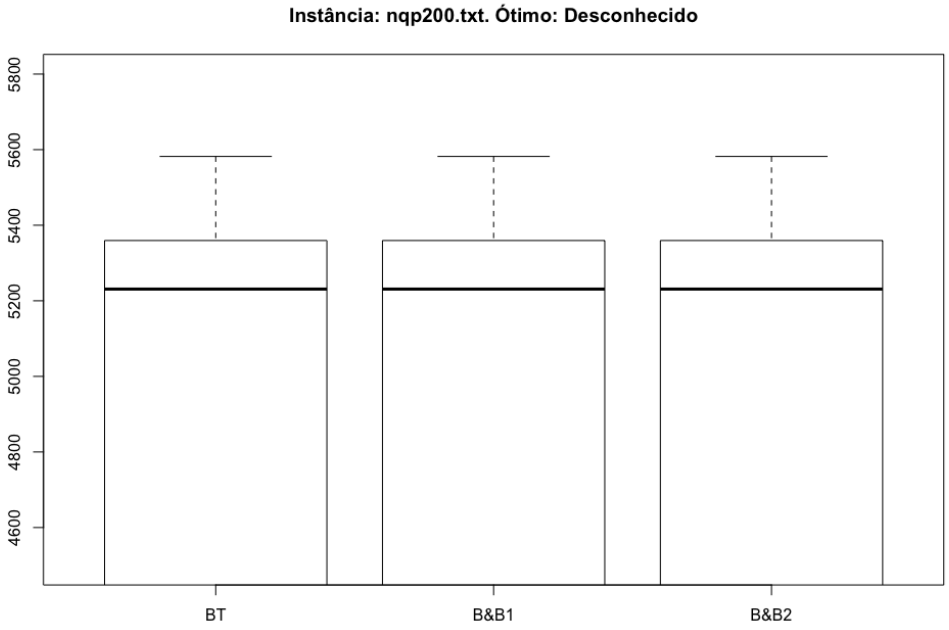

Figura 16. BoxPlot da Instância nqp200.txt.

6. 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.

Foram implementados um total de três algoritmos, sendo um com técnica *Backtracking* e outros dois variantes com *Branch and Bound*.

A primeira implementação utiliza-se de um *Backtracking* simples que verifica todas as soluções possíveis. A segunda e terceira é uma variação do primeiro algoritmo adicionando filtros que verificam quão bom é determinado subramo. Foi implementados dois tipos de filtros diferentes e por este motivo, implementou-se dois algoritmos *Branch and Boudn* com o propósito de comparar os resultados dos três algoritmos sobre o mesmo intervalo de tempo.

É possível ver que o *B&B 2* obteve melhores resultados na maioria das instâncias, principalmente nas maiores instâncias. Entretanto, não foi um resultado consideravelmente maior já que o algoritmo de *BT* também obteve um resultado tão bom quanto. A análise desses resultados são descritos à seguir.

Percebeu-se que os algoritmos implementados utilizam de muitas divisões o que torna seu processamento custoso. É claro que o compilador realiza otimizações automáticas visando a melhora da execução do código, entretanto, em alto nível, ainda existe forma de aprimorá-lo reduzindo a quantidade de operações repetidas armazenando seu resultado numa variável e utilizá-la quando necessário usando de memória primária para melhorar o tempo de processamento.

A definição de parâmetros de execução dos filtros também é um item fundamental para uma execução com boas soluções num intervalo de tempo. Parâmetros muito restritos podem eliminar soluções que a priori são insuficientes, mas que podem ter bons resultados no decorrer do algoritmo ou até mesmo não encontrando nenhum. Já parâmetros com valores relaxados fazem com que o o *Branch and Bound* aproxime-se da técnica *Backtracking* voltando o problema inicial.

Não somente a variação dos parâmetros, mas também o gerenciamento de novos filtros. A adição ou remoção de filtros é uma variação de algoritmo de *Branch and Bound* que também permite filtrar os resultados usando um novo método. Lembrando que qualquer variação realizada, dever-se-á configurar os parâmetros a fim de obter bons resultados em tempo reduzido.

E possivelmente o fator mais importante a ser trabalhado neste algoritmo seria a execução do algoritmo de força bruta inicial. Para casos onde o intervalo de tempo é limitado (tal como este trabalho), o algoritmo de força bruta inicial (*Bactracking*) que é executado em todos os três algoritmos, torna item fundamental para uma boa solução inicial. Dependendo do tamanho da instância, seu tempo de execução pode ser ruim o suficiente para utilizar o tempo limite todo para encontrar uma solução inicial válida. Observando os resultados da Tabela 3 e 4 é possível ver que os valores para todos os algoritmos foram exatamente os mesmos na última instância. Pode-se concluir então que no intervalo de tempo de 60 segundos, somente a solução inicial foi executada não permitindo assim a execução do *Branch and Bound*. Assim, ter um algoritmo para geração

de uma solução inicial rápida, é fundamental para obter um *Branch and Bound* mais eficiente. Com o algoritmo de força bruta inicial mais veloz, é possível que o *Branch and Bound* realize mais podas (mesmo que pequenas) saltando caminhos que possuem soluções ruins e conseguindo resultados melhores ao longo de sua execução.

7. Código dos Algoritmos

Os algoritmos *Shell Script*, *Backtracking*, *Branch and Bound 1* e *Branch and Bound 2* estão distribuídos nas páginas 21, 22, 38 e 38 respectivamente.

7.1. *Shell Script*

```
#!/bin/bash

echo "Quantas iteracoes?"
read quantidade_iteracoes;
echo

eval "rm algori*"
eval "gcc n-Queens-Prize-Backtracking/n-queens-prize-backtracking.c -Ofast -o n-Queens-Prize-Backtracking/n-Queens-Prize-Backtracking"
eval "gcc n-Queens-Prize-BranchAndBound-1/n-queens-prize-branchAndBound-1.c -Ofast -o n-Queens-Prize-BranchAndBound-1/n-Queens-Prize-BranchAndBound-1"
eval "gcc n-Queens-Prize-BranchAndBound-2/n-queens-prize-branchAndBound-2.c -Ofast -o n-Queens-Prize-BranchAndBound-2/n-Queens-Prize-BranchAndBound-2"

instancias=( nqp005.txt nqp008.txt nqp010.txt nqp020.txt nqp030.txt nqp040.txt nqp050.txt nqp060.txt nqp070.txt nqp080.txt nqp090.txt nqp100.txt nqp200.txt )

algoritmos=( n-Queens-Prize-Backtracking n-Queens-Prize-BranchAndBound-1 n-Queens-Prize-BranchAndBound-2 )

for algoritmo in "${algoritmos[@]}"
do
   echo $algoritmo

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

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

         cmd="./$algoritmo/$algoritmo $instancia $i*1234 60 0"
         date
         echo $cmd
         $cmd
```

```
31        done
32        echo
33
34    done
35    echo
36
37 done
```

7.2. *Backtracking*

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  // variável global sobre a quantidade de colunas/linhas da matriz
6  int global_tamanho_matriz = -1;
7  // variável que ativa impressoes na tela
8  char global_imprime_tela_ativado = 0;
9
10 // Variáveis de tempo para cálculo do intervalo de tempo de execução
11 time_t endwait, start;
12
13
14 /*
15  * Estrutura que representa um tabuleiro.
16  *
17  * Um ponteiro para um vetor de onde representa cada posicao da rainha
    ↪  na coluna
18  * O valor de prêmrio do tabuleiro no estado atual
19  */
20 typedef struct Struct_Tabuleiro {
21    int * colunas;
22    int   premio;
23 } Tabuleiro;
24
25
26 /*
27  * Estrutura de representação de Fila
28  *
29  * Esta fila é utilizada para organizar a ordem de seleção das rainhas
30  */
31 typedef struct Struct_Fila {
32    int           lugar; // Posição da rainha
33    struct Struct_Fila * proximo;
34 } Fila;
35
```

```
36
37
38  /*
39   * Procedimento que verifica se a fila passada por parâmetro está
     ↪  vazia.
40   *
41   * Retorna valores boleanos
42   */
43  char Fila_Esta_Vazia(Fila * f) {
44     if (f == NULL) {
45        return 1;
46     } else
47        return 0;
48  }
49
50
51
52  /*
53   * Procedimento que adiciona um novo valor à fila
54   */
55  void Enfilera(Fila ** f, int l) {
56     Fila * atual = 0, * novo = 0;
57
58     // Cria um novo nó e adiciona informações
59     novo = calloc(1, sizeof(Fila));
60     novo->lugar = l;
61     novo->proximo = NULL;
62
63
64     // Se a fila estiver vazia, coloca na cabeça
65     if (Fila_Esta_Vazia(*f)) {
66
67        * f = novo;
68
69     } else {
70        // Caso contrário, coloca na calda
71
72        atual = * f;
73
74        while(atual->proximo != NULL) {
75           atual = atual->proximo;
76        }
77
78        atual->proximo = novo;
79     }
80  }
```

```
81
82
83
84  /*
85   * Procedimento que realiza a retirada de um elemento da fila seguindo
     ↪  sua natureza
86   */
87  int Desenfilera(Fila ** f) {
88     Fila * retirar = 0, * fila = * f;
89     int posicao = 0;
90
91     // Se a fila estiver vazia, informa erro e termina o programa
92     if (Fila_Esta_Vazia(*f)) {
93        printf("\n\n[ERRO] Impossível retirar de uma fila vazia.
           ↪  \tFinalizando o programa.");
94        return -1;
95     } else {
96        // caso contrário, retira o primeiro elemento.
97        posicao = fila->lugar;
98        retirar = fila;
99
100       fila = fila->proximo;
101
102       free(retirar);
103
104       *f = fila;
105
106       return posicao;
107    }
108 }
109
110
111
112 /*
113  * Procedimento que realiza o processo de retirar todos os elementos da
     ↪  fila
114  */
115 void Desaloca_Fila(Fila ** f) {
116    Fila * retirar = 0;
117
118    // Retira todos os elementos da fila
119    while (*f != NULL) {
120       retirar = *f;
121
122       *f = (*f)->proximo;
123
```

```
124         free (retirar);
125     }
126 
127 }
128 
129 
130 
131 /*
132  * Procedimento para impressão da Fila na tela
133  */
134 void Imprime_Fila(Fila * f) {
135     Fila * atual = f;
136 
137     if (global_imprime_tela_ativado) {
138 
139         printf("\n\n\t\t");
140         while (atual != NULL) {
141             printf("%d -> ", atual->lugar);
142             atual = atual->proximo;
143         }
144         printf("NULL");
145 
146         fflush(stdout);
147     }
148 }
149 
150 
151 
152 /*
153  * Procedimento responsável por gerar uma fila de posições de rainhas
     ↪  em ordem aleatória.
154  *
155  * Primeiro cria-se um vetor com os valores de 1..n ordenado
     ↪  simbolizando a ordem das rainhas.
156  * Em seguida, é gerado dois valores inteiros posicao1 e posicao2 de
     ↪  forma aleatória e estes
157  *       são utilizados para a comutação dos valores situados em
     ↪  vetor[posicao1] e vetor[posicao2]
158  * No final de algumas iterações, estes respectivos valores já
     ↪  desordenados são copiados para
159  *       a fila para que o algoritmo use.
160  */
161 Fila * Cria_Fila_Aleatoria() {
162     Fila * f = 0, * cabeca = 0;
163 
164     int vetor_valores_fila [global_tamanho_matriz],
```

```
165         i = 0,
166         permutacoes = global_tamanho_matriz * 10,
167         posicao1 = 0, posicao2 = 0, temp = 0;
168
169     // Cria-se um vetor com valores ordenados de 1..n_rainhas
170     for (i = 0; i < global_tamanho_matriz; i++) {
171        vetor_valores_fila[i] = i;
172     }
173
174     // Realiza várias permutações dos valores deste vetor de acordo com
        ↪  os dois índices gerados
175        // Realiza-se global_tamanho_matriz * 10 permutações
176     i = 0;
177     while (i < permutacoes) {
178        // Gera primeiro índice
179        posicao1 = rand() % global_tamanho_matriz;
180
181        // Gera segundo índice
182        posicao2 = rand() % global_tamanho_matriz;
183
184        // Realiza a comutação dos valores dos respectivos índices
185        temp = vetor_valores_fila[posicao1];
186        vetor_valores_fila[posicao1] = vetor_valores_fila[posicao2];
187        vetor_valores_fila[posicao2] = temp;
188
189        i++;
190     }
191
192     // Aloca a fila e adiciona os valores após realizar a desordem
193     cabeca = calloc(1, sizeof(Fila));
194
195     f = cabeca;
196
197     // Copia os valores do vetor para a fila
198     i = 0;
199     while (i < global_tamanho_matriz - 1) {
200
201        f->lugar = vetor_valores_fila[i];
202
203        f->proximo = calloc(1, sizeof(Fila));
204
205        f = f->proximo;
206
207        i++;
208     }
209
```

```c
    f->lugar = vetor_valores_fila[i];

    f->proximo = NULL;

    // Retorna a fila desordenada
    return cabeca;
}



/*
 * Procedimento para retorndo do de prêmio específico de uma célula
 */
int Retorna_Premio(int * premios, int linha, int coluna) {

    // Acessa o vetor como se fosse uma matriz comum nxn.
    return premios[linha * global_tamanho_matriz + coluna];
}



/*
 * Procedimento para realizar a leitura dos arquivo prêmio
 *
 * Este também armazena o maior premio lido e o retorna.
 */
void Le_Premios(char * diretorio, int ** premios){
    FILE * file = 0;
    int i = 0;
    int premio_lido = 0;
    int quantidade_celulas = 0;

    // Abre o arquivo de prêmios em forma de leitura
    file = fopen(diretorio, "r");

    // Verifica se a abertura foi feita com sucesso
    if (file != NULL) {

        //lê a primeira informação (quantas linhas existem)
        fscanf(file, "%d", &global_tamanho_matriz);
        if (global_imprime_tela_ativado)
            printf("\n[INFO] Quantidade de colunas do tabuleiro = %d.\n",
              ↪  global_tamanho_matriz);

        // Se o valor de tamanho da matriz for um valor válido
        if (global_tamanho_matriz > 0) {
```

```
255
256             quantidade_celulas = global_tamanho_matriz *
                 ↪  global_tamanho_matriz;
257
258             // Aloca a matriz de prêmios
259             * premios = calloc(quantidade_celulas, sizeof(int));
260
261             // Começa a coleta dos prêmios
262             fscanf(file, "%d", &premio_lido);
263
264             // Salvas os prêmios e coleta o próximo
265             while(i < quantidade_celulas) {
266
267                 if (global_imprime_tela_ativado) {
268                     // Imprime na tela o prêmio lido
269                     printf("\n[INFO] Lendo premio [%d,%d] = %d.", i /
                         ↪  global_tamanho_matriz + 1, i %
                         ↪  global_tamanho_matriz + 1, premio_lido);
270                     fflush(stdout);
271                 }
272
273                 // Salva no vetor
274                 (*premios)[i++] = premio_lido;
275
276                 // Lê o próximo prêmio
277                 fscanf(file, "%d", &premio_lido);
278             }
279         } else {
280             // Informa Erro
281             printf("\n[ERRO] Quantidade de rainhas insuficiente!\n\n");
282             exit(-1);
283         }
284
285         // Fecho o arquivo
286         fclose(file);
287     }
288     else {
289         // Informa Erro
290         printf("\n[ERRO] Falha na leitura do arquivo de
             ↪  configuração!\n\n");
291         exit(-1);
292     }
293 }
294
295
296
```

```
297 /*
298  * Como utiliza-se uma estrutura fila com todas as opções e sem
    ↪  repetição, não
299  *      existe a possibilidade de duas rainhas ficarem numa mesma linha
300  *      vertical, horizontal (isso pois os valores não se repetem).
301  *
302  * Com isso, basta verificar se as diagonais estão conflitando.
303  */
304 char Posicao_Eh_Valida(Tabuleiro t, int col, int pos) {
305    int i = 0;
306    char esquerda_diagonal = 0, direita_diagonal = 0;
307
308    // Inicializa dizendo que as diagonais não estão ocupadas
309    esquerda_diagonal = direita_diagonal = 1;
310
311    // Da posição da rainha até a coluna 0, faça:
312    i = col;
313    while (i > 0) {
314
315       // Verifica:
316       //      Diagonal esquerda-direita
317       if (t.colunas[col - i]  == pos + i) {
318          esquerda_diagonal = 0;
319
320       // Verifica:
321       //      Diagonal direita-esquerda
322       } else
323          if (t.colunas[col - i] == pos - i) {
324          direita_diagonal = 0;
325       }
326
327       // Verifica validade
328       // Caso alguma diagonal já esteja ocupada, aborta o procedimento
329          // informando que esta posição é inválida
330       if (esquerda_diagonal == 0 || direita_diagonal == 0) {
331          return 0;
332       }
333
334       i--;
335    }
336
337    // Verifica validade das diagonais
338    if (esquerda_diagonal == 1 && direita_diagonal == 1)
339       return 1;
340    else
341       return 0;
```

```
342 }
343
344
345
346 /*
347  * Procedimento que copia os dados de um tabuleiro para outro.
348  *
349  * Este procedimento é utilizando quando encontra-se um novo valor de
     ↪ prêmio maior
350  *      que o atual e assim, realiza-se a substituição do tabuleiro
     ↪ antigo pelo novo
351  *      encontrado.
352  */
353 void Copia_Novo_Tabuleiro(Tabuleiro tabuleiro_atual, Tabuleiro *
     ↪ tabuleiro_maior) {
354   int i = 0;
355
356   tabuleiro_maior->premio           = tabuleiro_atual.premio;
357
358   for (i = 0; i < global_tamanho_matriz; i++) {
359      tabuleiro_maior->colunas[i]    = tabuleiro_atual.colunas[i];
360   }
361 }
362
363
364
365 /*
366  * Procedimento que adiciona uma nova rainha no tabuleiro já calculando
     ↪ o prêmio desta.
367  *
368  * Este procedimento não precisa verificar a validade da posição já que
     ↪ este já foi calculado
369  *      quando a rainha foi selecionada.
370  */
371 void Adiciona_Rainha_Tabuleiro_Calculando_Premio(Tabuleiro * t, int
     ↪ posicao,  int r, int * premios) {
372
373   t->premio += Retorna_Premio(premios, r, posicao);
374   t->colunas[posicao] = r;
375 }
376
377
378
379 /*
380  * Procedimento que retira a rainha do tabuleiro calculando o novo
     ↪ prêmio
```

```
381   */
382  void Retira_Rainha_Tabuleiro_Calculando_Premio(Tabuleiro * t, int
      ↪  posicao, int r, int * premios) {
383     t->premio -= Retorna_Premio(premios, r, posicao);
384     t->colunas[posicao] = -1;
385  }
386
387
388
389  /*
390   * Procedimento que inicializa um novo tabuleiro
391   */
392  Tabuleiro * Cria_Tabuleiro(){
393     int i = 0;
394     Tabuleiro * t = 0;
395
396     t = calloc(1, sizeof(Tabuleiro));
397     t->colunas = calloc(global_tamanho_matriz, sizeof(int));
398
399     for (i = 0; i < global_tamanho_matriz; i++)
400        t->colunas[i] = -1;
401
402     return t;
403  }
404
405
406
407  /*
408   * Procedimento que imprime o tabuleiro junto com os prêmios
409   */
410  void Imprime_Tabuleiro (Tabuleiro t, Tabuleiro maior, int * premios) {
411     int i = 0, j = 0;
412
413     if (global_imprime_tela_ativado) {
414
415        printf("\n");
416
417        for (i = 0; i < global_tamanho_matriz; i++) {
418              printf("\n\t");
419           for (j = 0; j < global_tamanho_matriz * 2; j++) {
420              if ( j < global_tamanho_matriz) {
421                 if (i == t.colunas[j])
422                    printf("%3d ", i);
423                 else
424                    printf(" -1 ");
425              }
```

```
426                 else {
427                     if (j == global_tamanho_matriz)
428                         printf("\t\t");
429                     printf("%3d ", premios[i * global_tamanho_matriz + j -
                        ↪  global_tamanho_matriz]);
430                 }
431 
432             }
433 
434             fflush(stdout);
435         }
436 
437         printf("\tPremio_Atual: %d;\tMaior_Premio: %d.", t.premio,
                ↪  maior.premio);
438 
439         fflush(stdout);
440     }
441 }
442 
443 
444 /*
445  * Procedimento que imprime o tabuleiro, junto com os prêmios
        ↪  informando fim da execução
446  */
447 void Imprime_Tabuleiro_Final (Tabuleiro maior, int * premios) {
448     int i = 0, j = 0;
449     FILE * saida = 0;
450 
451     if (global_imprime_tela_ativado) {
452 
453         printf("\n\n[INFO] Resultado Final do Processamento:");
454 
455         for (i = 0; i < global_tamanho_matriz; i++) {
456                 printf("\n\t");
457             for (j = 0; j < global_tamanho_matriz * 2; j++) {
458                 if ( j < global_tamanho_matriz) {
459                     if (i == maior.colunas[j])
460                         printf("%3d ", i);
461                     else
462                         printf(" -1 ");
463                 }
464                 else {
465                     if (j == global_tamanho_matriz)
466                         printf("\t\t");
467                     printf("%3d ", premios[i * global_tamanho_matriz + j -
                        ↪  global_tamanho_matriz]);
```

```
468                 }
469             }
470 
471             fflush(stdout);
472         }
473 
474         printf("\tPremio: %d.",maior.premio);
475         printf("\n[INFO] Resultado Final do Processamento.\n");
476     } else {
477         saida = fopen("algoritmo0.txt", "a");
478 
479         if (saida) {
480             fprintf(saida, "%d\n", maior.premio);
481         }
482 
483         fclose(saida);
484     }
485 
486     fflush(stdout);
487 }
488 
489 
490 
491 /*
492  * Procedimento que libera memória
493  */
494  void Desaloca_Tabuleiro (Tabuleiro ** f) {
495     if (*f != 0) {
496         free((*f)->colunas);
497         free(*f);
498     }
499  }
500 
501 
502 
503 /*
504  * Procedimento que libera memória
505  */
506 void Desaloca_Premios (int ** premios) {
507     if (*premios != 0)
508         free(*premios);
509 }
510 
511 
512 
513 /*
```

```
514   * Método de branch and bound desenvolvido
515   */
516 void n_Rainhas_Prize(int coluna_atual, Tabuleiro * tabuleiro_atual,
      ↪  Tabuleiro * tabuleiro_maior, Fila ** posicoes_restantes, int *
      ↪  premios) {
517     int iteracoes = 0, linha_atual_temp = 0;
518 
519     // Recebe o tempo atual.
520     start = time(NULL);
521 
522     // Verifica se o tempo excedeu o liminte estabelecido.
523     if (start > endwait) {
524        // Se sim, cancela totalmente a continuação da recursão
525        return ;
526     }
527 
528 
529     // Varre a fila com os valores que sobraram
530        // A cada recursão, um item é retirado
531 
532     // Caso tenha percorrido todas as rainhas deste contexto de
        ↪  recursão,
533        // o while será impedido de ser executando forçando a realizar o
534        // retorno à um nível acima de recursao
535     while (iteracoes < global_tamanho_matriz - coluna_atual) {
536 
537        Imprime_Fila(* posicoes_restantes);
538        Imprime_Tabuleiro(*tabuleiro_atual, *tabuleiro_maior, premios);
539 
540        // Retira um item da fila e salva numa variável local
541        linha_atual_temp = Desenfilera( posicoes_restantes );
542 
543           printf("\nALERTA%d\n", linha_atual_temp);
544        if (linha_atual_temp == -1) {
545           Desaloca_Premios(&premios);
546 
547           Desaloca_Tabuleiro(&tabuleiro_maior);
548           Desaloca_Tabuleiro(&tabuleiro_atual);
549           exit(-1);
550        }
551 
552        Imprime_Fila(* posicoes_restantes);
553        if (global_imprime_tela_ativado) {
554           printf("\t\tBuffer_Atual: %d", linha_atual_temp);
555           fflush(stdout);
556        }
```

```
557
558
559         // Testa a validade da rainha retirada no momento
560            // Se for posição válida realizará o processamento desta
561            // Caso contrário, ela será posta no final da fila=
562         if(Posicao_Eh_Valida(*tabuleiro_atual, coluna_atual,
             ↪  linha_atual_temp)) {
563
564            // Adiciona a rainha no tabuleiro calculando o prêmio com sua
                ↪  inclusão
565            Adiciona_Rainha_Tabuleiro_Calculando_Premio(tabuleiro_atual,
                ↪  coluna_atual, linha_atual_temp, premios);
566
567            Imprime_Tabuleiro(*tabuleiro_atual, *tabuleiro_maior,
                ↪  premios);
568
569
570            // Se a fila estiver vazia, significa que acabou de ser gerado
                ↪  uma solução
571               // válida.
572            // Assim, será verificado se o prêmio é melhor que o atual.
573            if (Fila_Esta_Vazia(* posicoes_restantes)) {
574
575               if (tabuleiro_atual->premio > tabuleiro_maior->premio) {
576
577                  if (global_imprime_tela_ativado)
578                     printf("\n\n[INFO] Novo Recorde Encontrado!");
579                  Copia_Novo_Tabuleiro(*tabuleiro_atual, tabuleiro_maior);
580
581                  Imprime_Tabuleiro(*tabuleiro_atual, *tabuleiro_maior,
                      ↪  premios);
582               }
583
584
585               Imprime_Fila(*posicoes_restantes);
586
587               // Após chegar na folha, a recursão é revertida até que
                   ↪  encontre
588                  // uma próxima solução pra explorar.
589               Enfilera(posicoes_restantes, linha_atual_temp);
590
591               Imprime_Fila(*posicoes_restantes);
592
593               // Como mencionado, a raínha é posta novamente na fila para
                   ↪  a procura
594                  // de novas soluções
```

```
595             Retira_Rainha_Tabuleiro_Calculando_Premio(tabuleiro_atual,
                  ↪  coluna_atual, linha_atual_temp, premios);
596
597
598             return ;
599
600
601         // Se não for a última rainha, então realiza as análises de
              ↪  bound
602         } else {
603
604             n_Rainhas_Prize(coluna_atual + 1, tabuleiro_atual,
                  ↪  tabuleiro_maior, posicoes_restantes, premios);
605
606
607             Enfilera(posicoes_restantes, linha_atual_temp);
608
609             Retira_Rainha_Tabuleiro_Calculando_Premio(tabuleiro_atual,
                  ↪  coluna_atual, linha_atual_temp, premios);
610
611             Imprime_Fila(*posicoes_restantes);
612
613             iteracoes++;
614         } // if
615
616
617     // Se a posição escolhida não for válida
618     } else {
619
620         if(global_imprime_tela_ativado) {
621             printf("\n[INFO] Posição Inválida. Retornando o valor %d à
                  ↪  fila", linha_atual_temp);
622             fflush(stdout);
623         }
624
625         // Readiciona-la no final da fila
626         Enfilera(posicoes_restantes, linha_atual_temp);
627
628         Imprime_Fila(*posicoes_restantes);
629
630         // Passa pra próxima rainha
631         iteracoes++;
632
633     } // if
634
635 } // While
```

```
636
637
638     // Após ter percorrido todos as soluções deste nível e seus
        ↪  subníveis
639       // é retornado um nível para continuar a busca.
640
641     if(global_imprime_tela_ativado)
642        printf("\n[INFO] Saindo do nível %d", coluna_atual);
643
644     return ;
645
646  }
647
648
649  int main(int argc, char** argv) {
650     int * premios = 0;
651     char * diretorio = 0;
652     Tabuleiro * tabuleiro_atual = 0, * tabuleiro_maximo_encontrado = 0;
653     Fila * fila_rainhas = 0;
654     time_t seconds = 0;
655
656     if (argc != 4) {
657        diretorio = argv[1];
658        srand(atoi(argv[2]));
659        seconds = atoi(argv[3]);
660        global_imprime_tela_ativado = atoi(argv[4]);
661
662     } else {
663        exit(-1);
664     }
665
666     Le_Premios(diretorio, &premios);
667
668     tabuleiro_atual = Cria_Tabuleiro();
669     tabuleiro_maximo_encontrado = Cria_Tabuleiro();
670
671     fila_rainhas = Cria_Fila_Aleatoria();
672
673     start = time(NULL);
674
675     endwait = start + seconds;
676
677
678     n_Rainhas_Prize(0, tabuleiro_atual, tabuleiro_maximo_encontrado,
        ↪  &fila_rainhas, premios);
679
```

```
680
681     Imprime_Tabuleiro_Final(*tabuleiro_maximo_encontrado, premios);
682
683     Desaloca_Fila(&fila_rainhas);
684     Desaloca_Premios(&premios);
685     Desaloca_Tabuleiro(&tabuleiro_maximo_encontrado);
686     Desaloca_Tabuleiro(&tabuleiro_atual);
687
688     return (EXIT_SUCCESS);
689 }
```

7.3. *Branch and Bound* 1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  // variável global sobre a quantidade de colunas/linhas da matriz
6  int global_tamanho_matriz = -1;
7  // variável que ativa impressoes na tela
8  char global_imprime_tela_ativado = 0;
9
10 // Variáveis de tempo para cálculo do intervalo de tempo de execução
11 time_t endwait, start;
12
13
14 /*
15  * Estrutura que representa um tabuleiro.
16  *
17  * Um ponteiro para um vetor de onde representa cada posicao da rainha
    ↪  na coluna
18  * O valor de prêmrio do tabuleiro no estado atual
19  */
20 typedef struct Struct_Tabuleiro {
21     int * colunas;
22     int   premio;
23 } Tabuleiro;
24
25
26 /*
27  * Estrutura de representação de Fila
28  *
29  * Esta fila é utilizada para organizar a ordem de seleção das rainhas
30  */
31 typedef struct Struct_Fila {
32     int           lugar; // Posição da rainha
```

```
33      struct Struct_Fila * proximo;
34  } Fila;
35
36
37
38  /*
39   * Procedimento que verifica se a fila passada por parâmetro está
     ↪  vazia.
40   *
41   * Retorna valores boleanos
42   */
43  char Fila_Esta_Vazia(Fila * f) {
44      if (f == NULL) {
45          return 1;
46      } else
47          return 0;
48  }
49
50
51
52  /*
53   * Procedimento que adiciona um novo valor à fila
54   */
55  void Enfilera(Fila ** f, int l) {
56      Fila * atual = 0, * novo = 0;
57
58      // Cria um novo nó e adiciona informações
59      novo = calloc(1, sizeof(Fila));
60      novo->lugar = l;
61      novo->proximo = NULL;
62
63
64      // Se a fila estiver vazia, coloca na cabeça
65      if (Fila_Esta_Vazia(*f)) {
66
67          * f = novo;
68
69      } else {
70          // Caso contrário, coloca na calda
71
72          atual = * f;
73
74          while(atual->proximo != NULL) {
75              atual = atual->proximo;
76          }
77
```

```
78          atual->proximo = novo;
79      }
80  }
81
82
83
84  /*
85   * Procedimento que realiza a retirada de um elemento da fila seguindo
     ↪  sua natureza
86   */
87  int Desenfilera(Fila ** f) {
88      Fila * retirar = 0, * fila = * f;
89      int posicao = 0;
90
91      // Se a fila estiver vazia, informa erro e termina o programa
92      if (Fila_Esta_Vazia(*f)) {
93          printf("\n\n[ERRO] Impossível retirar de uma fila vazia.");
94          return -1;
95
96      } else {
97          // caso contrário, retira o primeiro elemento.
98          posicao = fila->lugar;
99          retirar = fila;
100
101         fila = fila->proximo;
102
103         free(retirar);
104
105         *f = fila;
106
107         return posicao;
108     }
109 }
110
111
112
113 /*
114  * Procedimento que realiza o processo de retirar todos os elementos da
     ↪  fila
115  */
116 void Desaloca_Fila(Fila * f) {
117     Fila * retirar = 0;
118
119     // Retira todos os elementos da fila
120     while (f != NULL) {
121         retirar = f;
```

```
122
123         f = f->proximo;
124
125         free (retirar);
126     }
127
128 }
129
130
131
132 /*
133  * Procedimento para impressão da Fila na tela
134  */
135 void Imprime_Fila(Fila * f) {
136     Fila * atual = f;
137
138     if (global_imprime_tela_ativado) {
139
140         printf("\n\n\t\t");
141         while (atual != NULL) {
142             printf("%d -> ", atual->lugar);
143             atual = atual->proximo;
144         }
145         printf("NULL");
146
147         fflush(stdout);
148     }
149 }
150
151
152
153 /*
154  * Procedimento responsável por gerar uma fila de posições de rainhas
    ↪  em ordem aleatória.
155  *
156  * Primeiro cria-se um vetor com os valores de 1..n ordenado
    ↪  simbolizando a ordem das rainhas.
157  * Em seguida, é gerado dois valores inteiros posicao1 e posicao2 de
    ↪  forma aleatória e estes
158  *       são utilizados para a comutação dos valores situados em
    ↪  vetor[posicao1] e vetor[posicao2]
159  * No final de algumas iterações, estes respectivos valores já
    ↪  desordenados são copiados para
160  *       a fila para que o algoritmo use.
161  */
162 Fila * Cria_Fila_Aleatoria() {
```

```
163     Fila * f = 0, * cabeca = 0;
164
165     int vetor_valores_fila [global_tamanho_matriz],
166           i = 0,
167           permutacoes = global_tamanho_matriz * 10,
168           posicao1 = 0, posicao2 = 0, temp = 0;
169
170     // Cria-se um vetor com valores ordenados de 1..n_rainhas
171     for (i = 0; i < global_tamanho_matriz; i++) {
172        vetor_valores_fila[i] = i;
173     }
174
175     // Realiza várias permutações dos valores deste vetor de acordo com
        ↪  os dois índices gerados
176        // Realiza-se global_tamanho_matriz * 10 permutações
177     i = 0;
178     while (i < permutacoes) {
179        // Gera primeiro índice
180        posicao1 = rand() % global_tamanho_matriz;
181
182        // Gera segundo índice
183        posicao2 = rand() % global_tamanho_matriz;
184
185        // Realiza a comutação dos valores dos respectivos índices
186        temp = vetor_valores_fila[posicao1];
187        vetor_valores_fila[posicao1] = vetor_valores_fila[posicao2];
188        vetor_valores_fila[posicao2] = temp;
189
190        i++;
191     }
192
193     // Aloca a fila e adiciona os valores após realizar a desordem
194     cabeca = calloc(1, sizeof(Fila));
195
196     f = cabeca;
197
198     // Copia os valores do vetor para a fila
199     i = 0;
200     while (i < global_tamanho_matriz - 1) {
201
202        f->lugar = vetor_valores_fila[i];
203
204        f->proximo = calloc(1, sizeof(Fila));
205
206        f = f->proximo;
207
```

```
208         i++;
209     }
210 
211     f->lugar = vetor_valores_fila[i];
212 
213     f->proximo = NULL;
214 
215     // Retorna a fila desordenada
216     return cabeca;
217 }
218 
219 
220 
221 /*
222  * Procedimento para retorndo do de prêmio específico de uma célula
223  */
224 int Retorna_Premio(int * premios, int linha, int coluna) {
225 
226     // Acessa o vetor como se fosse uma matriz comum nxn.
227     return premios[linha * global_tamanho_matriz + coluna];
228 }
229 
230 
231 
232 /*
233  * Procedimento para realizar a leitura dos arquivo prêmio
234  *
235  * Este também armazena o maior premio lido e o retorna.
236  */
237 void Le_Premios(char * diretorio, int ** premios){
238     FILE * file = 0;
239     int i = 0;
240     int premio_lido = 0;
241     int quantidade_celulas = 0;
242 
243     // Abre o arquivo de prêmios em forma de leitura
244     file = fopen(diretorio, "r");
245 
246     // Verifica se a abertura foi feita com sucesso
247     if (file != NULL) {
248 
249         //lê a primeira informação (quantas linhas existem)
250         fscanf(file, "%d", &global_tamanho_matriz);
251         if (global_imprime_tela_ativado)
252             printf("\n[INFO] Quantidade de colunas do tabuleiro = %d.\n",
                ↪  global_tamanho_matriz);
```

```
253
254         // Se o valor de tamanho da matriz for um valor válido
255         if (global_tamanho_matriz > 0) {
256
257             quantidade_celulas = global_tamanho_matriz *
                  ↪  global_tamanho_matriz;
258
259             // Aloca a matriz de prêmios
260             * premios = calloc(quantidade_celulas, sizeof(int));
261
262             // Começa a coleta dos prêmios
263             fscanf(file, "%d", &premio_lido);
264
265             // Salvas os prêmios e coleta o próximo
266             while(i < quantidade_celulas) {
267
268                 if (global_imprime_tela_ativado) {
269                     // Imprime na tela o prêmio lido
270                     printf("\n[INFO] Lendo premio [%d,%d] = %d.", i /
                          ↪  global_tamanho_matriz + 1, i %
                          ↪  global_tamanho_matriz + 1, premio_lido);
271                     fflush(stdout);
272                 }
273
274                 // Salva no vetor
275                 (*premios)[i++] = premio_lido;
276
277                 // Lê o próximo prêmio
278                 fscanf(file, "%d", &premio_lido);
279             }
280         } else {
281             // Informa Erro
282             printf("\n[ERRO] Quantidade de rainhas insuficiente!\n\n");
283             exit(-1);
284         }
285
286         // Fecho o arquivo
287         fclose(file);
288     }
289     else {
290         // Informa Erro
291         printf("\n[ERRO] Falha na leitura do arquivo de
              ↪  configuração!\n\n");
292         exit(-1);
293     }
294 }
```

```
295
296
297
298 /*
299  * Como utiliza-se uma estrutura fila com todas as opções e sem
     ↪  repetição, não
300  *      existe a possibilidade de duas rainhas ficarem numa mesma linha
301  *      vertical, horizontal (isso pois os valores não se repetem).
302  *
303  * Com isso, basta verificar se as diagonais estão conflitando.
304  */
305 char Posicao_Eh_Valida(Tabuleiro t, int col, int pos) {
306    int i = 0;
307    char esquerda_diagonal = 0, direita_diagonal = 0;
308
309    // Inicializa dizendo que as diagonais não estão ocupadas
310    esquerda_diagonal = direita_diagonal = 1;
311
312    // Da posição da rainha até a coluna 0, faça:
313    i = col;
314    while (i > 0) {
315
316       // Verifica:
317       //      Diagonal esquerda-direita
318       if (t.colunas[col - i]  == pos + i) {
319          esquerda_diagonal = 0;
320
321       // Verifica:
322       //      Diagonal direita-esquerda
323       } else
324          if (t.colunas[col - i] == pos - i) {
325          direita_diagonal = 0;
326       }
327
328       // Verifica validade
329       // Caso alguma diagonal já esteja ocupada, aborta o procedimento
330          // informando que esta posição é inválida
331       if (esquerda_diagonal == 0 || direita_diagonal == 0) {
332          return 0;
333       }
334
335       i--;
336    }
337
338    // Verifica validade das diagonais
339    if (esquerda_diagonal == 1 && direita_diagonal == 1)
```

```
340         return 1;
341      else
342         return 0;
343   }
344
345
346
347   /*
348    * Procedimento que copia os dados de um tabuleiro para outro.
349    *
350    * Este procedimento é utilizando quando encontra-se um novo valor de
       ↪ prêmio maior
351    *       que o atual e assim, realiza-se a substituição do tabuleiro
       ↪ antigo pelo novo
352    *       encontrado.
353    */
354   void Copia_Novo_Tabuleiro(Tabuleiro tabuleiro_atual, Tabuleiro *
       ↪ tabuleiro_maior) {
355      int i = 0;
356
357      tabuleiro_maior->premio           = tabuleiro_atual.premio;
358
359      for (i = 0; i < global_tamanho_matriz; i++) {
360         tabuleiro_maior->colunas[i]    = tabuleiro_atual.colunas[i];
361      }
362   }
363
364
365
366   /*
367    * Procedimento que adiciona uma nova rainha no tabuleiro já calculando
       ↪ o prêmio desta.
368    *
369    * Este procedimento não precisa verificar a validade da posição já que
       ↪ este já foi calculado
370    *       quando a rainha foi selecionada.
371    */
372   void Adiciona_Rainha_Tabuleiro_Calculando_Premio(Tabuleiro * t, int
       ↪ posicao,  int r, int * premios) {
373
374      t->premio += Retorna_Premio(premios, r, posicao);
375      t->colunas[posicao] = r;
376   }
377
378
379
```

```
380 /*
381  * Procedimento que retira a rainha do tabuleiro calculando o novo
     ↪  prêmio
382  */
383 void Retira_Rainha_Tabuleiro_Calculando_Premio(Tabuleiro * t, int
     ↪  posicao, int r, int * premios) {
384    t->premio -= Retorna_Premio(premios, r, posicao);
385    t->colunas[posicao] = -1;
386 }
387
388
389
390 /*
391  * Procedimento que inicializa um novo tabuleiro
392  */
393 Tabuleiro * Cria_Tabuleiro(){
394    int i = 0;
395    Tabuleiro * t = 0;
396
397    t = calloc(1, sizeof(Tabuleiro));
398    t->colunas = calloc(global_tamanho_matriz, sizeof(int));
399
400
401    for (i = 0; i < global_tamanho_matriz; i++)
402       t->colunas[i] = -1;
403
404    return t;
405 }
406
407
408
409 /*
410  * Procedimento que imprime o tabuleiro junto com os prêmios
411  */
412 void Imprime_Tabuleiro (Tabuleiro t, Tabuleiro maior, int * premios) {
413    int i = 0, j = 0;
414
415    if (global_imprime_tela_ativado) {
416
417       printf("\n");
418
419       for (i = 0; i < global_tamanho_matriz; i++) {
420             printf("\n\t");
421          for (j = 0; j < global_tamanho_matriz * 2; j++) {
422             if ( j < global_tamanho_matriz) {
423                if (i == t.colunas[j])
```

```
424                     printf("%3d ", i);
425                  else
426                     printf(" -1 ");
427               }
428               else {
429                  if (j == global_tamanho_matriz)
430                     printf("\t\t");
431                  printf("%3d ", premios[i * global_tamanho_matriz + j -
                     ↪  global_tamanho_matriz]);
432               }
433
434            }
435
436            fflush(stdout);
437         }
438
439         printf("\tPremio_Atual: %d;\tMaior_Premio: %d.", t.premio,
            ↪  maior.premio);
440
441         fflush(stdout);
442      }
443   }
444
445
446   /*
447    * Procedimento que imprime o tabuleiro, junto com os prêmios
       ↪  informando fim da execução
448    */
449   void Imprime_Tabuleiro_Final (Tabuleiro maior, int * premios) {
450      int i = 0, j = 0;
451      FILE * saida = 0;
452
453      if (global_imprime_tela_ativado) {
454
455         printf("\n\n[INFO] Resultado Final do Processamento:");
456
457         for (i = 0; i < global_tamanho_matriz; i++) {
458               printf("\n\t");
459            for (j = 0; j < global_tamanho_matriz * 2; j++) {
460               if ( j < global_tamanho_matriz) {
461                  if (i == maior.colunas[j])
462                     printf("%3d ", i);
463                  else
464                     printf(" -1 ");
465               }
466               else {
```

```
467                 if (j == global_tamanho_matriz)
468                    printf("\t\t");
469                 printf("%3d ", premios[i * global_tamanho_matriz + j -
                      ↪  global_tamanho_matriz]);
470              }
471           }
472
473           fflush(stdout);
474        }
475
476        printf("\tPremio: %d.",maior.premio);
477        printf("\n[INFO] Resultado Final do Processamento.\n");
478     } else {
479        saida = fopen("algoritmo2.txt", "a");
480
481        if (saida) {
482           fprintf(saida, "%d\n", maior.premio);
483        }
484
485        fclose(saida);
486     }
487
488     fflush(stdout);
489  }
490
491
492
493  /*
494   * Procedimento que libera memória
495   */
496  void Desaloca_Tabuleiro (Tabuleiro ** f) {
497     if (*f != 0) {
498        free((*f)->colunas);
499        free(*f);
500     }
501  }
502
503
504
505  /*
506   * Procedimento que libera memória
507   */
508  void Desaloca_Premios (int ** premios) {
509     if (*premios != 0)
510        free(*premios);
511  }
```

```
512
513
514
515 /*
516  * Método de branch and bound desenvolvido
517  */
518 void n_Rainhas_Prize(int coluna_atual, Tabuleiro * tabuleiro_atual,
     ↪  Tabuleiro * tabuleiro_maior, Fila ** posicoes_restantes, int *
     ↪  premios) {
519    int iteracoes = 0, linha_atual_temp = 0;
520    float fator_continua_recursao = 0;
521
522    // Recebe o tempo atual.
523    start = time(NULL);
524
525    // Verifica se o tempo excedeu o liminte estabelecido.
526    if (start > endwait) {
527       // Se sim, cancela totalmente a continuação da recursão
528       return ;
529    }
530
531
532    // Varre a fila com os valores que sobraram
533       // A cada recursão, um item é retirado
534
535    // Caso tenha percorrido todas as rainhas deste contexto de
       ↪  recursão,
536       // o while será impedido de ser executando forçando a realizar o
537       // retorno à um nível acima de recursao
538    while (iteracoes < global_tamanho_matriz - coluna_atual) {
539
540       Imprime_Fila(* posicoes_restantes);
541       Imprime_Tabuleiro(*tabuleiro_atual, *tabuleiro_maior, premios);
542
543       // Retira um item da fila e salva numa variável local
544       linha_atual_temp = Desenfilera( posicoes_restantes );
545
546       if (linha_atual_temp == -1) {
547          Desaloca_Premios(&premios);
548
549          Desaloca_Tabuleiro(&tabuleiro_maior);
550          Desaloca_Tabuleiro(&tabuleiro_atual);
551          exit(-1);
552       }
553
554       Imprime_Fila(* posicoes_restantes);
```

```
555
556         if (global_imprime_tela_ativado) {
557            printf("\t\tBuffer_Atual: %d", linha_atual_temp);
558            fflush(stdout);
559         }
560
561
562         // Testa a validade da rainha retirada no momento
563            // Se for posição válida realizará o processamento desta
564            // Caso contrário, ela será posta no final da fila=
565         if(Posicao_Eh_Valida(*tabuleiro_atual, coluna_atual,
             ↪  linha_atual_temp)) {
566
567            // Adiciona a rainha no tabuleiro calculando o prêmio com sua
                ↪  inclusão
568            Adiciona_Rainha_Tabuleiro_Calculando_Premio(tabuleiro_atual,
                ↪  coluna_atual, linha_atual_temp, premios);
569
570            Imprime_Tabuleiro(*tabuleiro_atual, *tabuleiro_maior,
                ↪  premios);
571
572
573            // Se a fila estiver vazia, significa que acabou de ser gerado
                ↪  uma solução
574               // válida.
575            // Assim, será verificado se o prêmio é melhor que o atual.
576            if (Fila_Esta_Vazia(* posicoes_restantes)) {
577
578               if (tabuleiro_atual->premio > tabuleiro_maior->premio) {
579
580                  if (global_imprime_tela_ativado)
581                     printf("\n\n[INFO] Novo Recorde Encontrado!");
582                  Copia_Novo_Tabuleiro(*tabuleiro_atual, tabuleiro_maior);
583
584                  Imprime_Tabuleiro(*tabuleiro_atual, *tabuleiro_maior,
                      ↪  premios);
585               }
586
587
588               Imprime_Fila(*posicoes_restantes);
589
590               // Após chegar na folha, a recursão é revertida até que
                   ↪  encontre
591                  // uma próxima solução pra explorar.
592               Enfilera(posicoes_restantes, linha_atual_temp);
593
```

```
594             Imprime_Fila(*posicoes_restantes);
595
596             // Como mencionado, a raínha é posta novamente na fila para
                  ↪  a procura
597                 // de novas soluções
598             Retira_Rainha_Tabuleiro_Calculando_Premio(tabuleiro_atual,
                  ↪  coluna_atual, linha_atual_temp, premios);
599
600
601             return ;
602
603
604         // Se não for a última rainha, então realiza as análises de
              ↪  bound
605         } else {
606
607             if (global_imprime_tela_ativado)
608                 printf("\n%10f\n", ((float) coluna_atual) /
                      ↪  global_tamanho_matriz);
609
610
611             //Considerações
612                 // - Ler o relatório que acompanha o código.
613                 // - Enquanto o maior premio encontrado ainda for 0,
                      ↪  então os filtros não
614                      // serão aplicados
615
616
617             // Filtro 1
618             if (tabuleiro_maior->premio != 0  &&  ((float)
                  ↪  coluna_atual) / global_tamanho_matriz >= 0.60  &&
                  ↪  ((float) coluna_atual) / global_tamanho_matriz < 0.7)
                  ↪  {
619
620                 fator_continua_recursao =  (float)
                      ↪  tabuleiro_atual->premio / tabuleiro_maior->premio;
621
622                 if (global_imprime_tela_ativado) {
623                     printf("\n[INFO] Premio Atual: %d;\tMaior Premio:
                          ↪  %d\tFator Continua Recursão: %.7f",
                          ↪  tabuleiro_atual->premio,
                          ↪  tabuleiro_maior->premio,
                          ↪  fator_continua_recursao);
624                     fflush(stdout);
625                 }
626
```

```
627                 //if (fator_continua_recursao > 0.50)
628                 if (fator_continua_recursao > 0.6)
629                     n_Rainhas_Prize(coluna_atual + 1, tabuleiro_atual,
                        ↪  tabuleiro_maior, posicoes_restantes, premios);
630
631
632             // Filtro 2
633             } else if (tabuleiro_maior->premio != 0  &&  ((float)
                ↪  coluna_atual) / global_tamanho_matriz >= 0.7 &&
                ↪  ((float) coluna_atual) / global_tamanho_matriz < 0.8)
                ↪  {
634
635                 fator_continua_recursao =  (float)
                    ↪  tabuleiro_atual->premio / tabuleiro_maior->premio;
636
637                 if (global_imprime_tela_ativado) {
638                     printf("\n[INFO] Premio Atual: %d;\tMaior Premio:
                        ↪  %d\tFator Continua Recursão: %.7f",
                        ↪  tabuleiro_atual->premio,
                        ↪  tabuleiro_maior->premio,
                        ↪  fator_continua_recursao);
639                     fflush(stdout);
640                 }
641
642                 //if (fator_continua_recursao > 0.60)
643                 if (fator_continua_recursao > 0.7)
644                     n_Rainhas_Prize(coluna_atual + 1, tabuleiro_atual,
                        ↪  tabuleiro_maior, posicoes_restantes, premios);
645
646
647             // Filtro 3
648             } else if (tabuleiro_maior->premio != 0  &&  ((float)
                ↪  coluna_atual) / global_tamanho_matriz >= 0.8) {
649
650                 fator_continua_recursao =  (float)
                    ↪  tabuleiro_atual->premio / tabuleiro_maior->premio;
651
652                 if (global_imprime_tela_ativado) {
653                     printf("\n[INFO] Premio Atual: %d;\tMaior Premio:
                        ↪  %d\tFator Continua Recursao: %.7f",
                        ↪  tabuleiro_atual->premio,
                        ↪  tabuleiro_maior->premio,
                        ↪  fator_continua_recursao);
654                     fflush(stdout);
655                 }
656
```

```
657                 //if (fator_continua_recursao > 0.82)
658                 if (fator_continua_recursao > 0.8)
659                     n_Rainhas_Prize(coluna_atual + 1, tabuleiro_atual,
                        ↪  tabuleiro_maior, posicoes_restantes, premios);
660
661
662             // Enquanto o maior premio encontrado ainda for 0, então os
                ↪  filtros não
663                 // serão aplicados e a recursão segue normalmente usando
                    ↪  força
664                 // bruta
665             } else {
666                 n_Rainhas_Prize(coluna_atual + 1, tabuleiro_atual,
                    ↪  tabuleiro_maior, posicoes_restantes, premios);
667             }
668
669             // Ao retornar das recursões, a rainha atual será retirada
                ↪  do tabuleiro e colocada na fila novamente
670                 // e será procurado a próxima solução disponível
671
672             Enfilera(posicoes_restantes, linha_atual_temp);
673
674             Retira_Rainha_Tabuleiro_Calculando_Premio(tabuleiro_atual,
                ↪  coluna_atual, linha_atual_temp, premios);
675
676             Imprime_Fila(*posicoes_restantes);
677
678             iteracoes++;
679         } // if
680
681
682     // Se a posição escolhida não for válida
683     } else {
684
685         if(global_imprime_tela_ativado) {
686             printf("\n[INFO] Posição Inválida. Retornando o valor %d à
                    ↪  fila", linha_atual_temp);
687             fflush(stdout);
688         }
689
690         // Readiciona-la no final da fila
691         Enfilera(posicoes_restantes, linha_atual_temp);
692
693         Imprime_Fila(*posicoes_restantes);
694
695         // Passa pra próxima rainha
```

```
696             iteracoes++;
697 
698         } // if
699 
700     } // While
701 
702 
703     // Após ter percorrido todos as soluções deste nível e seus
      ↪  subníveis
704         // é retornado um nível para continuar a busca.
705 
706     if(global_imprime_tela_ativado)
707         printf("\n[INFO] Saindo do nível %d", coluna_atual);
708 
709     return ;
710 }
711 
712 
713 
714 int main(int argc, char** argv) {
715     int * premios;
716     char * diretorio;
717     Tabuleiro * tabuleiro_atual, * tabuleiro_maximo_encontrado;
718     Fila * fila_rainhas;
719      time_t seconds;
720 
721     if (argc != 4) {
722         diretorio = argv[1];
723         srand(atoi(argv[2]));
724         seconds = atoi(argv[3]);
725         global_imprime_tela_ativado = atoi(argv[4]);
726 
727     } else {
728         exit(-1);
729     }
730 
731     Le_Premios(diretorio, &premios);
732 
733     tabuleiro_atual = Cria_Tabuleiro();
734     tabuleiro_maximo_encontrado = Cria_Tabuleiro();
735 
736     fila_rainhas = Cria_Fila_Aleatoria();
737 
738     start = time(NULL);
739 
740      endwait = start + seconds;
```

```
741
742     n_Rainhas_Prize(0, tabuleiro_atual, tabuleiro_maximo_encontrado,
          ↪  &fila_rainhas, premios);
743
744
745     Imprime_Tabuleiro_Final(*tabuleiro_maximo_encontrado, premios);
746
747
748     Desaloca_Fila(fila_rainhas);
749
750     Desaloca_Premios(&premios);
751
752     Desaloca_Tabuleiro(&tabuleiro_maximo_encontrado);
753     Desaloca_Tabuleiro(&tabuleiro_atual);
754
755     return (EXIT_SUCCESS);
756 }
```

7.4. *Branch and Bound* 2

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  // variável global sobre a quantidade de colunas/linhas da matriz
6  int global_tamanho_matriz = -1;
7  // variável que ativa impressoes na tela
8  char global_imprime_tela_ativado = 0, global_imprime_filtro = 0;
9
10 // Variáveis de tempo para cálculo do intervalo de tempo de execução
11 time_t endwait, start;
12
13
14 /*
15  * Estrutura que representa um tabuleiro.
16  *
17  * Um ponteiro para um vetor de onde representa cada posicao da rainha
     ↪  na coluna
18  * O valor de prêmrio do tabuleiro no estado atual
19  */
20 typedef struct Struct_Tabuleiro {
21     int * colunas;
22     int   premio;
23 } Tabuleiro;
24
25
```

```c
/*
 * Estrutura de representação de Fila
 *
 * Esta fila é utilizada para organizar a ordem de seleção das rainhas
 */
typedef struct Struct_Fila {
   int            lugar; // Posição da rainha
   struct Struct_Fila * proximo;
} Fila;



/*
 * Procedimento que verifica se a fila passada por parâmetro está
 ↪  vazia.
 *
 * Retorna valores boleanos
 */
char Fila_Esta_Vazia(Fila * f) {
   if (f == NULL) {
      return 1;
   } else
      return 0;
}



/*
 * Procedimento que adiciona um novo valor à fila
 */
void Enfilera(Fila ** f, int l) {
   Fila * atual = 0, * novo = 0;

   // Cria um novo nó e adiciona informações
   novo = calloc(1, sizeof(Fila));
   novo->lugar = l;
   novo->proximo = NULL;


   // Se a fila estiver vazia, coloca na cabeça
   if (Fila_Esta_Vazia(*f)) {

      * f = novo;

   } else {
      // Caso contrário, coloca na calda
```

```
71
72          atual = * f;
73
74          while(atual->proximo != NULL) {
75             atual = atual->proximo;
76          }
77
78          atual->proximo = novo;
79       }
80    }
81
82
83
84    /*
85     * Procedimento que realiza a retirada de um elemento da fila seguindo
       ↪  sua natureza
86     */
87    int Desenfilera(Fila ** f) {
88       Fila * retirar = 0, * fila = * f;
89       int posicao = 0;
90
91       // Se a fila estiver vazia, informa erro e termina o programa
92       if (Fila_Esta_Vazia(*f)) {
93          printf("\n\n[ERRO] Impossível retirar de uma fila vazia.");
94          return -1;
95
96       } else {
97          // caso contrário, retira o primeiro elemento.
98          posicao = fila->lugar;
99          retirar = fila;
100
101         fila = fila->proximo;
102
103         free(retirar);
104
105         *f = fila;
106
107         return posicao;
108      }
109   }
110
111
112
113   /*
114    * Procedimento que realiza o processo de retirar todos os elementos da
       ↪  fila
```

```
115  */
116 void Desaloca_Fila(Fila * f) {
117    Fila * retirar = 0;
118 
119    // Retira todos os elementos da fila
120    while (f != NULL) {
121       retirar = f;
122 
123       f = f->proximo;
124 
125       free (retirar);
126    }
127 
128 }
129 
130 
131 
132 /*
133  * Procedimento para impressão da Fila na tela
134  */
135 void Imprime_Fila(Fila * f) {
136    Fila * atual = f;
137 
138    if (global_imprime_tela_ativado) {
139 
140       printf("\n\n\t\t");
141       while (atual != NULL) {
142          printf("%d -> ", atual->lugar);
143          atual = atual->proximo;
144       }
145       printf("NULL");
146 
147       fflush(stdout);
148    }
149 }
150 
151 
152 
153 /*
154  * Procedimento responsável por gerar uma fila de posições de rainhas
     ↪  em ordem aleatória.
155  *
156  * Primeiro cria-se um vetor com os valores de 1..n ordenado
     ↪  simbolizando a ordem das rainhas.
157  * Em seguida, é gerado dois valores inteiros posicao1 e posicao2 de
     ↪  forma aleatória e estes
```

```
158   *      são utilizados para a comutação dos valores situados em
      ↪  vetor[posicao1] e vetor[posicao2]
159   * No final de algumas iterações, estes respectivos valores já
      ↪  desordenados são copiados para
160   *      a fila para que o algoritmo use.
161   */
162  Fila * Cria_Fila_Aleatoria() {
163     Fila * f = 0, * cabeca = 0;
164
165     int vetor_valores_fila [global_tamanho_matriz],
166           i = 0,
167           permutacoes = global_tamanho_matriz * 10,
168           posicao1 = 0, posicao2 = 0, temp = 0;
169
170     // Cria-se um vetor com valores ordenados de 1..n_rainhas
171     for (i = 0; i < global_tamanho_matriz; i++) {
172        vetor_valores_fila[i] = i;
173     }
174
175     // Realiza várias permutações dos valores deste vetor de acordo com
        ↪  os dois índices gerados
176        // Realiza-se global_tamanho_matriz * 10 permutações
177     i = 0;
178     while (i < permutacoes) {
179        // Gera primeiro índice
180        posicao1 = rand() % global_tamanho_matriz;
181
182        // Gera segundo índice
183        posicao2 = rand() % global_tamanho_matriz;
184
185        // Realiza a comutação dos valores dos respectivos índices
186        temp = vetor_valores_fila[posicao1];
187        vetor_valores_fila[posicao1] = vetor_valores_fila[posicao2];
188        vetor_valores_fila[posicao2] = temp;
189
190        i++;
191     }
192
193     // Aloca a fila e adiciona os valores após realizar a desordem
194     cabeca = calloc(1, sizeof(Fila));
195
196     f = cabeca;
197
198     // Copia os valores do vetor para a fila
199     i = 0;
200     while (i < global_tamanho_matriz - 1) {
```

```
201
202         f->lugar = vetor_valores_fila[i];
203
204         f->proximo = calloc(1, sizeof(Fila));
205
206         f = f->proximo;
207
208         i++;
209     }
210
211     f->lugar = vetor_valores_fila[i];
212
213     f->proximo = NULL;
214
215     // Retorna a fila desordenada
216     return cabeca;
217 }
218
219
220
221 /*
222  * Procedimento para retorndo do de prêmio específico de uma célula
223  */
224 int Retorna_Premio(int * premios, int linha, int coluna) {
225
226     // Acessa o vetor como se fosse uma matriz comum nxn.
227     return premios[linha * global_tamanho_matriz + coluna];
228 }
229
230
231
232 /*
233  * Procedimento para realizar a leitura dos arquivo prêmio
234  *
235  * Este também armazena o maior premio lido e o retorna.
236  */
237 void Le_Premios(char * diretorio, int ** premios, float **
      ↪  media_coluna){
238     FILE * file = 0;
239     int i = 0;
240     int premio_lido = 0;
241     int quantidade_celulas = 0;
242
243     // Abre o arquivo de prêmios em forma de leitura
244     file = fopen(diretorio, "r");
245
```

```
246     // Verifica se a abertura foi feita com sucesso
247     if (file != NULL) {
248
249        //lê a primeira informação (quantas linhas existem)
250        fscanf(file, "%d", &global_tamanho_matriz);
251        if (global_imprime_tela_ativado)
252           printf("\n[INFO] Quantidade de colunas do tabuleiro = %d.\n",
             ↪  global_tamanho_matriz);
253
254        // Se o valor de tamanho da matriz for um valor válido
255        if (global_tamanho_matriz > 0) {
256
257           quantidade_celulas = global_tamanho_matriz *
             ↪  global_tamanho_matriz;
258
259           // Aloca a matriz de prêmios
260           * premios      = calloc(quantidade_celulas, sizeof(int));
261           * media_coluna = calloc(global_tamanho_matriz, sizeof(int));
262
263           // Começa a coleta dos prêmios
264           fscanf(file, "%d", &premio_lido);
265
266           // Salvas os prêmios e coleta o próximo
267           while(i < quantidade_celulas) {
268
269              if (global_imprime_tela_ativado) {
270                 // Imprime na tela o prêmio lido
271                 printf("\n[INFO] Lendo premio [%d,%d] = %d.", i /
                   ↪  global_tamanho_matriz + 1, i %
                   ↪  global_tamanho_matriz + 1, premio_lido);
272                 fflush(stdout);
273              }
274
275              (*media_coluna)[i % global_tamanho_matriz] += premio_lido;
276
277              // Salva no vetor
278              (*premios)[i++] = premio_lido;
279
280              // Lê o próximo prêmio
281              fscanf(file, "%d", &premio_lido);
282           }
283        } else {
284           // Informa Erro
285           printf("\n[ERRO] Quantidade de rainhas insuficiente!\n\n");
286           exit(-1);
287        }
```

```
288
289         // Fecho o arquivo
290         fclose(file);
291      }
292      else {
293         // Informa Erro
294         printf("\n[ERRO] Falha na leitura do arquivo de
             ↪  configuração!\n\n");
295         exit(-1);
296      }
297
298      // calcula a média de cada coluna.
299         // Lembrando que os valores já foram somados
300      for (i = 0; i < global_tamanho_matriz; i++) {
301         (*media_coluna)[i] /= global_tamanho_matriz;
302      }
303   }
304
305
306
307   /*
308    * Como utiliza-se uma estrutura fila com todas as opções e sem
       ↪  repetição, não
309    *       existe a possibilidade de duas rainhas ficarem numa mesma linha
310    *       vertical, horizontal (isso pois os valores não se repetem).
311    *
312    * Com isso, basta verificar se as diagonais estão conflitando.
313    */
314   char Posicao_Eh_Valida(Tabuleiro t, int col, int pos) {
315      int i = 0;
316      char esquerda_diagonal = 0, direita_diagonal = 0;
317
318      // Inicializa dizendo que as diagonais não estão ocupadas
319      esquerda_diagonal = direita_diagonal = 1;
320
321      // Da posição da rainha até a coluna 0, faça:
322      i = col;
323      while (i > 0) {
324
325         // Verifica:
326         //       Diagonal esquerda-direita
327         if (t.colunas[col - i]  == pos + i) {
328            esquerda_diagonal = 0;
329
330         // Verifica:
331         //       Diagonal direita-esquerda
```

```
332         } else
333            if (t.colunas[col - i] == pos - i) {
334            direita_diagonal = 0;
335         }
336
337         // Verifica validade
338         // Caso alguma diagonal já esteja ocupada, aborta o procedimento
339            // informando que esta posição é inválida
340         if (esquerda_diagonal == 0 || direita_diagonal == 0) {
341            return 0;
342         }
343
344         i--;
345      }
346
347      // Verifica validade das diagonais
348      if (esquerda_diagonal == 1 && direita_diagonal == 1)
349         return 1;
350      else
351         return 0;
352   }
353
354
355
356   /*
357    * Procedimento que copia os dados de um tabuleiro para outro.
358    *
359    * Este procedimento é utilizando quando encontra-se um novo valor de
       ↪ prêmio maior
360    *       que o atual e assim, realiza-se a substituição do tabuleiro
       ↪ antigo pelo novo
361    *       encontrado.
362    */
363   void Copia_Novo_Tabuleiro(Tabuleiro tabuleiro_atual, Tabuleiro *
       ↪ tabuleiro_maior) {
364      int i = 0;
365
366      tabuleiro_maior->premio           = tabuleiro_atual.premio;
367
368      for (i = 0; i < global_tamanho_matriz; i++) {
369         tabuleiro_maior->colunas[i]    = tabuleiro_atual.colunas[i];
370      }
371   }
372
373
374
```

```
375 /*
376  * Procedimento que adiciona uma nova rainha no tabuleiro já calculando
     ↪  o prêmio desta.
377  *
378  * Este procedimento não precisa verificar a validade da posição já que
     ↪  este já foi calculado
379  *      quando a rainha foi selecionada.
380  */
381 void Adiciona_Rainha_Tabuleiro_Calculando_Premio(Tabuleiro * t, int
     ↪  posicao,  int r, int * premios) {
382
383    t->premio += Retorna_Premio(premios, r, posicao);
384    t->colunas[posicao] = r;
385 }
386
387
388
389 /*
390  * Procedimento que retira a rainha do tabuleiro calculando o novo
     ↪  prêmio
391  */
392 void Retira_Rainha_Tabuleiro_Calculando_Premio(Tabuleiro * t, int
     ↪  posicao, int r, int * premios) {
393    t->premio -= Retorna_Premio(premios, r, posicao);
394    t->colunas[posicao] = -1;
395 }
396
397
398
399 /*
400  * Procedimento que inicializa um novo tabuleiro
401  */
402 Tabuleiro * Cria_Tabuleiro(){
403    int i = 0;
404    Tabuleiro * t = 0;
405
406    t = calloc(1, sizeof(Tabuleiro));
407    t->colunas = calloc(global_tamanho_matriz, sizeof(int));
408
409
410    for (i = 0; i < global_tamanho_matriz; i++)
411       t->colunas[i] = -1;
412
413    return t;
414 }
415
```

```
416
417
418 /*
419  * Procedimento que imprime o tabuleiro junto com os prêmios
420  */
421 void Imprime_Tabuleiro (Tabuleiro t, Tabuleiro maior, int * premios) {
422    int i = 0, j = 0;
423
424    if (global_imprime_tela_ativado) {
425
426       printf("\n");
427
428       for (i = 0; i < global_tamanho_matriz; i++) {
429             printf("\n\t");
430          for (j = 0; j < global_tamanho_matriz * 2; j++) {
431             if ( j < global_tamanho_matriz) {
432                if (i == t.colunas[j])
433                   printf("%3d ", i);
434                else
435                   printf(" -1 ");
436             }
437             else {
438                if (j == global_tamanho_matriz)
439                   printf("\t\t");
440                printf("%3d ", premios[i * global_tamanho_matriz + j -
                     ↪  global_tamanho_matriz]);
441             }
442
443          }
444
445          fflush(stdout);
446       }
447
448       printf("\tPremio_Atual: %d;\tMaior_Premio: %d.", t.premio,
             ↪  maior.premio);
449
450       fflush(stdout);
451    }
452 }
453
454
455 /*
456  * Procedimento que imprime o tabuleiro, junto com os prêmios
      ↪  informando fim da execução
457  */
458 void Imprime_Tabuleiro_Final (Tabuleiro maior, int * premios) {
```

```
459     int i = 0, j = 0;
460     FILE * saida = 0;
461 
462     if (global_imprime_tela_ativado) {
463 
464        printf("\n\n[INFO] Resultado Final do Processamento:");
465 
466        for (i = 0; i < global_tamanho_matriz; i++) {
467              printf("\n\t");
468           for (j = 0; j < global_tamanho_matriz * 2; j++) {
469              if ( j < global_tamanho_matriz) {
470                 if (i == maior.colunas[j])
471                    printf("%3d ", i);
472                 else
473                    printf(" -1 ");
474              }
475              else {
476                 if (j == global_tamanho_matriz)
477                    printf("\t\t");
478                 printf("%3d ", premios[i * global_tamanho_matriz + j -
                    ↪  global_tamanho_matriz]);
479              }
480           }
481 
482           fflush(stdout);
483        }
484 
485        printf("\tPremio: %d.",maior.premio);
486        printf("\n[INFO] Resultado Final do Processamento.\n");
487     } else {
488        saida = fopen("algoritmo3.txt", "a");
489 
490        if (saida) {
491           fprintf(saida, "%d\n", maior.premio);
492        }
493 
494        fclose(saida);
495     }
496 
497     fflush(stdout);
498  }
499 
500 
501 
502  /*
503   * Procedimento que libera memória
```

```
504   */
505   void Desaloca_Tabuleiro (Tabuleiro ** f) {
506     if (*f != 0) {
507        free((*f)->colunas);
508        free(*f);
509     }
510   }
511
512
513
514  /*
515   * Procedimento que libera memória
516   */
517  void Desaloca_Premios_E_Media_Coluna (int ** premios, float **
      ↪  media_coluna) {
518     if (*premios != 0)
519        free(*premios);
520
521     if (*media_coluna != 0)
522     free(*media_coluna);
523  }
524
525
526
527  /*
528   *
529   */
530  float Calcula_Fator_Continua_Recursao(int tabuleiro_atual_premio, int
      ↪  tabuleiro_maior_premio, float * media_coluna, int coluna_atual,
      ↪  int filtro) {
531     float fator = 0, soma = 0;
532
533     soma = tabuleiro_atual_premio + media_coluna[coluna_atual];
534
535     fator = ((float) soma) / tabuleiro_maior_premio;
536
537     if (global_imprime_tela_ativado || global_imprime_filtro) {
538        printf("\n[INFO] Premio Atual: %d;\tPeso c/ Media: %f;\tMaior
            ↪  Premio: "
539              "%d\tFator Continua Recursão: %.7f Filtro: %d",
                 ↪  tabuleiro_atual_premio,
540              tabuleiro_atual_premio + media_coluna[coluna_atual],
                 ↪  tabuleiro_maior_premio, fator, filtro);
541        fflush(stdout);
542     }
543
```

```
544     return fator;
545 }
546 
547 
548 /*
549  * Método de branch and bound desenvolvido
550  */
551 void n_Rainhas_Prize(int coluna_atual, Tabuleiro * tabuleiro_atual,
552     Tabuleiro * tabuleiro_maior, Fila ** posicoes_restantes, int *
        ↪  premios, float * media_coluna) {
553     int iteracoes = 0, linha_atual_temp = 0;
554     float fator_continua_recursao = 0;
555 
556     // Recebe o tempo atual.
557     start = time(NULL);
558 
559     // Verifica se o tempo excedeu o liminte estabelecido.
560     if (start > endwait) {
561         // Se sim, cancela totalmente a continuação da recursão
562         return ;
563     }
564 
565     // Varre a fila com os valores que sobraram
566         // A cada recursão, um item é retirado
567 
568     // Caso tenha percorrido todas as rainhas deste contexto de
        ↪  recursão,
569         // o while será impedido de ser executando forçando a realizar o
570         // retorno à um nível acima de recursao
571     while (iteracoes < global_tamanho_matriz - coluna_atual) {
572 
573         Imprime_Fila(* posicoes_restantes);
574         Imprime_Tabuleiro(*tabuleiro_atual, *tabuleiro_maior, premios);
575 
576         // Retira um item da fila e salva numa variável local
577         linha_atual_temp = Desenfilera( posicoes_restantes );
578 
579         if (linha_atual_temp == -1) {
580             Desaloca_Premios_E_Media_Coluna(&premios, &media_coluna);
581 
582             Desaloca_Tabuleiro(&tabuleiro_maior);
583             Desaloca_Tabuleiro(&tabuleiro_atual);
584             exit(-1);
585         }
586 
587         Imprime_Fila(* posicoes_restantes);
```

```
588
589         if (global_imprime_tela_ativado) {
590            printf("\t\tBuffer_Atual: %d", linha_atual_temp);
591            fflush(stdout);
592         }
593
594
595         // Testa a validade da rainha retirada no momento
596            // Se for posição válida realizará o processamento desta
597            // Caso contrário, ela será posta no final da fila=
598         if(Posicao_Eh_Valida(*tabuleiro_atual, coluna_atual,
             ↪  linha_atual_temp)) {
599
600            // Adiciona a rainha no tabuleiro calculando o prêmio com sua
                ↪  inclusão
601            Adiciona_Rainha_Tabuleiro_Calculando_Premio(tabuleiro_atual,
                ↪  coluna_atual, linha_atual_temp, premios);
602
603            Imprime_Tabuleiro(*tabuleiro_atual, *tabuleiro_maior,
                ↪  premios);
604
605
606            // Se a fila estiver vazia, significa que acabou de ser gerado
                ↪  uma solução
607               // válida.
608            // Assim, será verificado se o prêmio é melhor que o atual.
609            if (Fila_Esta_Vazia(* posicoes_restantes)) {
610
611               if (tabuleiro_atual->premio > tabuleiro_maior->premio) {
612
613                  if (global_imprime_tela_ativado)
614                     printf("\n\n[INFO] Novo Recorde Encontrado!");
615                  Copia_Novo_Tabuleiro(*tabuleiro_atual, tabuleiro_maior);
616
617                  Imprime_Tabuleiro(*tabuleiro_atual, *tabuleiro_maior,
                      ↪  premios);
618               }
619
620
621               Imprime_Fila(*posicoes_restantes);
622
623               // Após chegar na folha, a recursão é revertida até que
                   ↪  encontre
624                  // uma próxima solução pra explorar.
625               Enfilera(posicoes_restantes, linha_atual_temp);
626
```

```
627             Imprime_Fila(*posicoes_restantes);
628
629             // Como mencionado, a raínha é posta novamente na fila para
                  ↪  a procura
630               // de novas soluções
631             Retira_Rainha_Tabuleiro_Calculando_Premio(tabuleiro_atual,
                  ↪  coluna_atual, linha_atual_temp, premios);
632
633
634             return ;
635
636
637         // Se não for a última rainha, então realiza as análises de
              ↪  bound
638         } else {
639
640             if (global_imprime_tela_ativado)
641                 printf("\n%10f\n", ((float) coluna_atual) /
                      ↪  global_tamanho_matriz);
642
643
644             //Considerações
645                 // - Ler o relatório que acompanha o código.
646                 // - Enquanto o maior premio encontrado ainda for 0,
                      ↪  então os filtros não
647                       // serão aplicados
648
649
650             // Filtro 1
651             if (tabuleiro_maior->premio != 0  &&  ((float)
                  ↪  coluna_atual) / global_tamanho_matriz >= 0.60 &&
                  ↪  ((float) coluna_atual) / global_tamanho_matriz < 0.7)
                  ↪  {
652
653               fator_continua_recursao =
                    ↪  Calcula_Fator_Continua_Recursao(tabuleiro_atual->premio,
                    ↪  tabuleiro_maior->premio, media_coluna,
                    ↪  coluna_atual, 1);
654
655               if (fator_continua_recursao > 0.6)
656                   n_Rainhas_Prize(coluna_atual + 1, tabuleiro_atual,
                        ↪  tabuleiro_maior, posicoes_restantes, premios,
                        ↪  media_coluna);
657
658
659             // Filtro 2
```

```
660             } else if (tabuleiro_maior->premio != 0  &&  ((float)
                 ↪  coluna_atual) / global_tamanho_matriz >= 0.7  &&
                 ↪  ((float) coluna_atual) / global_tamanho_matriz < 0.8)
                 ↪  {
661
662                 fator_continua_recursao =
                     ↪  Calcula_Fator_Continua_Recursao(tabuleiro_atual->premio,
                     ↪  tabuleiro_maior->premio, media_coluna,
                     ↪  coluna_atual, 2);
663
664                 if (fator_continua_recursao > 0.7)
665                     n_Rainhas_Prize(coluna_atual + 1, tabuleiro_atual,
                         ↪  tabuleiro_maior, posicoes_restantes, premios,
                         ↪  media_coluna);
666
667
668             // Filtro 3
669             } else if (tabuleiro_maior->premio != 0  &&  ((float)
                 ↪  coluna_atual) / global_tamanho_matriz >= 0.8) {
670
671                 fator_continua_recursao =
                     ↪  Calcula_Fator_Continua_Recursao(tabuleiro_atual->premio,
                     ↪  tabuleiro_maior->premio, media_coluna,
                     ↪  coluna_atual, 3);
672
673                 if (fator_continua_recursao > 0.8)
674                     n_Rainhas_Prize(coluna_atual + 1, tabuleiro_atual,
                         ↪  tabuleiro_maior, posicoes_restantes, premios,
                         ↪  media_coluna);
675
676
677             // Enquanto o maior premio encontrado ainda for 0, então os
                 ↪  filtros não
678                 // serão aplicados e a recursão segue normalmente usando
                     ↪  força
679                 // bruta
680             } else {
681                 n_Rainhas_Prize(coluna_atual + 1, tabuleiro_atual,
                     ↪  tabuleiro_maior, posicoes_restantes, premios,
                     ↪  media_coluna);
682             }
683
684             // Ao retornar das recursões, a rainha atual será retirada
                 ↪  do tabuleiro e colocada na fila novamente
685                 // e será procurado a próxima solução disponível
686
```

```
687             Enfilera(posicoes_restantes, linha_atual_temp);
688
689             Retira_Rainha_Tabuleiro_Calculando_Premio(tabuleiro_atual,
                  ↪  coluna_atual, linha_atual_temp, premios);
690
691             Imprime_Fila(*posicoes_restantes);
692
693             iteracoes++;
694          } // if
695
696
697       // Se a posição escolhida não for válida
698       } else {
699
700          if(global_imprime_tela_ativado) {
701             printf("\n[INFO] Posição Inválida. Retornando o valor %d à
                  ↪  fila", linha_atual_temp);
702             fflush(stdout);
703          }
704
705          // Readiciona-la no final da fila
706          Enfilera(posicoes_restantes, linha_atual_temp);
707
708          Imprime_Fila(*posicoes_restantes);
709
710          // Passa pra próxima rainha
711          iteracoes++;
712
713       } // if
714
715    } // While
716
717
718    // Após ter percorrido todos as soluções deste nível e seus
       ↪  subníveis
719       // é retornado um nível para continuar a busca.
720
721    if(global_imprime_tela_ativado)
722       printf("\n[INFO] Saindo do nível %d", coluna_atual);
723
724    return ;
725 }
726
727
728 int main(int argc, char** argv) {
729    int * premios = 0;
```

```
730     char * diretorio = 0;
731     Tabuleiro * tabuleiro_atual = 0, * tabuleiro_maximo_encontrado = 0;
732     Fila * fila_rainhas = 0;
733     time_t seconds = 0;
734     float* media_coluna_premios = 0;
735 
736     if (argc != 4) {
737        diretorio = argv[1];
738        srand(atoi(argv[2]));
739        seconds = atoi(argv[3]);
740        global_imprime_tela_ativado = atoi(argv[4]);
741 
742     } else {
743        exit(-1);
744     }
745 
746     Le_Premios(diretorio, &premios, &media_coluna_premios);
747 
748     tabuleiro_atual = Cria_Tabuleiro();
749     tabuleiro_maximo_encontrado = Cria_Tabuleiro();
750 
751     fila_rainhas = Cria_Fila_Aleatoria();
752 
753     start = time(NULL);
754 
755     endwait = start + seconds;
756 
757     n_Rainhas_Prize(0, tabuleiro_atual, tabuleiro_maximo_encontrado,
        ↪  &fila_rainhas, premios, media_coluna_premios);
758 
759     Imprime_Tabuleiro_Final(*tabuleiro_maximo_encontrado, premios);
760 
761     Desaloca_Fila(fila_rainhas);
762     Desaloca_Premios_E_Media_Coluna(&premios, &media_coluna_premios);
763     Desaloca_Tabuleiro(&tabuleiro_maximo_encontrado);
764     Desaloca_Tabuleiro(&tabuleiro_atual);
765 
766     return (EXIT_SUCCESS);
767 }
```

8. Anexos

Tabela 4: Tabela com todos os valores obtidos.

Arquivo	*Seed*	Prêmio *BT*	Prêmio *B&B* 1	Prêmio *B&B* 2	Ótimo
nqp005.txt	0	167	167	167	167
nqp005.txt	1234	167	167	167	167
nqp005.txt	2468	167	167	167	167
nqp005.txt	3702	167	167	167	167
nqp005.txt	4936	167	167	167	167
nqp005.txt	6170	167	167	167	167
nqp005.txt	7404	167	167	167	167
nqp005.txt	8638	167	167	167	167
nqp005.txt	9872	167	167	167	167
nqp005.txt	11106	167	167	167	167
nqp005.txt	12340	167	167	167	167
nqp005.txt	13574	167	167	167	167
nqp008.txt	0	298	298	298	298
nqp008.txt	1234	298	298	298	298
nqp008.txt	2468	298	298	298	298
nqp008.txt	3702	298	298	298	298
nqp008.txt	4936	298	298	298	298
nqp008.txt	6170	298	298	298	298
nqp008.txt	7404	298	298	298	298
nqp008.txt	8638	298	298	298	298
nqp008.txt	9872	298	298	298	298
nqp008.txt	11106	298	298	298	298
nqp008.txt	12340	298	298	298	298
nqp008.txt	13574	298	298	298	298
nqp010.txt	0	381	381	381	381
nqp010.txt	1234	381	381	381	381
nqp010.txt	2468	381	381	381	381
nqp010.txt	3702	381	381	381	381
nqp010.txt	4936	381	381	381	381
nqp010.txt	6170	381	381	381	381
nqp010.txt	7404	381	381	381	381
nqp010.txt	8638	381	381	381	381
nqp010.txt	9872	381	381	381	381
nqp010.txt	11106	381	381	381	381
nqp010.txt	12340	381	381	381	381
nqp010.txt	13574	381	381	381	381
nqp020.txt	0	743	797	787	883
nqp020.txt	1234	777	797	797	883
nqp020.txt	2468	778	797	797	883
nqp020.txt	3702	762	774	774	883
nqp020.txt	4936	734	789	785	883
nqp020.txt	6170	763	809	809	883
nqp020.txt	7404	733	767	767	883
nqp020.txt	8638	786	823	823	883
nqp020.txt	9872	768	795	789	883
nqp020.txt	11106	767	808	808	883
nqp020.txt	12340	755	765	765	883
nqp020.txt	13574	729	801	768	883
nqp030.txt	0	960	995	1021	1372
nqp030.txt	1234	957	1046	1050	1372
nqp030.txt	2468	1023	1094	1104	1372
nqp030.txt	3702	937	1008	1025	1372
nqp030.txt	4936	973	1041	1044	1372
nqp030.txt	6170	974	1097	1106	1372
nqp030.txt	7404	1025	1056	1074	1372
nqp030.txt	8638	1018	1096	1096	1372
nqp030.txt	9872	1019	1132	1145	1372
nqp030.txt	11106	1021	1107	1107	1372
nqp030.txt	12340	1071	1154	1154	1372
nqp030.txt	13574	966	1062	1063	1372
nqp040.txt	0	1275	1430	1430	1883
nqp040.txt	1234	1456	1563	1509	1883
nqp040.txt	2468	1294	1459	1435	1883
nqp040.txt	3702	1173	1276	1283	1883
nqp040.txt	4936	1228	1351	1388	1883

Tabela 4: (continuação)

nqp040.txt	6170	1191	1222	1253	1883
nqp040.txt	7404	1291	1458	1417	1883
nqp040.txt	8638	1164	1346	1393	1883
nqp040.txt	9872	1274	1444	1454	1883
nqp040.txt	11106	1348	1452	1454	1883
nqp040.txt	12340	1221	1342	1376	1883
nqp040.txt	13574	1246	1350	1359	1883
nqp050.txt	0	1490	1628	1665	2380
nqp050.txt	1234	1530	1607	1676	2380
nqp050.txt	2468	1666	1825	1771	2380
nqp050.txt	3702	1520	1573	1642	2380
nqp050.txt	4936	1479	1679	1723	2380
nqp050.txt	6170	1565	1660	1727	2380
nqp050.txt	7404	1572	1694	1698	2380
nqp050.txt	8638	1524	1643	1643	2380
nqp050.txt	9872	1535	1594	1630	2380
nqp050.txt	11106	1463	1629	1680	2380
nqp050.txt	12340	1437	1621	1656	2380
nqp050.txt	13574	1573	1743	1786	2380
nqp060.txt	0	1612	1705	1763	2874
nqp060.txt	1234	1700	1959	1936	2874
nqp060.txt	2468	1829	1997	2032	2874
nqp060.txt	3702	1798	1823	1798	2874
nqp060.txt	4936	1816	1911	2019	2874
nqp060.txt	6170	1864	2110	2064	2874
nqp060.txt	7404	1910	2003	2045	2874
nqp060.txt	8638	1800	1866	1907	2874
nqp060.txt	9872	1720	1852	1987	2874
nqp060.txt	11106	1739	1830	1872	2874
nqp060.txt	12340	1743	1827	1889	2874
nqp060.txt	13574	1696	1716	1763	2874
nqp070.txt	0	1939	2149	2143	Desconhecido
nqp070.txt	1234	2170	2386	2389	Desconhecido
nqp070.txt	2468	1870	2000	1951	Desconhecido
nqp070.txt	3702	1991	2181	2289	Desconhecido
nqp070.txt	4936	2012	2265	2296	Desconhecido
nqp070.txt	6170	1830	1663	1737	Desconhecido
nqp070.txt	7404	2073	2051	2080	Desconhecido
nqp070.txt	8638	2048	2362	2307	Desconhecido
nqp070.txt	9872	2077	2197	2185	Desconhecido
nqp070.txt	11106	2110	2292	2309	Desconhecido
nqp070.txt	12340	1832	2084	1898	Desconhecido
nqp070.txt	13574	2279	2397	2444	Desconhecido
nqp080.txt	0	2185	2402	2415	Desconhecido
nqp080.txt	1234	2199	2269	2364	Desconhecido
nqp080.txt	2468	2271	2231	2304	Desconhecido
nqp080.txt	3702	2321	2376	2386	Desconhecido
nqp080.txt	4936	1938	2269	2184	Desconhecido
nqp080.txt	6170	2457	2750	2712	Desconhecido
nqp080.txt	7404	2120	2357	2410	Desconhecido
nqp080.txt	8638	2256	2528	2536	Desconhecido
nqp080.txt	9872	0	0	0	Desconhecido
nqp080.txt	11106	2434	2534	2591	Desconhecido
nqp080.txt	12340	2408	2496	2492	Desconhecido
nqp080.txt	13574	2445	2691	2656	Desconhecido
nqp090.txt	0	2614	2863	2838	Desconhecido
nqp090.txt	1234	2707	2759	2764	Desconhecido
nqp090.txt	2468	2488	2574	2564	Desconhecido
nqp090.txt	3702	2574	2713	2709	Desconhecido
nqp090.txt	4936	2652	2942	2948	Desconhecido
nqp090.txt	6170	2441	2854	2815	Desconhecido
nqp090.txt	7404	2620	2640	2577	Desconhecido
nqp090.txt	8638	2485	2631	2815	Desconhecido
nqp090.txt	9872	2642	2878	2881	Desconhecido
nqp090.txt	11106	2565	2625	2674	Desconhecido
nqp090.txt	12340	2558	2622	2673	Desconhecido
nqp090.txt	13574	2296	2623	2588	Desconhecido
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
nqp100.txt	4936	2644	2802	2833	Desconhecido
nqp100.txt	6170	2810	2817	2820	Desconhecido
nqp100.txt	7404	2628	2564	2603	Desconhecido
nqp100.txt	8638	2708	2926	2971	Desconhecido
nqp100.txt	9872	2674	2798	2881	Desconhecido
nqp100.txt	11106	2769	2863	2902	Desconhecido
nqp100.txt	12340	2860	3210	2922	Desconhecido
nqp100.txt	13574	2789	2908	2909	Desconhecido
nqp200.txt	0	5329	5329	5329	Desconhecido
nqp200.txt	1234	0	0	0	Desconhecido
nqp200.txt	2468	0	0	0	Desconhecido
nqp200.txt	3702	4994	4994	4994	Desconhecido
nqp200.txt	4936	5582	5582	5582	Desconhecido
nqp200.txt	6170	5479	5479	5479	Desconhecido
nqp200.txt	7404	5179	5179	5179	Desconhecido
nqp200.txt	8638	0	0	0	Desconhecido
nqp200.txt	9872	5390	5390	5390	Desconhecido
nqp200.txt	11106	5283	5283	5283	Desconhecido
nqp200.txt	12340	0	0	0	Desconhecido
nqp200.txt	13574	5289	5289	5289	Desconhecido