Contador de Tarefas

André Luiz Marques Macrini Leite, Giovanni Frozza

27 de setembro de 2020

Resumo

O presente artigo descreve uma solução para um problema proposto na disciplina de Algoritmos e Estruturas de Dados II que tem como objetivo exercitar os conhecimentos adquiridos sobre árvores, análise de complexidade e capacidade de desenvolvimento de algoritmos ao longo da disciplina. O problema consiste em automatizar um cálculo para descobrir o tempo gasto em uma sequencia de tarefas utilizando multiprocessadores para processar essas tarefas, ou não, considerando e comparando duas políticas para executar a próxima tarefa. A política de pegar sempre a menor tarefa e a política de pegar sempre a maior tarefa. Serão apresentados testes e seus resultados, assim como a conclusão chegada.

Palavras-chaves: Algoritmos. Árvore. Processador.

Introdução

O problema proposto consiste na criação de um algoritmo capaz de calcular o tempo que uma empresa irá gastar para realizar determinadas tarefas. As tarefas podem ser realizadas simultaneamente e o algoritmo deve calcular o tempo gasto utilizando duas políticas:

- Política Min
- Política Max

Na política Min, quando as tarefas forem ser executadas, serão consideradas primeiramente as tarefas com o menor tempo de execução. Já na política Max, as tarefas primeiramente consideradas são as tarefas que possuem um maior tempo para serem concluídas.

Partindo das definições dadas no enunciado do trabalho, deve ser possível responder a questão:

- Quanto tempo é gasto para executar as tarefas utilizando cada política?

A entrada recebida é um arquivo texto composto por uma string, que representa o código da tarefa e um número que representa o tempo que a tarefa demora para ser executada. Esses dois são separados pelo símbolo "underline". Cada linha do arquivo representa uma tarefa e a sua próxima, estas separadas pelos símbolos ->". Além disso, como as tarefas podem ser executadas simultaneamente, na entrada é recebido na primeira linha do arquivo, a quantidade de processadores que irão processar as tarefas recebidas.

Para a resolução do problema analisamos a entrada e chegamos a uma solução. Em seguida serão apresentados os resultados obtidos e a conclusão levantadas ao decorrer do trabalho.

Segue abaixo o exemplo de um de uma árvore demonstrando as sequencias das tarefas que foi dado no enunciado do trabalho.

No exemplo utilizando a política min foi obtido 2178 unidades de tempo, já na política max foi obtido 2337 unidades de tempo.

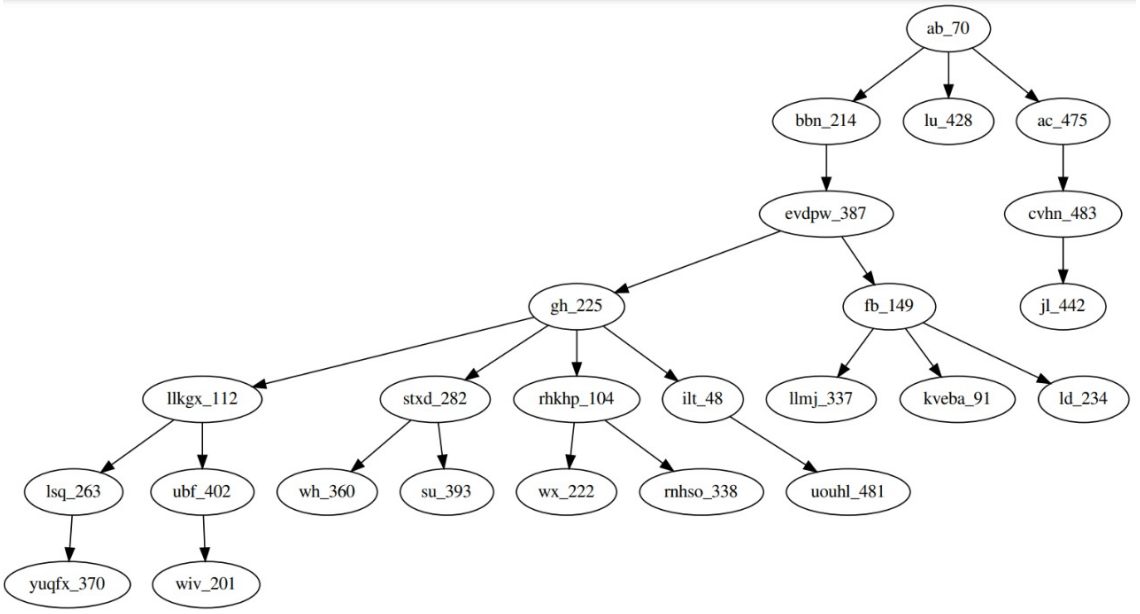


Figura 1 – Exemplo

O Problema

Você está trabalhando em uma grande empresa que virtualiza soluções na nuvem. Isso é tudo muito motivador, disruptivo, inovador e quântico, mas tem problemas como qualquer empresa e você agora tem que resolver um deles: seus clientes querem processar grandes conjuntos de tarefas que precisam ser feitas na ordem certa e levam tempos diferentes para serem realizadas. Se eles comprarem apenas um processador da sua nuvem, é claro que o tempo de completar tudo será a soma de todos os tempos das tarefas, mas quanto tempo demora se eles comprarem dois processadores? E três? E quatro? E cento e quarenta e dois? Você está cansado de fazer continhas à mão para responder esse tipo de pergunta e resolveu automatizar o cálculo.

Visto isso, o problema consiste em descobrir quanto tempo as tarefas vão demorar para serem executadas levando em consideração a quantidade de processadores contratados pelo cliente.

Primeiramente, analisamos a entrada dada no exemplo do enunciado para identificar algum padrão. A entrada do exemplo era:

```
# Proc 4
lsq_263 -> yuqfx_370
llkgx_112 -> lsq_263
llkgx_112 -> ubf_402
gh_225 -> llkgx_112
gh_225 -> stxd_282
gh_225 -> rhkhp_104
gh_225 -> ilt_48
evdpw_387 -> gh_225
evdpw_387 -> fb_149
ubf_402 -> wiv_201
stxd_282 -> wh_360
stxd_282 -> su_393
bbn_214 -> evdpw_387
rhkhp_104 -> wx_222
rhkhp_104 -> rnhso_338
fb_149 -> llmj_337
fb_149 -> kveba_91
fb_149 -> ld_234
ab_70 -> bbn_214
ab_70 -> lu_428
ab_70 -> ac_475
ilt_48 -> uouhl_481
ac_475 -> cvhn_483
cvhn_483 -> jl_442
```

Figura 2 – Entrada

A entrada é composta por diversas linhas contendo letras, números e símbolos. Cada linha da entrada representa uma tarefa e qual é a sua próximo, tendo em vista que a sua próxima só poderá ser executada após o seu término. Essas duas são separadas pelos símbolos "->", onde a parte que fica na esquerda se refere a tarefa de "entrada"e a que fica na esquerda se refere a tarefa de "Saída"(próxima tarefa). Além disso, cada tarefa possui duas informações, o código da tarefa e o tempo gasto na tarefa. Essas duas informações estão presentes em cada tarefa da entrada e como dito anteriormente, cada linha possui duas tarefas. As informações das tarefas são separadas pelo símbolo "_". A informação logo a esquerda deste símbolo é o código da tarefa, já a informação à direita deste símbolo é o tempo gasto na tarefa.

A linha

blabla_213 -> tititi_53

Significa que existe uma tarefa com código **blabla** que demora **213** unidades de tempo para ser executada e uma tarefa com código **tititi**, que demora **53** unidades de tempo para ser executada e esta última será liberada somente depois do término da primeira.

Primeira Solução

Com base no que foi dado, pensamos em uma maneira de resolver o problema com Árvores. Foi utilizada então uma estrutura encadeada para a geração da Árvore. Partindo disso, a solução começou a ser elaborada.

Seguindo com a ideia, pensamos inicialmente sobre como adicionar as tarefas na árvore. Analisando a entrada do exemplo, pudemos perceber que em todas as entradas já estavam as definições da árvore. Cada linha da entrada representava uma ligação de pai e filho na árvore. Pensando nisso, bastou irmos adicionando essas ligações na árvore seguindo as seguintes regras para realizar essa inclusão:

- Se a tarefa pai e a tarefa filho já existirem na árvore:
  - Remove a tarefa filho do seu atual pai.
  - Adiciona a tarefa filho no pai correto.
  - Altera o atual pai da tarefa filho, para o pai correto.
- Se a tarefa pai já existir na árvore, mas a tarefa filho não:
  - Cria uma nova referencia para a tarefa filho.
  - Adiciona a tarefa filho nos filhos da tarefa pai.
  - Altera a tarefa pai para o pai da tarefa filho.
- Se a tarefa filho já existir na árvore, mas a tarefa pai não:
  - Cria uma nova referencia para a tarefa pai.
  - Adiciona nos filhos da tarefa pai a tarefa filho.
  - Se o filho possuir um pai não nulo.

        * Remove o filho encontrado dos filhos da tarefa pai atual.
        * Adiciona na referencia do pai da tarefa pai criada, o pai da tarefa filho.
    - Adiciona nova tarefa pai na referencia do pai da tarefa filho.
    - Se a tarefa filho for igual à tarefa que esta na raiz da árvore.
        * A raiz da árvore passa a ser o pai criado.

- Se a tarefa filho e a tarefa pai não existirem na árvore:
    - Cria uma nova referencia para a tarefa pai.
    - Cria uma nova referencia para a tarefa filho.
    - Adiciona nos filhos da tarefa pai, a tarefa filho criada.
    - Adiciona no pai da tarefa filho, o pai criado.
    - Adiciona nos filhos da raiz da árvore, o pai criado.
    - Altera o pai da tarefa pai criada, para a raiz da árvore.

Após implementada essa lógica e conseguirmos montar as árvores corretamente, fizemos o cálculo do tempo gasto nas tarefas.

Para isso, pensamos em utilizar uma lista para representar os processadores que executam as tarefas. Esses processadores são responsáveis por armazenar as tarefas e por fazer o cálculo para descobrir o tempo gasto nas tarefas. Fizemos uma classe interna Node que é um nó na árvore que criamos. Utilizamos uma marcação para sinalizar em qual nodo já passamos. Primeiramente o programa pega a raiz da árvore, que foi montada utilizando as regras descritas anteriormente, e marca a raiz para informar que já passou por lá. Após isso, adicionamos o tempo gasto na tarefa que estava na raiz em um somador que será somado toda que uma tarefa for concluída. Então chamamos um método que percorre a árvore recursivamente buscando os filhos liberados, adicionando eles na listagem de processadores calculando o tempo gasto e somando o tempo no somador.

Esta implementação deu certo até certo ponto. Nos casos menores funcionou corretamente, porém nos casos a partir do 100, começou a dar StackOverflow.

Por esse motivo, decidimos rever os conceitos e utilizar uma nova abordagem para a resolução do problema, o que nos leva para a nossa segunda solução.

Segunda Solução

Voltando para estaca zero, decidimos analisar a entrada de uma outra forma. Pensamos nela não como diversas linhas, mas sim como duas colunas. Uma coluna com as "entradas"e uma coluna com as "saídas", representando o lado esquerdo e direito do símbolo "->". A coluna de entradas teria as tarefas pais e a coluna de saída teria as tarefas filhas.

Primeiramente lemos o arquivo e separamos as colunas em duas listas, uma "listaEntradas"e uma "listaSaidas". Tratamos essas duas listas em nosso algoritmos como duas listas espelho, ou seja, cada uma das posições da "listaEntradas"tem uma ligação com a mesma posição na "listaSaidas".

O problema pede para que sejam calculados os tempos utilizando duas políticas: a política de sempre pegar a tarefa com menor tempo para processar (política min) e a política de sempre pegar a tarefa com maior tempo para processar (política max). Para

utilizarmos a política min, nós ordenamos as duas listas, "listaEntradas"e "listaSaidas", em ordem crescente com base nos tempos das tarefas contidas na "listaSaidas".

Já para utilizarmos a política max, nós ordenamos as mesmas duas listas em ordem decrescente com base nos tempos das tarefas contidas na "listaSaidas".

Segue abaixo o trecho que utilizamos para fazer ordenação (o trecho abaixo foi o utilizado para política min):

```
boolean trocou = false;
do {
    trocou = false;
    for (int i = 0; i < listaSaida.size() - 1; i++) {
        if (listaSaida.get(i).time > listaSaida.get(i + 1).time) {
            Task auxEntrada = listaEntrada.get(i);
            listaEntrada.set(i, listaEntrada.get(i + 1));
            listaEntrada.set(i + 1, auxEntrada);
            Task auxSaida = listaSaida.get(i);
            listaSaida.set(i, listaSaida.get(i + 1));
            listaSaida.set(i + 1, auxSaida);
            trocou = true;
        }
    }
} while (trocou);
```

Após termos essas duas listas ordenadas, devemos descobrir quem é o root, para podermos iniciar o processamento. Para saber quem é o root, basta encontrar um elemento da listaEntradas que não esteja na listaSaidas, pois o root é o único elemento que não possui um pai, mas possui filhos.

Segue abaixo o trecho que fizemos a busca pelo root nas listas:

```
Task root = null;
for (Task tIn : listaEntrada) {
    boolean achou = false;
    for (Task tOut : listaSaida) {
        achou = false;
        if (tIn.taskCode.equals(tOut.taskCode)) {
            achou = true;
            break;
        }
    }
    if (!achou) {
        root = tIn;
        break;
    }
}
```

Ao encontrarmos o root podemos iniciar o processamento das tarefas.

Para realizar o processamento tivemos uma ideia parecida com a ideia da primeira solução. Pensamos em criar uma lista de processadores que armazenam as tarefas em execução. Essa lista de processadores é quem cuida da execução simultânea de tarefas, assim

como o cálculo dos tempos de execução. Cada vez que uma tarefa termina sua execução, seu tempo deve ser subtraído das demais tarefas no processador. Por exemplo, se temos três processadores, **1**, **2** e **3**, com as tarefas **aa_30**, **bb_20** e **cc_40** respectivamente, o processo com a tarefa com menor tempo irá terminar primeiro, ou seja o processo **2** irá encerrar antes. Como os as tarefas estavam sendo executadas em paralelo, entende-se que o tempo passado para todas as tarefas é o mesmo, então após ocorrer o término da execução do processo **2**, deve ser subtraído o tempo gasto, no caso **20** unidades de tempo, de todas as demais tarefas que estão nos outros processadores. logo, no final da execução da tarefa **bb_30**, os processadores irão ficar assim:

aa_10 - bb_0 - cc_20

Para fazer essa subtração ordenamos a lista de processadores em ordem crescente, pegamos sua primeira posição, que seria o com menor tempo, e percorremos todos os processadores não vazios, subtraindo seus tempos com o menor. Ao terminar a execução de um processo, ou seja, a subtração der **0**, o mesmo é adicionado em uma lista de finalizados, para que possamos buscar os processos que já foram finalizados, com a intenção de encontrar os filhos dessas tarefas. Além de ser adicionado nessa lista, quando um processo é finalizado, o tempo dele é somado em uma variável utilizada para fazer o somatório do tempo gasto na execução de todas as tarefas.

Abaixo segue o método responsável por esse controle dos processadores:

```
public void calcula(List<String> liberados) {
    this.processadores.sort((Task n1, Task n2) -> n1.time - n2.time);
    List<Task> listaProcessos = this.processadores
                                                .stream()
                                                .filter(p -> p.time > 0)
                                                .collect(Collectors.toList());
    if (listaProcessos == null || listaProcessos.size() == 0) return;
    int timeLess = listaProcessos.get(0).time;
    for (Task p : this.processadores) {
        if (p.time > 0) {
            int sub = p.time - timeLess;
            if (sub == 0) {
                if (!liberados.contains(p.taskCode)) {
                    liberados.add(p.taskCode);
                }
            }
            p.time = sub;
        }
    }
    this.soma += timeLess;
}
```

Após encontrarmos o root, adicionamos ele nos processadores e chamamos a função "calcula()", descrita anteriormente, passando por parâmetro a lista de tarefas já finalizadas.

Para cada processador disponível iteramos a "listaEntradas"verificando se cada item está presente na lista de tarefas finalizadas. Caso uma tarefa esteja nessa lista, pegamos a tarefa correspondente a ela na mesma posição na lista espelho, ou seja, a

"listaSaidas"e adicionamos a tarefa nos processadores para que seja processada. Após isso removemos o elemento da "listaEntradas"e da "listaSaidas", para não voltarmos a processar a tarefa novamente. Esse procedimento nós denominamos de "popular", que é seguido do procedimento "calcular", que é o método "calcula()", mencionado anteriormente. Esses dois procedimentos são o core do nosso algoritmo. Isso tudo é feito enquanto a "listaEntradas"possuir algum elemento. Abaixo segue a implementação desse trecho:

```
List<String> liberados = new ArrayList<>();
proc.add(root);
proc.calcula(liberados);
while(!listaEntrada.isEmpty()) {
    for (int j = 0; j < proc.processadores.size(); j++) {
        if (proc.numeroProcessadoresDisponiveis() > 0) {
            for (int i = 0; i < listaEntrada.size(); i++) {
                if (proc.numeroProcessadoresDisponiveis() > 0
                    && liberados.contains(listaEntrada.get(i).taskCode)) {
                    proc.add(listaSaida.get(i));
                    listaEntrada.remove(listaEntrada.get(i));
                    listaSaida.remove(listaSaida.get(i));
                }
            }
        }
    }
    proc.calcula(liberados);
}
while (proc.numeroProcessadoresDisponiveis() < proc.processadores.size()) {
    proc.calcula(liberados);
}
```

Por fim, enquanto os processadores tiverem alguém sendo processado, é feita uma iteração calculando os processos até que todos os que sobraram nos processadores sejam zerados.

Teste de mesa

Segue abaixo um teste de mesa realizado em um exemplo para exemplicar de uma maneira prática o funcionamento do nosso algoritmo:

Entrada recebida:

Demonstração na forma de árvore para visualização da hierarquia de tarefas:

Execução do nosso algoritmo focando nos procedimentos "popula"e "calcula"mencionados anteriormente:

```
# Proc 2
aax_61 -> amqnr_493
aax_61 -> hocr_273
amqnr_493 -> vbmj_213
amqnr_493 -> txi_174
hocr_273 -> tb_24
aax_61 -> nco_204
```

Figura 3 – Entrada

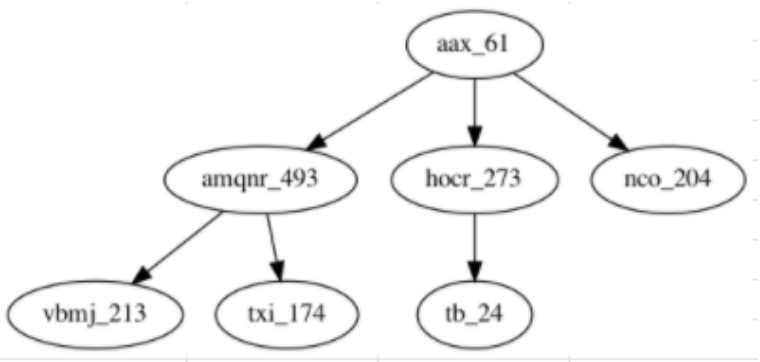


Figura 4 – Visualização em Árvore

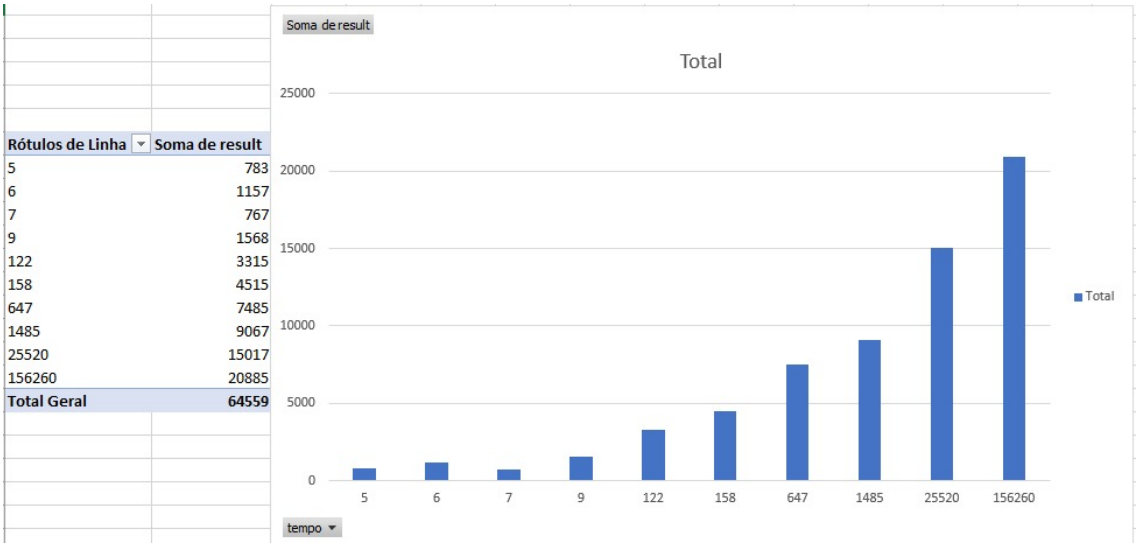
PROCEDIMENTO	TAREFA	1	TAREFA	2	SOMA	TOTAL
Popula	aax	61	-	0	61	971
Calcula	aax	0	-	0	204	
Popula	nco	204	hocr	273	69	
Calcula	nco	0	hocr	69	24	
Popula	amqnr	493	hocr	69	400	
Calcula	amqnr	424	hocr	0	174	
Popula	amqnr	424	tb	24	39	
Calcula	amqnr	400	tb	0		
Popula	amqnr	400	tb	0		
Calcula	amqnr	0	tb	0		
Popula	txi	174	vbmj	213		
Calcula	txi	0	vbmj	39		
Popula	txi	0	vbmj	39		
Calcula	txi	0	vbmj	0		

Figura 5 – Procedimento

Análise do Algoritmo

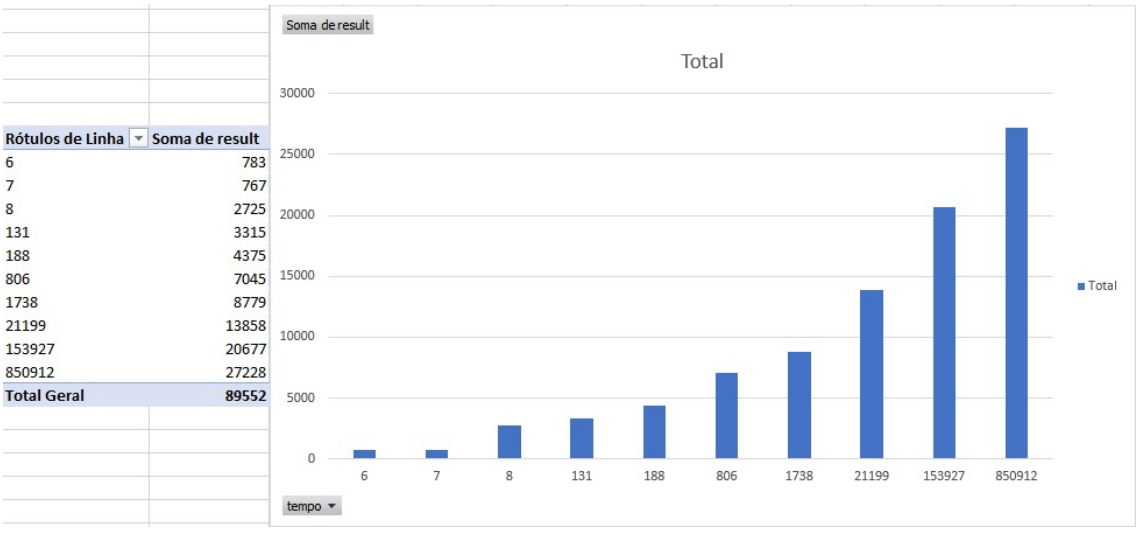
Referente a complexidade do algoritmo, analisamos os resultados obtidos e plotamos em um gráfico representando no eixo Y os valores obtidos nos resultados e no eixo X os

tempos levados para execução. Separamos em dois gráficos, um para a política min e outro para política max.



Rótulos de Linha	Soma de result
5	783
6	1157
7	767
9	1568
122	3315
158	4515
647	7485
1485	9067
25520	15017
156260	20885
Total Geral	**64559**

Figura 6 – Gráfico análise do algoritmo Política min



Rótulos de Linha	Soma de result
6	783
7	767
8	2725
131	3315
188	4375
806	7045
1738	8779
21199	13858
153927	20677
850912	27228
Total Geral	**89552**

Figura 7 – Gráfico análise do algoritmo Política max

Com base na análise feita, chegamos na conclusão de que o algoritmo possui complexidade de O(n^2).

Resultados

Depois de implementar o algoritmo acima na linguagem Java e executá-lo, obtivemos os seguintes resultados:

CASOS	RESULTADO		tempo(milissegundos)	
	Politica Min	Politica Max	Politica Min	Politica Max
caso5	783	783	5	6
caso6	1157	1157	6	8
caso7	767	767	7	7
caso10	1568	1568	9	8
caso100	3315	3315	122	131
caso200	4515	4375	158	188
caso500	7485	7045	647	806
caso1000	9067	8779	1485	1738
caso2000	15017	13858	25520	21199
caso5000	20885	20677	156260	153927
caso10000		27228		850912

Figura 8 – Resultados

O algoritmo não conseguiu executar o caso10000 com a política min e acabou entrando em um laço infinito, por esse motivo não colocamos na tabela de resultados os dados referente à este caso na política min.

Considerações finais

No decorrer do desenvolvimento do trabalho tivemos problemas ao tentar implementar um algoritmo que utilizava árvore e por isso resolvemos mudar nossa implementação para não utilizarmos mais essa estrutura de árvore. Após o término do trabalho pudemos visualizar o funcionamento do algoritmo e verificar o seu desempenho. Identificamos que em alguns casos o algoritmo não chegou exatamente na resposta esperada. Não conseguimos identificar o problema em questão, mas para vários casos ele rodou corretamente e em um tempo satisfatório.