**Trabalho Prático 2:**

**SURPRESA NA PROVA**

**Fábio Dayrell Ferreira Martins Rosa**

dayrell@dcc.ufmg.br

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais

**Resumo:** este relatório descreve a implementação de um programa do tipo NP-Completo. A solução foi feita utilizando a representação de grafos por matrizes de adjacências alocadas dinamicamente. O trabalho permitiu praticar o uso dos itens solicitados: (1) modularização; (2) alocação dinâmica de memória; (3) compilação do código através da ferramenta makefile; (4) solução ótima/exata de problemas do tipo NP-Completo; (5) solução aproximada/heurística de problemas do tipo NP-Completo.

**1. INTRODUÇÃO**

Esta documentação descreve a implementação da solução do problema de encontrar o maior conjunto de alunos que não marcaram nenhuma resposta igual em uma prova. Para isso, serão disponibilizadas na entrada a lista de todos os alunos que marcaram cada resposta.

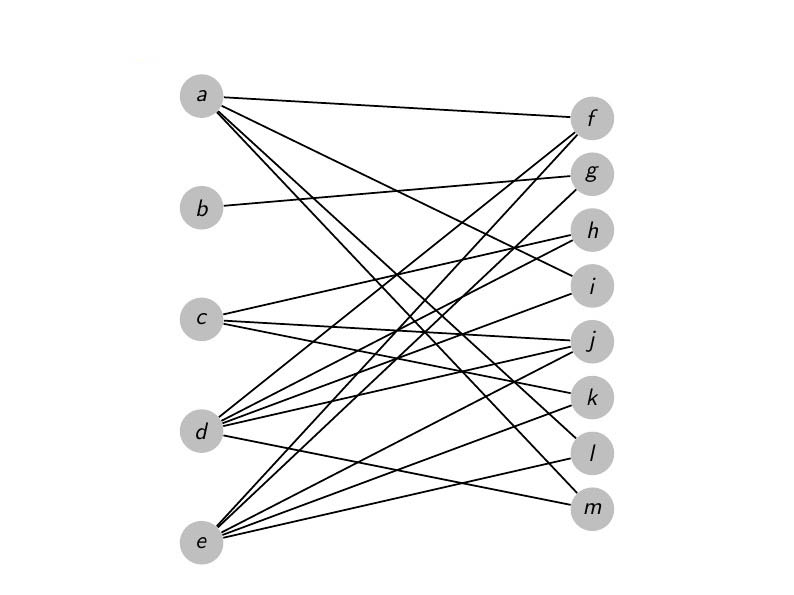
O restante deste relatório está organizado da seguinte forma:

* Seção 2: detalhes de implementação;
* Seção 3: análise de complexidade das principais funções;
* Seção 4: análise experimental e demais testes realizados durante o desenvolvimento do programa;
* Seção 5: conclusão sobre o trabalho;
* Seção 6: referencias bibliográficas.

**2. IMPLEMENTAÇÃO**

A solução para o problema foi implementada através de um grafo não direcionado, representado por uma matriz de duas dimensões alocada dinamicamente. Nessa matriz, as linhas de *i* a *j* representam os alunos e as colunas de *k* a *l* representam as respostas. Caso o aluno *i* tenha marcado a resposta *k*, a posição *Matriz[i][k]* receberá o valor inteiro 1. Caso contrário, o valor será 0. Desse modo, só existirão arestas ligando os alunos às respostas.

O exemplo dado na especificação gerou o seguinte grafo e a seguinte matriz:



No lado esquerdo estão os alunos de ‘*a’* a ‘*e*‘ e do lado direito estão as respostas de ‘*f*’ a ‘*m*’. As arestas indicam que um determinado aluno marcou uma determinada resposta.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 |
| 01 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 02 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 03 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 04 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 05 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |

Como pode ser visto acima, as matrizes terão dimensão A\*R, onde *A* é o número de alunos e *R* é o número de opções de respostas.

**Solução exata:**

Para se obter uma solução exata para esse problema é necessário testar todos os conjuntos possíveis de alunos e verificar se em cada um desses conjuntos existem apenas alunos que não responderam respostas iguais e, por esse motivo, a complexidade de tempo de execução do programa será exponencial.

As funções relacionadas à solução exata estão no arquivo funcoesOtimo.c e serão listadas abaixo.

A função ***reiniciaVetor*** recebe um vetor ***V*** do tamanho do número de opções de respostas (que também é o número de colunas da matriz) e insere o inteiro *0* em todas as suas posições.

O vetor ***V*** é essencial para testar se algum dos conjuntos de alunos marcou apenas respostas diferentes.

A função ***preencheConjuntoVertices*** recebe a matriz que representa o grafo, o número de opções de respostas, o número de um aluno e o vetor ***V*** já citado acima.

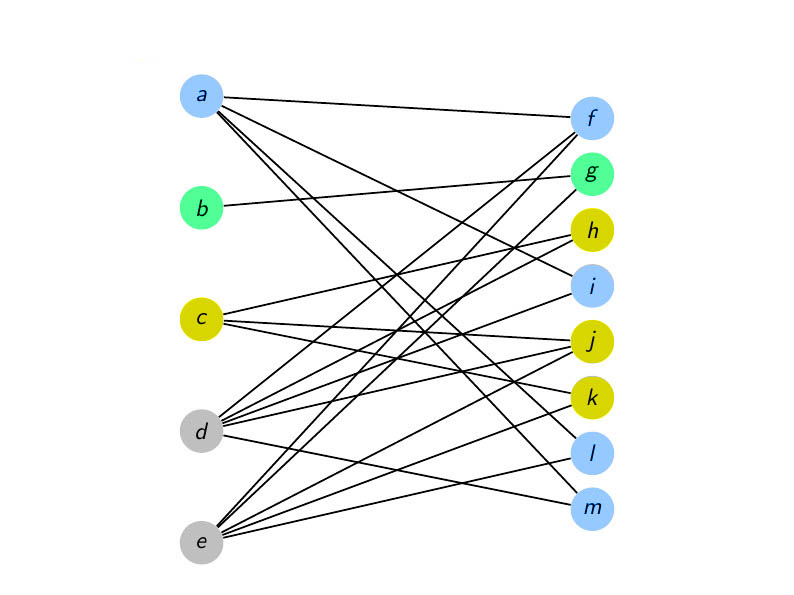
Esse vetor ***V*** é inicializado com 0 em todas posições a cada conjunto de alunos gerado. Para cada resposta *i* marcada por um aluno *k*, é adicionado o inteiro *1* na posição *i* do vetor. Se já existe o inteiro *1* numa posição a qual eu irei adicionar o inteiro *1*, isso significa que outro aluno já marcou essa opção, logo, esse conjunto não é válido.

Por exemplo: foi gerado o conjunto de alunos *AB*. *A* marcou as opções 1 e 2, já *B* marcou as respostas 3 e 4. Na primeira chamada dessa função, o vetor **V** será inicialmente *0000*, já na segunda chamada ele será *1100* e ao término dessa chamada ele será *1111* e será considerado um conjunto válido, já que nenhum aluno marcou respostas iguais e de tamanho 2.

Já se o aluno *A* marcou as opções 1, 2, 3 e o aluno *B* marcou as opções 3 e 4, o programa detectará que o conjunto *AB* é inválido, já que na segunda chamada da função o vetor **V** será *1110* e, ao tentar adicionar o inteiro 1 na posição 3 do vetor ocorrerá um erro, já que ela já está preenchida com 1.

A função ***verificaConjunto*** gera todos os conjuntos de alunos possíveis e aplica as funções citadas acima a cada um a fim de encontrar o maior conjunto válido, ou seja, o maior conjunto no qual nenhum aluno marcou opções iguais.

A imagem abaixo mostra um conjunto válido no exemplo dado na especificação e que já foi citado acima:



No exemplo, os alunos *a*, *b* e *c* formam um conjunto válido, já que eles não marcaram nenhuma resposta igual. Eles também formam o maior conjunto válido.

**Solução aproximada:**

Como a solução exata possui tempo de execução exponencial, ele se tornará inviável a partir de um determinado número de alunos. Por isso, é importante apresentar uma solução aproximada para o problema que apresente não necessariamente a solução exata, mas sim uma suficientemente boa.

As funções relacionadas à solução aproximada estão no arquivo funcoesHeuristica.c e serão listadas abaixo.

A função ***verificaQuantidadedeRespostas*** recebe um vetor **V2** do tamanho do número de alunos e as informações relacionadas ao grafo. Cada posição *i* do vetor **V2**será preenchida com a quantidade de respostas marcadas pelo aluno *i*.

A função ***escolheMenor*** recebe o vetor **V2** e as informações relacionadas ao grafo. Ela retorna o índice do aluno que respondeu menos respostas, mas, antes disso, muda o número de respostas desse aluno para -1, de modo que ele não será escolhido na próxima vez que essa função for chamada.

A função ***verificaEPreencheVetor*** recebe o índice **i** de um aluno, o vetor **V3**, que funciona de maneira semelhante aovetor **V** já citado anteriormente, e as informações relacionadas ao grafo. Essa função verifica se o aluno **i** possui respostas iguais a de outros alunos que já foram adicionadas ao conjunto. Caso não possua, as respostas marcadas por esse aluno são adicionadas ao vetor **V3** e a função retorna o inteiro 0. Caso contrário, a função apenas retorna 1.

A função ***verificaConjunto*** aplica as funções citadas acima. Um *while* repete o A vezes, onde A é o número de alunos. A cada repetição, é escolhido o aluno que marcou menos opções através da função ***escolheMenor*** e, depois, esse aluno é enviado para a função ***verificaEPreencheVetor***. Caso essa função retorne 0, o tamanho do conjunto é incrementado.

Essa solução aproximada foi implementada baseada no fato de que alunos que marcaram um número menor de respostas possuem uma probabilidade maior de não terem marcado respostas iguais.

**Leitura de dados e construção do grafo**

O programa principal (*main*) lê o número de instâncias, alunos e respostas. Um *while* dentro da *main* irá criar um grafo e aplicar as funções para cada instância do arquivo de entrada.

**Funções e estruturas de grafo**

No arquivo *grafo.c* estão as duas funções relativas à construção da matriz que representará os grafos.

A função ***inicializaGrafo*** simplesmente preenche uma matriz de dimensão (número de vértices) \* (número de vértices) com o número 0 em todas as posições.

A função **preencheGrafo** pega a matriz gerada pela função anterior e insere o número 1 nas posições onde existem arestas. Para isso, os vértices que possuem uma aresta entre eles são lidos no arquivo de entrada.

Também é importante ressaltar que os arquivos de texto devem possuir um *\n* ou *\r* em cada quebra de linha para que todas as linhas das opções de respostas sejam lidas corretamente. No entanto, esse é o padrão para arquivos de texto gerados no Mac, Windows ou Linux, portanto isso não deve gerar problemas na leitura dos arquivos.

**3. ANÁLISE DE COMPLEXIDADE**

|  |  |
| --- | --- |
| A | Alunos |
| R | Opções de Respostas |

**Funções relacionadas ao algoritmo exato**

A função ***reiniciaVetor*** e ***preencheConjuntoVertices***possuem complexidade de tempo O(R), já que apenas preenchem um vetor de tamanho R. A complexidade de espaço também é O(R).

Já a função ***verificaConjuntos*** possui complexidade de tempo O(2A\*R), já que o primeiro *for* repete 2A vezes e o *for* interno se repete R vezes. A complexidade de espaço é O(A\*R+R), já que essa função usa uma matriz de tamanho A\*R e um vetor de tamanho R.

Desse modo, a complexidade do programa com a solução exata possui complexidade de tempo O(2A\*R).

**Funções relacionadas ao algoritmo aproximado**

A função ***verificaEPreencheVetor*** possui complexidade de tempo O(R), já que preenche um vetor de tamanho R. Já a ***escolheMenor***  tem complexidade de tempo O(A), já que preenche percorre uma das linhas da matriz. A função ***verificaQuantidadeDeRespostas*** percorre toda a matriz, então possui complexidade de tempo O(A\*R).

A complexidade de tempo da função ***verificaConjunto*** é O(R+A\*A\*R), já que, logo no início da função é preenchido um vetor de tamanho R e, logo depois, as funções ***escolheMenor*** e ***verificaEPreencheVetor*** são chamadas A vezes.

Desse modo, a complexidade de tempo do programa com o algoritmo aproximado é O(A2\*R).

**4. ANÁLISE EXPERIMENTAL**

**Ambiente utilizado**

* Processador: Intel Core i5, 2,5 GHz;
* Memória: 10 GB, 1600 MHz, DDR3;
* Sistema operacional: Mac OS X, versão 10.9.1;
* Compilador: GCC.

**Compilação**

O programa deve ser compilado através do terminal através do comando *make*, que irá gerar um executável chamado *tp2*.

De modo alternativo, é possível digitar apenas *make run-e, ou make run-h* no terminal para que o executável do algoritmo exato, ou o aproximado, respectivamente, seja gerado e compilado, gerando automaticamente a saída do programa.

**Execução**

A execução do programa tem como parâmetros:

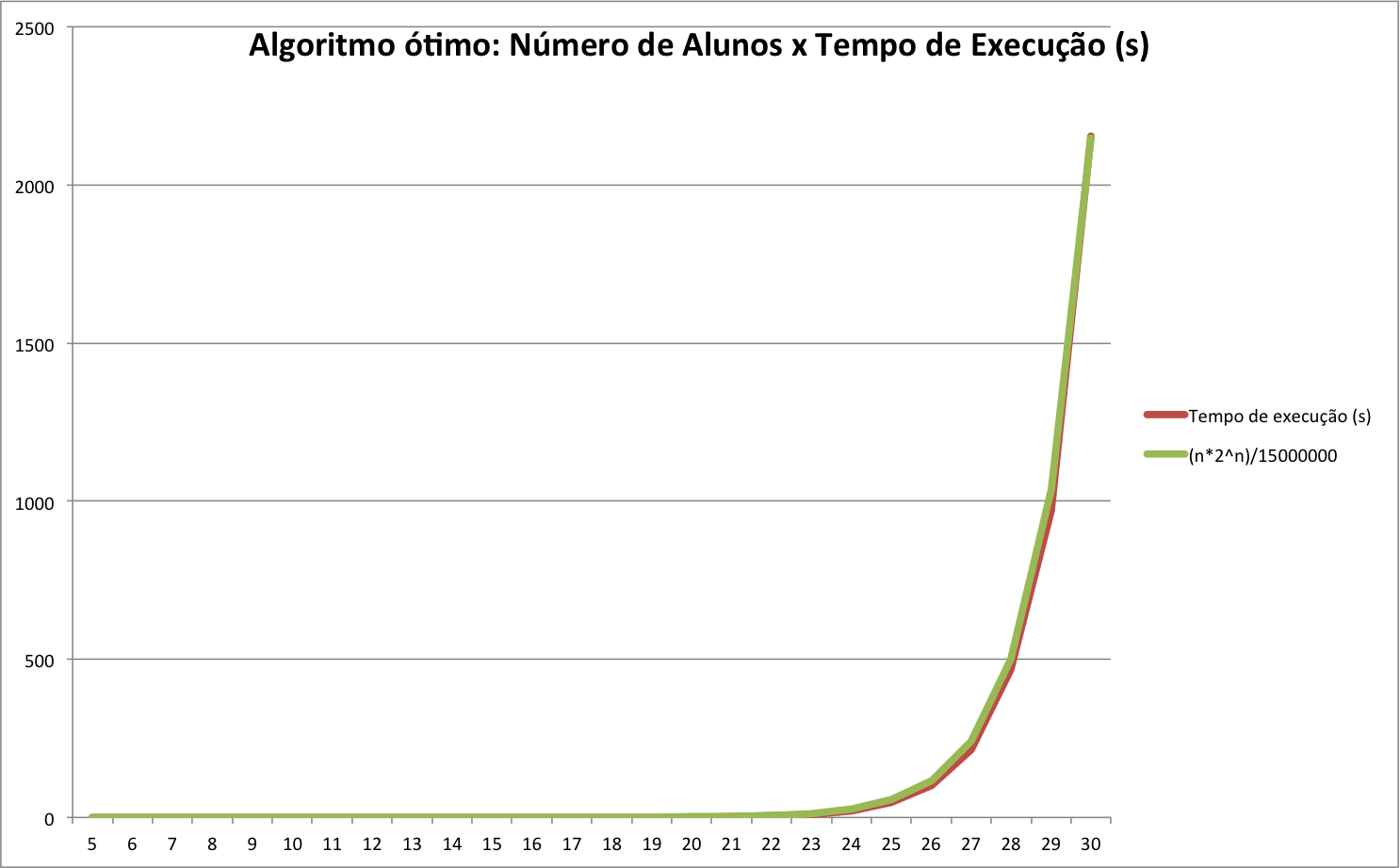
* Um arquivo de entrada;
* Um arquivo de saída.

O comando de execução do programa é da forma: *./tp2e input.txt output.txt* ou *./tp2h input.txt output.txt*.

Ou então, os programas poderão ser executado com os comandos make run-e e make run-h.

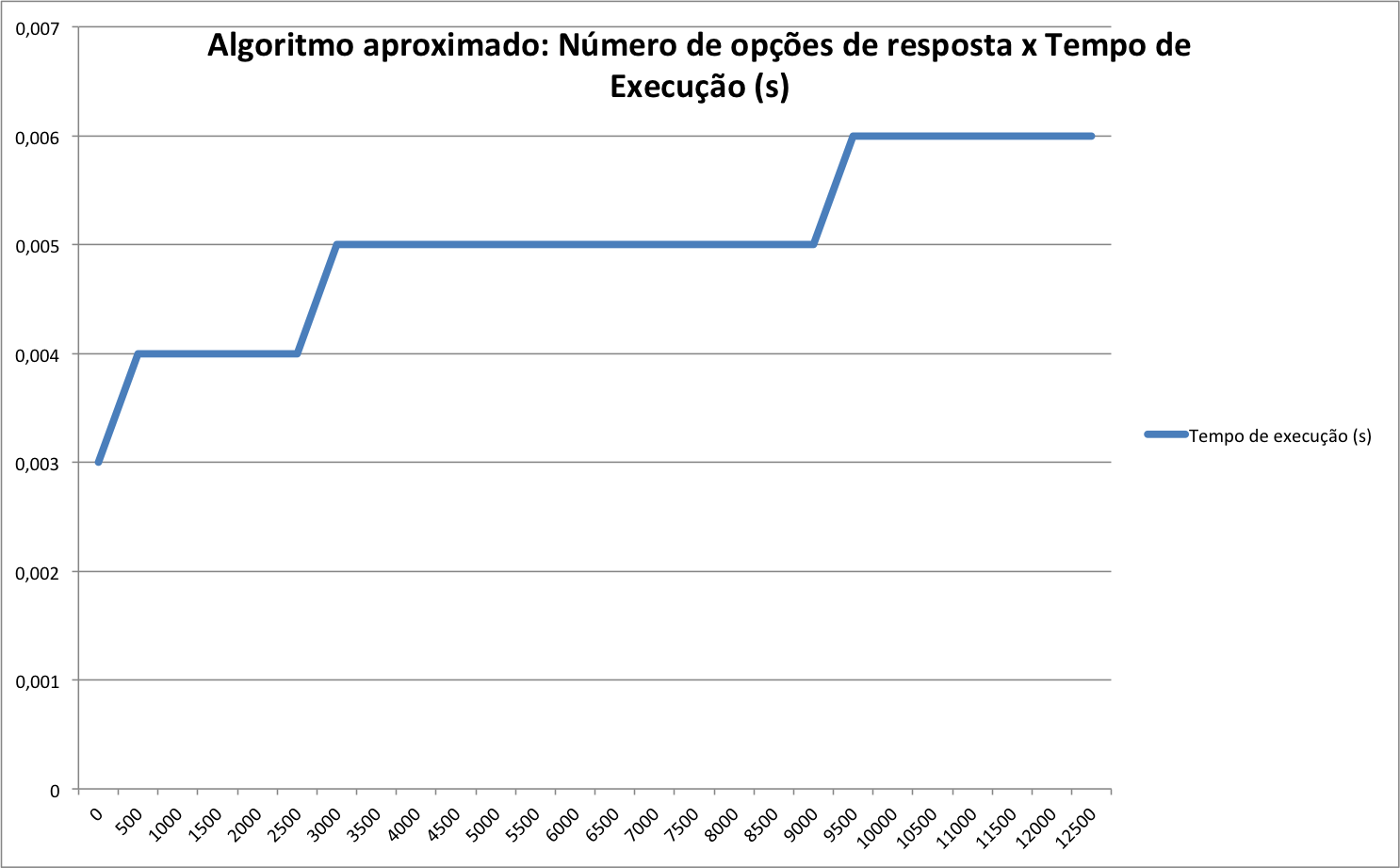
**Experimentos**

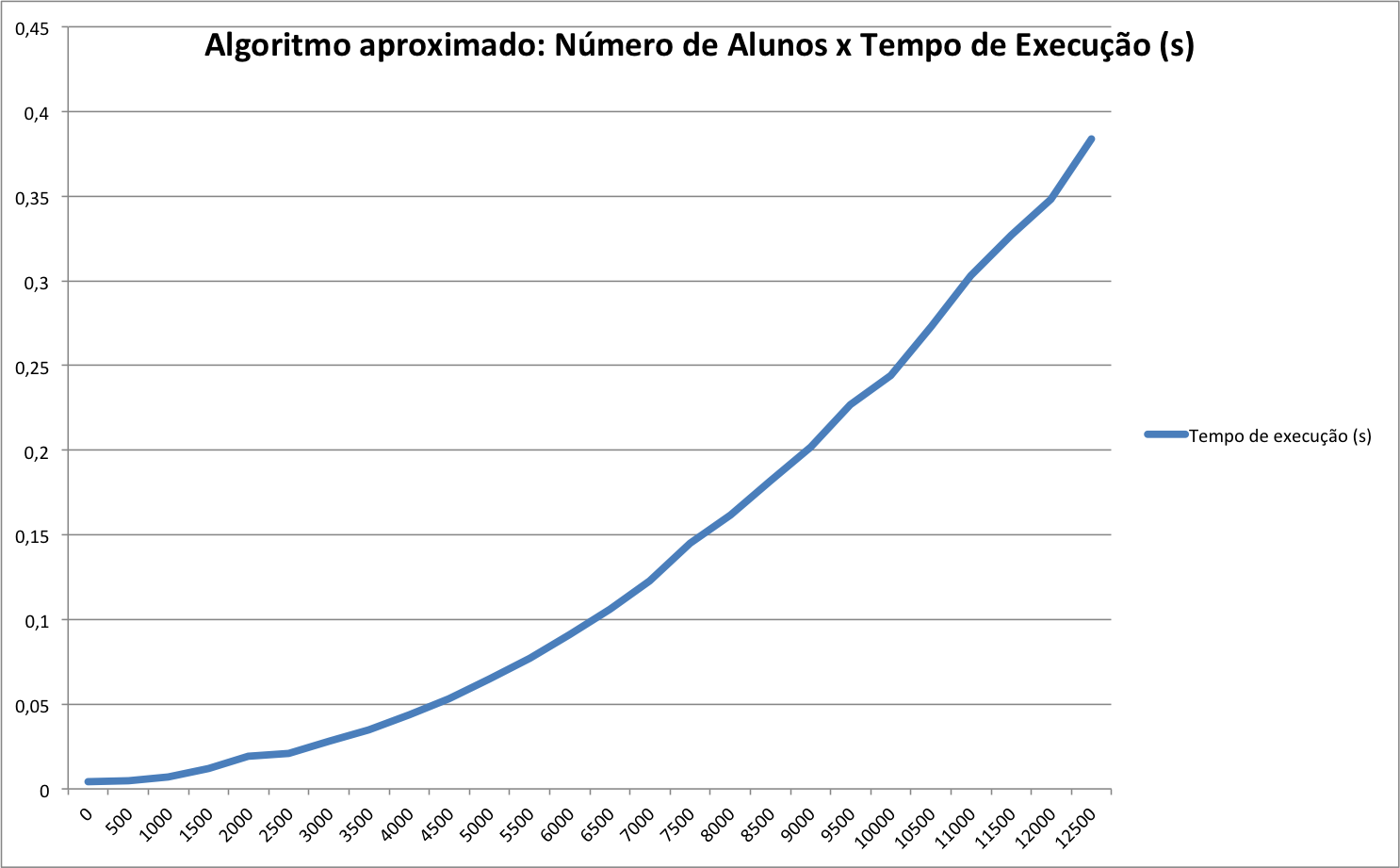
Na imagem abaixo é possível ver um experimento do algoritmo exato onde o número de alunos é aumentado.



No eixo X está o número de alunos e no eixo Y o tempo de execução em segundos. A linha vermelha é o tempo de execução do programa. Já a linha verde é a função f(x)=(x\*2x)/15000000 e foi adicionada para mostrar que o tempo de execução do programa realmente possui tempo de execução exponencial.

No gráfico abaixo é possível ver o tempo de execução do algoritmo aproximada quando variamos o número de alunos e deixamos o número de opções de respostas fixo, em seguida está o gráfico do tempo de execução quando variamos o número de opções e fixamos o número de alunos.





É possível constatar que a complexidade de tempo desse algoritmo é realmente O(A2\*R), já que o gráfico do tempo de execução de quando se varia o número de alunos torna-se semelhante a um gráfico de uma função quadrática e quando obtemos o gráfico do tempo de execução de quando se varia o número de opções de respostas obtemos um gráfico de uma função de parece ser linear.

De fato, a função f(x)=10\*x2 irá crescer muito mais rápido do que a função f(z)=102\*z, como pode ser visto no gráfico abaixo.

**5. CONCLUSÕES**

Neste trabalho, foi descrita a implementação de uma solução exata e outra solução aproximada de um problema do tipo NP-Completo. É importante criar soluções heurísticas para problemas do tipo NP-Completo, já que o tempo de execução da solução exata torna-se inviável a partir de um certo número de entradas. Mesmo que a solução heurística não retorne valor exatos, ela pode ser usada, já que, caso ela seja feita de uma maneira inteligente, a margem de erro do resultado será pequena.

Após a conclusão do trabalho, foi constatado que o problema poderia ser modelado como um Conjunto Independente. No entanto, optei por não modificar o código, uma vez que a solução para encontrar um Conjunto Independente é semelhante à solução feita por mim.

**6. REFERÊNCIAS BIBLIOGRÁFICAS**

1 Livro Projeto de Algoritmos, Nívio Ziviani, 3ª edição.