



Projeto e Análise de Algoritmo

Marcos Paulo Ferreira Rodrigues - 14.2.4341
Thiago Gonçalves Resende - 13.2.4667
Victor Lott Galvão de Oliveira - 15.1.4240



Sumário

- Introdução
- Gerador de Instância
- Problemas
 - Conjunto Independente
 - Problema
 - Solução
 - Clique
 - Satisfabilidade
- Principais dificuldades
- Perguntas



Introdução

- Trabalho prático desenvolvido em C++;
- Bibliotecas utilizadas
 - Algorithm
 - Math
- Header das principais funções:
 - `inline unsigned LimiteInferior()`
 - `inline unsigned LimiteSuperior()`
 - `unsigned NumeroExploraveis(vector<bool>& exploraveis)`
 - `vector<int> Vizinhos(vector<bool>& exploraveis, unsigned v)`
 - `Solucao BranchAndBound(Solucao& best, Solucao& s, vector<bool>& exploraveis, unsigned& liminf, unsigned limsup)`
 - `Solucao BnB()`



Variáveis

```
typedef vector<vector<int> > Grafo;    // Matriz de adjacencias de um grafo
typedef vector<vector<int> > Formula;  // Matriz de clausulas e variaveis

typedef vector<int> Solucao;           // Solucao para os 3 problemas
// Para o clique, a solucao eh uma
// lista contendo o indice dos vertices
// que fazem parte do clique
// Para o conjunto independente, idem
// Para a satisfabilidade, a solucao eh
// uma lista das variaveis e o valor
// 1 ou 0 indica atribuicao V ou F na
// formula
```



Gerador de instâncias

```
// Instancias de grafo
const int n_tamanhos = 4; // 4 * 4 = 16 tamanhos e densidades diferentes de grafo
int tamanhos[n_tamanhos] = {10, 20, 30, 40}; // Numero de vertices das instancias
float densidades[n_tamanhos] = {0.2, 0.4, 0.6, 0.8}; // Densidades das instancias

string dir = "instancias_grafos/";

for(int i = 0; i < n_tamanhos; i++) {
    for(int j = 0; j < n_tamanhos; j++) {
        Grafo g = CriaGrafo(tamanhos[i], densidades[j]);
        |
        string nomearq = dir + "grafo_" + to_string(tamanhos[i])
        | | | | | | | | + "_" + to_string(densidades[j])[2] + ".txt";
        SalvaGrafo(g, nomearq.c_str());
    }
}
```

Funções auxiliares

```
uniform_real_distribution<float> distrib(0.0, 1.0);  
// Cria um grafo aleatorio com n numero de vertices  
// A chance de ter uma aresta entre dois vertices eh determinada pela probabilidade p (entre 0 e 1)  
Grafo CriaGrafo(unsigned n, float p) {  
    Grafo g(n, vector<int>(n, 0));  
  
    for(unsigned i = 0; i < n - 1; i++) {  
        for(unsigned j = i + 1; j < n; j++) {  
            // Se o numero aleatorio for menor que p, cria adjacencia entre vertices  
            float chance = distrib(gerador);  
            if(chance < p) {  
                g[i][j] = g[j][i] = 1;  
            }  
        }  
    }  
  
    return g;  
}
```



Gerador de instâncias

```
// Instancias de formula
const int n_clausulas = 4; // 4 numero de clausulas diferentes
const int n_variaveis = 4; // 4 numero de variaveis diferentes

int clausulas[n_clausulas] = {2, 4, 6, 8};
int variaveis[n_variaveis] = {3, 5, 7, 9};
float probabilidade = 0.5; // Probabilidade unica

dir = "instancias_formulas/";

for(int i = 0; i < n_clausulas; i++) {
    for(int j = 0; j < n_variaveis; j++) {
        Formula f = CriaFormula(clausulas[i], variaveis[j], probabilidade);

        string nomearq = dir + "formula_" + to_string(clausulas[i])
            + "_" + to_string(variaveis[j]) + ".txt";
        SalvaFormula(f, nomearq.c_str());
    }
}
```

Funções auxiliares

```
// Cria uma formula booleana com c clausulas e v variaveis,  
// Com a chance p de uma variavel participar de cada clausula,  
// Com 50% de chance de ter valor negado se participar  
Formula CriaFormula(unsigned c, unsigned v, float p) {  
    Formula f(c, vector<int>(v, 2));  
  
    for(unsigned i = 0; i < c; i++) {  
        for(unsigned j = 0; j < v; j++) {  
            // Se o numero aleatorio for menor que p, a variavel j participa da clausula i  
            // Se o numero for menor que p/2, a variavel tera valor negado  
            float chance = distrib(gerador);  
            if(chance < p / 2)  
                f[i][j] = 0;  
            else if(chance < p)  
                f[i][j] = 1;  
        }  
    }  
    return f;  
}
```




Conjunto Independente

- Problema
 - Dado um grafo, encontrar o maior número de vértices independentes, isto é, não existe aresta entre nenhum par deles.
- Solução
 - Carrega o Grafo
 - Executa o Branch and Bound
 - Imprime a Solução



Principais funções

```
// Funcao preparatoria para o BranchAndBound
Solucao BnB() {
    unsigned liminf = LimiteInferior();
    unsigned limsup = LimiteSuperior();

    Solucao best; // Melhor solucao encontrada
    Solucao s; // Solucao atual

    // Vertices que podem participar do conjunto independente atual
    vector<bool> exploraveis(g.size(), true);
    return BranchAndBound(best, s, exploraveis, liminf, limsup);
}
```



Principais funções

```
// Limite inferior, 0 caso seja um grafo vazio, 1 caso contrario
```

```
inline unsigned LimiteInferior() {  
    return (g.size() > 0 );  
}
```

```
// Tamanho maximo possivel de um conjunto independente em um grafo
```

```
// eh dado pela formula a seguir, vide https://www8.cs.umu.se/kurser/5DA001/HT08/lab2.pdf
```

```
inline unsigned LimiteSuperior() {  
    unsigned n = g.size();  
    unsigned m = NumeroArestas(g);  
    return floor((1 + (sqrt(1 - 8 * m - 4 * n + 4 * n * n))) / 2.0);  
}
```

Funções auxiliares

```
// Computa numero de arestas do grafo
unsigned NumeroArestas(Grafo& g) {
    unsigned n = 0;

    for(unsigned i = 0; i < g.size() - 1; i++)
        for(unsigned j = i + 1; j < g.size(); j++)
            n += g[i][j];

    return n;
}

// Gera complemento h do grafo g
Grafo Complemento(Grafo& g) {
    Grafo h(g);

    for(unsigned i = 0; i < h.size() - 1; i++)
        for(unsigned j = i + 1; j < h.size(); j++)
            h[i][j] = h[j][i] = !h[i][j];

    return h;
}
```



Principais funções

```
// Quantidade de vertices que podem ser ainda explorados
unsigned NumeroExploraveis(vector<bool>& exploraveis) {
    unsigned e = 0;
    for(unsigned i = 0; i < exploraveis.size(); i++) {
        e += exploraveis[i];
    }
    return e;
}

// Indice dos vertices vizinhos do vertice atual que ja nao foram considerados
vector<int> Vizinhos(vector<bool>& exploraveis, unsigned v) {
    vector<int> vizinhos;

    for(unsigned j = 0; j < g.size(); j++)
        if(g[v][j] && exploraveis[j])
            vizinhos.push_back(j);

    return vizinhos;
}
```

Branch and Bound

```
// BranchAndBound, algoritmo que percorre a arvore de solucoes
// de forma a percorrer apenas ramos promissores para melhorar
// a solucao corrente
Solucao BranchAndBound(Solucao& best, Solucao& s, vector<bool>& exploraveis,
    unsigned& liminf, unsigned limsup) {

    if(s.size() == limsup) return s; // 1: Solucao confirmadamente otima encontrada
    // 2: Quantidade de vertices que faltam explorar nao sao suficientes para melhorar solucao
    if(s.size() + NumeroExploraveis(exploraveis) <= liminf) return s;

    // Verifica se encontrou nova melhor solucao (novo limite inferior)
    if(s.size() > liminf) {
        liminf = s.size();
        best = s;
        return s;
    }

    // 3: Conjunto independente maximal atingido
    if(find(exploraveis.begin(), exploraveis.end(), true) == exploraveis.end()) return s;
```



Branch and Bound

```
// Percorrendo os vertices nao visitados. Adiciona na solucao atual o próximo
// Vertice que puder participar e marca os seus vizinhos como nao exploraveis
for(unsigned i = 0; i < exploraveis.size(); i++) {
    if(exploraveis[i]) {
        s.push_back(i);
        vector<int> vizinhos = Vizinhos(exploraveis, i);

        for(unsigned j = 0; j < vizinhos.size(); j++) exploraveis[vizinhos[j]] = false;
        exploraveis[i] = false;

        // Explora esse ramo da arvore de decisoes
        BranchAndBound(best, s, exploraveis, liminf, limsup);

        s.pop_back(); // Desfaz a acao deste ramo

        for(unsigned j = 0; j < vizinhos.size(); j++) exploraveis[vizinhos[j]] = true;
        exploraveis[i] = true;
    }
}
// Ja percorreu todos os ramos da arvore
return best;
```


Conjunto Independente

INSTANCIAS (vértice_prob)	TEMPO (em segundos)	MELHOR SOLUÇÃO	VÉRTICES
grafo_10_2.txt	0.000180206	6	0 2 3 7 9 1
grafo_10_4.txt	3.6397e-05	4	0 3 1 9
grafo_10_6.txt	3.7534e-05	4	0 3 6 8
grafo_10_8.txt	1.1418e-05	2	0 1
grafo_20_2.txt	1.16709	9	0 3 4 5 6 7 10 15 19
grafo_20_4.txt	0.0037577	7	3 4 6 7 14 16 18
grafo_20_6.txt	0.000136365	3	1 3 5
grafo_20_8.txt	8.9785e-05	3	0 10 8

Conjunto Independente

INSTANCIAS (vértice_prob)	TEMPO (em segundos)	MELHOR SOLUÇÃO	VÉRTICES
grafo_30_2.txt	190.296	12	1 2 4 6 11 12 13 15 19 24 26 28
grafo_30_4.txt	0.0520036	7	1 4 8 10 11 13 26
grafo_30_6.txt	0.00121473	5	1 2 3 19 20
grafo_30_8.txt	0.000136752	4	2 9 23 27
grafo_40_2.txt	0	0	0
grafo_40_4.txt	0	0	0
grafo_40_6.txt	0	0	0
grafo_40_8.txt	0	0	0

[illegible]



Clique

- Problema:
 - Dado um grafo, encontrar o conjunto máximo de vértices tal que todas as possíveis arestas entre eles estejam presentes (subgrafo completo)
- Solução:
 - Carregar o Grafo
 - Obter o Complemento deste Grafo a partir da função Complemento(Grafo & g)
 - Executar o Branch and Bound
 - Imprimir a melhor solução encontrada



Clique

INSTANCIAS (vértice_prob)	TEMPO (em segundos)	MELHOR SOLUÇÃO	VÉRTICES
grafo_10_2.txt	9.712e-06	3	2 4 5
grafo_10_4.txt	1.8599e-05	3	0 6 4
grafo_10_6.txt	2.4438e-05	4	0 4 9 1
grafo_10_8.txt	0.000254146	5	0 3 4 5 2
grafo_22_2.txt	4.5743e-05	3	1 6 9
grafo_20_4.txt	0.000181581	5	7 10 11 12 19
grafo_20_6.txt	0.00538006	8	0 2 4 6 9 12 19 1
grafo_20_8.txt	0.428168	9	0 3 1 5 11 14 15 17 18



Clique

INSTANCIAS (vértice_prob)	TEMPO (em segundos)	MELHOR SOLUÇÃO	VÉRTICES
grafo_30_2.txt	0.000140754	3	0 25 1
grafo_30_4.txt	0.00098644	5	1 6 12 16 17
grafo_30_6.txt	0.242446	9	0 2 8 10 12 18 25 27 6
grafo_30_8.txt	1525.45	13	1 5 6 7 8 10 11 17 20 22 25 26 29
grafo_42_2.txt	0.000360072	4	0 23 32 25
grafo_40_4.txt	0.00278079	6	0 4 22 23 27 32
grafo_40_6.txt	0.308666	8	0 7 1 12 19 27 37 38
grafo_40_8.txt	0	0	0



Satisfabilidade

- Problema
 - Dado uma fórmula booleana na forma normal conjuntiva, encontrar uma atribuição de valores-verdade às variáveis da fórmula que a torne verdadeira, ou informe que não existe tal atribuição.
- Solução
 - Carrega a Fórmula
 - Reduz a fórmula em um grafo
 - Cada cláusula da fórmula é um grafo completo, onde cada variável é um vértice.
 - Conecta os vértices com as suas negações dos demais grafos).
 - Calcula o Complemento do grafo.
 - Executa o Branch and Bound
 - Verifica se a quantidade de valores na solução é igual ao número de cláusulas do problema
 - SATISFAZÍVEL: se true;
 - NÃO SATISFAZÍVEL: se false;

Redução de Fórmula para Grafo

```
// Reduz uma fórmula para um grafo
Grafo ReducaoSatGrafo(Formula& f) {
    vector<int> indice_i; // Indice i da variavel na matriz f
    vector<int> indice_j; // Indice j da variavel na matriz f
    vector<int> sinal;    // Se variavel esta sem not ou com not

    unsigned lin = f.size();
    unsigned col = f[0].size();

    // Anota as variaveis que vao virar vertices
    for(unsigned i = 0; i < lin; i++) {
        for(unsigned j = 0; j < col; j++) {
            if(f[i][j] != 2) {
                indice_i.push_back(i);
                indice_j.push_back(j);
                sinal.push_back(f[i][j] == 1);
            }
        }
    }
}
```

Redução de Fórmula para Grafo

```
// Numero de vertices do grafo
unsigned quant = indice_i.size();

// Cria um grafo sem arestas
Grafo g;
if(quant > 0) { // Para evitar criar um grafo nulo
    g.resize(quant, vector<int>(quant, 0));
}

// Conecta todos as variaveis da mesma clausula
for(unsigned i = 0; i < quant - 1; i++) {
    for(unsigned j = i + 1; j < quant; j++) {
        if(indice_i[i] == indice_i[j]) {
            g[i][j] = g[j][i] = 1;
        }
    }
}
```


Redução de Fórmula para Grafo

```
// Conecta todas as variaveis com sinal diferente em clausulas diferentes
for(unsigned i = 0; i < quant - 1; i++) {
    for(unsigned j = i + 1; j < quant; j++) {
        if(indice_j[i] == indice_j[j] && sinal[i] != sinal[j]) {
            g[i][j] = g[j][i] = 1;
        }
    }
}

return g;
}
```



Satisfabilidade

INSTANCIAS (vértice_prob)	TEMPO (em segundos)	SATISFAZ	VÉRTICES
formula_2_3.txt	8.465e-06	SIM	0 1
formula_2_5.txt	8.153e-06	SIM	0 1
formula_2_7.txt	4.9199e-05	NÃO	3 4 5 6
formula_2_9.txt	1.2806e-05	NÃO	0 2 1
formula_4_3.txt	5.911e-06	NÃO	0 1
formula_4_5.txt	1.5239e-05	SIM	3 4 6 5
formula_4_7.txt	2.2063e-05	NÃO	3 4 5 7 6
formula_4_9.txt	3.0058e-05	NÃO	4 5 6 7 8

Satisfabilidade

INSTANCIAS (vértice_prob)	TEMPO (em segundos)	SATISFAZ	VÉRTICES
formula_6_3.txt	1.4249e-05	NÃO	0 1
formula_6_5.txt	2.7158e-05	NÃO	0 2 1
formula_6_7.txt	6.1052e-05	NÃO	3 4 5 7 6
formula_6_9.txt	9.7938e-05	NÃO	6 7 8 9 11 12 10
formula_8_3.txt	2.683e-05	NÃO	7 8 9
formula_8_5.txt	6.1032e-05	NÃO	0 2 3 1
formula_8_7.txt	4.6848e-05	NÃO	0 2 3 1
formula_8_9.txt	0.00019888	NÃO	27 28 29 30 31 32 33



Principais dificuldades

- Gerador de Instâncias
 - Criar Fórmula
 - Criar Grafo
- Converter a fórmula em grafo.
- Conectar as variáveis que são negação na fórmula



Dúvidas?