Trabalho Prático 02 - Projeto e Análise de Algoritmos

Pedro Henrique Reis Rodrigues

Novembro 2021

1 Introdução

Para este trabalho prático foi proposto a pesquisa e interpretação dos métodos e estratégias no desenvolvimento de algoritmos, nos quais são `Branch and Bound` e `BackTracking`, na qual será discorrido ao longo do relatório seus conceitos, implementações, diferenças e semelhanças com outros algoritmos usados no desenvolvimento de softwares, e por fim o código implementado do Problema do `Cavalo`, representando a técnica de Backtracking, e o Problema da `Mochila` para representar a técnica de Branch And Bound.

2 Branch and Bound

2.1 Conceito

Branch and bound é um paradigma de concepção de algoritmos para problemas de optimização discreta e combinatória, bem como de optimização matemática. Um algoritmo de branch-and-bound consiste numa enumeração sistemática de soluções candidatas. Ou seja, o conjunto de soluções candidatas é pensado como formando uma árvore enraizada com o conjunto completo na raiz. O algoritmo explora os ramos desta árvore, que representam os subconjuntos do conjunto de soluções. Antes de enumerar as soluções candidatas de um ramo, o ramo é verificado em relação aos limites superiores e inferiores estimados da solução óptima e é descartado se não conseguir produzir uma solução melhor do que a melhor encontrada até agora pelo algoritmo.

Branch and bound é um paradigma de concepção de algoritmos que é geralmente utilizado para resolver problemas de optimização combinatória. Estes problemas são tipicamente exponenciais em termos de complexidade temporal e podem exigir a exploração de todas as permutações possíveis na pior das hipóteses. A técnica Branch and Bound Algorithm resolve estes problemas de forma relativamente rápida.

Neste relatório iremos ver a abordagem de Branch and Bound para resolver o `Problema de Mochila`: A solução utilizando Backtracking pode ser optimizada se conhecermos uma sub-árvore da melhor solução possível, enraizada em

cada nó. Se o melhor na sub-árvore for pior que o melhor actual, podemos simplesmente ignorar este nó e as suas sub-árvores. Assim, calculamos o limite (melhor solução) para cada nó e comparamos o limite com a melhor solução actual antes de explorar o nó.

Branch and bound é uma técnica muito útil para procurar uma solução, mas na pior das hipóteses, precisamos de calcular completamente a árvore inteira. Na melhor das hipóteses, só precisamos de calcular completamente um caminho através da árvore e podar o resto da árvore

2.2 Semelhanças e Diferenças

Algumas semelhanças e diferenças que podemos observar nesse neste paradigma de concepção que é o Branch-and-bounch, podemos ressaltar as semelhanças que são algoritmos que se assemelham bastante com a programação dinâmica, e divisão e conquista, uma vez que recorrem sempre na resolução parcial dos problemas, que se assemelha bastante ao que foi visto em Divisão e Conquista, e também é um problema que sempre acaba trabalhando com resultados passados, que também se qualifica como semelhança, de forma superficial, com os algoritmos que vimos utilizando Programação Dinâmica.

Agora em termos de eficiência e processo dos algoritmos, para casos de retrocedência, sem dúvidas que o Branch and Bound é mais eficiente que a programação dinâmica e algoritmos gulosos, e em alguns casos até mais eficiente que a Divisão e Conquista, porém são casos muito específicos para poder analisar essas diferenças, é necessário um problema complexo que seja possível resolver utilizando as 4 metodologias para realizar fortemente a comparação de complexidade e eficiência.

2.3 Implementação

O problema que iremos utilizar o Branch and Bound para resolver é um problema comum na área da programação dinâmica, mais conhecido como o Problema da Mochila, na qual temos um peso de mochila, e uma sequência de itens que possuem quantidade e peso, e o Problema é simples, quantos e quais itens devemos alocar na mochila para pegar um peso máximo com maior valor. Este código foi realizado utilizando o método de Branch and Bound, na qual faz retrocesso para saber a melhor decisão de alocação da mochila, feito em C++.

2.3.1 Código Fonte

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 // Aluno: Pedro Henrique Reis Rodrigues
5 // Professor: Silvio Guimar es
6 // Mat ria: PAA (Projeto e An lise de Algoritmos)
7 // Problema da Mochila, utilizando Branch and Bound
```

```
8
9  typedef struct Item
10 {
11     float peso;
12     int quantidade;
13 } Item;
14
15 typedef struct No
16 {
17     int nivel;
18     int max;
19     int bound;
20     float peso;
21 } No;
22
23 bool cmp(Item a, Item b)
24 {
25     double r1 = (double)a.quantidade / a.peso;
26     double r2 = (double)b.quantidade / b.peso;
27     return r1 > r2;
28 }
29
30 int bound(No u, int items, int peso, Item K[])
31 {
32     if (u.peso >= peso)
33     { // se ultrapassar o proprio peso, retorna 0, n vai entrar
34         return 0;
35     }
36     int lucro = u.max;    // pega o max q aguenta, que seria o lucro
37     int j = u.nivel + 1; // come a o index no atual + 1
38     int peso_aux = u.peso;
39
40     while ((j < items) && (peso_aux + K[j].peso <= peso))
41     { // condicoes de peso e capacidade
42         peso_aux += K[j].peso;
43         lucro += K[j].quantidade;
44         j++;
45     }
46     if (j < items)
47     { // se k for diferente de n, ent o inclua o ultimo item
48         lucro += (peso - peso_aux) * K[j].quantidade / K[j].peso;
49     }
50     return lucro;
51 }
52 // retorna a capacidade maxima de itens K, q suprota a mochila com capacidade "p
53 int mochila(int peso, Item K[], int items)
```

```
54 {
55     queue<No> Fila;
56     No u, v;
57     u.nivel = -1;
58     u.max = 0; // inicializa   o das auxiliares da arvore
59     u.peso = 0;
60
61     Fila.push(u); // adiciona na fila
62     int max = 0;
63
64     while (!Fila.empty())
65     {
66         // tira um no
67         u = Fila.front();
68         Fila.pop();
69         if (u.nivel == -1)
70         { // no inicial = 0
71             v.nivel = 0;
72         }
73         if (u.nivel == items - 1)
74         { // se n tiver next no = continua
75             continue;
76         }
77         v.nivel = u.nivel + 1;
78         v.peso = u.peso + K[v.nivel].peso; // adiciona no n   v, todos os itens
79         v.max = u.max + K[v.nivel].quantidade;
80
81         if (v.peso <= peso && v.max > max) // se o peso acumlado for menor q o p
82         {
83             max = v.max;
84         }
85         v.bound = bound(v, items, peso, K); // recorrencia upper bound para veri
86
87         if (v.bound > max)
88         {
89             Fila.push(v);
90         }
91         v.peso = u.peso;
92         v.max = v.max;                          // repete a ao so q sem tirar o iter
93         v.bound = bound(v, items, peso, K); // recorrencia upper bound para veri
94
95         if (v.bound > max)
96         {
97             Fila.push(v);
98         }
99     }
```

```
100     return max;
101 }
102
103 int main()
104 {
105
106     int peso; // peso da mochila
107     scanf("%d", &peso);
108     int items;
109     scanf("%d", &items); // numero de itens
110     Item K[items];
111     for (int i = 0; i < items; i++)
112     {
113         scanf("%d %f", &K[i].quantidade, &K[i].peso); // pega quantidade e peso
114     }
115
116     sort(K, K + items, cmp);
117
118     int resultado = mochila(peso, K, items);
119     printf("Resultado ->%d<", resultado);
120 }
```

O código fonte enviado juntamente com este documento está devidamente comentado nas partes importantes do código. Os problemas de optimização combinatória são na sua maioria exponenciais em termos de complexidade temporal, portanto `O(2^n)`.

3 Backtracking

3.1 Conceito

O Backtracking é uma técnica algorítmica para resolver problemas de forma recorrente, tentando construir uma solução de forma incremental, uma peça de cada vez, removendo as soluções que não satisfazem as necessidades do problema em qualquer altura, como por exemplo, em uma árvore de pesquisa, o caso base poderia ser o tempo decorrido até chegar certo nível dessa árvore.

O Backtracking é um algoritmo geral para encontrar todas as soluções para alguns problemas computacionais, nomeadamente problemas de satisfação de constrangimento, que constrói gradualmente possíveis candidatos às soluções e abandona um candidato assim que determina que o candidato não pode ser completado para finalmente se tornar uma solução válida.

Neste artigo iremos ver a abordagem de Backtracking para resolver o Problema do `Cavalo`. Será usado o Backtracking uma vez que ele é perfeito para essa situação de em que é tentado diversas soluções diferentes até encontrar uma solução que funcione. Estes problemas só podem ser resolvidos tentando todas as configurações possíveis e cada configuração é tentada apenas uma vez. Uma solução ingénua para estes problemas é tentar todas as configurações e produzir

uma configuração que siga dadas as limitações do problema.

O Backtracking funciona de forma incremental e é uma optimização sobre a solução `Naive` onde todas as configurações possíveis são geradas e tentadas.

3.2 Semelhanças e Diferenças

Semelhantemente ao que foi discutido a respeito das semelhanças e diferenças do Branch and Bound em relação aos algoritmos gulosos, programação dinâmica e divisão conquista, o Backtracking não se difere bastante neste caso.

Algumas semelhanças e diferenças que podemos observar nesse neste paradigma de concepção que é o Backtracking, podemos ressaltar as semelhanças que são algoritmos que se assemelham bastante com a programação dinâmica, e divisão e conquista, uma vez que recorrem sempre na resolução parcial dos problemas, que se assemelha bastante ao que foi visto em Divisão e Conquista, e também é um problema que sempre acaba trabalhando com resultados passados, que também se qualifica como semelhança, de forma superficial, com os algoritmos que vimos utilizando Programação Dinâmica.

Agora em termos de eficiência e processo dos algoritmos, há casos em que só é possível resolver certo algoritmo utilizando Backtracking. Em questão de eficiência, teóricamente é fato afirmar que em maioria utilizar backtracking será mais eficiente, porém não é todo caso. É necessário um caso muito específico para poder analisar essas diferenças de desempenho, é necessário um problema complexo que seja possível resolver utilizando as 4 metodologias para realizar fortemente a comparação de complexidade e eficiência.

3.3 Implementação

O problema que iremos utilizar o Backtracking para resolver é um problema comum na área da programação dinâmica, mais conhecido como o Problema da Cavalo, na qual temos tabuleiro NxN, e uma posição inicial do cavalo, que deve se movimentar nas regras do xadrez (em L), e assim deve procurar andar em todas as casas exatamente uma vez, ou seja, sem repetir uma casa. Este código foi realizado utilizando o método de Backtracking, na qual faz retrocesso para saber a melhor decisão de alocação da mochila, feito em C++.

3.3.1 Código Fonte

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 // Aluno: Pedro Henrique Reis Rodrigues
5 // Professor: Silvio Guimar es
6 // Mat ria: PAA (Projeto e An lise de Algoritmos)
7 // Problema do Cavalo, utilizando Backtracking
8
9 int tab[8][8];
```

```
10
11 int cavalo(int x, int y, int atual, int tam, int movesX[], int movesY[])
12 {
13     int prox_x, prox_y;
14     if (atual == tam * tam)
15     {
16         return 1; // caso base
17     }
18     // tenta todas as posicoes do tabuleiro, verificando se h  possibilidade (p
19     for (int i = 0; i < 8; i++)
20     {
21         prox_x = x + movesX[i];
22         prox_y = y + movesY[i];
23         if (prox_x >= 0 && prox_x < tam && prox_y >= 0 && y < tam && tab[prox_x]
24         { // verificando se a posicao do tabuleiro eh valida
25             tab[prox_x][prox_y] = atual;
26             if (cavalo(prox_x, prox_y, atual + 1, tam, movesX, movesY) == 1)
27             {
28                 return 1;
29             }
30             else
31             {
32                 tab[prox_x][prox_y] = -1; // backtracking
33             }
34         }
35     }
36 }
37
38 int main()
39 {
40
41     int tam = 8; // tamanho do tabuleiro
42
43     int movesX[8] = {2, 1, -1, -2, -2, -1, 1, 2};
44     int movesY[8] = {1, 2, 2, 1, -1, -2, -2, -1};
45
46     tab[0][0] = 0;
47
48     if (cavalo(0, 0, 1, tam, movesX, movesY) == 0) // o 0, 0 ali decide qual a p
49     {
50         printf("Nao existe solucao");
51     }
52     else
53     {
54         for (int i = 0; i < tam; i++)
55         {
```

```
56              for (int j = 0; j < tam; j++)
57              {
58                  printf("%d ", tab[i][j]);
59              }
60              printf("\n");
61          }
62      }
63
64      return 0;
65  }
```

O código fonte enviado juntamente com este documento está devidamente comentado nas partes importantes do código. A complexidade do algoritmo varia, entre `O(N^2)` para casos auxiliares, ou em um pior caso `O(8^(n^2))` uma vez que temos um máximo de 8 possiveis movimentos para realizar, em um espaço de tabuleiro `N^2`.

4 Conclusão

Através desse trabalho, foi possível fortemente concluir e observar as diferenças e semelhanças dentre os algorítmos de Branch and Bound e Backtracking, tendo os dois como principio o paradigma de retrocesso para resolução dos problemas, foi possível observar também a importância desses algoritmos para resolução desses problemas que foram citados, dentre outros milhares de problemas na computação que é necessário a utilização desses para um melhor desempenho.

Referências

[1] Bader, David A.; Hart, William E.; Phillips, Cynthia A. (2004). "Parallel Algorithm Design for Branch and Bound"(PDF). In Greenberg, H. J. (ed.). Tutorials on Emerging Methodologies and Applications in Operations Research. Kluwer Academic Press. Archived from the original (PDF) on 2017-08-13. Retrieved 2015-09-16.

[2] Little, John D. C.; Murty, Katta G.; Sweeney, Dura W.; Karel, Caroline (1963). "An algorithm for the traveling salesman problem"(PDF). Operations Research. 11 (6): 972–989. doi:10.1287/opre.11.6.972. hdl:1721.1/468

[3] Clausen, Jens (1999). Branch and Bound Algorithms—Principles and Examples (PDF) (Technical report). University of Copenhagen. Archived from the original (PDF) on 2015-09-23. Retrieved 2014-08-13.

[4] http://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf.

[5] http://www.cis.upenn.edu/~matuszek/cit594-2009/Lectures/35-backtracking.ppt.