Projeto e Análise de Algoritmos

2º Trabalho de Implementação

PUC-Rio
Departamento de Informática
Pós Graduação em Informática

Professor
Marcus Poggi

Grupo
Daniela Pacheco
Marcelo Arza
Marcelo Garnier
Rafael Diniz
Willian Oizumi

Turma
2013.1

Data
09/07/2013

Sumário

h

1. Introdução

Este 2º trabalho de implementação de Projeto e Análise de Algoritmos trata do tema de problemas de otimização NP-difícil, cuja versão de decisão é um problema NP-completo.

Um problema é NP-difícil se sua versão NP-completa pode ser resolvida em tempo polinomial por uma Máquina de Turing Não Determinística

Na teoria da complexidade computacional, a classe de complexidade NP-completo é o subconjunto dos problemas de decisão em NP de tal modo que todo problema em NP se pode reduzir, com uma redução de tempo polinomial, a um dos problemas NP-completo.

Pode-se dizer que os problemas NP-completo são os problemas mais difíceis de NP e muito provavelmente não formem parte da classe de complexidade P. A razão é que se se conseguisse encontrar uma maneira de resolver qualquer problema NP-completo rapidamente (em tempo polinomial) , então poderiam ser utilizados algoritmos para resolver todos problemas NP rapidamente.

Na prática, o conhecimento de NP-completude pode evitar que se desperdice tempo tentando encontrar um algoritmo de tempo polinomial para resolver um problema quando esse algoritmo não existe.

Problemas NP-difíceis podem ser de qualquer tipo: problemas de decisão, problemas de pesquisa ou problemas de otimização.

Uma vez que os problemas NP-completos transformam-se uns aos outros por redução um-para-muitos em tempo polinomial (também chamada de transformação polinomial), todos os problemas NP-completos podem ser resolvidos em tempo polinomial por uma redução para qualquer outro problema NP-completo. Assim, todos os problemas em NP reduzem para os problemas NP-completos; note-se, porém, que isso envolve a combinação de duas transformações diferentes: de problemas de decisão NP-completos para o problema NP-completo L por transformação polinomial, e de L para H pela redução em tempo polinomial de Turing;

Um erro comum é pensar que NP em NP-difícil representa não-polinomial. Embora seja amplamente suspeito de que não existem algoritmos de tempo polinomial para problemas NP-difíceis, isto nunca foi provado. Além disso, a classe NP contém também todos os problemas que podem ser resolvidos em tempo polinomial.

Atualmente todos os algoritmos conhecidos para problemas NP-completos utilizam tempo exponencial quanto ao tamanho da entrada. Desconhece-se a existência de melhores algoritmos para se resolver um problema NP-completo de tamanho arbitrário se utiliza de um dos seguintes enfoques:

Aproximação: Também chamado de Relaxação. Um algoritmo que rapidamente encontra uma solução não necessariamente ótima, contudo dentro de um certo intervalo de erro. Em alguns casos, encontrar uma boa aproximação é o suficiente para resolver o problema, porém nem todos os problemas NP-completos tem bons algoritmos de aproximação. Se uma Relaxação é uma boa aproximação do modelo original, então detecta infactibilidade rapidamente, obtém limitantes mais precisos, tem maior chance de fornecer a solução ótima e tem arredondamento mais fácil.

Probabilístico: Um algoritmo que pode obter em média uma boa solução para um problema apresentado de uma distribuição de dados de entrada.

Restrição: Restringindo a estrutura da entrada, algoritmos mais rápidos são possíveis.

Parametrização: Geralmente há algoritmos rápidos se certos parâmetros da entrada são fixos.

Heurísticas: Um algoritmo que trabalha razoavelmente bem em muitos casos, mas não há prova de que são sempre rápidos e que produzam sempre bons resultados.

2. Problema

O problema a ser tratado é o Problema da Árvore Geradora com Restrições de Capacidade (CAP-MST):

Dado um grafo $G = (V \cup \{r\}, E)$, custos c(i, j) >= 0 associados às arestas (i, j) em E, demandas q(v) associadas aos vértices v em V, e uma capacidade Q. Deseja-se encontrar uma árvore geradora de G, cujas sub-árvores enraizadas no vértice r possuam a soma das demandas dos seus vértices inferiores ou iguais à Q, cujo custo total de suas arestas seja mínimo.

A CAP-MST já foi resolvida utilizando heurísticas, métodos de corte e programação inteira. Abordagens heurísticas normalmente são mais rápidas que métodos de corte e de programação inteira e portanto são mais adequadas para resolver instâncias maiores, porém a qualidade das soluções tipicamente não são tão boas.

A proposta deste trabalho é utilizar uma Aproximação e, em seguida, Branch and bound, um algoritmo para encontrar soluções ótimas para vários problemas de otimização, especialmente em otimização combinatória. Consiste em uma enumeração sistemática de todos os candidatos a solução, através da qual grandes subconjuntos de candidatos infrutíferos são descartados em massa utilizando os limites superior e inferior da quantia otimizada.

Branch and bound é de longe a ferramenta mais utilizada para resolver problemas NP-difícil de otimização combinatória. No entanto, ele é um paradigma de algoritmo, que deve ser preenchido para cada tipo específico de problema e existe uma grande quantidade de opções para cada um de seus componentes.

3. Implementação

Para a implementação deste trabalho, utilizou-se a técnica de busca de solução chamada branch and bound [1]. Essa técnica utiliza soluções parciais que representam subproblemas. São eliminadas subsoluções que superem um limite superior (valor mínimo no caso da CAP-MST). Esse limite deve ser calculado de forma eficiente, ou seja, deve haver um algoritmo polinomial para calculá-lo. Além disso, durante a execução do algoritmo deve haver uma variável que armazene a melhor solução viável encontrada até o dado momento. Tal variável é retornada ao final da execução do algoritmo. A Figura 1 exibe um pseudocódigo genérico para o branch and bound.

```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
bestsofar = ∞
Repeat while S is nonempty:
  choose a subproblem (partial solution) P ∈ S and remove it from S
  expand it into smaller subproblems P_1, P_2, ..., P_k
  For each P_i:
    If P_i is a complete solution:  update bestsofar
    else if lowerbound(P_i) < bestsofar:  add P_i to S
return bestsofar
```

Figura 1. Pseudocódigo para o *branch and bound* (retirado de [1])

Para a implementação do branch and bound devem ser escolhidos alguns critérios e técnicas. Sendo assim, a listagem a seguir define os critérios e técnicas utilizados na implementação deste trabalho:

1. Critério de Particionamento do espaço de soluções: Conforme sugerido no enunciado do trabalho, o critério de particionamento do espaço de soluções escolhido foi a divisão em dois conjuntos: (i) árvore geradora contendo obrigatoriamente uma dada aresta α, e (ii) árvore geradora obrigatoriamente sem a aresta α.
2. Critério para percorrer subconjuntos do espaço de soluções: Seguindo a sugestão do enunciado, o método utilizado para percorrer os subconjuntos do espaço de soluções foi a busca em profundidade.
3. Relaxação Utilizada: A relaxação escolhida para o branch and bound foi a Árvore Geradora Mínima (AGM) sem restrições de capacidade. A diferença nesse caso é que a AGM deverá conter a aresta fixada no subconjunto de soluções selecionado.
4. Método para obter primeira solução viável: A primeira solução viável será gerada para se possuir uma primeira melhor solução conhecida antes de se iniciar o branch and bound. Neste trabalho, conforme as orientações do enunciado, para gerar tal solução será utilizado um método guloso onde cada vértice existente no grafo é conectado como uma subárvore no vértice raiz. Ou seja, a solução inicial possuirá n subárvores conectadas ao nó raiz.
5. Critério de seleção do particionamento: Para seleção das arestas no particionamento, será feita a ordenação crescente das arestas por custo. Dessa forma, as arestas de menor custo devem ser consideradas antes das arestas de maior custo.

3.1. Trecho de código relevantes

Nesta seção exibimos os trechos de código mais relevantes. Para gerar a primeira solução viável, utilizou-se um algoritmo simples que liga todos os vértices à raiz. Apesar desse algoritmo não gerar uma "boa" solução, ele encontra uma solução viável em tempo polinomial (Trecho 1).

```
int root_connect(struct Graph* graph) {

        struct Edge next_edge;

        struct Edge* viavel = (struct Edge*) malloc(graph->V * sizeof(struct Edge));

        int total = 0, i = 0, j = 0;

        while (j < graph->V - 1) {

                next_edge = graph->edge[i];

                if (next_edge.dest == graph->V - 1) {

                        viavel[j] = next_edge;

                        total += next_edge.weight;

                        j++;

                }

                i++;

        }

        delete viavel;

        return total;

}
```

Trecho 1: Algoritmo para a primeira solução viável

Para o branch and bound, os algoritmos de Find e Union (Trecho 2) do Kruskal foram reaproveitados. Conforme explicado, o branch and bound particiona o espaço de soluções por meio das arestas. Nesse contexto, os algoritmos Find e Union unem vértices com arestas que não formam ciclo (Trecho 3).

```
int find(struct subset subsets[], int i) {

        if (subsets[i].parent != i)

                subsets[i].parent = find(subsets, subsets[i].parent);

        return subsets[i].parent;

}
```

```
void Union(struct subset subsets[], int x, int y) {
        int xroot = find(subsets, x);
        int yroot = find(subsets, y);
        if (subsets[xroot].rank < subsets[yroot].rank) {
                subsets[xroot].parent = yroot;
        } else if (subsets[xroot].rank > subsets[yroot].rank) {
                subsets[yroot].parent = xroot;
        }
        else {
                subsets[yroot].parent = xroot;
                subsets[xroot].rank++;
        }
}
```

Trecho 2: Funções de Find e Union

```
while(subproblems.size() > 0 && totaltime.getCPUTotalSecs() < 3600) {
        totaltime.start();subproblem s = subproblems.top();    subproblems.pop();
int edge_index = 0; bool forma_ciclo = false;
for (int i = 0; i < graph->E; ++i) {
        if (s.mandatory[i] == 1) {
                int x = find(s.subsets, graph->edge[i].src);
                int y = find(s.subsets, graph->edge[i].dest);
                if (x != y) {
                        result[edge_index] = graph->edge[i];
                        edge_index++;
                        Union(s.subsets, x, y);
                } else
                        forma_ciclo = true; break;
        }
}
```

```
}
```

Trecho 3: Utilização do Find e Union para selecionar arestas

Em seguida, após a seleção de uma aresta, o branch and bound calcula uma nova AGM com a aresta selecionada "fixa", como obrigatória (mandatory) ou não permitida (disallowed). Se o custo da AGM for menor do que o upper bound (última solução viável encontrada), o algoritmo verifica se é uma solução viável. Por fim, são gerados sub-problemas, com base no problema atual (Trecho 4).

```
if (!forma_ciclo) {
        int total_cost = KruskalMST(graph, result, edge_index, &s, capacity);
        if (total_cost < upper_bound) {
                if (solucao_viavel(result)) {
                        upper_bound = total_cost;
                }
                subproblem filho1;
                memcpy(filho1.disallowed, s.disallowed, 8192);
                memcpy(filho1.mandatory, s.mandatory, 8192);
                memcpy(filho1.subsets, s.subsets, 81);
                filho1.mandatory[s.next_edge] = 1;
                s.disallowed[s.next_edge] = 1;
                s.next_edge = s.next_edge + 1;
                filho1.next_edge = s.next_edge + 1;
                subproblems.push(s);
                subproblems.push(filho1);
        }
}
```

Trecho 4: Geração de sub-problemas

Conforme explicado anteriormente, o problema utilizado para gerar um lower bound foi a árvore geradora mínima. Nossa implementação foi baseada no algoritmo de Kruskal (Trecho 5).

```
int KruskalMST(struct Graph* graph, struct Edge result[], int e, subproblem *s, int c) {
    int V = graph->V;
    int i = 0;  // váriável de indice para pegar arestas ordenadas
    while (e < V) {
        // Step 2: Pegar a menor aresta. E incrementar o indice
        struct Edge next_edge = graph->edge[i++];
            if (s->disallowed[i - 1] == 0) {
                int x = find(s->subsets, next_edge.src);
                int y = find(s->subsets, next_edge.dest);
                // Verifica se a aresta não gera ciclo e faz a união
                if (x != y) {
                        result[e++] = next_edge;
                        Union(s->subsets, x, y);
                }
                // se formar ciclo descarta a aresta
            }
    }
    // calcula a soma dos pesos das arestas
    int total = 0;
    for (i = 0; i < e; ++i) {
        total += result[i].weight;
    }
    return total;
}
```

Trecho 5: AGM com Kruskal

Para ser viável uma solução deve satisfazer os seguintes critérios: (i) ser uma árvore com origem na raiz da instância, (ii) respeitar as restrições de capacidade e (iii) não possuir vértices desconectados da árvore. Tal verificação é feita pelo método solucao_viavel (Trecho 6).

```
bool solucao_viavel(struct Graph* graph, struct Edge result[], int Cap) {
    int Raiz = graph->V - 1;
    stack<int> nos;
    int demanda = 0;
    int v = -1;
    for (int i = 0; i < graph->V; i++) {
            if (result[i].src == Raiz) {
                nos.push(result[i].dest);
                demanda++;
            }
    }
    while(!nos.empty()) {
            v = nos.top();
            nos.pop();
            demanda++;
            if (demanda > Cap) {
                return false;
            }
            eles {
                for(int j = 0; j < graph->V; j++) {
                        if (result[j].src == v) {
                            nos.push(result[j].dest);
                        }
                }
            }
    }
    return true;
}
```

Trecho 6: Verificação de solução viável

4. Resultados

Esta seção tem por objetivo apresentar os resultados obtidos na execução do algoritmo CAP-MST com as instâncias fornecidas. Ao todo, foram testadas 39 instâncias, com capacidades 3, 5 e 10, e quantidade de vértices 16, 40 e 80.

Tabela 1: Resultados Obtidos para todas as instâncias.

Instância	Capacidade	LB	UP	Solução Ótima
TC40-1.TXT	3	476	1607	742
TC40-1.TXT	5	476	1607	588
TC40-1.TXT	10	476	1607	498
TC40-2.TXT	3	460	1511	717
TC40-2.TXT	5	460	1511	583
TC40-2.TXT	10	460	1511	490
TC40-3.TXT	3	470	1480	716
TC40-3.TXT	5	470	1480	577
TC40-3.TXT	10	470	1480	500
TC40-4.TXT	3	480	1666	775
TC40-4.TXT	5	480	1666	617
TC40-4.TXT	10	480	1666	512
TC40-5.TXT	3	478	1531	741
TC40-5.TXT	5	478	1531	605
TC40-5.TXT	10	478	1531	504
TC80-1.TXT	3	830	3332	1572
TC80-1.TXT	5	830	3332	1120
TC80-1.TXT	10	830	3332	896
TE16.TXT	3	194	482	258
TE16.TXT	5	194	482	222
TE16.TXT	10	194	482	200
TE40-1.TXT	3	496	2915	1190
TE40-1.TXT	5	496	2915	835
TE40-1.TXT	10	496	2915	608
TE40-2.TXT	3	484	2677	1103
TE40-2.TXT	5	484	2677	794
TE40-2.TXT	10	484	2677	573
TE40-3.TXT	3	452	2758	1117
TE40-3.TXT	5	452	2758	797
TE40-3.TXT	10	452	2758	572
TE40-4.TXT	3	496	2747	1134
TE40-4.TXT	5	496	2747	815
TE40-4.TXT	10	496	2747	596
TE40-5.TXT	3	470	2711	1115
TE40-5.TXT	5	470	2711	797
TE40-5.TXT	10	470	2711	572
TE80-1.TXT	3	1142	9944	3287
TE80-1.TXT	5	1142	9944	2555
TE80-1.TXT	10	1142	9944	1672

Referências

[1] S. Dasgupta, C. H. Papadimitriou, U. V. Vazirani. “Algorithms”, 2006.

[2] L. Gouveia, A comparison of directed formulations for the capacitated minimal spanning tree problem, Telecommunication Systems vol 1, 1993, 51-76.