PROJETO E ANÁLISE DE ALGORITMOS II - SIMULATED ANNEALING

Caroline Castello Letizio*, Rafael Fonseca dos Santos†, Rafael Godoi Dias‡, Victor Matheus de Araujo Oliveira§

*RA059664 - Ciência da Computação 2006

†RA072146 - Ciência da Computação 2007

‡RA072149 - Ciência da Computação 2007

§RA072589 - Ciência da Computação 2007

Emails: caroline.letizio@gmail.com, fonsecasantos.rafael@gmail.com, gdrafael@gmail.com, victormatheus@gmail.com

Resumo— Nesse trabalho para a disciplina MC548 (ministrada pelo prof. Eduardo C. Xavier) mostramos uma heurística para encontrar uma solução para o problema da Árvore de Steiner em um grafo completo. Fizemos uso do algoritmo H [3] para a construção da solução inicial e de um algoritmo de vizinhança baseado em caminhos mínimos na meta-heurística *Simulated Annealing*.

Palavras-chave— Steiner Tree, Simulated Annealing, Heuristic

1 Descrição do problema

Seja $G = (V, E, c)$, um grafo ponderado não dirigido, onde $V = \{v_1, v_2, ..., v_n\}$ é o conjunto de vértices de G, $E \subseteq \{\{v_i, v_j\} \mid v_i \in V, v_j \in V \, e \, v_i \neq v_j\}$[1] é o conjunto de arestas de G e $c : E \to R$ é a função custo que mapeia E no conjunto R de números não-negativos. G é *completo* se, para todos os pares de vértices distintos v_i e v_j, $\{v_i, v_j\} \in E$. Seja $T \subset V$ um subconjunto de vértices diferenciados de V que chamaremos de *vértices terminais*.

Denotaremos um *caminho* em G por uma sequência de vértices, $u_1, u_2, ..., u_p$, tal que para todo k, $1 \leq k < p$, $\{u_k, u_{k+1}\} \in E$ e $u_k \in V$. Dizemos que o caminho é de u_1 para u_k e sua distância é $\sum_{k=1}^{p-1} d(\{u_k, u_{k+1}\})$. O caminho é *simples* se todos os vértices no caminho são distintos. Um *caminho mínimo* de u_1 para u_p é um caminho de u_1 para u_p cujo custo é mínimo entre todos os possíveis caminhos de u_1 para u_p.

Uma *árvore* de G é um subgrafo conexo de G tal que a remoção de qualquer aresta do subgrafo o fará desconexo. Seja Q qualquer subconjunto de vértices em um subgrafo conexo G' de G. Diremos que G' gera Q. Uma *árvore geradora* de G é uma árvore que gera V. A *árvore geradora mínima* de G é a árvore geradora de G tal que a peso total nas suas arestas é mínimo entre todas as árvores geradoras. Dado um grafo não dirigido ponderado G e um conjunto T de *vértices terminais*, uma árvore de Steiner para G e T é uma árvore em G que gera T. A árvore mínima de Steiner para G e T é uma árvore de Steiner para G e T tal que o custo total nas suas arestas é mínimo entre todas as árvores de Steiner para G e T.

O problema de encontrar uma árvore de Steiner mínima para quaisquer G e T é encontrar uma árvore de G que gera T com custo total mínimo das arestas. Esse problema mostrou-se ser NP-completo, mesmo para uma classe restrita de funções de mapeamento para a função custo.

[1] Autoloops foram desconsiderados nesse trabalho

2 A heurística implementada

A meta-heurística *Simulated Annealing* consiste em uma busca local probabilística e se baseia na equivalência entre o processo físico de formação de cristais e a otimização de um problema combinatório. Na Física, a obtenção de cristais consiste do seguinte: colocar a matéria em alta temperatura e resfriá-la lentamente. No início, com temperatura alta, a matéria pode mudar para estados de mais alta *ou* mais baixa energia. No final, com temperatura baixa, praticamente só é possível passar de um estado para outro que tenha energia menor. Esse processo se encerra quando o sistema está "congelado", ou seja, foi atingido o estado mínimo local de energia.

O modelo de Metropolis de simulação de um sistema físico é baseado na idéia de que a probabilidade do sistema estar em um estado de energia E é proporcional a função de Gibbs-Boltzmann

$$\frac{1}{e^{E/(kT)}}, \; para \; T > 0$$

onde T representa a temperatura e k é uma constante de normalização.

Para simular esse processo, o algoritmo de busca local passa de uma solução para outra na sua vizinhança com uma probabilidade que é maior para soluções de menor custo e que tende a zero para soluções que provocam aumento de custo à medida que o número de iterações aumenta. O comportamento do algoritmo assemelha-se àquele de uma busca aleatória no início e a de uma busca local determinística no fim (Figura 1). Para mais informações veja [4].

```
Simulated-Annealing    (* problema de minimização *)
    T ← T0;
    S ← Gera-Solucao-Inicial;
    S* ← S;
    Repita (L1) vezes
        Repita (L2) vezes
            S' ← Escolha-Vizinho-Aleatorio(S);
            δ ← custo(S') − custo(S);
            Se δ < 0 então
                S ← S' ;
                Se custo(S) < custo(S*) então
                    S* ← S;
            Senão
                Fazer S ← S' com probabilidade e^{−(δ/(kT))};
        T ← αT;
    Retornar S*.
```

Figura 1: Algoritimo para a heurística *Simulated Annealing*

3 Heurística Construtiva Utilizada

Para gerar uma solução inicial para o problema Steiner Tree, utilizamos uma heurística que recebe como entrada um grafo ponderado não-dirigido $G = (V, E, c)$ e um conjunto T, tal que V é o conjunto de vértices de G, E é o conjunto de arestas de G, c é uma função com o custo das arestas de G e $T \subset V$ é um subconjunto de V que representa os vértices terminais.

Considere o grafo ponderado não dirigido $G_1 = (V_1, E_1, c_1)$ construído de G e T tal que $V_1 = T$ e, para todo $\{v_i, v_j\} \in E_1$, $d(\{v_i, v_j\})$ é igual ao custo do caminho mínimo de v_i para v_j em G. Note que, para cada aresta de G_1, há um correspondente caminho mínimo em G. Dada qualquer árvore geradora em G_1, podemos construir um subgrafo de G por substituir cada aresta da árvore por seu corresponde caminho mínimo em G.

Nessa heurística são executados simplesmente cinco passos fundamentais e que geram uma solução para o problema (Figura 2):

Algoritmo H

1. Constrói-se uma grafo completo e não-dirigido $G_1 = (V_1, E_1, c_1)$ a partir de G e T.

2. Encontra-se a árvore geradora mínima (AGM) T_1 a partir de G_1 (se houverem várias AGM, pega-se uma arbitrária).

3. Constrói-se um subgrafo, G_S, a partir de G, de modo que cada aresta em T_1 seja substituída pelo seu respectivo caminho mínimo em G (se houver mais de um caminho mínimo, utiliza-se um arbitrário).

4. Constrói-se uma outra AGM, T_S, a partir de G_S (no caso de encontrar mais de uma árvore, seleciona-se uma arbitrária).

5. Constrói-se uma árvore de Steiner, T_H, a partir de T_S, tal que as arestas desnecessárias são removidas e todas as folhas da árvore resultante são vértices terminais.

Quanto a complexidade desse algoritmo, no pior caso, temos que a etapa 1 pode ser feita em tempo $O(|T||V|^2)$, a etapa 2 em tempo $O(|T|^2)$, a etapa 3 em tempo $O(|V|)$, a etapa 4 em tempo $O(|V|^2)$ e a etapa 5 pode ser feita em tempo $O(|V|)$. Portanto, esse algoritmo tem complexidade de pior caso de $O(|T||V|^2)$. Pode-se mostrar que C_H, o custo total das arestas da árvore de Steiner produzida por esse algoritmo não está muito distante de C_{MIN}, o custo total das arestas da árvore de Steiner mínima. De fato, $C_H/C_{MIN} \leq 2\left(1 - \frac{1}{l}\right)$, onde l é o número de folhas na árvore de Steiner mínima. Mais detalhes sobre a complexidade e a prova de corretude do algoritmo podem ser vistos em [3].

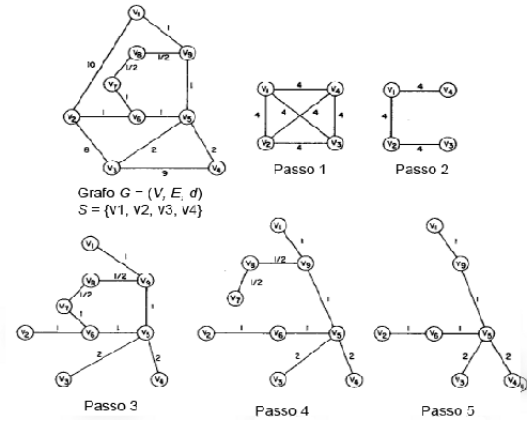

Figura 2: Demonstração de execução do algoritmo H

4 Algoritmo para geração de vizinhança

Uma parte fundamental na execução do algoritmo Simulated Annealing é a geração da vizinhança de uma solução (equivalente à perturbação no estado da matéria para o processo físico). Para gerar as vizinhanças, partimos do seguinte pressuposto: uma solução T é possível se T é uma árvore geradora de todos os vértices terminais e onde todos os não-terminais tem grau pelo menos 2. Uma vizinhança de T é qualquer árvore T' que pode ser obtida de T por (para detalhes, veja [2]):

Algoritmo para geração de vizinhanças

1. remover uma aresta e de T,

2. reconetar as duas componentes de $T - e$ por um caminho de custo mínimo,

3. remover os não-terminais de grau 1 (um de cada vez).

Visto que um dos problemas de outras heurística, como Busca Local e Busca Tabu, é permanecer iterando sobre mínimos locais, nosso objetivo ao utilizar esse algoritmo foi tentar gerar soluções o mais diversificadamente possível com a

esperança de atingirmos o mínimo global (ou próximo dele).

5 Implementação

Neste trabalho utilizamos as linguagem Python e C++. Apesar de ser uma linguagem interpretada, o Python se saiu relativamente bem na questão de desempenho, além da clareza, rapidez e facilidade de programação.

O único problema encontrado foi que o tempo de execução para o algoritmo de caminhos mínimos Floyd-Warshall usado na seção 3 era muito grande, fazendo o nosso algoritmo passar o tempo-limite em algumas instâncias.

A solução para esse problema foi utilizar a ferramenta swig e criar um binding C++ para o Floyd-Warshall. Como vemos na figura 3, o desempenho melhorou consideravelmente [2] , o que nos leva a conclusão de que a linguagem Python padrão não é eficiente para a manipulação de matrizes, característica do Floyd-Warshall.

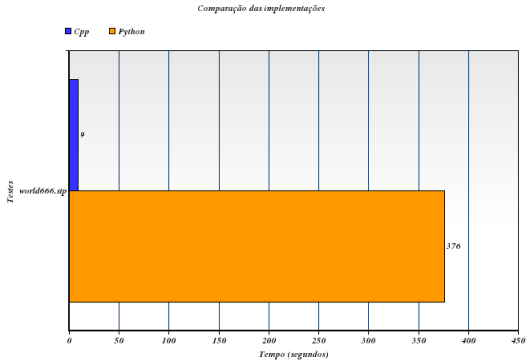

Figura 3: Comparação das implementações em python e em C++ do algoritmo de Floyd-Warshall

Uma outra solução seria usar uma biblioteca de computação científica (como a Numpy), mas na especificação do trabalho não se podia utilizar bibliotecas externas, então não tentamos essa alternativa.

5.1 Descrição do código-fonte

graph.py: possui a classe Node (vértice), Edge (aresta) e Graph (grafo) que representam a estrutura de um grafo. Também possui as classes Tree (árvore) e SteinerTree (árvore que possui nós terminais e um custo associado).

graph_utils.py: são as classes utilitárias do projeto como UnionFind (mantém uma estrutura de conjuntos disjuntos), FloydWarshall (calcula o caminho mínimo entre todos os pares de vértices de um grafo), HAlgorithm (gera uma solução inicial para o problema de Steiner), Neighborhood (gera a vizinhança de uma solução) e GraphGen (gera grafos aleatórios para teste).

simulannealing.py: Contém a classe SimulatedAnnealing que executa a respectiva heurística, baseada no algoritmo de Metropolis.

stein.py: Arquivo responsável por fazer a interface com o usuário e executar a instância passada como entrada.

teste.py: Usado para executar testes sobre as classes implementadas.

floydwarshall.cpp: Implementa o algoritmo de Floyd-Warshall em c++.

floydwarshall.py: Faz a comunicação do Binding C++ do algoritmo de Floyd-Warshall com o Python.

floydwarshall_wrap.cpp: Binding C++ do algoritmo de Floyd-Warshall.

floydwarshall.i: Binding C++ do algoritmo de Floyd-Warshall.

5.2 Estruturas de dados utilizadas

5.2.1 Dicionários:

Em vez de nos preocuparmos com a representação por lista ou matriz de adjacência, utilizamos dicionários na representação do grafo. Dicionários são estruturas de dados já incluídas por padrão na linguagem Python que fazem uma ligação entre uma chave e um valor.

Desse modo, nossa representação para um grafo nada mais é do que 2 dicionários: em um deles, dado um nó do grafo, tem-se a lista de adjacência dele; no outro, dado um par (u,v) de nós, tem-se a aresta entre u e v.

Como a relação chave-valor é dada por uma função de hash, o tempo de acesso aos elementos é constante, além disso, operações que percorrem todos os elementos do dicionários são em tempo linear, pois as chaves estão em uma lista. Assim, temos uma representação eficiente da matriz e da lista de adjacência na mesma estrutura.

6 Resultados

A execução dos testes nesse trabalho foi fundamental para a verificação da corretude do programa. Para isso, nos baseamos em comparações das soluções ótimas conhecidas com as soluções geradas.

6.1 Testes de soluções

O objetivo desse conjunto de testes é comparar as soluções geradas pela heurística implementada com as soluções ótimas conhecidas de determinadas instâncias. Utilizamos como referência o site [1] onde se pode encontrar uma série de instâncias para o problema da árvore de Steiner com suas

[2] Os testes dessa série foram executados num computador com processador Intel(R) Core(TM) 2 Duo T8100 de 2.10GHz, cache de 3072 KB e com 3GB de memória RAM.

características e soluções ótimas. Dentre essas, optamos pelas a seguir[3]:

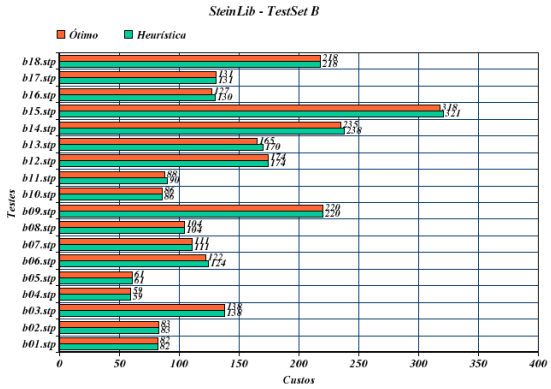

Figura 4: TestSet B - Grafos esparsos com custos aleatórios

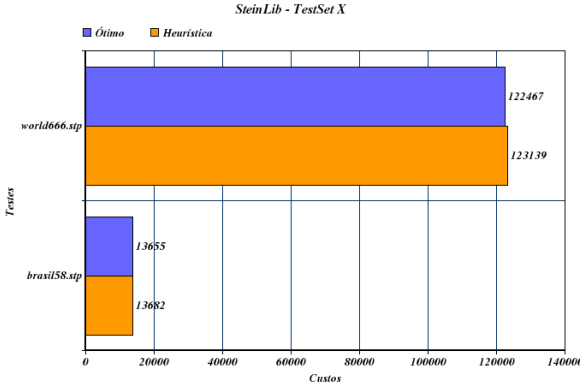

Figura 5: TestSet X - grafos completos com custos euclidianos

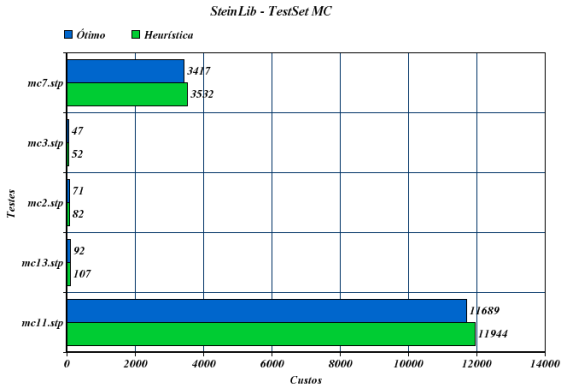

Figura 6: TestSet MC - Grafos esparsos e completos

Para a execução desses testes foi elaborado um script Shell (localizado em *'tests/test_steinlib.sh'*), que faz uma série de chamadas ao programa principal e processa os arquivos de instâncias. Além disso, foi usado o comando *time* (do Linux) para medir os tempos de execução que, por sinal, ficarama abaixo do limite estabelecido de 2 minutos.

6.2 Teste de tempo

Este conjunto de testes tem por objetivo analisar o tempo de execução de grafos completos gerados aleatoriamente.[4] Utilizamos a classe *GraphGen* e um script Python de testes (localizado em *'tests/test_random.py'*) que usa funções do pacote *time* para exibir o tempo gasto em cada uma das etapas do algoritmo. Segue abaixo uma tabela com os resultados obtidos:

NV	TG (s)	NT	FW (s)	HA (s)	SA (s)	TT (s)
10	0.002	5	0.005	0.002	3.100	3.108
50	0.028	18	0.016	0.014	12.986	13.017
100	0.141	72	0.072	0.298	15.606	15.979
200	0.874	196	0.388	3.416	11.388	15.208
500	11.370	51	4.468	0.172	29.706	34.507
700	33.149	372	12.881	20.239	61.905	95.359
1000	97.438	733	42.922	130.608	127.481	301.745
1500	346.814	271	177.566	12.062	54.050	245.589

Tabela 1: Legenda: NV = número de vértices, TG = Tempo de geração, NT = número de terminais, FW = Floyd-Warshall, HA = Algoritmo H, SA = Simulated Annealing, TT = tempo total (FW + HA + SA)

Os dados da tabela mostram que, quanto maior o número de vértices, maior tende a ser o tempo total de execução e o tempo para executar Floyd-Warshall (que é de complexidade de tempo $O(|V|^3)$) e que quanto maior o número de vértices terminais, maior é o tempo de execução do algoritmo H (poi, como já vimos, sua complexidade é $O(|T||V|^2)$). Também nota-se que é difícil prever qual das etapas (FW, HA ou SA) demorará mais visto que temos instâncias onde cada uma delas aparece pelo menos uma vez como a etapa mais demorada.

Referências

[1] Steinlib testdata library.

[2] Frank Hwang, Dana Richards, P. W. The steiner tree problem. *Annals of Discrete Mathematics 53*, 163.

[3] Kahng, A. B., Member, A., Robins, G., and Member, S. A new class of iterative steiner tree heuristics with good performance. *IEEE Trans. Computer-Aided Design 11* (1992), 893–902.

[4] Souza, C. C., and Xavier, E. C. Tratamento de problemas np-dificeis: Simulated annealing.

[3] Para os gráficos mostrados, entende-se por "Ótimo" o resultado ótimo fornecido por [1] e "Heurística" o resultado obtido pela nossa implementação

[4] Os teste dessa seção foram executados num computador com processador Intel(R) Core(TM) 2 Duo T5550 de 1.83GHz e com 2GB de memória RAM.