Trabalho Prático 3

Thiago Felicíssimo César

1 Introdução

Neste trabalho prático, objetiva-se a resolução de dois problemas, que partem de uma mesma contextualização. Dado uma trilha em uma floresta, que é formada por m caminhos e n junções de caminhos, deseja-se colocar o menor número possível de bandeiras em junções de caminhos, de forma que todo caminho tenha uma bandeira em alguma de suas extremidades.

Para o primeiro problema, denominado Tarefa 1, deseja-se achar o número de bandeiras que satisfaz o problema em uma trilha sem ciclos. Já no segundo problema, denominado Tarefa 2, deseja-se obter uma solução aproximada para o problema para trilhas que possivelmente contém ciclos. Nesse caso, o número de bandeiras encontrado deve ser no máximo o dobro do número ótimo de bandeiras, e além disso deseja-se também saber em quais junções de caminhos as bandeiras devem ser colocadas.

Com relação a complexidade do código, tem-se a restrição de que o tempo de execução não deve exceder dois segundos. Como essa restrição não é muito precisa, será adotada durante o trabalho a restrição de que a complexidade deve ser polinomial em n e m, e na seção 5 será mostrado que o código de fato cumpre a restrição original.

2 Solução Proposta

O problema tradado pode ser modelado por meio de grafos de uma maneira muito natural. De fato, se a trilha for modelada por um grafo no qual as arestas representam os caminhos e os vértices representam as junções, obtém-se exatamente o problema da cobertura mínima de vértices, que é um problema bastante conhecido.

Infelizmente, tal problema é conhecidamente NP completo, o que na prática acaba com qualquer possibilidade de achar uma solução para um grafo genérico. Convenientemente, isso não será um grande problema neste trabalho, uma vez que algumas particularidades no enunciado do problema facilitam bastante sua solução.

De fato, como na Tarefa 1 a trilha não apresenta ciclos, então ela pode ser modelada como uma árvore - isto é, como um grafo conexo sem ciclos. Na subseção 2.1 é apresentado o principio de funcionamento de um algorítimo que possibilita resolver o problema em árvores em tempo linear.

No caso da Tarefa 2, passa-se a lidar com grafos genéricos que podem conter ciclos. Todavia, para este problema não é necessário achar a solução ótima, já bastando uma solução que seja no máximo duas vezes maior que a ótima. Na subseção 2.2 é apresentado um algorítimo guloso que resolve esse problema em tempo linear.

2.1 Algorítimo para a Tarefa 1

O fato dos grafos tratados na Tarefa 1 não apresentarem ciclos faz com que se torne possível encontrar a solução ótima em tempo linear. Para isso, será apresentado um algorítimo de programação dinâmica que faz exatamente isso, e na subseção 4.1 será mostrado como é possível adaptá-lo para uma outra classe de algorítimos. Antes disso, contudo, é importante fazer algumas definições para lidar com o problema da forma mais natural possível.

Dada uma árvore $G = (V, E)$, considera-se a situação em que ele é enraizado em um $r \in V$ qualquer - isto é arbitra-se um vértice r qualquer para ser a raiz. Com isso, gera-se uma divisão em camadas no grafo, o que induz relações de descendência. Define-se P como a função pai relacionada a essa divisão em camadas induzida pela escolha da raiz. Ou seja, para um determinado $w \in V$, o seu $P(w)$ será aquela nó que conecta o vértice w a camada superior. Arbitra-se para o nó raiz r que $P(r) =$ inválido.

A partir da função P, define-se para cada $v \in V$ o conjunto v^* de seus descendentes como o conjunto daqueles nós que compartilham alguma aresta com v, com exceção de seu pai. Uma outra definição que será importante aqui é o de subárvore gerada por um vértice v. Nesse caso, considera-se as duas componentes conexas formadas pela remoção da aresta $\{v, P(v)\}$ de G. A componente que contém o nó v, quando enraizada nele, será chamada de subárvore gerada por v. Essas definições apresentadas serão suficientes para tratar o problema.

Assim como em todos os algorítimos de programação dinâmica, objetiva-se encontrar uma forma de expressar o problema em função de subproblemas menores. Para explicar como isso pode ser feito, considera-se o problema de cobrir a subárvore gerada por um $v \in V$ qualquer (perceba que esse é precisamente o problema tratado neste trabalho quando o nó em questão é a raiz de G). O menor número de vértices necessários para resolver esse problema em particular será dado por $OPT(v)$.

Nota-se que, para cobrir a subárvore gerada por v, um de dois casos deve acontecer: ou o nó v faz parte da cobertura, ou todos seus descendentes fazem parte dela - caso contrário, a aresta que liga v em algum $w \in v^*$ ficaria de fora da cobertura. Se o nó v fizer parte da cobertura, então considera-se em seguida os subproblemas de cobrir as subárvores geradas por cada um de seus descendentes. Caso contrário, então é necessário cobrir todos os nós em v^* e, em seguida, considerar os subproblemas de cobrir as árvores geradas pelos nós em w^* para cada $w \in v^*$ - ou seja, as subárvores geradas pelos descendentes dos descendentes de v. Com isso, para saber o valor de fato de $OPT(v)$ é necessário tomar o menor número de vértices necessários ao considerar cada uma dessas duas situações.

A partir desse raciocínio, observa-se que é possível expressar o valor de $OPT(v)$ por

$$OPT(v) = \min\{1 + \sum_{w \in v^*} OPT(w), \sum_{w \in v^*} 1 + \sum_{z \in w^*} OPT(z)\}.$$

Convém comentar que, no caso de um vértice sem descendentes, o número de vértices necessários para cobrir sua subárvore é evidentemente zero - uma vez que não há aresta para cobrir. Para facilitar a compreensão da expressão (e será visto que isso ajudará também na implementação do algorítimo), define-se para um vértice $v \in V$ o $COPT(v)$ por

$$COPT(v) := \sum_{w \in v^*} OPT(w)$$

(ou seja, o somatório do OPT de cada um de seus descendentes). Utilizando essa definição, pode-se expressar o $OPT(v)$ por

$$OPT(v) = \min\{1 + \sum_{w \in v^*} OPT(w), \sum_{w \in v^*} 1 + COPT(w)\},$$

uma fórmula muito mais agradável.

Nota-se que a corretude do algorítimo já está automaticamente estabelecida. De fato, pela sua construção, ele pega sempre o menor valor possível de OPT que garanta que todas as arestas estejam cobertas.

A introdução da função $COPT$ não traz benefícios de natureza somente estética para a equação do problema. Na verdade, por mais que o algorítimo tenha sido apresentado até aqui como um algorítimo de programação dinâmica, utilizando a função $COPT$ torna-se possível expressar o problema de achar o OPT e o $COPT$ de um vértice em função da solução desse problema para cada vértice em v^*. Fazendo uso dessa observação, na subseção 4.1 será mostrado como implementar esse algorítimo em uma versão divisão e conquista. Na seção 5 será feita sua análise de complexidade.

2.2 Algorítimo para a Tarefa 2

Na falta de garantias com respeito a não-ciclicidade do grafo considerado, torna-se impossível achar a solução ótima para o problema em tempo polinomial. Todavia, nessa subseção é demonstrado como pode-se obter uma solução aproximada para o problema, de forma que o número de vértices necessários não seja maior que duas vezes o número de vértices na solução ótima.

Para que seja possível explicar o algorítimo, antes é preciso apresentar algumas definições importantes. Para um dado grafo $G = (V, E)$, um conjunto $S \subset E$ é chamado de emparelhamento quando nenhum par de arestas $e_i, e_j \in S$, $i \neq j$, é incidente a um mesmo vértice. O emparelhamento é chamado de maximal quando para todo $e \in E \setminus S$, existe algum $e' \in S$ tal que e e e' são incidentes a um mesmo vértice. Nota-se que um determinado grafo pode ter vários emparelhamentos maximais, inclusive com tamanhos diferentes.

Agora que o conceito de emparelhamento está definido, pode-se utiliza-lo para obter alguns resultados interessantes para o problema tratado aqui. Suponha que S seja um emparelhamento maximal qualquer e que U seja a cobertura ótima do grafo em questão. Nota-se que o tamanho de U não pode ser menor do que o de S, uma vez que, para que todas as arestas em S estejam cobertas, é necessário para cada aresta $e \in S$ tomar um vértice v que esteja conectado a ela. Como os conjuntos de vértices conectados por cada aresta $e \in S$ são disjuntos - caso contrário teria-se duas arestas incidentes a um mesmo vértice -, então é preciso tomar ao menos $|S|$ vértices para se ter uma cobertura. Com isso, estabelece-se a relação de que $|U| \geq |S|$ para todo emparelhamento S que seja maximal.

Finalmente, considera-se agora o seguinte algorítimo. Começando com $M = \emptyset$, toma-se uma aresta $e \in E$ qualquer e adiciona-se ela em M. Em seguida, remove-se do grafo a aresta previamente selecionada e todas as arestas que compartilham um nó com ela. Repete-se o mesmo procedimento até que não haja mais nenhuma aresta em E. Não é difícil ver que o conjunto M obtido é um emparelhamento maximal - afinal, nenhuma das arestas em M são incidentes a um mesmo vértice, e se o algorítimo parou é porque não havia mais nenhuma aresta no grafo que não compartilhasse ao menos um nó com algum $e \in M$. Já que M é um emparelhamento maximal, estabelece-se que $|M| \leq |U|$.

Definindo $Q \subset V$ como o conjunto dos vértices tal que, para todo $v \in Q$ tem-se que algum $e \in M$ seja incidente a v, obtém-se que $|Q| = 2|M|$ - uma vez que cada aresta conecta dois vértices, e nenhum par de arestas em M é incidente a um mesmo vértice. Além disso, nota-se que Q é uma cobertura de vértices, uma vez que se se supusesse por absurdo que não, então haveria alguma aresta em $E \setminus M$ que não compartilha nenhum vértice com nenhuma aresta em M, e portanto o algorítimo não teria terminado. Sabendo que $|M| \leq |U|$, e que $|Q| = 2|M|$, obtém-se que $|Q| \leq 2|U|$. Segue que o algorítimo apresentado acima pode ser utilizado para se obter uma cobertura de vértices Q tal que $|Q| \leq 2|U|$.

Agora que o algorítimo já foi apresentado e sua corretude já está estabelecida, falta apresentar sua implementação e mostrar que sua complexidade é de fato linear em $|V|$ e $|E|$, o que será feito, respectivamente, na subseção 4.2 e na seção 5.

3 Estruturas de Dados para o Grafo

Agora que o algoritmo utilizado já foi discutido de forma clara em alto nível, é necessário precisar os detalhes de como ele foi implementado. Todavia, antes de falar propriamente do algoritmo e de como as decisões são tomadas, faz-se necessário discutir a estrutura de dados utilizado para representar o grafo.

A Figura 1 ilustra a implementação da estrutura de dados que armazena os dados do grafo. Nela, é representado como exemplo um grafo de sete vértices.

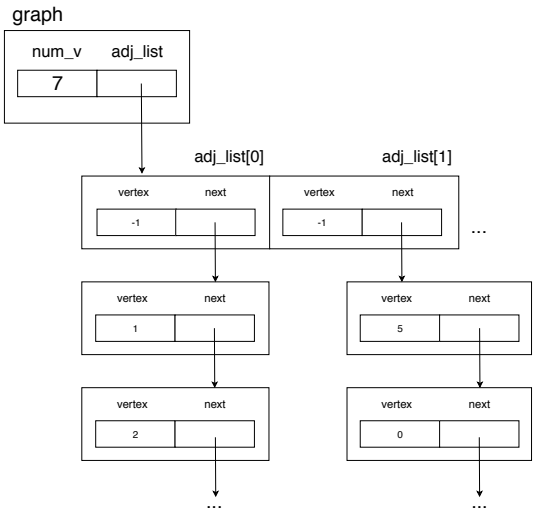


Figura 1: Estrutura de dados do grafo.

Como pode ser observado na imagem, dois tipos de *struct* são utilizados. Um deles, do tipo *graph*, representa o grafo todo e contém seu número de vértices e um ponteiro. Esse ponteiro aponta para um arranjo, alocado dinamicamente na inicialização do grafo, de estruturas do tipo *edge*, organizado de tal maneira que na posição i do arranjo se encontra a cabeça da lista que armazena as arestas incidentes ao vértice i. Cada célula guarda um ponteiro para a próxima célula, bem como o identificador do outro vértice ligado pela aresta (naturalmente, esses valores são inicializados como -1 para as cabeças).

4 Algoritmos

Esta seção é dividida em duas partes, nas quais se explica de maneira separada a implementação dos algorítimos das Tarefas 1 e 2. Será dado enfoque nos algorítimos que de fato resolvem o problema, uma vez que procedimentos como ler a entrada, criar um grafo e adicionar uma aresta são de implementação trivial.

4.1 Implementação da Tarefa 1

Assim como comentado na subseção 2.1, aqui será mostrado como implementar um algorítimo de divisão e conquista utilizando como ponto de partida a abordagem de programação dinâmica anteriormente descrita. A principal característica a ser notada da implementação apresentada é a ausência de estruturas de *memoization*. De fato, com a introdução da variável COPT foi possível eliminar completamente a necessidade

de manter um arranjo com os valores de OPT para cada vértice, removendo a característica principal que enquadrava o algorítimo como dinâmico.

A implementação do algorítimo faz uso de dois procedimentos. O primeiro, que é o Algorítimo 1, simplesmente faz a chamada do procedimento MVC_Recursivo (MVC = *Minimum Vertex Cover*) que calcula os valores de OPT e COPT para o nó passado como argumento. Em particular, o nó 1 é passado como argumento, e como seu pai é definido como inválido, então o nó 1 passa a ser o pai de todos os nós conectados a ele - e, consequentemente, passa a ser a raiz de todo o grafo G. Em seguida, o valor de OPT do nó 1 é retornado, que é justamente o valor procurado.

Algorítimo 1 Algorítimo para a Tarefa 1

Dados: G
OPT_r, COPT_r ←MVC_Recursivo($G, v = 1$, PAI_v $= -1$)
retorna OPT_r

Algorítimo 2 MVC_Recursivo

Dados: G, v, PAI_v
L ← $prox(G.cabeca\_lista\_adj[v])$
OPT_não_coberto, COPT_v ← 0
OPT_coberto ← 1
enquanto $L \neq 0$ **faça**
  se PAI_v $\neq$ *nó(L)* **então**
    OPT_w, COPT_w ←MVC_Recursivo(G, nó(L), v)
    OPT_coberto ← OPT_coberto + OPT_w
    OPT_não_coberto ← OPT_não_coberto + 1 + COPT_w
    COPT_v ← COPT_v + OPT_w
  fim
  L ← $prox(L)$
fim
OPT_v ←min{OPT_coberto,OPT_não_coberto}
retorna OPT_v, COPT_v

No procedimento MVC_Recursivo, começa-se por pegar a primeira aresta da lista de adjacência do nó v passado como argumento, e em seguida inicializa-se as variáveis OPT_coberto, OPT_não_coberto e COPT_v. Como mostrado na subseção 2.1, para calcular o OPT de um determinado nó, considera-se as possibilidades dele estar ou não na cobertura e calcula-se o valor mínimo entre os tamanhos de cobertura associados a cada possibilidade. As variáveis OPT_coberto e OPT_não_coberto representam exatamente esses valores, que vão sendo calculados durante o laço **enquanto**. Já o COPT_v, assim como definido previamente, vai sendo calculado somando os valores dos OPT de cada nó descendente de v.

Para cada nó w na lista de adjacência de v, primeiramente verifica-se se o nó examinado é o pai de v - isto é, verifica-se se $P(v) = w$. Caso não seja o caso, é feita uma chamada recursiva em w, passando como argumento o nó v como o pai de w - de fato, qualquer nó conectado em v que não seja seu pai tem de ser um de seus descendentes. Observa-se que, como essa chamada só interfere na subárvore gerada por w, então não há qualquer risco das chamadas recursivas geradas precisarem dos valores de OPT e COPT em v, o que impossibilitaria a divisão do problema.

Uma vez terminada a chamada recursiva em w, obtém-se os valores de OPT e COPT do nó w, que são utilizados nos cálculos de OPT_coberto, OPT_não_coberto e COPT_v. Após o laço **enquanto** ter sido

executado para todo $w \in v^*$, o COPT_v já está completamente calculado, e para calcular o OPT_v basta tomar o mínimo de OPT_coberto e OPT_não_coberto.

4.2 Implementação da Tarefa 2

O Algorítimo 3 exibe a implementação da solução para a Tarefa 2. Inicialmente, dois arranjos são inicializadas com entradas todas em *falso*. O arranjo **cobertos** é atualizado ao longo do algorítimo a medida em que os vértices vão sendo cobertos - aqui, utiliza-se um abuso de notação para dizer que todas suas arestas estão cobertas. Já o arranjo **cobertura** vai sendo atualizado para indicar quais vértices vão fazer parte da cobertura. Nota-se que todo vértice que passa a fazer parte da cobertura também passa a estar coberto (o contrário obviamente não é verdade).

Algorítimo 3 Algorítimo para a Tarefa 2

```
Dados: G
cobertos[1, ..., n] ← falso
cobertura[1, ..., n] ← falso
para i = 1, ..., n faça
    se cobertos[i] = falso então
        L ← prox(G.cabeca_lista_adj[i])
        enquanto L ≠ 0 e cobertos[nó(L)] = verdadeiro faça
            L ← prox(L)
        fim
        se L = 0 então
            cobertos[i] ← verdadeiro
        senão
            cobertura[i, nó(L)] ← verdadeiro
            cobertos[i, nó(L)] ← verdadeiro
        fim
    fim
fim
retorna cobertura
```

Para cada um dos nós do grafo, o algorítimo primeiro verifica se ele já está coberto. Se esse é o caso, então não há nada a fazer com o nó considerado. Caso o nó não esteja coberto, então o algorítimo passa a procurar por um vizinho dele que ainda não esteja coberto - lembrando que isso é equivalente a procurar por uma aresta não coberta, uma vez que o nó vizinho só estará coberto se todas suas arestas assim estiverem.

Caso não se ache uma aresta que não esteja coberta, e como as trilhas analisadas são todas conexas, então a única conclusão possível é que todos seus vizinhos já foram cobertos - ou seja, todas suas arestas foram cobertas por nós vizinhos -, e portanto o vértice analisado é marcado como coberto. Caso haja um vizinho não coberto, então o nó analisado e o vizinho encontrado são ambos adicionadas na cobertura - e consequentemente passam a estar cobertos também.

Uma observação importante é que o algorítimo implementado aqui é equivalente ao algorítimo apresentado na subseção 2.2. De fato, a única diferença é que ao invés de obter o conjunto das arestas selecionadas, ao final do algorítimo obtém-se o conjunto dos vértices ligados por cada uma das arestas que estaria no emparelhamento obtido, o que é uma informação muito mais útil e fácil de representar.

5 Análise de Complexidade

Verifica-se facilmente que, utilizando a estrutura de dados apresentada, a complexidade de tempo e espaço para a inicialização e a deleção do grafo é $O(|V| + |E|)$ em ambas as tarefas.

5.1 Análise de Tempo

5.1.1 Tarefa 1

O primeiro ponto a se notar no algorítimo da Tarefa 1 é que, para cado nó $v \in V$, é feita somente uma chamada de MVC_Recursivo. Além disso, excluindo o custo das chamadas recursivas que são feitas para os nós em v^*, os únicos procedimentos que sobram consistem em iterar a lista de adjacência de v e algumas outras operações de custo constante.

Observa-se que cada cada aresta $\{v, w\} = e \in E$ só pode ser analisada duas vezes: uma por v e outra por w. Essa observação estabelece um invariante para a análise de quantas vezes o laço **enquanto** pode ser executado. Assim sendo, conclui-se que, excluindo o custo das chamadas recursivas de MVC_Recursivo de dentro do laço **enquanto**, tem-se que o custo total de todos os laços **enquanto** gerados pela chamada de MVC_Recursivo na raiz da árvore é $O(|E|)$. Além disso, como os outros procedimentos de custo constante em MVC_Recursivo são executados uma vez para cada $v \in V$, segue que o custo total do código é $O(|V|+|E|)$. Adicionando os custos de inicialização e deleção, a complexidade continua a mesma. Como o grafo analisado é uma árvore, vale que $|E| = |V| - 1$, e portanto a complexidade se escreve como $O(|V|)$, que também é igual a $O(|E|)$.

5.1.2 Tarefa 2

Inicialmente, o código inicializa os arranjos cobertos e cobertura - como ambos tem $n = |V|$ posições, a inicialização tem custo $O(|V|)$. Para cada $v \in V$, no pior caso (isto é, quando a condição do laço **se** é verdadeira) itera-se sobre a lista de adjacências do nó v, e em seguida são realizadas outras operações de custo constante. Como cada aresta é analisada no máximo duas vezes, segue que o custo total de todos os laços enquanto é $O(|E|)$. As outras operações que estão dentro do **se** mas fora do **enquanto** tem um custo total de $O(|V|)$, pois são realizadas no máximo uma vez para cada nó. Segue que o custo total do procedimento é $O(|V| + |E|)$. Adicionando os custos de inicialização e deleção, a complexidade continua a mesma.

5.2 Análise de Espaço

5.2.1 Tarefa 1

A cada chamada de MVC_Recursivo é feito um número constante de alocações em memoria. Dessa forma, se l é a maior a distância entre o nó da árvore e uma folha, então o custo em memória é $O(l)$. No pior caso, tem-se que $l = |V|$ (quando os nós formam uma lista, e a raiz escolhida é justamente uma de suas extremidades). Ou seja, no pior caso, o custo em memória é $O(|V|)$. Adicionando os custos de inicialização e deleção, obtém-se uma complexidade de $O(|V| + |E|)$. Como o grafo tratado é uma árvore, pelo mesmo argumento já utilizado, a complexidade se escreve como $O(|V|)$, que também é igual a $O(|E|)$.

5.2.2 Tarefa 2

As únicas variáveis presentes no código são cobertos, cobertura, L e i. L e i tem custo claramente constante, enquanto que cobertos e cobertura são ambos $O(|V|)$. Segue que o custo em memória é $O(|V|)$. Adicionando os custos de inicialização e deleção, obtém-se uma complexidade de $O(|V| + |E|)$.

6 Avaliação Estatística

Para a avaliação estatística, objetivou-se inicialmente verificar o crescimento do tempo de execução com o valor do número de vértices na entrada. Essa análise permite também verificar a complexidade temporal do algoritmo.

Para cada uma das tarefas, foi programado um gerador de entradas aleatórias para que o código deste trabalho pudesse ser rodado várias vezes e os tempos comparados. Para a Tarefa 1, para números de vértices de $50, 100, 150, ..., 15000$ foram geradas 10 entradas para cada tamanho (lembrando que, como os grafos da Tarefa 1 são árvores, o número de arestas em cada um é igual ao número de vértices menos um). Já no caso da Tarefa 2, foram geradas 10 entradas para números de vértices de $20, 40, 60, ..., 3000$, e com $|E| = 2|V|$ para cada grafo. Não foi possível gerar entradas com números de vértices tão altos para a Tarefa 2, pois o gerador programado toma muito mais tempo para gerar entradas para a Tarefa 2 do que para a Tarefa 1.

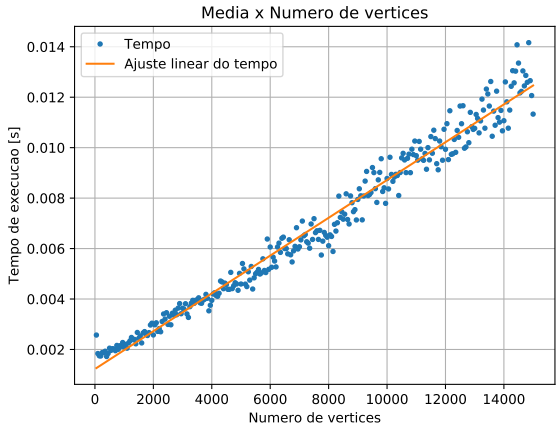


Figura 2: Média dos tempos de execução para a Tarefa 1.

As Figuras 2 e 3 mostram as médias dos tempos de execução para as Tarefas 1 e 2 para cada tamanho de entrada, bem como o ajuste linear de cada um dos conjuntos de pontos. No caso da Tarefa 1 é possível perceber claramente uma tendência linear da curva de tempo, o que comprova o resultado da complexidade temporal ser $O(|V|)$. Para a Tarefa 2, havia-se deduzido que a complexidade temporal seria $O(|V| + |E|)$, mas como nas entradas geradas tinha-se que $|E| = 2|V|$, então a complexidade teórica se reduziria para $O(|V| + 2|V|) = O(|V|)$, resultado que é comprovado pela Figura 3 - apesar da correlação ser bem mais fraca do que na Figura 2.

As Figuras 4a e 4b exibem os desvios padrões do tempo de execução para cada uma das tarefas. Como a grande maioria dos pontos apresenta desvio padrão inferior a 25%, considera-se que a amostra obtida é estatisticamente válida.

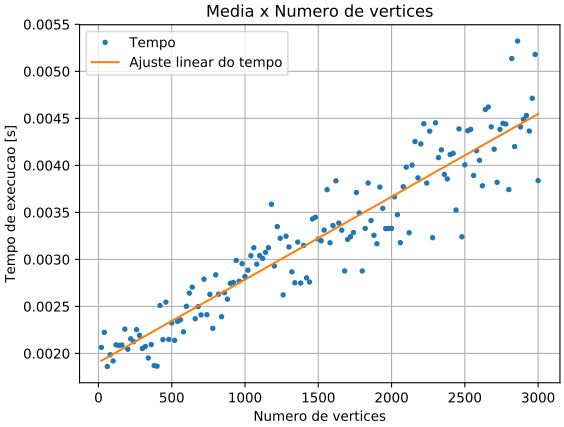
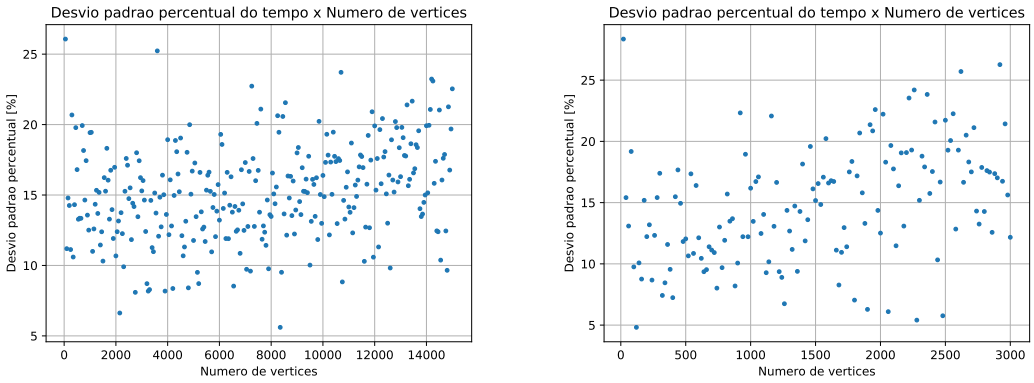


Figura 3: Média dos tempos de execução para a Tarefa 2.



(a) Tarefa 1.

(b) Tarefa 2.

Figura 4: Desvio padrão dos tempos de execução.

Para analisar as soluções obtidas por meio do algorítimo da Tarefa 2, seu código foi rodado com as entradas geradas para a Tarefa 1. Como as entradas para a Tarefa 1 têm suas coberturas ótimas conhecidas - uma vez que o algorítimo para a Tarefa 1 não é um algorítimo aproximado, e de fato acha a solução ótima -, então foi possível comparar o tamanho das coberturas encontradas pelo algorítimo da Tarefa 2 com os tamanhos ótimos. A Figura 5 exibe o tamanho normalizado da cobertura encontrada pelo algorítimo da Tarefa 2 - ou seja, o tamanho da cobertura encontrada pela Tarefa 2 dividido pelo tamanho encontrado pela Tarefa 1. Verifica-se que em todos os casos o tamanho normalizado não passou de 2, assim como desejado.

Uma observação importante a ser feita é que, como esse teste foi feito utilizando apenas árvores como entrada, então seus resultados não podem ser completamente generalizados, uma vez que o algorítimo foi implementado justamente para ser usado em grafos arbitrários que não sejam necessariamente árvores. Ainda assim, esse teste é uma forma interessante de verificar experimentalmente a corretude do algorítimo, por mais que em uma situação restrita.

Também é interessante fazer alguns estudos de caso para verificar quais vértices o algorítimo aproximado escolhe em casos específicos. As Figuras 6a e 6b exibem a cobertura ótima e a cobertura encontrada pelo

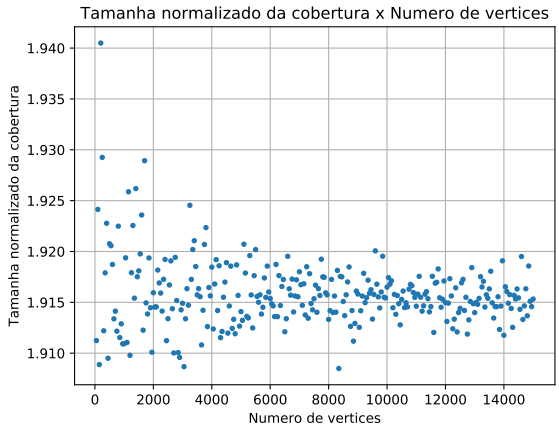
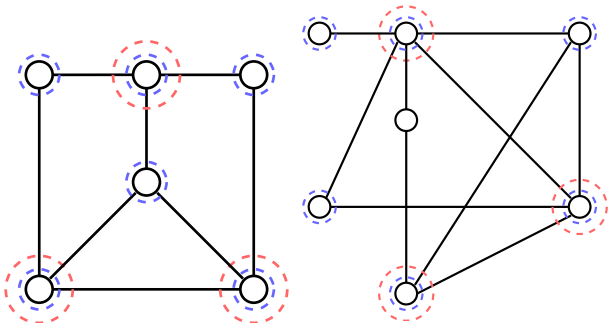


Figura 5: Otimidade da solução da Tarefa 2.



(a) Exemplo 1. (b) Exemplo 2.

Figura 6: Exemplos de saídas geradas pelo algorítimo da Tarefa 2.

algorítimo da Tarefa 2 em dois exemplos específicos. Os vértices na cobertura ótima são marcados em vermelho, enquanto que aqueles selecionados pelo algorítimo aproximado são marcados em azul.

Nota-se que no exemplo 1 o algorítimo aproximado acabou selecionando todos os vértices, o que apesar de ser uma solução bastante ruim ainda assim não é maior que duas vezes o tamanho da cobertura mínima - que nesse caso é 3. No exemplo 2 também verificou-se algo parecido, no qual o algorítimo pegou de novo exatamente o dobro de vértices na cobertura mínima, deixando somente um vértice de fora. De qualquer forma, a corretude do algorítimo foi verificada nos dois casos.

7 Conclusão

Considera-se esse trabalho um sucesso, uma vez que foi possível implementar os algorítimos para resolver ambas as tarefas. Além disso, ambos os algorítimos atenderam as restrições impostas. Um ponto interessante a se destacar é que, ao pesquisar na *internet*, não foi encontrado nenhuma outra proposta de algorítimo que resolve a Tarefa 1 utilizando divisão e conquista. Isso talvez sugira que o algorítimo apresentado nesse documento possa ser o primeiro de seu tipo para este problema. Entretanto, faria-se necessário uma pesquisa bibliográfica mais rigorosa para se poder afirmar isso.