Relatório do problema 2

2019215917 Inês Martins Marçal

2019219433 Noémia Quintano Mora Gonçalves

1. **Descrição do algoritmo**

- O grafo é representado por um u*nordered_map* nodes onde a *key* é o número do nó e o *value* é *uma struct node (explicada no tópico structs).*

- Em primeiro lugar, começou-se por organizar os nós por níveis de profundidade, no sentido de se efetuar uma abordagem *bottom up*. Por sua vez, estes níveis de profundidade foram calculados através de uma função recursiva que calcula o nível de profundidade de cada nó e, posteriormente, o adiciona a um *unordered_map treedepth* onde a *key* é o nível de profundidade e o *value* os nós desse mesmo nível.

- A função *pyramidscheme* percorre os níveis de profundidade e para cada nó desse nível calcula os pares (número de nós, valor), mais especificamente um par para o caso em que o nó vai ser usado e um par para o que não vai ser usado. Se o nó for uma folha a sua não utilização define-se como (0,0) e a sua utilização como (1, valor do nó). Pelo contrário, nos restantes nós ao longo da árvore, no caso de o nó não vir a ser utilizado, devido às restrições do problema, os seus filhos terão de ser selecionados obrigatoriamente. Assim, este caso define-se como a soma dos pares (n, v) correspondentes à seleção de cada filho que foram calculados anteriormente e guardados na respetiva *struct node*. Por outro lado, o caso em que o nó irá vir a ser selecionado define-se como a soma do melhor par de cada filho com o valor da seleção do próprio nó. Para calcular o melhor par recorre-se à função *getBest* que compara os pares e devolve o que tem menos número de nós utilizados e em caso de igualdade o que tem maior valor.

- À medida que vamos subindo na árvore, os resultados do cálculo dos casos (selecionado, não selecionado) para esses nós tem em conta os resultados calculados para os casos anteriores, sendo esses cálculos os overlapping sub-problems.

- Depois de percorrida a árvore, utiliza-se o *getBest* explicado anteriormente para retirar o melhor par.

2. **Structs**

As estruturas utilizadas no algoritmo implementado foram as seguintes:

- Struct Node:

int value: indica o valor de cada nó

array<array<int,2>,2> achievedValues: onde o achievedValues[0] contém o caso do nó não ser selecionado e o achievedValues[1] o caso em que é selecionado

vector<int> children: representa os filhos do nó em causa

- unordered_map nodes:

key: o número do nó

value: struct node correspondente a esse nó

- unordered_map treedepth:

key: nível de profundidade

value: vetor com o número dos nós que se encontram nesse nível

3. **Correctness**

O problema pode ser dividido em subproblemas que consistem em escolher o nó ou não o escolher. A escolha ou não dos nós e as restrições de verificação de conexões influenciam o resultado dos nós nos níveis superiores. Considerando que o resultado final é dado pelo melhor resultado obtido para o nó inicial, este foi construído com base nos resultados dos nós filhos e esses por sua vez com base nos seus filhos, garantindo assim que o resultado final tem em conta a árvore inteira.

Substrutura ótima:

- Dado uma árvore A = {N_0, N_1, …, N_n}, dizemos que S(N_k) é a solução para a árvore que tem como raiz N_k. Então S(N_{ki}) será a solução para os subproblemas que começam em N_{ki} (filhos de N_k).

Suposição:

- Assume-se que S(N_k) é a solução ótima para a árvore que tem como raiz N_k e que deste modo S(N_{ki}) é a solução ótima para os filhos de N_k que tenham sido utilizados em S(N_k).

Negação:

- S(N_{ki}) não é a solução ótima para os filhos de N_k que tenham sido utilizados em S(N_k).

Consequência:

- Então existe uma solução ótima melhor S'(N_{ki}) que tem um número de nós menor ou custo de recrutamento maior do que em S(N_{ki}).

Contradição:

- No entanto, ao se voltar a juntar todos os nós conclui-se que S(N_k) não é a solução ótima, o que é uma contradição. Deste modo, S(N_{ki}) é a solução ótima para o problema em questão.

4. **Algorithm Analysis**

A função recursiva que efetua o pré-processamento é utilizada para mapear os nós de acordo com a sua profundidade, tendo assim complexidade temporal de O(|V|+|E|), uma vez que a estratégia assenta num algoritmo de DFS.

Posteriormente, através da função *pyramidscheme*, já com os nós guardados por profundidade, percorrem-se todos os vértices e para cada vértice não terminal percorrem-se os seus filhos. No caso dos nós terminais são feitas operações de complexidade temporal constante. Perante isto, a função referida acima tem uma complexidade temporal de O(|V| + |E|) e consequentemente o algoritmo apresenta uma complexidade temporal total de O(2|V| + 2|E| + c), onde o c representa a complexidade constante das restantes operações.

Em termos de complexidade espacial é utilizado N para o *unordered_map* em que o *value* é a *struct node*, como referenciado no tópico *structs*, e ainda N para o *unordered_map* em que são colocados os índices dos nós do nível de profundidade correspondente. Sendo assim um total de complexidade espacial de 2N.

5. References

[1] https://www.cplusplus.com/

[2] https://en.cppreference.com/w/