Algoritmo para calcular o custo de um megaprojeto

Gabriel Weich* e João Paulo Cecilio†
Escola Politécnica — PUCRS

13 de outubro de 2018

Resumo

Este artigo descreve alternativas de solução para o segundo problema proposto na disciplina de Algoritmos e Estruturas de Dados II no terceiro semestre, o qual nos desafia a criar uma solução ótima para calcular custos de atividades em um software de gestão de projetos usando estrutura de grafos. São apresentados alguns caminhos para a solução do problema e um deles, que é a nossa solução final, é analisado mais detalhadamente. Em seguida são mostrados os resultados obtidos para diferentes casos de teste.

Introdução

O problema descrito na proposição deste trabalho é de através de uma lista de atividades, suas dependências, quantidades e custo unitários, calcular o custo total do projeto. Cada atividade pode ou não depender de outras atividades n vezes, e.g., uma atividade ABC custa $10 e usa quatro vezes a atividade FGH (que custa $5), o custo final de ABC será $30.

A entrada dos dados é constituída de duas partes: na primeira recebemos uma lista de todas as atividades envolvidas e seu custo unitário. Na segunda é descrito o número de ligações entre as atividades e quantas vezes uma atividade faz uso da outra.

Com base nos conceitos aprendidos em aula sobre algoritimos, serão testadas algumas soluções possíveis para o problema, analisadas as suas complexidades e tempo de execução, para ao final apontar e detalhar de forma mais aprofundada a melhor solução encontrada, assim como as conclusões obtidas a partir dessas tentativas.

1 Primeira solução

Inicialmente procuramos uma solução simples para o primeiro caso de teste, e para tal criamos um script em python que consistiu em:

1. Modelar uma classe grafo usando como vértice cada uma das atividades.
2. Modelar uma classe aresta onde tínhamos como atributo os vértices e a quantidade de vezes que a atividade inicial fazia uso da subsequente.
3. Ler a primeira parte do arquivo, criando uma estrutura chave-valor para os custos unitários de cada atividade.

*`gabriel.weich@acad.pucrs.br`
†`joao.cecilio@acad.pucrs.br`

4. Ler a segunda parte do arquivo, adicionando à estrutura de dados do grafo cada um dos vértices e suas respectivas arestas.

5. Calcular o custo total do projeto percorrendo o grafo em sua totalidade.

A estrutura de dados que usamos tanto para os custos, quanto para o grafo foram dicionários nativos da linguagem python. No primeiro caso usamos a string do nome da atividade e inteiros com os custos, no segundo o nome e instâncias da classe aresta. O crítico para essa, e para outra soluções que testamos, acabou sendo a forma como o custo total seria calculado, ou seja o caminhamento no grafo.

Desenvolvemos uma versão adaptada de um caminhamento de busca em profundidade (depth-first search = DFS), pois precisávamos percorrer cada um dos vértices e somar seus custos para compor o custo total. Segue abaixo a primeira versão do algoritimo de caminhamento, que faz uso de uma recursão simples para o cálculo.

```python
def soma_custo(grafo ,custos ,vertice):
    vizinhos = []
    for aresta in grafo.dict_principal[vertice]:
        vizinhos.append([aresta.vertice_final ,aresta.peso])
    if len(vizinhos) == 0:
        return int(custos[vertice])
    else:
        custo = int(custos[vertice])
        for i in vizinhos:
            custo = custo + (int(i[1])*soma_custo(grafo ,custos ,i[0]))
        return custo
```

1.1 Conclusões e resultados preliminares da primeira solução

O algoritmo acima resolve o problema proposto, pois consegue realizar o cálculo do custo total do projeto. No entanto, não possui uma performance adequada pois o tempo de execução cresce exponencialmente dado o número de tarefas e suas dependências (para o caso de 100 tarefas o algoritmo excedeu 1 hora de execução).

Isso acontece por conta de sucessivos cálculos do custo das atividades que já foram calculadas anteriormente. Ou seja, já percebemos que quando visitássemos um vértice e as arestas ligadas a ele teríamos que guardar essa informação a fim de não realizar o caminhamento naquele subgrafo novamente. Verificamos então, que para as próximas soluções que iríamos desenvolver, precisaríamos realizar uma marcação de visita no vértice e guardar seu custo total, sendo necessária assim, a criação de uma classe nodo.

Além disso, para o teste, sabíamos de antemão a atividade inicial do projeto, à qual estavam ligadas todas as outras atividades. Precisaríamos também, realizar um procedimento para selecionar por onde o caminhamento iniciaria.

Por último, sendo uma primeira solução bastante simples, não nos preocupamos com a existência de ciclos no grafo. O que poderia deixar o programa em looping. Nas próximas soluções sabíamos que deveríamos criar um tratamento para essa questão também.

2 Segunda solução

Na segunda solução buscamos resolver os problemas deixados em aberto pela primeira solução. Buscamos uma forma mais eficiente de calcular o custo do projeto sem percorrer novamente um subgrafo de uma atividade com custo já calculado. O novo algoritmo também deveria ser capaz de encontrar o nodo inicial do grafo e detectar um possível ciclo.

Para esta solução também fizemos uso de uma estrutura de dicionários para o grafo principal. Usamos o nome da atividade como chave do dicionário e uma segunda estrutura Nodo como valor.

A estrutura Nodo é responsável por armazenar as informações da atividade com aquela chave, guardando assim o seu custo e as atividades filhas junto com a quantidade de repetições das mesmas. Para guardar as atividades filhas de uma tarefa, o nodo usa um dicionário em que a chave é o nome da atividade filha e o valor é a quantidade de repetições.

A estrutura Nodo também possuí um atributo para armazenar o custo calculado da atividade, levando em consideração as atividades dependentes. Dessa forma, uma vez que o algoritmo percorre todos os filhos de um nodo ele guarda a informação de seu custo e usa essa informação para o caso de uma outra atividade depender dele, evitando assim, percorrer novamente todo o subgrafo de um nodo com seu custo já calculado. O custo final do projeto sendo, portanto, o custo calculado do nodo inicial.

2.1 Descobrindo o nodo inicial

Para descobrir por qual nodo começar a percorrer o grafo deveríamos encontrar um vértice com grau de entrada zero, ou seja, nenhuma aresta deveria ter como destino aquele vértice.

Para isso, usamos uma estrutura de Set em que adicionamos todos os vértices do grafo no momento da leitura dos mesmos e ao ler as arestas descartamos do Set todos os vértices destinos, restando apenas o vértice que não houvesse aresta com destino ao mesmo, sendo esse o nodo inicial.

2.2 Detecção de ciclo

A presença de um ciclo no grafo inviabilizaria o cálculo do custo do projeto, pois se uma atividade ABC dependente de outra FGH e FGH dependente de ABC não haveria como saber o custo dessas atividades e consequentemente, do projeto.

Para detectar um possível ciclo o algoritmo percorre o grafo em profundidade (junto com o cálculo do custo) e registra o status "1"em todos os nodos percorridos, considera-se que todos os nodos tenham status inicial "0". Quando o algoritmo termina de visitar todos os filhos de um nodo ele registra status "2"naquele nodo. Mas, se enquanto o algoritmo estiver percorrendo os filhos de um nodo, encontrar algum com status igual a "1"significa que o caminho leva a um ponto já percorrido e então é detectada a ocorrência de um ciclo no grafo. que, conforme a revisão da literatura nos mostra, acaba

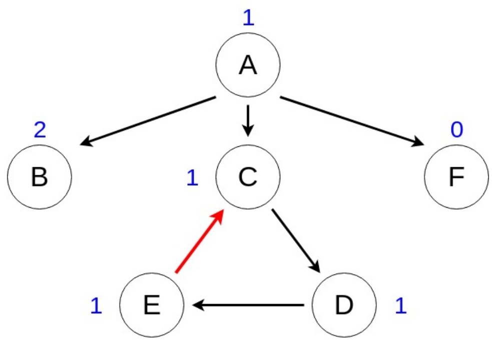

Figura 1: Ao percorrer os filho do nodo E o algoritmo detecta que o nodo C já está com status "1", indicando ciclo.

2.3 O algoritmo

Para calcular o custo de uma atividade o algoritmo percorre as atividade filhas recursivamente, caminhando em profundidade até que uma atividade não possua mais atividadades dependentes, quando isso ocorre significa que o algoritmo já calculou o custo total de uma tarefa e assim irá registrar esse custo, marcando o nodo como visitado e retornando para próxima atividade que ainda tenha filhos para serem calculados.

O algoritmo que calcula o custo do projeto, verificando também um possível ciclo, pode ser escrito da seguinte forma:

```
procedimento CALCULA_CUSTO_PROJETO(tarefa):
    nodo ← grafo.get_nodo(tarefa)
    se (nodo.visitado = verdadeiro): retorna nodo.custo_calculado
    nodo.status_ciclo ← 1
    para cada tarefa_filha em nodo.tarefas_filhas faca:
        nodo_filho ← grafo.get_nodo(tarefa_filha)
        se nodo_filho.status_ciclo = 1: gera excecao ("Grafo possuí ciclo")
        quantidade_repeticoes ← nodo.get_repeticoes(tarefa_filha)
        nodo.cuso_calculado ← nodo.custo_calculado +
                quantidade_repeticoes∗(CALCULA_CUSTO_PROJETO(tarefa_filha))

    nodo.status_ciclo ← 2
    nodo.visitado ← verdadeiro
    retorna nodo.custo_calculado
```

Resultados

Depois de implementar o algoritmo acima em Python obtivemos os seguintes resultados para os casos de teste dados no problema.

Caso de teste: 10 atividades
Tempo: 0.0000909 segundos
Resultado Final: 22706

Caso de teste: 100 atividades
Tempo: 0.0018784 segundos
Resultado Final: 380441814026746001850808315202

Caso de teste: 200 atividades
Tempo: 0.0058062 segundos
Resultado Final: 182832945051831624950529345863608760503550600332

Caso de teste: 400 atividades
Tempo: 0.016147 segundos
Resultado Final: 446985329942843776062350031346685068625985073721304298464727
926045647953

Caso de teste: 600 atividades
Resultado Final: Grafo Cíclico

Caso de teste 800 atividades
Tempo: 0.049968 segundos
Resultado Final: 5594652178285989998134059543749222654633051270102276919795492
2616843495029982728884241934596986334215409163035080004689

Caso de teste 1000 atividades
Tempo: 0.069042 segundos
Resultado Final: 1436266384739004255267672798310851016187488603393038182588671
91772865340108359630385195611583256808433792649539259698946044156010010

Conclusões

Podemos verificar através do que foi exposto que, inicialmente, a contribuição de uma primeira versão mais simplificada foi a de termos uma compreensão mais aprofundada do problema que estávamos propostos a resolver. Foi necessária essa compreensão aliada ao conhecimento das propriedades dos grafos e seus algoritimos de caminhamento para desenvolver uma solução final simples e eficiente.

O entendimento do problema nos levou a escolha do caminhamento adequado, e às adaptações que utilizamos, tais como guardar o custo total no nodo e o uso de um Set para seleção do vértice de início.

Foi possível perceber também, que a escolha do algoritimo de busca em profundidade nos possibilitou a resolução de camadas necessárias da proposição inicial, que foram a detecçaão de ciclos e a redução do tempo de execução. Essa se deu principalmente pela utilização da marcação de "visita"no nodo, qua acabou dividindo as arestas do grafo em arestas de descoberta e arestas de retorno, convergindo para uma complexidade linear *O(n+m)*, sendo *n* o número de vértices e *m* o número de arestas.

Um dos pontos que poderia ser abordado com mais profundidade em um trabalho futuro, seria o gerenciamento de memória das estruturas usadas, pois parte delas fica sem uso prático, no entanto ainda referenciada, após a visita do nodo e o cálculo de custo total.

Por fim, acreditamos que construimos uma solução simples, eficiente, de fácil compreensão e que atende a todos requisitos da proposição inicial.

Referências

[1] Cormen, T. H.; Leiserson, E. C.; Rivest, R. L.: **“Introduction to Algorithms”**. Mc-Graw Hill Book Co., The MIT Electrical Engineering and Computer Science Series, Cambridge, 1990.