Trabalho Prático 2 - Algoritmos 2

Leandro Diniz Silva[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brazil

leandrodinizsilva9@gmail.com

Abstract. *This meta-paper describes the implementation and a analises of the algorithm used in the knapsack problem. The solutions found to solve this np-hard problem were by backtracking and branch-and-bound.*

Resumo. *Este meta-artigo descreve a implementação e a análise do algoritmo do problema da mochila. As soluções usadas para esse problema np-difícil foram por backtrack e branch-and-bound.*

1. Introdução

Este trabalho prático tem como objetivo usar os algoritmos estudados em aula para produzir um programa capaz de atender as especificações do mesmo, com isso melhorando o entendimento por parte do aluno sobre a implementação de algoritmos np-difícies.

O problema escolhido nesse caso foi o problema da mochila binário, ou *knapsack problem*, para solucionar esse algoritmo foi implementado duas soluções diferentes, uma usando *backtrack* recursivamente e outra usando *branch-and-bound* interativamente.

2. Estruturas de Dados

A classe *Node*, foi a primeira a estrutura de dados a ser implementada, ela é usada como uma folha de um árvore com quatro atributos, esses são:

Level, salva o nível da folha
Valor, salva o valor da folha
Peso, salva o peso da folha
Limite, salva o limite superior da folha

Para salvar a fila de prioridades usadas no *branch-and-bound* foi usado uma *list* nativa do Python, que com suas funções *append* e *pop* se comportam igualmente a um *heap* implementado por bibliotecas externas.

3. Compilação e Uso

Todo o trabalho foi realizado em um ambiente *python* com auxílio do *jupyter notebook*, o arquivo disponibilizado no *Microsoft Teams* é um notebook que pode ser aberto e executado facilmente usando ferramentas como o *Google Collaboratory* ou o próprio

jupyter notebook.

Na função "*executaTodosTestes*" a mesma usa de um valor constante na variável *"testFiles"* em que a mesma busca os arquivos usados no teste seguindo da pasta raiz do notebook e buscando uma pasta na mesma localização chamada "Testes" e dentro dessa pasta os arquivos de testes tem que serem encontrados e seus valores precisam estar no formato passado nas especificações do trabalho práticos.

4. Implementação

Como foi citado anteriormente existem duas implementações diferentes usadas neste trabalho, o *backtracking* recursivo, que não ignora de nenhum caso específico e passa por todos os valores encontrados no arquivo e o *branch-and-bound* que analisa os limites superiores de cada folha de um árvore e de acordo com os mesmos corta casos que não precisam mais ser analisados pois seus valores não irão ser necessários no resultado final.

4.1. Implementação BackTracking

O algoritmo principal usado no *backtrack* passa por todos os valores de entrada, verificando se o peso analisado é maior que a capacidade da mochila ainda disponível no momento, se ela for ele pula a mesma e se não chama uma função *max*, que recupera qual o maior valor entre duas chamadas recursivas do próprio *backtrack*, onde a primeira chamada recursiva soma o valor do item atual mais o próximo item e a segunda chamada recursiva sem a soma do valor atual usando somente os valores do item adjacente.

Essas duas chamadas recursivas usadas no *max* são a parte do sistema que verifica qual a melhor escolha de item no momento, passando por todos os itens nos dois casos, onde um adiciona o item atual e outro onde ele só é ignorado, e o *max* seleciona o melhor valor dos resultados para retornar no final o melhor valor dos itens usados.

4.2. Implementação Branch-and-Bound

O *branch-and-bound* usa uma implementação interativa sobre uma fila, que termina sua execução quando a mesma se encontra vazia, além de um *Node* que auxilia na busca das informações de um item.

Para auxiliar o algoritmo os dados usados no mesmo ficam organizados de forma decrescente de acordo com o resultado de seu valor dividido pelo peso, onde o primeiro item encontrado no *array* é o que tem o maior valor por peso e o último o menor.

A fila se inicia em um estado em que ela tem um único item que tem todos seus valores iguais a zero, esse estado inicial é usado simplesmente para poder verificar o próximo item e seu adjacente. Nas verificações dos itens não iniciais é feito o cálculo do limite superior contendo o *node* atual e sem ele.

Com os limites superiores calculados a lista de prioridade e o valor do melhor

total é atualizado, se o peso atual permite adicionar um node com seu peso disponível no momento, o valor do total é atualizado e a folha é adicionada na lista de prioridades, mas logo após essa verificação com o node atual é feita a mesma verificação sem o mesmo, em que se essa condição for analisada como melhor, o total e a lista será atualizado com os novos valores.

Além da verificação do peso o que difere esse algoritmo do *backtrack* é sua habilidade de armazenar um resultado total no momento de execução em que se esse valor já for melhor que o limite superior do próximo *node* o mesmo é ignorado e todos os seus descendentes também são cortados, economizando cálculos desnecessários e tempo de execução do algoritmo.

4.3. Implementação Algoritmos Auxiliares

Foram implementadas algumas funções auxiliares para receber os valores e realizar os casos de teste dados nas especificações do trabalho prático, além de outros dois casos mais simples adicionados simplesmente para testar os resultados com valores já encontrados antes.

A primeira função auxiliar recebe o caminho para o arquivo contendo o caso de teste recebe seu nome, os valores da primeira linha para determinar a capacidade da mochila e quantos valores serão analisados no total e as demais linhas são convertidas para uma database em pandas com seus valores no formato *float64* e logo convertidas novamente para um *numpy array* que facilita os cálculos necessários em todo o sistema.

Outra função é a que executa todos os testes, a mesma recebe o caminho para a pasta contendo todos os arquivos de teste, recebe o caminho dos mesmos e sobre eles executa a função de transformar os mesmos em um *array*, cada *array* é passado sobre o *backtrack* e o *branch-and-bound*. Todos os valores encontrados nestes testes além do tempo de execução de cada é salvo em um arquivo *csv* chamado de “resultados.csv”.

5. Analisando a Diferença dos Algoritmos

Os resultados encontrados pelos dois algoritmos se encontram no arquivo chamado “resultados.csv”, o mesmo se encontra com um comentário em que o meu nome e minha matrícula na UFMG são exibidos seguido de uma linha com o cabeçalho dos dados encontrados, esses são:

Teste, nome do arquivo de teste
Resultado BackTrack, resultado encontrado pelo algoritmo *backtrack*
Tempo BackTrack, tempo total da execução do algoritmo no método *backtrack*
Resultado BranchBound, resultado encontrado pelo algoritmo *branch-and-bound*
Tempo BranchBound, tempo total da execução do algoritmo no método *branch-and-bound*

A (Figura 1) nos mostra um exemplo do resultado ao analisar todos os testes disponibilizados, é possível verificar que nestes testes o *branch-and-bound* encontra na maioria dos casos o resultado em um tempo menor comparado ao *backtrack,* em alguns casos chegando a encontrar em até três vezes mais rápido.

```
#Nome: Leandro Diniz Silva
#Matricula: 2019105718
Teste;Resultado BackTrack;Tempo BackTrack;Resultado BranchBound;Tempo BranchBound
f10_l-d_kp_20_879;798.0;0.004012107849121094;679.0;0.0
f1_l-d_kp_10_269;293.0;0.0;114.0;0.0
f2_l-d_kp_20_878;878.0;0.003011465072631836;678.0;0.0010023117065429688
f3_l-d_kp_4_20;28.0;0.0;35.0;0.0
f4_l-d_kp_4_11;23.0;0.0;22.0;0.0
f5_l-d_kp_15_375;408.81999933719635;0.0010025501251220703;96.529998421669;0.0
f6_l-d_kp_10_60;50.0;0.0;33.0;0.0
f7_l-d_kp_7_50;79.0;0.0;88.0;0.0
f8_l-d_kp_23_10000;9552.0;0.0010023117065429688;9673.0;0.0010046958923339844
f9_l-d_kp_5_80;118.0;0.0;130.0;0.0
teste.txt;280.0;0.0;160.0;0.0
teste2.txt;52.0;0.0;25.0;0.0
```

Figura 1. Exemplo csv

Uma observação a ser feita é que a função usada da biblioteca *time* para calcular o tempo retorna zero se o tempo que a função demorou a ser executada for menor que um décimo de segundo.

Olhando somente a (Figura 2) é possível também reparar em um caso em que a execução do *branch-and-bound* demorou o mesmo tempo que a execução do *backtracking* , isso aconteceu porque esse caso em específico tem seus valores muitos próximos uns dos outros fazendo com que o *branch-and-bound* entre no seu pior caso em que ele tem o mesmo custo do *backtracking*, pois precisa percorrer todos os valores sem "cortar" nenhum em toda sua execução.

```
f8_l-d_kp_23_10000;9552.0;0.0010023117065429688;9673.0;0.0010046958923339844
```

Figura 2. Exemplo f8_1-d_kp_23_10000

6. Conclusão

Com todos os dados analisados nesse trabalho é possível perceber que os algoritmos np-difíceis podem ter implementações simples de um algoritmo guloso, como o *backtracking,* que encontram um resultado para o problema, mas o algoritmo pode ser melhor desenvolvido evitando casos desnecessários fazendo com que o tempo para o mesmo ser concluído seja reduzido sucintamente e mesmo assim essa otimização pode cair em piores casos que a mesma não é nada melhor que sua contraparte gulosa.

Em resumo, problemas np-difíceis podem ter soluções ótimas difíceis de serem

encontradas, mas se as mesmas não forem obrigatórias podem existir algoritmos gulosos que conseguem resolver o problema

7. Referências

Levitin, Anany. (2012). Introduction to The Design & Analysis of Algorithms. 3th edition, pages 116–119, 436-438.