Trabalho Prático 2 – Problema da mochila com backtracking e branch-and-bound

Victor Hugo L. do Nascimento[1]

[1]Instituto de Ciências Exatas – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, MG, Brasil

vhln10@ufmg.br

Abstract. *This document describe two exact algorithms to solve the np-hard knapsack problem. This project discuss the implementation of the algorithm as well as a comparative analysis between both methods used with respect to their execution time for the given examples.*

Resumo. *Este trabalho descreve dois algoritmos exatos para solucionar o problema np-difícil da mochila binária. Foi discutida sua implementação e também uma análise comparativa entre os dois métodos utilizados a respeito do tempo de execução para os casos propostos.*

1. Introdução

O problema da mochila binária consiste em determinar a melhor solução para a escolha de um subconjunto de itens de forma a otimizar seu valor tendo o peso dos elementos como restrição. Neste trabalho é apresentado duas soluções exatas para esse desafio. Uma usando a técnica de backtracking e outra usando branch and bound. A seguir serão discutidas a implementação, ou seja, como o código funciona e foi feito, e a análise comparativa para determinar qual deles performar melhor dentre os casos propostos.

2. Implementação

Ambas as técnicas implementadas compartilham de uma base de dados quase idêntica. Os dois códigos lêem um arquivo csv contendos os dados e criam uma matriz. Além disso, a partir desses dados, é criada uma terceira coluna com o valor por unidade de peso do item. No caso específico do branch and bound também é adicionada uma linha de zeros na última linha da matriz. A explicação desse fato ficará clara no próximo tópico onde discutimos essa implementação com mais detalhes.

2.1. Branch and bound

Para implementar essa técnica foi seguido estritamente o código do slide da aula 10. A ideia é criar uma árvore onde seus nós armazenam o valor, o peso e a estimativa de qual valor o nó pode ter percorrendo esse caminho. A estimativa é feito supondo que o próximo item a ser adicionado possa encher a mochila com o maior valor por unidade de peso possível. Esclarecendo, supondo que a raiz seja um nó artificial no nível 0. Então no nível 1 é criado um nó supondo que o item com maior valor por unidade de peso está na mochila e outro nó supondo que ele não está. Em cada nível da árvore fazemos essa suposição para os filhos do nó anterior até analisarmos todos os items. Justamente por isso foi adicionado uma linha de zeros na matriz. Sem ela o último item do arquivo de

entrada não seria analisado. Agora com essa linha podemos fazer mais uma iteração, de forma que a estimativa para o próximo item depois do último é nula.

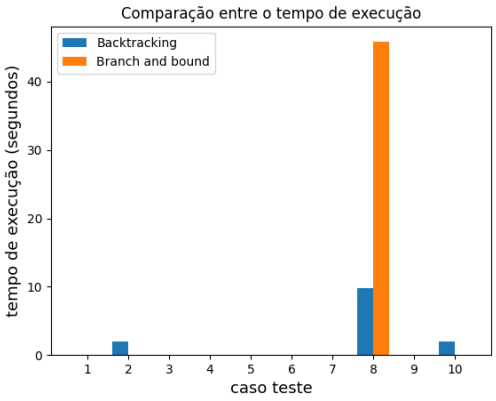
Falando o algoritmo em si, ele segue exatamente essa ideia. É criado um nó artificial e colocado em uma fila de prioridade (heap). Então verifica-se se todos os elementos já foram analisados. Se sim, então checa-se se o valor daquele nó é melhor do que a melhor solução atual. Mas, se ainda há elementos a serem analisados, é verificado se a partir daquele nó pode-se adicionar mais elementos dado sua restrição de peso e se faz sentido fazer isso baseado na estimativa do nó e no melhor valor da solução atual. Se sim, então é criado mais um nó adicionando o próximo item na ordem do valor por unidade de peso. Também cria-se um nó sem que o item em questão seja adicionado, caso a estimativa seja melhor do que a solução atual. Os nós que forem criado são adicionados na fila de prioridades, que é ordenada pela que tem maior estimativa. No código foi usada um valor negativo, isso se deve ao fato do heap ordenadar em forma crescente. Então pondo a estimativa como número negativo faz com a maior estimativa seja o primeiro elemento do heap (pois será um elemento menor devido ao sinal). O segundo campo do item adicionado no heap serve para critério de desempate de duas estimativas iguais. Caso contrário o heap não saberia qual elemento retornar resultando em erro. Dessa forma temos uma fila de prioridade ordenada pela estimativa de valor daquele nó. Por fim, o laço while retira o elemento de maior prioridade até que não haja mais nenhum no heap e então o algoritmo se encerra.

2.2. Backtracking

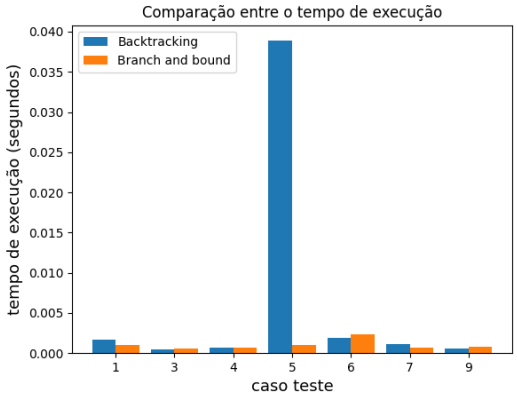
Aqui a ideia é mais simples do que na seção anterior. Baseando-se no código feito para o branch and bound, foi apenas necessário adaptar algumas ideias. Agora não é mais necessário fazer uma estimativa para cada nó, pois não há uma fila de prioridade. Em vez disso usa-se uma pilha, pois a ideia é explorar o um ramo da árvore até seu último nó e só então voltar ao topo. A pilha é perfeita para isso, pois, primeiramente, empilha os nós conforme são explorados até chegar onde todos os itens foram adicionados (caso a restrição de peso não impeça). Em seguida o backtracking volta desempilhando e, dessa foma, cria os nós que representam os ramos onde certos itens não foram adicionados. Sempre que se chega em um ramo onde todos os itens foram analisados, verifica-se se o valor do nó é melhor do que a solução atual.

3. Resultado dos testes

Nesta seção vamos analisar os resultados dos testes para ambas implementações. O gráfico abaixo mostra a diferença do tempo de execução entre os casos testes. No eixo x os números 1, 2,..., 10 representam os casos testes f1_ł-d__0_69, f2_ł-d__0_78,..., f10_ł-d__0_79. Ou seja, o número do eixo x representa o caso que tem o mesmo número logo após a primeira letra do nome do arquivo do teste.



Fica evidente que o caso 2, 8, 10 tem um tempo de execução muito maior e por questão de escala deixam os outros invisíveis. Abaixo um novo gráfico sem os casos mencionados anteriormente.



Nota-se como a diferença entre as implementações é insignificante para os casos pequenos com menos de 10 itens para completar a mochila. A diferença surge no caso 5 que tem 15 itens, onde o branch and bounch performar mais rapidamente.

De modo geral, os casos tem desempenho semelhante por serem pequenos. Em casos grandes o branch and bound tende a ser melhor, pois normalmente consegue cortar mais galhos da árvore usando sua estimativa, o que evita fazer uma busca por um ramo infrutífero. Entretanto, no caso 8 ocorre o contrário, onde o backtracking performar muito melhor. Isso ocorre, provavelmente, porque a estimativa do branch and bound está cortando poucos galhos. Dessa forma quase toda a árvore está sendo explorada. Isso também ocorre no backtracking, a diferença é que na outra implementação temos uma fila de prioridade e como poucos ramos estão sendo cortados muitos nós estão sendo inseridos no heap. O pior caso de inserção no heap é $O(\log n)$, devido a necessidade de ordenar o novo item adicionado. Isso faz com que o branch and bound leve mais tempo que o backtracking. Ou seja, normalmente o branch and bound performa melhor, exceto quando sua estimativa não corta muitos ramos. Nesse caso o backtracking pode ser preferível.

4. Detalhes e especificações

Para executar o código basta adicionar o arquivo no VS Code. Por padrão ele lê um arquivo com nome ”items.csv”. Ao final ele retorna, na primeira linha, o tempo de execução e, na segunda linha, o valor da solução encontrada.

O arquivo csv contendo os resultados dos testes contém uma coluna adicional do que foi especificada no trabalho. A sua última coluna representa se o teste foi executado com backtracking ou branch and bound.

As implementações foram testadas no WSL:Ubuntu-20.04 com Python 3.8.10 64-bit sem problemas para qualquer teste proposto.

5. Referências

Aulas 9 e 10 do curso de Algoritmos 2. Em especial o slide da aula 10 a respeito do problema da mochila binária.

https://www.sbc.org.br/documentos-da-sbc/category/169-templates-para-artigos-e-capitulos-de-livros