Lab Report Fabian Nilsson DVA 264 MDU 2023

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/7c96472a-2a73-4370-98eb-3298a581f1ec/Assignment_1.pdf

1. Explanation of the problem

a.

The goal of the knapsack problem is to find an assignment of values to the variables (selecting items) that maximizes the total benefit or value while satisfying the capacity constraint.

The solution of the knapsack problem can be represented as a list of binary digits with a list-length of n, where n corresponds to the quantity of items, in which the value 1 indicates that the respective item is included within the knapsack, while 0 denotes that it is excluded.

Set of variables:

X = {x1, x2, x3, ...,xi, …, xn}

where X is the set, each xi corresponds to a specific item, and n is the total number of items.

Domain

The variables in the knapsack problem are binary (0 or 1) and represent whether an item is included in the knapsack or not. Therefore, the domain for each variable xi is {0, 1}.

The domain can be expressed with the following notation:

For each item xi in the set X, the domain D(xi) is {0, 1}.

D = {D(xi) | xi ∈ X}

where X is the set of variables {x1, x2, x3, ..., xi, …, xn} and D(xi) = {0, 1} for each variable xi in X.

This is a logical representation of the problem as we use an imaginary decision tree where every right child exclude the current item, and every left child include the current item. Included items are represented with the binary digit 1, and excluded items are represented with the binary digit 0. By backtracking the path of the leaf that has the highest benefit and is below the weight limit, we find the optimal solution. The optimal solution can therefore be represented as following lists for BFS and DFS:

[1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1] for BFS

The solution has the benefit of 307 and weight of 419.
[1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1] for DFS

The solution has the benefit of 307 and weight of 399.

b

The objective function is to maximize the total benefit of the items selected to be placed in the knapsack. The equation of the objective function will therefore be written as:

f(solution) =∑(benefit_i * xi) for i in items

Where benefit_i is the benefit of item i, and xi is a binary variable that indicates if item i is included in the knapsack.

c. constraints

The main constraint is the max-capacity W of the knapsack, which limits the total weight of the items that can be put inside of it. The equation for the constraint C can therefore be written as:

C = {Σ(weight_i * xi) | xi ∈ X, i = 1, 2, ..., n} ≤ W

Where weight_i is the benefit of item i, xi is a binary variable that indicate if item i is included in the knapsack, and W is the maximum weight capacity of the knapsack.

The constraint that each item can be included in the knapsack at most once is implicitly enforced by the binary nature of the xi variables, since the domain D(xi) = {0, 1} for each variable xi in X, that contraint is already expressed in the domain.

d

The branching factor is 2 because at each node, there are two possibilities. We can include the current item or exclude it. The maximum depth (m) is the number of items, as each item has its own level in the tree.

2. Comparison of the algorithms

Lets assume that:

b = maximum branching factor, maximum number of actions.

d = depth of the shallowest solution in the search tree

m = maximum depth of the search tree

l = depth limit

Now lets explore the time and space complexity of DFS and BFS.

The time complexity of DFS is:

$$O(b^m)$$

It’s space complexity is:

$$O(b * m)$$

The time complexity of BFS is:

$$O(b^d)$$

it’s space complexity is:

$$O(b^d)$$

In the assignment problem, depth first search logically have better time and space complexity compared to breadth first search because of its ability to explore deeper in the search tree and prune branches more effectively. In this case, pruning refers to the process of eliminating branches of the search tree that do not need to be explored any further, as the weight capacity has already been reached and the benefit is not higher than the previous max benefit.

When it comes to space complexity, DFS uses a stack to keep track of the nodes to explore, hence only storing the nodes on the currenct path in the search tree. In contrast, BFS uses a queue and stores all the nodes at the current depth level in the tree. In the worst case, the number of node at the deepest level is much greater than the number of nodes on a single path. The conclusion of this is that BFS will often require more memory when storing nodes in the queue.

When it comes to time complexity, DFS clearly benefits from pruning branches in a more efficient way than BFS. As mentioned earlier, DFS can backtrack when the weight capacity is reached, which will save time. Similarly, DFS will backtrack immediately and avoid further exploration of a subtree if the algorithm has reached a lower bound that is not better than the current best solution. This will save considerable amount of time compared to the BFS algorithm. This is because the BFS algorithm has to explore all nodes at the current level of depth before moving to the next level, which means it cannot take advantage of early pruning as effectively as DFS.