CS301

2022-2023 Spring

Project Report

Group 048

Group Members

Ceren Dinç 28220

Alper Berber 28224

Problem Description

Intuitive Description:

The Bin Packing problem is a well-known optimization problem that requires packing a set of objects of varying sizes into the fewest possible bins (or containers) of a predetermined capacity.
Imagine that you need to fit a specific assortment of items into a specific number of boxes, and that you are aiming to use the least amount of boxes as possible. Each box has a fixed capacity, which means that the volume or weight it can hold is limited. Finding the most effective way to pack the items into the boxes to use the fewest number of boxes is the challenge.
Bin Packing problem can be used in a number of real-world contexts, including logistics, shipping, and storage, where reducing the number of containers used can result in time and resource savings.

Formal Description:

Input:

- A set of n items, where item i has weight w_i, for $i = 1, 2, ..., n$.
- A set of m identical bins, each with capacity c.

Output:

- An integer k that represents the minimum number of bins used.

Constraints:

- For all $i = 1, 2, ..., n$, we have $0 \leq w_i \leq c$.
- For all $j = 1, 2, ..., m$, we have $0 \leq c$.

Objective:

- Minimize k subject to the constraints above.

In other words, we want to find a way to assign each item to a bin such that the total weight of items in each bin does not exceed its capacity c, and we want to use the minimum number of bins possible.

Applications:

Logistics and Transportation:

- The Bin Packing problem is used in the logistics and transportation sectors to optimize the loading of cargo or freight onto vehicles like trucks, ships, and airplanes. The objective is to use the fewest possible containers or vehicles to transport the goods, which can decrease the cost of transportation and boost efficiency.

Manufacturing and Production:

- The Bin Packing problem is used in manufacturing and production processes to optimize the use of raw materials and decrease waste. Manufacturers can decrease the amount of unused materials and save on storage costs by packing the items into minimum number of containers.

Cloud Computing and Resource Allocation:

- The Bin Packing problem is used in cloud computing and resource allocation to optimize the allocation of virtual machines (VMs) among physical servers. Cloud providers can save money on infrastructure costs and power consumption by packing the VMs onto the minimum number of servers.

Cutting and Packing Materials:

- The Bin Packing problem is used to optimize material cutting and packing in sectors like textiles, metalworking, and woodworking to minimize waste and boost productivity.

Stock Management and Inventory Control:

- The Bin Packing problem is employed in the retail and distribution sectors to optimize product storage and distribution in warehouses and retail establishments. Retailers can lower storage costs and increase product availability by packing the products onto the fewest possible shelves or storage spaces.

Hardness of the Problem:

Theorem:

The Bin Packing Problem is NP-hard.

Proof:

The Bin Packing problem is NP-hard, which means that for large problem sizes, computing an optimal solution to this problem is computationally impractical.
In the book "Computers and Intractability: A Guide to the Theory of NP-Completeness" (1979), Garey and Johnson provided one of the earliest proofs of NP-hardness for the Bin Packing problem. By reducing the Subset Sum problem, which is known to be NP-Complete, into the Bin Packing problem, Garey and Johnson demonstrate that the problem is NP-hard. The Subset Sum problem can be reduced to the Bin Packing problem in polynomial time, meaning that the Bin Packing problem is NP-hard.

The Subset Sum Problem:

- Given a set of integers $S = \{s_1, s_2, \ldots, s_{n-1}, s_n\}$ and a target value t,
- Does there exist a subset of S, in which elements sum to t?

Reduction from the Subset Sum Problem:

Assume an instance of the Subset Sum problem is given, $S = \{s_1, s_2, \ldots, s_n\}$ and target value t,
To construct an instance of the Bin Packing problem:

- set the bin capacity to the target value t,
- for each s_i in the set S, create an item with size s_i
- let the number of items be n

It is claimed that reduction from the Subset Sum problem is valid and is an instance of the Bin Packing problem thus constructed is a yes-instance if and only if the Subset Sum problem instance is a yes-instance.

1. Assume that the Subset Sum problem instance has a solution (yes-instance), meaning that a subset S whose elements sum to t exists. Hence, it is possible to use a single bin with capacity t to pack elements in the given subset S.
2. Assume that the Bin Packing problem instance has a solution (yes-instance), meaning that n items can be packed into m bins with capacity t. Hence, it is possible to construct a subset of S by getting integers si which are items packed into the same bin. Since each bin has a capacity of t, integers in the constructed subset are sum to t.

Algorithm Description

```
bins = []
weights = [0.99, 0.01, 0.35, 0.06, 0.27]
best = 0
capacity = 1

def brute_force(index):
    count = 0
    for i in range(len(bins)):
        if bins[i] != 0:
            count += 1
    if index >= len(weights):
        if count < best:
            best = count
        return
    for i in range(len(bins)):
        if bins[i] + weights[index] <= capacity:
            bins[i] += weights[index]
            brute_force(index + 1)
            bins[i] -= weights[index]
```

The algorithm works by iterating over each bin and trying to add the next weight to it. If the weight can fit within the bin's capacity, the weight is added to the bin, and the function is recursively called with the index of the next weight to add. If the weight cannot fit in any bin, a new bin is created, and the function is recursively called with the index of the next weight to add.

The algorithm explores all possible combinations of adding weights to bins and keeps track of the minimum number of bins required to pack all the weights. This brute force algorithm implements the solution for bin packing problem in $O(n!)$ time.

Sample Generation (Random Instance Generator)

```
def create_test_case(size: int):
    # construct dataset
    temp_bins = [0.0] * size
    temp_weights = []
    for i in range(size):
        r = float(random.randint(0, 100)) / float(100)
        temp_weights.append(r)
    bins = temp_bins
    weights = temp_weights
    best = int(len(weights))
    return bins, weights, best
```

The code above generates random weights with the given size and creates empty bins. Since the maximum number of bins used can be as many as the size of *weights* input, *best* is set to it's local maximum *len(weights)*.

Algorithm Analysis

Theorem:

Subject to the restriction that each bin has a capacity of 1, the brute_force function for the Bin Packing problem determines the minimum number of bins required to pack a given set of weights.

Proof:

The brute_force function recursively tries all possible combinations of packing items into bins. Each item is either packed into a bin or is under consideration for packing into a bin is maintained as an invariant.

The first step of the algorithm is to determine whether every item has been packed into a bin. If all the items are packed and if fewer bins are used than the current best solution, it updates the best solution so far. If there are items that are not packed, the algorithm tries to pack the next item into each bin in turn, by recursively calling itself with the updated packing.

At each step, the algorithm determines whether the current item can be packed into each bin without exceeding the bin capacity, which is 1. If the capacity is not being exceeded, the item is packed into the bin and then the algorithm recursively calls itself with the updated packing. If the capacity is being exceeded, the algorithm tries to pack the item into the following bin.

Proof by Contradiction:

Assumption: The algorithm gives an incorrect solution. There exists a solution which uses fewer number of bins to pack the items.

As explained, the algorithm maintains an invariant that each item is either packed or being considered for packing into a bin. This demonstrates that the algorithm considers all possible packing of the items into bins and will eventually find the minimum number of bins required for the packing. This contradicts the assumption that the algorithm gives an incorrect solution.

By contradiction, it is proven that the brute_force function for the Bin Packing problem determines the minimum number of bins required to pack a given set of weights, subject to the restriction that each bin has a capacity of 1.

The Complexity Analysis:

The worst-case time complexity of the brute_force function is $\Theta(n!)$, where n is the total number of items.

The brute_force function tries to pack all possible combinations of items in all possible ways.

- The number of possible combinations of n items is n!.
- For each combination, the algorithm tries to pack the items into the available bins, which results in O(n) time.
- Thus, the worst-case time complexity of the brute_force algorithm takes $\Theta(n*n!)$, and simply $\Theta(n!)$.

Consider the case where the total number of items is n, each weighing 1, and k bins where k <= n. To pack the first item, any of the k bins can be used. For the second item, any of the remaining k-1 bins can be used. In each step, the number of available bins continues by decreasing by 1 for each item, until the last item is packed into the only available bin. In total, there are k * (k-1) * (k-2) * … * 1 possible combinations, which results in k! with only considering k items. Since n > k, all possible combinations of n items should be considered, which is n!. Therefore, the time complexity of the brute_force algorithm is $\Theta(n!)$.

Algorithm Implementation

The algorithm provided in the *Algorithm Description* part was used as the implementation. Tests were generated with the sample generation function provided in the *Sample Generation* section. Weights generated by the sample generation function are:

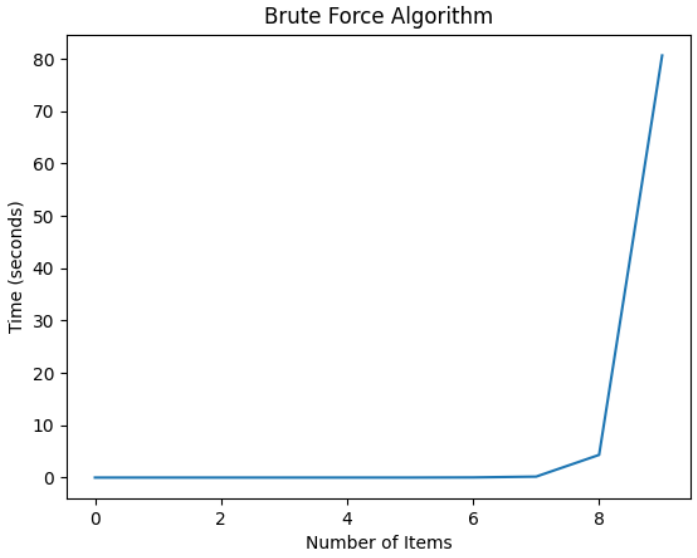
```
weights = [
    [],
    [0.65],
    [0.19, 0.01],
    [0.61, 0.83, 0.56],
    [0.06, 0.39, 0.27, 0.41],
    [0.77, 0.83, 0.68, 0.37, 0.53],
    [0.90, 0.45, 0.07, 0.55, 0.51, 0.44],
    [0.89, 0.65, 0.42, 1.0, 0.07, 0.01, 0.86],
    [0.17, 0.7, 0.43, 0.33, 0.35, 0.77, 0.23, 0.85],
    [0.72, 0.44, 0.1, 0.8, 0.32, 0.17, 0.68, 0.97, 0.24]
]
```

We kept the maximum size for our sample set to 9 since after 10 items, it takes ~45 minutes for 11 and for values bigger than 11 algorithm theoretically takes hours to find the solution.

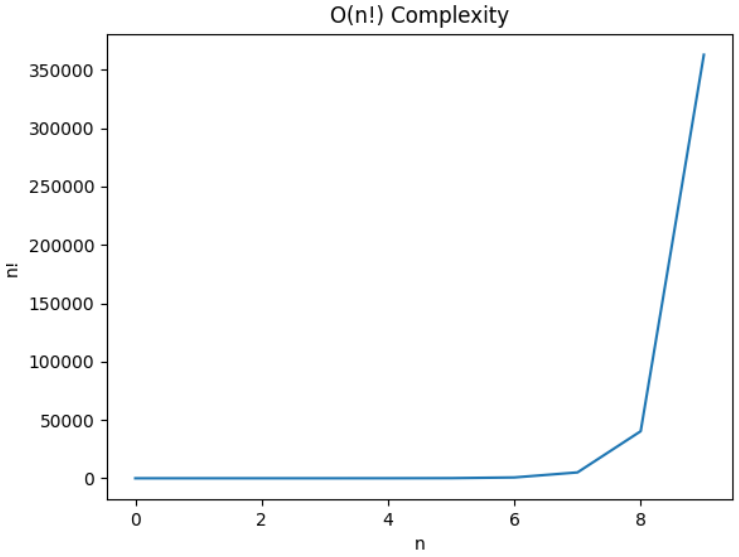
Performance tests of the brute force algorithm results are like this:

```
times = [
    0.000001907349,
    0.000010967255,
    0.000021696091,
    0.000025987625,
    0.000478029251,
    0.000792026520,
    0.027842998505,
    0.182782888412,
    4.333293676376,
    80.639649868011
]
```

Values are in seconds



This is the graph for the brute force algorithms results in terms of item size and their respective execution times.



This is the graph of *f(x)=x!* for *0 <= x <= 9*.

As you can see when two graphs are compares, they look exactly the same. This supports the fact that brute force algorithm works in O(n!).