Ocean Lu  
CS 3310   
Professor Damavandi / Johannsen  
12/03/2019

Homework #4: Report

Please view link for source code: <https://colab.research.google.com/drive/1A5GZejz7OMGhruiGJE0L3CTSy5Z-Wax7>

1. Does the branch-and-bound solution always perform better than the backtracking solution? If not, for what input did the backtracking solution check less nodes?   
   The branch-and-bound solution does not always perform better than the backtracking solution. The types of input the backtracking solution check less nodes are ones where the weight and value are already sorted when the data becomes mapped.
2. Did you find any inputs for which the brute-force algorithm is not much worse than either of the other two solutions?   
   The inputs for which the brute-force algorithm is not much worse than the other two solutions when there are less data inputs and when the data is already sorted.
3. On average, how much more efficient are the two solutions discussed in class over the brute-force solution?   
   The Backtracking solution may be effective dependent on the type of scenario, as it does iterate through the entirety of the tree, like Brute-force. The branch and bound solution has been by far the most efficient algorithm as it does not go through the entire tree, but focuses on bounded data sets.
4. After running your program on several different inputs, make a table displaying the # of nodes checked for each of the three solutions. (running my program on class exercises?)

|  |  |
| --- | --- |
| Test #1 (Lecture 20 Ch 6 Pt 1 Exercise slide) | |
| Data Items: | [‘1’, ‘2’, ‘3’, ‘4’, ‘5’] |
| Data Weight: | [2, 5, 7, 3, 1] |
| Data Value: | [20, 30, 35, 12, 3] |
| Data Mapped: | [('1', 2, 20), ('2', 5, 30), ('3', 7, 35), ('4', 3, 12), ('5', 1, 3)] |
| Max weight: | 13 |
| The Brute Force Solution: | |
| Nodes Visited | 31 |
| The Backtracking Solution: | |
| Nodes Visited | 13 |
| The Branch and Bound Solution | |
| Nodes Visited | 6 |
| Solution | |
| Total weight: | 13 |
| Total value: | 70 |
| Items: | 1, 3, 4, 5 |

|  |  |
| --- | --- |
| Test #2 (Lecture 18 Ch 5 Pt 2) | |
| Data Items: | [‘1’, ‘2’, ‘3’, ‘4’, ‘5’] |
| Data Weight: | [5, 6, 10, 11, 16] |
| Data Value: | [1, 1, 1, 1, 1] |
| Data Mapped: | [('1', 5, 1), ('2', 6, 1), ('3', 10, 1), ('4', 11, 1), ('5', 16, 1)] |
| Max weight: | 21 |
| The Brute Force Solution: | |
| Nodes Visited | 31 |
| The Backtracking Solution: | |
| Nodes Visited | 43 |
| The Branch and Bound Solution | |
| Nodes Visited | 5 |
| Solution | |
| Total weight: | 21 |
| Total value: | 3 |
| Items: | 1, 2, 3 |

|  |  |
| --- | --- |
| Test #3 (Lecture 18 Ch 5 Pt 3) | |
| Data Items: | [‘1’, ‘2’, ‘3’, ‘4’] |
| Data Weight: | [2, 5,10, 5] |
| Data Value: | [40, 30, 10, 5] |
| Data Mapped: | [('1', 2, 40), ('2', 5, 30), ('3', 10, 50), ('4', 5, 10)] |
| The Brute Force Solution: | |
| Nodes Visited | 15 |
| The Backtracking Solution: | |
| Nodes Visited | 18 |
| The Branch and Bound Solution | |
| Nodes Visited | 5 |
| Solution | |
| Total weight: | 12 |
| Total value: | 90 |
| Items: | 1, 3 |

|  |  |
| --- | --- |
| Test #4 (My own values) | |
| Data Items: | ['map', 'compass', 'water', 'sandwich', 'glucose', 'tin', 'banana', 'apple', 'cheese', 'beer', 'suntan cream', 'camera', 't-shirt', 'trousers', 'umbrella', 'waterproof trousers', 'waterproof overclothes', 'note-case', 'sunglasses', 'towel', 'socks', 'book'] |
| Data Weight: | [9, 13, 153, 50, 15, 68, 27, 39, 23, 52, 11, 32, 24, 48, 73, 42, 43, 22, 7, 18, 4, 30] |
| Data Value: | [150, 35, 200, 160, 60, 45, 60, 40, 30, 10, 70, 30, 15, 10, 40, 70, 75, 80, 20, 12, 50, 10] |
| Data Mapped: | [('map', 9, 150), ('socks', 4, 50), ('suntan cream', 11, 70), ('glucose', 15, 60), ('note-case', 22, 80), ('sandwich', 50, 160), ('sunglasses', 7, 20), ('compass', 13, 35), ('banana', 27, 60), ('waterproof overclothes', 43, 75), ('waterproof trousers', 42, 70), ('water', 153, 200), ('cheese', 23, 30), ('apple', 39, 40), ('camera', 32, 30), ('towel', 18, 12), ('tin', 68, 45), ('t-shirt', 24, 15), ('umbrella', 73, 40), ('book', 30, 10), ('trousers', 48, 10), ('beer', 52, 10)] |
| The Brute Force Solution: | |
| Nodes Visited | 4194303 |
| The Backtracking Solution: | |
| Nodes Visited | 781 |
| The Branch and Bound Solution | |
| Nodes Visited | 22 |
| Solution | |
| Total weight: | 396 |
| Total value: | 1030 |
| Items: | banana, compass, glucose, map, note-case, sandwich, socks, sunglasses, suntan cream, water, waterproof overclothes, waterproof trousers |

Code Analysis:

1. The Brute-Force solution that implicitly builds the entire state space tree.
   1. Code:  
      A screenshot of a cell phone

      Description automatically generatedA screenshot of a cell phone

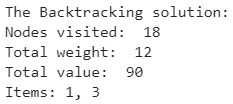
      Description automatically generatedA screenshot of a tree

      Description automatically generated
   2. Explanation: The anycomb function allows me to return combinations of any length from the items. The totalvalue function allows me to totalize a combination of items. By choosing the max value between a set of nodes after iterating through all the nodes (all the possibilities), it allows me to see and choose the best case.
   3. Output:   
      A screenshot of a cell phone

      Description automatically generated
2. The Backtracking solution.
   1. Code:   
      A screenshot of a cell phone

      Description automatically generatedA screenshot of a social media post

      Description automatically generatedA screenshot of a cell phone

      Description automatically generated
   2. Explanation: The totalvalue function allows me to totalize a combination of items. If weight of the nth item is more than Knapsack of capacity W, then this item cannot be included in the optimal solution.
   3. Output:  
      
3. The Branch-and-Bound solution. Make sure to use the best-first search version.
   1. Code:   
      A screenshot of a social media post

      Description automatically generatedA screenshot of a cell phone

      Description automatically generated
   2. Explanation: We first sort the data based on their efficiency (value/weight). The class state allows us to utilize a list of markings if a task is taken, or available. We define the upper bound with fractional knapsack. The initial upperbound is defined as 0. We want to accumulate weight used to stop the upperbound summation. In the develop(self) function, if it is not overweight, we go towards the left child. Otherwise, we go towards the right. Our root state is defined where we start off with nothing. We initialize a waiting\_states list, which will define a list of states waiting to be explored. We sort the waiting\_states list based on their upperbound, and explore the one with the largest upperbound, implementing the Greedy algorithm.
   3. Output:   
      A screenshot of a cell phone

      Description automatically generated

When the user starts the program, they should be asked for the following input:

* How many items are there to potentially take?
  + Code: A screenshot of a cell phone

    Description automatically generated
  + Output: (highlighted is user input)  
    
* What is the weight and profit of each item?
  + Code:   
    A screenshot of a cell phone

    Description automatically generated
  + Output: (highlighted is user input)  
    A close up of text on a white background

    Description automatically generated
* What is the max weight the bag can hold?
  + Code:  
    A close up of a logo

    Description automatically generated
  + Output: (highlighted is user input)  
    
* Then, I map the data in such that I can use:
  + Code:  
    A screenshot of a cell phone

    Description automatically generated
  + Output:  
    