BSc verslag Technische Wiskunde

Het schatten van de duur van het Branch and Bound algoritme voor 0-1 Knapsack problemen.
Estimating the duration of the Branch and Bound algorithm for 0-1 Knapsack problems.

Jelle de Jong

Technische Universiteit Delft

Contents

1 Abstract

The Branch and Bound method is a very useful method for finding solutions to optimization problems. However it does come with some problems as the search tree used for the Branch and Bound algorithm can be very large in size and can result in very long processing times as well as a large memory use. There is currently no known way to exactly know how large this tree will be and how long the algorithm will take, therefore having more information on this algorithm can be of great interest.

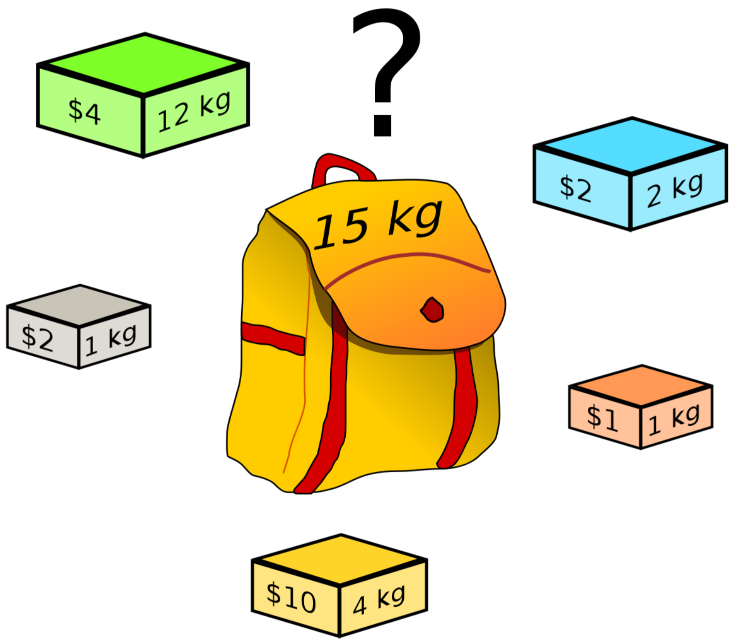
This report will look into the method used in the article Early Estimates of the Size of Branch-and-Bound Trees by Gerard Cornuéjols, Miroslav Karamanov and Yanjun Li [1]. In this report their method will be tested against a specific set of problems, the 0-1 Knapsack problems. Their time estimation method will be tested against a variety of 0-1 Knapsack problems and the estimation will be compared to the real time it takes to solve the problems. This will give an idea of how good the method works for the method works for the 0-1 Knapsack problems and show any issues it might run into.

2 Branch and Bound algorithm for 0-1 Knapsack problems

2.1 0-1 Knapsack Problems

The knapsack problem can be best explained by imagining a real knapsack. The knapsack only has the capability to carry a certain weight of items. There is a number of items, which each have their own weight and a price. This price indicates how much value you would give bringing this item with you in the knapsack. The goal is to bring an as high as possible value of items with you, while not going over the maximum weight constraint the knapsack has.

In the case of a 0-1 knapsack problem, each item can only be picked at most once, so there are no extra copies of items available to pick. A variable x_i is created, this x_i is equal to 1 if the item in included in a solution, and is 0 when an item is not included in the solution.



Source: `https://en.wikipedia.org/wiki/Knapsack_problem#/media/File:Knapsack.svg`

This is an example of what a simple knapsack problem can look like. There is a set of 5 items which each have their own value and weight. The optimal solution would be to carry as much value as possible while staying under the weight limit of 15 kg. In this case of a simple problem like this, it can easily be seen that the optimal solution would include all boxes except the green one. This would stay under the weight limit since it only reaches a weight of 8 kg, while having a value of $15.

2.2 ILP Problems

This knapsack problem is what is known as an ILP problem or Integer Linear Programming problem. A minimization or maximization optimization problem with linear constraints and a linear objective function. Each problem consists of a set of items, all of these items have two properties, a value and a weight. The value of item i will be called v_i and the weight of the item will be called c_i.

Each problem will have an objective function, in this case the objective function is to maximize the sum of $v_i x_i$ for all i, which would be the cumulative value of all the items included in the knapsack. There is also a restriction on the problem, the maximum weight that can be carried, this value will be indicated by b. Now a general knapsack problem can be defined in the following way:

$$
\begin{aligned}
\max \ & \sum_{i=0}^{n} v_i x_i \\
& \sum_{i=0}^{n} c_i x_i \leq b \\
& \quad x_i \in \{0,1\} \qquad\qquad i = 0, 1, ..., n
\end{aligned}
$$

2.3 LP relaxation

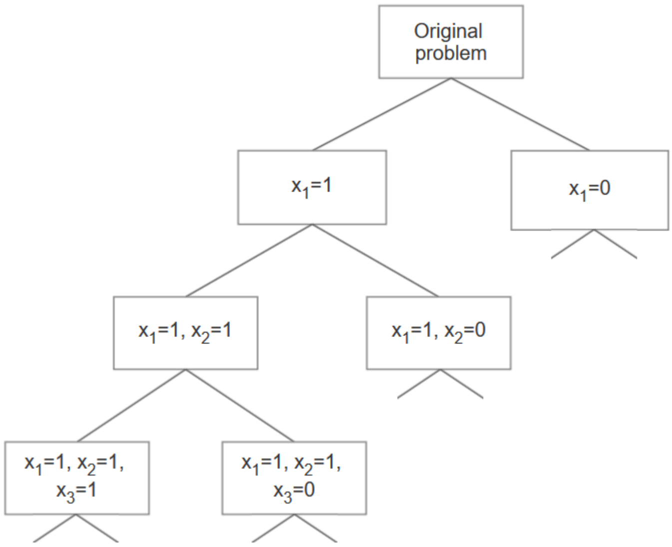
An LP relaxation is a simplified version of an ILP problem, the integral constraint is dropped from the problem. This means that the variable x_i can now take values between 0 and 1 as well.

$$\begin{aligned} \max \sum_{i=0}^{n} v_i x_i & \\ \sum_{i=0}^{n} c_i x_i \leq b & \\ x_i \in [0,1] & \qquad\qquad i = 0, 1, ..., n \end{aligned}$$

With this the exact weight limit b can always exactly be hit allowing for a potentially more optimal solution than the ILP would allow. This means the LP relaxation of the problem will give an upper bound to the maximum value the ILP problem can reach.

3 Branch and Bound algorithm

The Branch and Bound method is based on the idea of intelligently enumerating over the feasible points of a combinatorial optimization problem. [5] Branch and Bound breaks a problem into a series of smaller problems that are easier to solve, and the information from the smaller problems help solve the main problem. This is typically done using an enumeration tree. In the case of the Knapsack problem, a binary enumeration tree is created, this tree is created by first splitting the main problem into two subproblems: One with the first item included in the solution ($x_1 = 1$), and one with the first item excluded from the solution ($x_1 = 0$). Then each of these subproblems gets split again into two new subproblems with the second item included ($x_2 = 1$) or excluded ($x_2 = 0$). This can be repeated for every single item for a tree consisting of $2^{n+1} - 1$ subproblems. The binary enumeration tree will be referred to as the search tree.



Solving $2^{n+1} - 1$ subproblems becomes impossible to solve for larges values of n. [3] So a more intelligent way of looking at subproblems is needed.

3.1 Using bounds to prune subproblems

This intelligent way of looking at subproblems is done by calculating a lower bound and upper bound to the solution every time a new subproblem is created. Then if a subproblem is created that has an upper bound that is worse than the lower bound of another subproblem, this new subproblem can not lead to the optimal solution of the original problem. Because of the structure of the enumeration tree created, none of the new subproblems coming from this original problem can lead to the optimal solution either so a large part of the tree can be pruned this way.

3.2 Depth First Search

Pruning subproblems is very important for speeding up the Branch and Bound algorithm so it is of interest to prune non-optimal subproblems as soon as possible. To make this process faster, it is important to find a high lower bound as soon as possible, as many new subproblems can be found to be infeasible from that point on. This is why a Depth-First Search strategy is used. [4] This means that when choosing a subproblem to split into new subproblems, the best option would be to take the subproblem that previously had the highest lower bound, as the problems created from this are more likely to improve that bound. Having this high lower bound makes it much more likely that new subproblems can be found to be incapable of being an optimal solution, and therefore increasing the speed of the algorithm. [2]

4 Time Estimation

4.1 The importance of time estimation

Estimating the duration of the algorithm can be important as bigger problems can easily take an extremely long time to complete. In case of the 0-1 Knapsack Problem, every item has two options, it is either included as an item in the final solution, or left out. In a problem with n items this would mean that there are 2^n possibilities of item combinations that could be a final solution. This also means that the Branch and Bound algorithm has a worst case running time for $O(2^n)$.

Knapsack problems can easily contain thousands or more items in real world applications. If a program could check millions of possibilities per second, a computer would still not ever in millions of years be able to check all 2^{1000} possibilities. This means that there is a chance that you will not be able to solve this problem in any reasonable time. However there is also a possibility the program will be able to solve the problem in a fraction of a second.

It can be important to know in advance if the solution is easily solvable before you start working on a solution. Having an estimation in advance will give a lot of insight in the problem and what can be expected.

4.2 Overview of the model

The model used for estimating the duration of the Branch and Bound method for 0-1 Knapsack problems is based on the article: "Early Estimates of the Size of Branch-and-Bound Trees" by Gerard Cornuéjols, Miroslav Karamanov and Yanjun Li [1]
The model starts by running the Branch and Bound method as normal to find the optimal solution for a set problem. It will continue doing so until two requirements are met: At least five seconds have passed, and the amount of nodes checked is at least 20 times the depth of the tree created thus far. From here the total tree of subproblems that have been looked at will be summarized into a couple of key variables. Those variables will then be used to make a size estimation of what the full tree would look like if the program would keep running until an optimal solution is found. Using this size and an average computation time per subproblem up to that point. An estimate for the running time can be made.

4.3 Branch and Bound algorithm in the model

The Branch and Bound solver uses a method to solve the algorithm by first sorting the items by the best value over weight ratio. This way the most value efficient items will be checked first as these are very likely to be part of the final solution.

It starts with a single problem, the original problem we want to eventually solve. Then upper and lower bounds will be created for this problem. The lower bound is calculated by adding the items in the list in order until the next item does not fit anymore. The items are ordered by best value to weight ratio, so this creates decent lower bound for the problem. The upper bound is the calculated by finding an optimal solution to the LP-Relaxation of the problem. The LP-relaxation can always find a solution that is equal or better than the solution the ILP problem could give, so this is guaranteed to be an upper bound of the actual solution.

4.4 Pseudocode of the Branch and Bound Algorithm

1. Order the items from highest value over weight to lowest.

2. Calculate the upper and lower bounds of the problem and mark the problem as checked. Set the highest lower bound to the calculate lower bound.

3. Split the problem into two new subproblems with the first item on the list included or excluded.

4. Repeat until there are no unchecked problems:

   (a) Pick the subproblem with the highest depth in the search tree that has not yet been checked.

   (b) Calculate the upper and lower bounds of the subproblem. If the new lower bound is higher than the previous highest lower bound, make this the new highest lower bound.

   (c) If the upper bound is lower than the highest lower bound: Go to the next unchecked problem.

   (d) If the upper bound is higher than the highest lower bound: Split the subproblems into two new subproblems with the next item in the list included and excluded, then go to the next Unchecked subproblem.

5. The subproblem with the highest lower bound will have the optimal solution to the original problem.

5 Estimating the duration of the algorithm

The goal is to find a time estimation for the algorithm based on running part of the Branch and Bound algorithm. This estimation will be based on the shape of the tree created by the subproblems that are looked at. Three properties describing the shape of the tree will be used to make the final estimation:

1. $w_T(i)$, will be the width of level l, this means the amount of nodes at level l of the tree.

2. d_T will be the depth of the tree, which is the lowest level of the tree that contains nodes.

3. l_T is called the last full level, before this level all levels will have exactly twice as many nodes and the tree will form a complete binary tree.

4. b_T is the waist of the tree, the level with the maximum width, in case there are multiple levels with the maximum width, b_T is defined as the average level of the lowest and highest level with this maximum width: $b_T = \frac{b_1+b_2}{2}$ where $b_1 = min\{i : w_T(i) = t\}$ and $b_2 = max\{i : w_T(i) = t\}$ with t being the maximum width.

5.1 Estimating the γ-sequence

A γ-sequence is defined as the sequence $\gamma_0, \gamma_1, ..., \gamma_{d_T-1}$ where $\gamma_i = \frac{w_T(i+1)}{w_T(i)}$ for $0 \leq i \leq d_T$ which shows the relative size difference between levels of the tree. It is another way to describe the way a tree looks.

A time estimation is made using the estimated size of the final tree described by the γ-sequence of the final tree. However the width values of the final tree are not known, so the γ-sequence will have to be estimated, for this a linear estimation will be used based on the properties derived from the partial tree.

This linear estimation is based on the fact that the γ-sequence is generally decreasing for i greater than the last full level. This sequence is also approximately 1 at the waist and is 0 at the deepest level. Using this a γ-sequence is defined using the following formula:

$$\gamma_i = \begin{cases} 2, & \text{for } 0 \leq i \leq l_t - 1 \\ 2 - \frac{i-l_t+1}{b_t-l_t+1}, & l_t \leq i \leq b_t - 1 \\ 1 - \frac{i-b_t+1}{d_t-b_t+1}, & b_t \leq i \leq d_t \end{cases}$$

A number of nodes at every level can now be estimated using $w^*_{i+1} = w^*_i \gamma_i$ with $w^*_0 = 1$. And adding all of these w^*_i values will give an estimation of the nodes in the tree.

5.2 Time estimation

With an estimation of the full size of the tree, the duration of the algorithm can finally be estimated. To do this, the average time per node up until the time of estimation is used. This average time per node can now be used with the estimated amount of nodes in the full tree to get a total estimated time.

This now gives an estimated time to compare to the real time, there will always be an error though, so this error is required to be limited in size. It is most important that the estimated time is of the same magnitude as the real time, if the estimate is 5 minutes the program should not take a full day to complete. A time range will be constructed of five times the estimation in which the estimation is considered to be of the same magnitude.
This means the requirement is that the real time is between 0.2 times and 5 times the estimated time. This lower bound can be slightly improved by making it the maximum of 0.2 times the estimated time and the already elapsed time when the estimation is made. Since it can obviously not be shorter than the already elapsed time.

6 Data analysis

The goal is to test the performance of the estimation method for 0-1 Knapsack Problems, therefore multiple types of Knapsack problems will have to be tested. Knapsack problems with a small number of items but with very small variety in the weights and values, and problems with a larger amount of items but also a larger difference between the weights and values each item can have.

These two problems will cause very different solution times as the algorithm has to check many more possibilities for the problems with a small variety in cost and weight. This will result in a very wide estimation tree but not a very deep one. Whereas in the problems with a large number of items the estimation tree will become very deep but it will not be very wide.

By testing this estimation method for a variety of different Knapsack problems, it can be properly tested how well it performs and when the method lacks accuracy.

6.1 Problems

Using the method created the estimation method can now be tested against a variety of different problems. It is important to test different types of problems as they can create different issues for the estimation method. The problems will be created by generating a list of random numbers for the weights and another list for the value of the items. These two lists will then be paired to create an item with both a value and cost. Now these items can be put into the program which will start solving the problem, make an estimate when the estimation tree fits the restrictions for estimation, and then continue solving the problem to create a final actual solving time.

6.2 Small Variety Problems

To keep the problems hard to solve the difference in value and weight between the items has to be small. These problems will be randomly generated to have the weight and value of each item in an interval of around $[0.9x, x]$ with x being a set value. This way the values for each item will be very close to each other making the ratio of value over weight very similar. It will be very difficult for the algorithm to find a solution as even the difference between the "best" and "worst" items is not that large.

6.3 Large size problems

These problems will have a large amount of items to see how the algorithm preforms when there are lots of possibilities. Overall these problems will have any more cuts in the algorithm so a lot of problems will not have to be checked. However the large amount should create extra problems for the estimation method as the linear estimation for the amount of nodes could easily heavily overestimate the amount of nodes

6.4 Test results

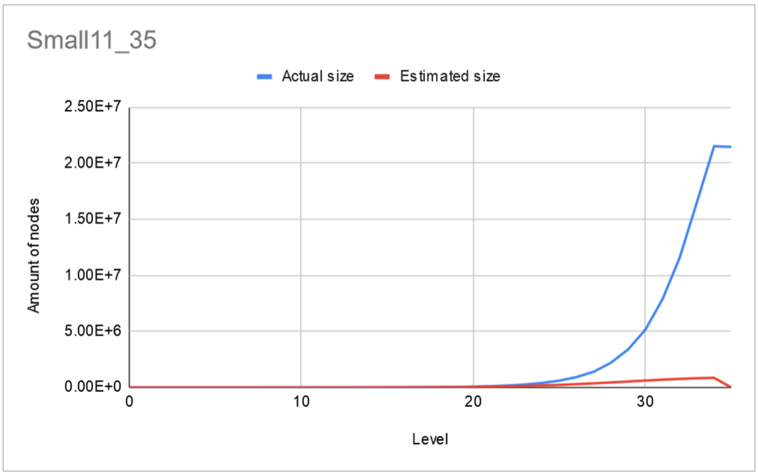
Problem	Algorithm Duration (s)	Time estimate interval (s)
Small1_30	247.582752	[5.008063554763794 , 71.93847929101015]
Small2_30	254.8848522	[5.0066423416137695 , 67.069068864531]
Small3_30	235.1424277	[5.0275163650512695 , 50.69532322382676]
Small4_30	229.1956658	[5.006619453430176 , 59.23833057770466]
Small5_35	1027.761663	[25.13548160951973 , 628.3870402379931]
Small6_35	896.1050351	[26.782633641928665 , 669.5658410482165]
Small7_30	529.3638763	[19.06282932966888 , 476.570733241722]
Small8_100	19.62864876	[6165736541437.09 , 154143413535927.25]
Small9_100	85.3552711	[5828753016363.98 , 145718825409099.53]
Small10_100	6.37242198	[1653025304221.0215 , 41325632605525.54]
Small11_35	1584.0622961521149	[20.082528583670687 , 502.06321459176706]
Large1_500	3431.878951	[8.006646766827597e+80 , 2.0016616917068997e+82]
Large2_500	102.8649545	[9.256545299339401e+80 , 2.3141363248348506e+82]
Large3_500	5.360163927	[8.399670225483611e+62 , 2.0999175563709023e+64]
Large4_600	213.7400203	[6.321416736974491e+97 , 1.5803541842436226e+99]
Large5_600	254.5948122	[2.0677817904892242e+95 , 5.1694544762230605e+96]

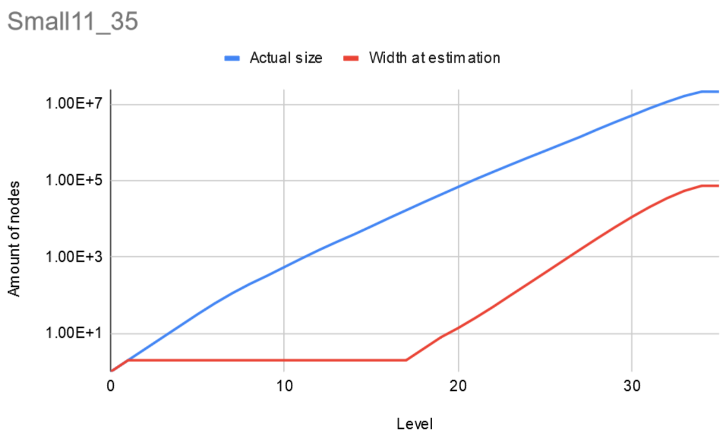
It seems that the estimation does not work properly as it the true algorithm duration does not fall within any of the intervals that were estimated during the tests. The large problems seem to have very extreme mistakes as they give incredibly high times while the actual problem sometimes only took a couple of minutes to solve. To see why these mistakes are happening, the problems need to be looked at at an individual scale.

Two problems will be looked at further to see what went wrong with the estimation. Small11_35 as it has a much longer solving time than estimated, and Large1_500 as it has the opposite problem, of the actual solving time being much lower than estimated.

6.5 Problem Small11_35

Small11 has an actual solving time of around 26 minutes, while the estimation interval only goes to about 8 minutes. When the linear size estimation made by the program is compared to the actual size once the problem is solved, it is clear that the problem was expected to be a lot smaller towards higher levels of the tree than it actually was. To see how this problem could arise, another look has to be taken at how the estimated size is created.

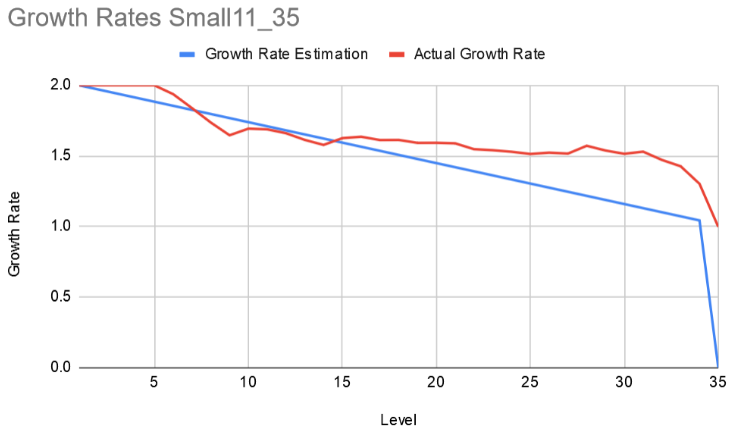




This causes issues for the three variables on which the estimation is based. The first value, the last full level l_t will be at 1, as not enough iterations have been done at lower levels to know anything more about this value. The next value, the level of maximum width b_t is set at 34.5 as both level 34 and 35 contain the same amount of nodes. The final value the depth of the tree is 35 which means that every single item in the list will have been considered for an optimal solution and no cuts have been made, this makes sense as the nature of the small problem is to create a problem that is relatively hard to solve.

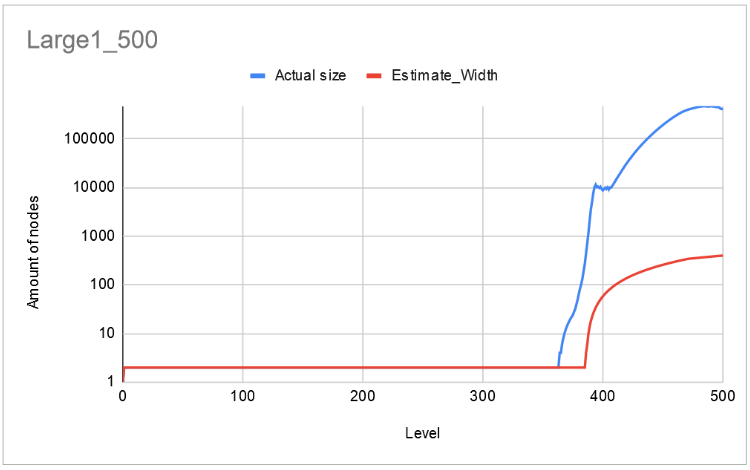
However since our estimation is only based on these three values, flaws of the estimation method are starting to show. The last full level is 1, so only for the first level will our estimation double in size. In the actual final width of the tree, this value ends up being 5. This means that our estimation does not grow as much at the start as the final solution does, resulting in a lower estimation.

The next problem is created because of the assumption the model would take a linear shape. This means that from l_t until b_t the growth between levels will slowly lower from a two times growth per level, to a value of one where the amount of nodes is stable. However when this is compared to the actual solution, the estimated growth rate is consistently under the actual growth rate.



6.6 Problem Large1_500

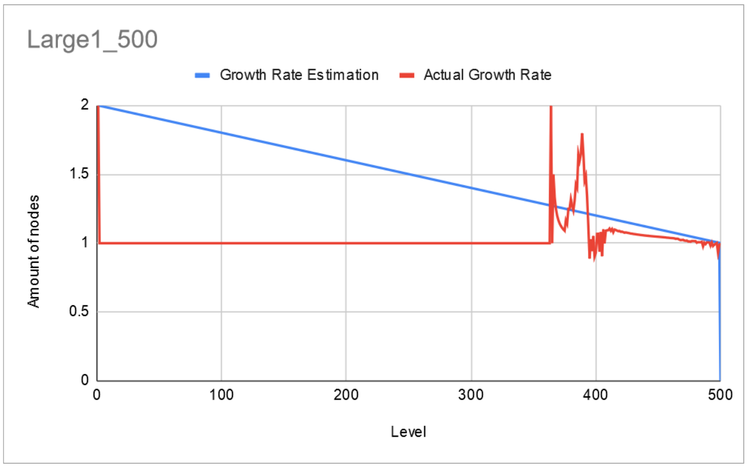
As with all of the estimations of large problems, the time estimates are extremely large. In this case a duration of around 10^{81} seconds is estimated by the program. Times that do not make any sense in the context of the problem. In reality the problem takes a little under an hour to solve. So the estimation method has some major issues with problems with a larger amount of items. To see what goes wrong with this problem, another look has to be taken into how the estimation is made.



Looking at the width levels at the time of estimation, this shape fairly closely resembles the final shape of the problem, although on a scale about a thousand times smaller than the final solution. This seems like a decent estimation of what the final shape could look like, however there is a problem with then making the time estimation. The time estimation is only based on the three variables: The last full level $l_t = 1$, the depth $d_t = 500$ and the level of maximum width $b_t = 499.5$.

The depth once again being the final level is not very surprising, since it is not unlikely that all items will be checked unless there are some severe outliers which will not even be taken into consideration by how bad they are. The last full level being 1 is actually correct with the final result since the first hundreds of levels will stay at a minimum value of a width of 2. The level of maximum width for the actual width of the tree is at level 485 which is also not very far off of the estimated level 499.5.

So the variables used for estimating the time duration seem good, however the final time estimation is not even close to the actual time duration. The problem lies in the assumption of linearity for the estimation. The growth rate is assumed to be linearly decreasing from 2 to 1 over the levels between l_t and b_t. However in this problem the growth rate actually stays at 1 for the first 385 levels. This means in the estimation, the width of each level grows enormously at the start, while in the problem stays very small. This huge increase assumed by the linearity causes a giant overestimation of the actual amount of nodes in the tree, and therefore the time estimation made for the problem.



Comparing the Growth Rate Estimation to the Actual Growth Rate, shows that this assumption of linearity does not at all work for the problem analyzed. It is clear that the Actual Growth rate moves around a lot and does not follow the shape a linear growth rate would have.

7 Conclusions

In both the smaller and larger problems, many issues occur preventing a proper time estimation using the method proposed in the paper "Early Estimates of the Size of Branch-and-Bound Trees" by Gerard Cornuéjols, Miroslav Karamanov and Yanjun Li when applied to 0-1 Knapsack Problems. The linear estimation does not seem to be a proper estimation method based on the actual growth rates found in the sample problems, and three variables does not give enough information about the shape of the tree the branch and bound algorithm creates when solving the problem.

Of course not only changing the linear estimation to another type of estimation could help improving the time estimation. Solutions like adding more variables that describe the behaviour of the tree, or adding more time before an estimation is made, could be made to improve the accuracy of the time estimation as well. However these solutions do come with a downside. The benefit of this method is that it is quickly able to give an estimation after about five seconds, and does not take a lot of computational power, adding more variables or more time loses a lot of these benefits. If an estimation is wanted, a good look has to be taken at the importance of the estimation and how much time they are willing to spend on making this estimation over just solving the problems.

Based on the results found in this paper, this method is not reliable enough to give a good time estimation that can be trusted. It would not be recommended to use this to estimate how long a branch and bound problem for 0-1 Knapsack problems would take to solve. If improvements were to be made to the method, it would be up to the user to determine if the extra time spend on making a better estimation is worth having the advanced knowledge of how long a problem will take, as the speed of estimation for this method is one of the big benefits of using it.

References

[1] G. Cornuéjols, M. Karamanov, Y. Li. 2004 Early Estimates of the Size of Branch-and-Bound Trees.

[2] L. A. Wolsey, 1998. *Integer Programming* p99 John Wiley and Sons, New York, NY.

[3] L. A. Wolsey, 1998. *Integer Programming* p9 John Wiley and Sons, New York, NY.

[4] D. E. Knuth, 1975. Estimating the efficiency of backtracking programs. *Mathematics of Computing* 29 121-136

[5] C. H. Papadimitriou, K. Steiglitz, 1998, *Combinatorial Optimization Algorithms and Complexity* p433 Prentice-Hall, Inc., New Jersey.

Attachments

```
import threading
from random import randrange
import time


class KnapsackInstance:
    def __init__(self, w, c, U):
        n = self.n = len(w)

        # We sort the weights and costs according to cost / weight
        # ratio.
        N = self.N = list(sorted(range(n), key=lambda i: -1.0 * c[i] / w[i]))
        self.w = [w[i] for i in N]
        self.c = [c[i] for i in N]
        self.U = U


def generate_random_instance(nitems):
    w = [randrange(866, 1001) for i in range(nitems)]  # Was xrange but python 3
    c = [randrange(866, 1001) for i in range(nitems)]
    return KnapsackInstance(w, c, 0.75 * sum(w))


# Create random weights and cost for every item, Capacity of weight U is 0.75* total sum of weights
# Returns a knapsackinstance as class above

class Subproblem:
    def __init__(self, fixed_one, fixed_zero, ub, depth):
        self.fixed_one = set(fixed_one)
        self.fixed_zero = set(fixed_zero)
        self.ub = ub
        self.depth = depth


# Items which are in/out of knapsack and upper bound.

class InfeasibleSubproblem(Exception):
    pass


def compute_bounds(inst, P):
    # inst is the random instance created (or any instance), P is the active nodes list.
```

```python
    lb = 0
    total_weight = 0

    # First we add up the fixed items.
    for i in P.fixed_one:
        lb += inst.c[i]
        total_weight += inst.w[i]

    if total_weight > inst.U:
        raise InfeasibleSubproblem

    # Then greedily fill up the rest to get a lower bound.
    items = []
    for i in range(inst.n):
        if i not in P.fixed_one and i not in P.fixed_zero:
            if total_weight + inst.w[i] <= inst.U:
                lb += inst.c[i]
                total_weight += inst.w[i]
                items.append(i)
            else:
                # First item not to fit.
                first_fail = i
                break
    else:
        # All items fit, so we have the optimal solution (upper bound
        # is equal to lower bound).
        return lb, lb, items

    # Compute the upper bound.
    ub = (lb + 1.0 * inst.c[first_fail]
          * (inst.U - total_weight) / inst.w[first_fail])

    return lb, ub, items


def check_estimate_periodically(width, estimate_width, initial_timeout_occured, estimated_time, start_
    initial_timeout_occured[0] = True
    if width[-1] != 0:
        first_zero_index = len(width)
    else:
        first_zero_index = width.index(0)
    if sum(width) > first_zero_index * 20:
        for i in range(len(width)):
            estimate_width[i] = width[i]
        estimated_time[0] = time.time()
```

```
        print(estimated_time[0]-start_time[0])
        print("5 second time limit exceeded and sum(width) > first_zero_index * 20, setting estimate_w


def do_branch_and_bound(inst):
    Pbest = 0
    start_time = []
    start_time.append(time.time())
    width = [1]
    for i in range(inst.n):
        width.append(0)
    estimated_time =[0]
    # Global upper and lower bounds, and best solution found so far.
    global_ub = sum(inst.c) + 1
    global_lb = -1
    best_solution = []
    depth = 0
    # After processing a node, the new global upper bound is the
    # maximum of the upper bounds of active nodes and integer
    # nodes. Since integer nodes get out of the list of active nodes,
    # we keep the maximum upper bound of integer nodes in the
    # following variable.
    integer_node_bound = -1

    # Initialization.
    active_nodes = [Subproblem([], [], global_ub, depth)]

    # Main loop.
    estimate_width = [0] * len(width)
    initial_timeout_occurred = [False]

    # Start 5 second timer, after that we check every iteration for estimate of appropiate size.
    threading.Timer(5, check_estimate_periodically, args=[width, estimate_width, initial_timeout_occur

    while active_nodes:
        Pbest = active_nodes[0]
        if initial_timeout_occurred[0] == True and estimate_width[0] == 0:
            print("Making an estimation is now possible.")
            check_estimate_periodically(width, estimate_width, initial_timeout_occurred, estimated_tim
            if estimate_width[0] != 0:
                print("")
                # The estimate was made!
```

```
        # Select an active node to process.
        for X in active_nodes:
            if X.depth > Pbest.depth:
                Pbest = X
        P = Pbest
        active_nodes.remove(P)
        #print(len(P.fixed_one)+len(P.fixed_zero))
        # Process the node.
        try:
            lb, ub, items = compute_bounds(inst, P)
        except InfeasibleSubproblem:
            # Pruned by infeasibility.
            continue

        # Update global lower bound.
        if lb > global_lb:
            global_lb = lb
            best_solution = list(P.fixed_one) + items
            print('Improved lower bound:', global_lb)

        # Update global upper bound.
        if lb == ub and lb > integer_node_bound:
            integer_node_bound = lb

        if active_nodes:
            new_global_ub = max(ub, integer_node_bound,
                                max(P.ub for P in active_nodes))
        else:
            new_global_ub = max(ub, integer_node_bound)

        if new_global_ub < global_ub:
            global_ub = new_global_ub
            print('Improved upper bound:', global_ub)

        # Prune by bound?
        if ub < global_lb:
            continue

        # Prune by optimality?
        if lb == ub:
            continue

        # Select variable for split and perform the split.
        for i in range(inst.n):
            if i not in P.fixed_one and i not in P.fixed_zero:
```

```
                    break
            else:
                raise RuntimeError('no variable to fix; this is a bug')

            width[P.depth + 1] = width[P.depth + 1] + 2
            Pl = Subproblem(list(P.fixed_one) + [i], P.fixed_zero, ub, P.depth + 1)
            Pr = Subproblem(P.fixed_one, list(P.fixed_zero) + [i], ub, P.depth + 1)
            active_nodes += [Pl, Pr]

        assert global_ub >= global_lb

        # Check that the solution is truly feasible.
        if sum(inst.w[i] for i in best_solution) > inst.U:
            raise RuntimeError('solution is infeasible; this is a bug')

        # Return optimal solution.
        return lb, best_solution, width, estimate_width, estimated_time, start_time



def main():
    inst = generate_random_instance(100)


    bthelp = []
    maxwidth = 0
    # w = [71, 71, 72, 71, 72, 72, 72, 72, 75, 73, 74, 76, 73, 71, 76, 77, 77, 74, 71, 75, 72, 73, 77
    # c = [100, 99, 100, 98, 99, 99, 98, 96, 100, 97, 98, 100, 96, 93, 99, 100, 100, 96, 92, 97, 93, 
    inst = KnapsackInstance(w, c, 0.75 * sum(w))

    print('Here is the output of the branch-and-bound method')
    opt, sol, width, estimate_width, estimated_time, start_time = do_branch_and_bound(inst)
    end_time = time.time()
    print(
        '\nOptimal solution =', opt, ' items =', sol)
    print(" ")
    print(width)
    print(estimate_width)
    dt=1
    for i in range(len(estimate_width)):
        if estimate_width[i] == 2 ** i:
            lt = i
        if estimate_width[i] != 0:
            dt = i
        if estimate_width[i] == maxwidth:
```

```
            bthelp.append(i)
        if estimate_width[i] > maxwidth:
            maxwidth = estimate_width[i]
            bthelp = [i]
    bt = (bthelp[0] + bthelp[len(bthelp) - 1]) / 2
    # print ('lt: ', lt, 'bt: ', bt, 'dt: ', dt)
    print('Start_time: ', start_time, 'Estimate time: ', estimated_time[0], 'End time: ', end_time)
    print('Time till estimate: ', estimated_time[0] - start_time[0], 'Total algorithm duration: ', enc


    gamma = [2]
    for i in range(0, dt+1):
        if i <= lt - 1:
            gamma.append(2)
            # print(i, 'a')
        if lt <= i <= bt - 1:
            gamma.append(2 - (i - lt + 1) / (bt - lt + 1))
            # print(i,'b')
        if bt <= i <= dt:
            gamma.append(1 - (i - bt + 1) / (dt - bt + 1))
            # print(i, 'c')

    time_per_node = (estimated_time[0] - start_time[0])/sum(estimate_width)
    measurement_tree = [1]
    for i in range(1, len(gamma)):
        measurement_tree.append(gamma[i] * measurement_tree[i-1])
    min_estimated_Duration_of_Algorithm = max((estimated_time[0]-start_time[0]), 0.2*sum(measurement_t
    max_estimated_Duration_of_Algorithm = 5*sum(measurement_tree) * time_per_node
    print("Algorithm duration estimation: [", min_estimated_Duration_of_Algorithm, ",", max_estimated_
    print('Estimation of amount of nodes', sum(measurement_tree))
    print('Actual amount of nodes', sum(width))
    print('W= ', inst.w)
    print('C= ', inst.c)
    print('lt= ', lt)
    print('bt= ', bt)
    print('dt= ', dt)




main()
```