Convex Optimization Project Report

Riccardo De Vidi

December 2023

This document presents a branch-and-bound algorithm for the binary knapsack problem. In particular, given a set of n items, each with profit p_j and weight w_j, and a single container of capacity W, the developed algorithm selects a subset of the items of maximum total profit that fits into the container. In other words, the algorithm finds a vector $xOpt$ such that $xOpt = argmax_{x \in \mathbb{R}^n, x^T w < W, x_j \in \{0,1\} j=1..n}(x^T p)$ where $p = [p_1..p_n]^T$ and $w = [w_1..w_n]^T$.

1 General solving procedure

A branch-and-bound algorithm for solving an ILP is a divide and conquer procedure that casts the solution of the problem into the solution of two "easier" IL subproblems. A branch-and-bound algorithm designed to solve the given problem should perform the following steps.

- Evaluate an optimal solution $xStar$ with possibly fractional $[xStar]_h$ for the continuous LP relaxation of the problem. If $xStar$ is integer it is the optimal solution of the ILP, otherwise the optimal solution of the ILP will have either $[xStar]_h = 0$ or $[xStar]_h = 1$.
- Evaluate the optimal integer solution $xStar0$ for the ILP problem with $[xStar]_h = 0$.
- Evaluate the optimal integer solution $xStar1$ for the ILP problem with $[xStar]_h = 1$.
- The optimal solution of the original problem is found choosing between $xStar0$ and $xStar1$ the solution with maximum cost.

2 LP relaxation solver

To develope a branch-and-bound algorithm to solve the binary knapsack problem is convinient to define a procedure to solve any LP relaxation of the problem. Defined

- $relativeProfits$ as the vector of the densities of the profits in respect to their associated weight $[relativeProfits]_j = p_j/w_j$;
- $sortedIndices$ as the list of indices that sort $relativeProfits$ in ascending order $flip(argsort(relativeProfits))$;
- $sortedProfits$ as the permutation of p according to $relativeProfits$ $p[sortedIndices]$;
- $sortedWeights$ as the permutation $w[sortedIndices]$;
- $fixedXs$ as the set of constraints that the solution of the LP relaxation $xStar$ should respect
  - if $[fixedXs]_i = -1$ then there is not any constraint on $[xStar]_i$;
  - if $[fixedXs]_i = 1$ then $[xStar]_i = 1$;
  - if $[fixedXs]_i = 0$ then $[xStar]_i = 0$.

the algorithm $solveContinuousRelaxation$ solves a LP relaxation of the problem in the following way.

- initialize $xStar$ so that $[xStar]_i = 1$ if $[fixedXs]_i = 1$ and $[xStar]_i = 0$ otherwise for all $i = 1..n$;
- evaluate the sum of weights $sumW$ given by $xStar$;
- modify the non-fixed entries of $xStar$ to maximize $xStar^T sortedProfits$ under the contraints described by $fixedXs$;

Algorithm 1: solveContinuousRelaxation

```
   Data: fixedXs, n, W, sortedWeights, sortedProfits
   Result: xStar, h, lowerBound, upperBound, sumW
1  Begin
2      xStar ← fixedXs = 1;
3      sumW ← xStar^T sortedWeights >;
4      h ← 0;
5      while sumW < W and h < n do
6          h ← h + 1;
7          if fixedXs[h] = −1 then
8              sumW ← sumW + sortedWeights[h];
9              if sumW > W then
10                 xStar[h] ← (W − sumW + sortedWeights[h])/sortedWeights[h];
11             end
12             else
13                 xStar[h] ← 1;
14             end
15         end
16     end
17     lowerBound ← int(xStar)^T sortedProfits;
18     upperBound ← xStar^T sortedProfits;
19     sumW ← int(xStar)^T sortedWeights;
```

3 ILP solver

The developed algorithm is based on the nodes of the tree linked to the branch-and-bound procedure. For every solved LP relaxation, the identified values are stored in two distinct node lists: one to keep track of all the open nodes, and another to keep track of the non-purged nodes (Fischetti 6.4.1). The nodes in the second list exclusively encapsulate the bounds provided by the LP solver. Briefly, the developed algorithm executes the following steps:

- Solve the LP relaxation of the problem: If its solution is integral, the optimality is achieved, and the algorithm ends (lines 1-12).
- If the solution is not integral, execute the following:
  - save the bounds in one node and add it to the second list (lines 1-12);
  - save the upper bound, the index of the fractional variable, and the fixed indices in another node, adding it to the first list (lines 1-16).
  - While the time limit is not exceeded and the second list is not empty:
    - randomly select an open node (removing it from the first list) (lines 24, 29-31);
    - from the selected data, generate two new sets of fixed indices, fixing the value of the fractional variable to either 0 or 1 (lines 25,26);
    - solve the two LP relaxations associated with the generated index sets (lines 27, 28);
    - apply the fathoming criteria to manage the solutions of the two subproblems, appending them to the appropriate lists (lines 32-49);
    - remove from the lists the nodes that can not contribute at improving the solution (lines 52-59).

The developed algorithm makes use of the lists *nodeLowerBounds* and *nodeUpperBounds* such that the node in position j in the second list can be seen as

$$([nodeLowerBounds]_j, [nodeUpperBounds]_j)$$

and of the lists *openUpperBounds*, *nodeFixedXs*, *nodeFracVarIndex* such that the node in position j in the first list can be seen as

$$([openUpperBounds]_j, [nodeFixedXs]_j, [nodeFracVarIndex]_j)$$

.

Algorithm 2: solve

```
Data: n, W, sortedWeights, sortedProfits, stopTime
Result: globalLowerBound, globalUpperBound, runtime, optimalityGap, xOpt
Begin
    startTime ← time;
    fixedXs ← (−1.. − 1);
    globalLowerBound ← −1;
    xOpt ← (0..0);
    xStar, fracVarIndex, lowerBound, upperBound ← SolveContinuousRelaxation(fixedXs, n);
    globalUpperBound ← upperBound;
    nodeLowerBounds ← (lowerBound);
    nodeUpperBounds ← (upperBound);
    openUpperBounds ← null;
    if lowerBound = upperBound then
        globalLowerBound ← upperBound;
        xOpt ← xStar;
    end
    else
        openUpperBounds ← (upperBound);
        nodeFixedXs ← fixedXs;
        nodeFracVarIndex ← (fracVarIndex);
    end
    stopped ← false;
    while openUpperBounds is not null and openUpperBounds.length > 0 do
        if time − startTime ≥ stopTime then
            stopped ← true;
            exit the loop;
        end
        index ← random integer in [0, openUpperBounds.length);
        fixedXs0 ← [nodeFixedXs]_index, [fixedXs0]_[nodeFracVarIndex]_index ← 0;
        fixedXs1 ← [nodeFixedXs]_index, [fixedXs1]_[nodeFracVarIndex]_index ← 1;
        xStar0, fracVarIndex0, lowerBound0, upperBound0, sumW0 ←
          solveContinuousRelaxation(fixedXs0, n);
        xStar1, fracVarIndex1, lowerBound1, upperBound1, sumW1 ←
          solveContinuousRelaxation(fixedXs1, n);
        openUpperBounds ← openUpperBounds without the element in position index;
        nodeFixedXs ← nodeFixedXs without the list in position index;
        nodeFracVarIndex ← nodeFracVarIndex without the element in position index;
        if sumW0 ≤ W then
            if upperBound0 ≥ globalLowerBound then
                append lowerBound0 to nodeLowerBounds;
                append upperBound0 to nodeUpperBounds;
                if lowerBound0 = upperBound0 then
                    if lowerBound0 > globalLowerBound then
                        xOpt ← xStar0;
                        globalLowerBound ← lowerBound0;
                    end
                end
                else
                    append upperBound0 to openUpperBounds;
                    append fixedXs0 to nodeFixedXs;
                    append fracVarIndex0 to nodeFracVarIndex;
                end
            end
        end
        Repeat the operations of the ended if with the other node
    end
    The loop continues in the next page
```

Algorithm 3: solve - continuation

while *(continuation)* **do**
  $keepIndices \leftarrow$ indices s.t. $nodeUpperBounds \geq globalLowerBound$;
  $nodeUpperBounds \leftarrow nodeUpperBounds[keepIndices]$;
  $nodeLowerBounds \leftarrow nodeLowerBounds[keepIndices]$;
  $keepIndices \leftarrow$ indices s.t. $openUpperBounds \geq globalLowerBound$;
  $openUpperBounds \leftarrow openUpperBounds[keepIndices]$;
  $nodeFixedXs \leftarrow nodeFixedXs[keepIndices]$;
  $nodeFracVarIndex \leftarrow nodeFracVarIndex[keepIndices]$;
  $globalUpperBound \leftarrow min(nodeUpperBounds)$;
$runtime \leftarrow time - startTime$;
$optimalityGap \leftarrow globalUpperBound - globalLowerBound$;

4 Computational evaluation

Runtime [s]	# of solved nodes
$N = 50$	
0.0536358356475830	147
0.0876007080078125	625
0.0986828804016113	507
0.3022103309631347	2175
0.0723311901092529	551
$N = 60$	
0.1484298706054687	861
0.0099222660064697	29
0.1021842956542968	471
0.1804924011230468	815
0.0034310817718505	9
$N = 70$	
0.1553301811218261	741
0.1012604236602783	399
0.0662117004394531	209
0.0003576278686523	1
0.0918653011322021	295
$N = 80$	
0.1403098106384277	649
0.2128927707672119	1277
0.0003657341003417	1
0.0542464256286621	303
0.0744769573211669	275
$N = 90$	
0.4708790779113769	2443
0.2748632431030273	1589
0.1209189891815185	519
0.0079255104064941	23
0.1528987884521484	819
$N = 100$	
0.1644334793090820	785
0.3006329536437988	1453
0.0936427116394043	279
0.2082011699676513	957
0.0935056209564209	533

Table 1: Runtime and number of solved nodes for every processed instance

The file *solver.py* presents an implementation of the described algorithm. The script can be executed by running the command `python3 solver.py input.txt output.txt`, where *output.txt* will store the optimal solution found. The algorithm is tested on instances with varying sizes ($N = 50, 60, 70, 80, 90, 100$) with a time limit of 300 seconds. All instances are successfully solved within the specified time limit, and the runtimes along with the number of solved nodes are reported in Table 1.

To gather more statistically relevant data, a broader analysis is conducted by solving 250 instances for each size ($N = 50, 60, 70, 80, 90, 100$). The results are summarized in Table 2 (every instance has been solevd within the time limit). Upon inspecting the scatter plots derived from the data in Table 2, a noticeable linear growth in the runtimes means and in their standard deviations (STDs) is observed for the chosen instance sizes. However, a similar linear trend is less apparent for the numbers of solved nodes and their STDs.

It is important to notice that the data in Table 2 exhibits a high level of uncertainty due its elevated STDs, adding a layer of complexity to the interpretation of the results. This uncertainty underlines the need for a cautious analysis and interpretation of the algorithm's performance across different instance sizes.

	Runtime [s]		# of solved nodes	
N	Mean	STD	Mean	STD
50	0.13415851	0.12522965	686.352	672.28726
60	0.12011925	0.09450382	738.416	717.52143
70	0.1615453	0.13200748	719.512	775.59178
80	0.16604083	0.14759301	908.704	865.84666
90	0.19577574	0.17269809	911.536	860.84284
100	0.21063516	0.18537437	897.072	844.26872

Table 2: Runtimes and numbers of solved nodes means and STDs

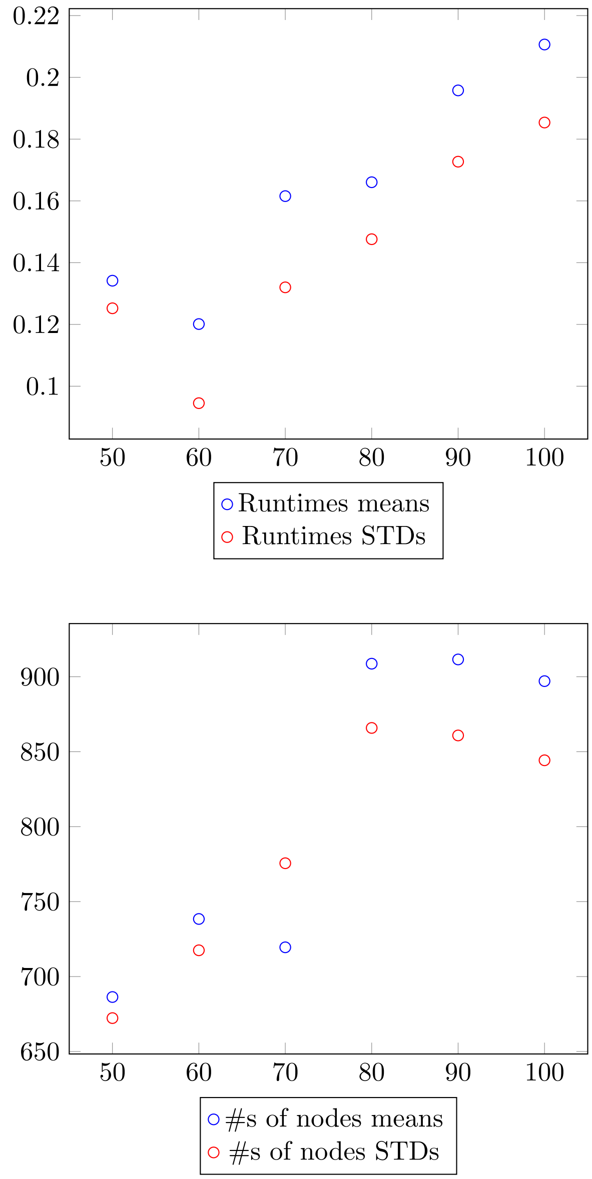

Figure 1: Scatter plots of the data in Table 2