CSC140 Advanced Algorithms
Spring 2023
Assignment 4 Report

0-1 KnapSack Problem

Student Name: Gabriele Nicula Student ID: 21996192

Test Machine(s) info

Type: MacBook Air 2020, 8GB memory
Cpu info: Apple M1
OS: macOS Ventura 13.2.1
Java: java 16.0.1 2021-04-20

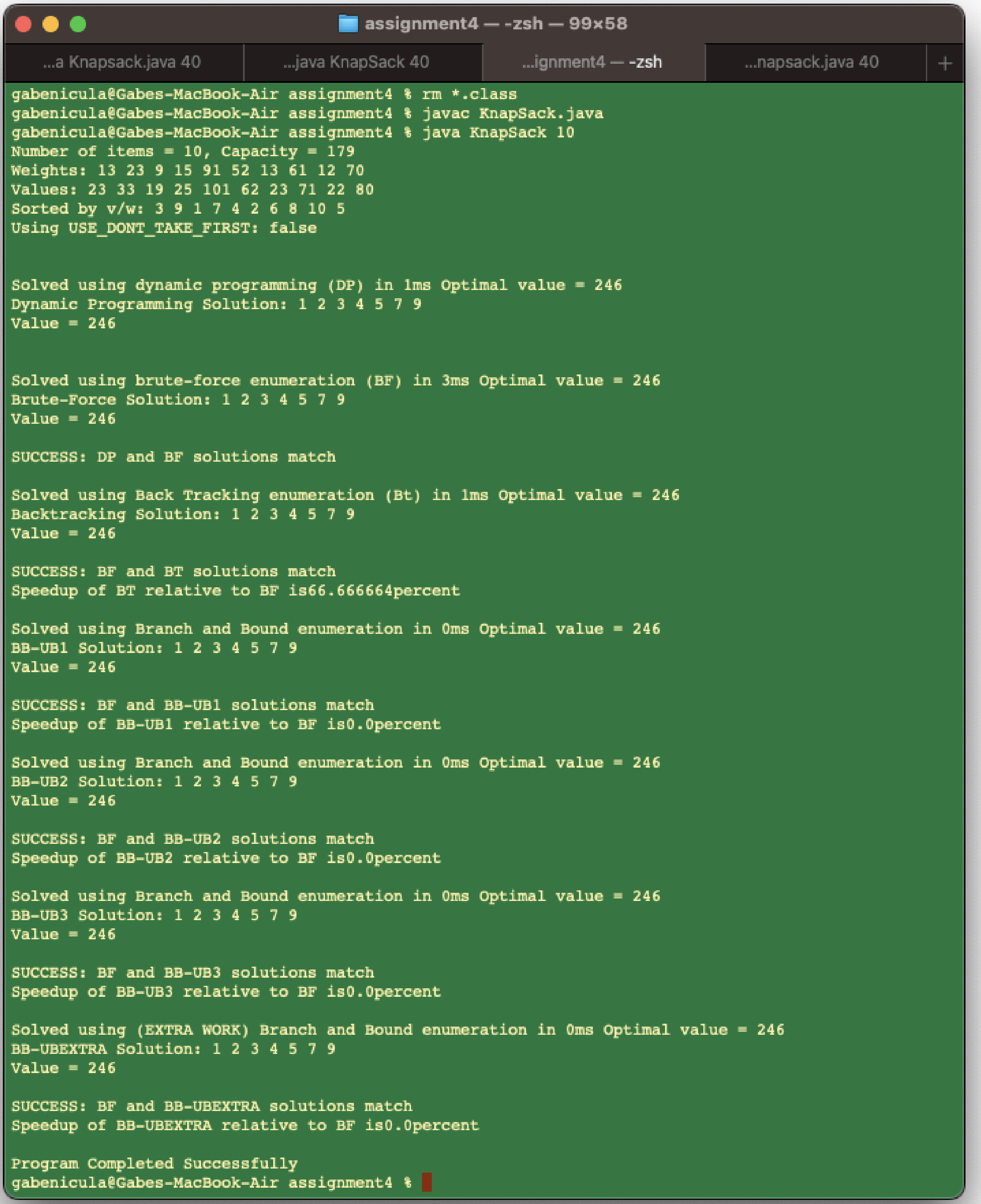
Also tested on: RVR2001 windows machine

Timing Results, all with TakeItem first (except row 4: N = 30 DontTake as noted*).

	Brute Force	Dynamic Prog	Back tracking	B&B UB1	B&B UB2	B&B UB3	B&B Fast UB3 (Extra Work)
N = 10	2ms	1ms	1ms	1ms	0ms	0ms	0ms
N = 20	87ms	9ms	34ms	41ms	6ms	2ms	0ms
N = 30	57660ms	11ms	30511ms	41732ms	2503ms	10ms	1ms
N = 30 DontTake 1st	77715ms	10ms	30953ms	43293ms	1979ms	4ms	2ms
N = 40	N/A[3] Timeout 1hr	12ms	N/A Timeout 1hr	N/A Timeout 1hr	3087233ms	401ms	4ms
Largest Input Solved in 10 seconds	27	410[1]	28	28[2]	32	50	100

(*) To enable DontTakeItem first strategy open KnapsackBFSolver.java and change:
`USE_DONT_TAKE_FIRST = true`
(1) Out of memory for values larger than 410 and the run times were around 4500ms.
(2) Times are very close to 10 seconds, for example: 10219ms
(3) Because BF for N = 30 takes about 60 seconds, N = 40 could take up to 60 * 1024. (Timeout)

Screenshot Image



```
gabenicula@Gabes-MacBook-Air assignment4 % rm *.class
gabenicula@Gabes-MacBook-Air assignment4 % javac KnapSack.java
gabenicula@Gabes-MacBook-Air assignment4 % java KnapSack 10
Number of items = 10, Capacity = 179
Weights: 13 23 9 15 91 52 13 61 12 70
Values: 23 33 19 25 101 62 23 71 22 80
Sorted by v/w: 3 9 1 7 4 2 6 8 10 5
Using USE_DONT_TAKE_FIRST: false


Solved using dynamic programming (DP) in 1ms Optimal value = 246
Dynamic Programming Solution: 1 2 3 4 5 7 9
Value = 246


Solved using brute-force enumeration (BF) in 3ms Optimal value = 246
Brute-Force Solution: 1 2 3 4 5 7 9
Value = 246

SUCCESS: DP and BF solutions match

Solved using Back Tracking enumeration (Bt) in 1ms Optimal value = 246
Backtracking Solution: 1 2 3 4 5 7 9
Value = 246

SUCCESS: BF and BT solutions match
Speedup of BT relative to BF is66.666664percent

Solved using Branch and Bound enumeration in 0ms Optimal value = 246
BB-UB1 Solution: 1 2 3 4 5 7 9
Value = 246

SUCCESS: BF and BB-UB1 solutions match
Speedup of BB-UB1 relative to BF is0.0percent

Solved using Branch and Bound enumeration in 0ms Optimal value = 246
BB-UB2 Solution: 1 2 3 4 5 7 9
Value = 246

SUCCESS: BF and BB-UB2 solutions match
Speedup of BB-UB2 relative to BF is0.0percent

Solved using Branch and Bound enumeration in 0ms Optimal value = 246
BB-UB3 Solution: 1 2 3 4 5 7 9
Value = 246

SUCCESS: BF and BB-UB3 solutions match
Speedup of BB-UB3 relative to BF is0.0percent

Solved using (EXTRA WORK) Branch and Bound enumeration in 0ms Optimal value = 246
BB-UBEXTRA Solution: 1 2 3 4 5 7 9
Value = 246

SUCCESS: BF and BB-UBEXTRA solutions match
Speedup of BB-UBEXTRA relative to BF is0.0percent

Program Completed Successfully
gabenicula@Gabes-MacBook-Air assignment4 %
```

Discussion:

1. *Exploring the Take option first is expected to be a better search order that completes the search faster, why?*

   When DontTakeItem() is called first, the recursion tree starts by completing the 'empty' solution with value 0 first and then incrementally adding/exchanging items starting from the last one and going toward the first. With this strategy, the final best solution (on random data) will most likely be discovered late in the recursion tree so branch and bounding will work with suboptimal current best solutions most of the time.

   When TakeItem is called first, the recursion tree starts with a full, potentially over-capacity knapsack and then incrementally removes/exchanges items. The first discovered and subsequently updated best solutions will be closer to the final solution so the branch and bounding algorithms have a chance to work with a close to optimal current best solution.

   Basically, TakeItem first offers a better quality pool of current best solutions vs. DontTakeItem first.

2. Interpretation of results [BF < UB1 < Backtracking < UB2 < UB3 < UB3 Fast]

   a) Brute Force performs worst, as expected.
   Brute Force considers all 2^N item combinations with no branch pruning. As N grows, Brute Force's execution time grows exponentially and becomes unusable. For a time complexity O(2^N) algorithm, theoretical running time increases about 1024 times more when N increases with 10.

   b) Dynamic programming performance:
   As seen in the above table, DP solver performs very well and it is competitive or better than B&B algorithms for the considered input sizes itemCnt <=40. When input size grows, ex: 50 or more, DP becomes faster than B&B methods 1 to 3 which are exponential in itemCnt.
   I have implemented DP solver with a table of size (itemCnt x Capacity) and each cell in the table must be traversed for the computation of the final value (best solution). For just computing the values, each cell is O(1) time and the total complexity of the DP solver is O(itemCnt x Capacity) time.
   This DP solver becomes faster with smaller capacities so it can take advantage of some inputs.

   However, this implementation must also track the structure of the best solution, not just compute the value of the best solution. This introduces additional overhead per cell to track which items are part of the solution per cell. In my implementation, this extra work is an array copy of sizes bounded by itemCnt. This DP solver is limited by the amount of program available (heap) memory and for example, on a MacBook Air, I was able to run it up to 410 items using -Xmx4g:

```
gabenicula@Gabes-MacBook-Air assignment4 % java -Xmx4g KnapSack.java 410
Number of items = 410, Capacity = 10191
Solved using dynamic programming (DP) in 4498ms Optimal value = 13101
```

c) Backtracking:
Performs better than Brute force and based on the collected data it manages to prune about half of the combinations for these runs (Backtracking running time is about half of the Brute Force).
This can be explained (for the collected data) by the relatively small capacity of the backpack set to half of the total weights: cap = wghtSum/2. With this setup Back Tracking will, on average, see an overweight partial solution about half the time.
It is still an exponential algorithm, with running time growing about 1024 times for each N+10 increase.

d) B&B UB1 - uses the sum of taken item values and undecided item values.
It performs better than Brute Force which shows it can prune some branches when partial solutions are already infeasible (worse than the current best). It is a bit worse than Backtracking which exploits the data generation a bit.
UB1 relies on pruning partial solutions that have discarded too many items already and it seems this doesn't happen as often as the Backtracking condition.
UB1 collected data verifies that it is still an exponential algorithm.

e) B&B UB2 uses the sum of taken item values and the values of the undecided items that fit in the remaining capacity at each node.
It performs better than Backtracking which shows that it is a "tighter" Upper Bound than UB1 and Backtracking. A possible explanation would be that on average, for the collected data, the sum of undecided items that fit is small maybe because due to the Generate() method, there are some large items that quickly become overweight for the remaining capacity.
UB2 collected data verifies that it is still an exponential algorithm.

f) B&B UB3 uses the best remaining value/weight to compute the bound.
It performs much better than {Backtracking, UB1, UB2} and it is on par with DP for N<=30.
UB3 prunes a lot of infeasible solutions because it considers a tight estimate of the undecided items when compared to UB2 which is more 'optimistic'.
UB3 is still an exponential algorithm in the worst case but in my tests, I did not encounter a very large running time on UB3.

g) B&B Fast UB3 - UB3 with Backtracking, see Extra Work (c) for more details
Performs best.

3. Extra work

a. *Implementing the Fractional Knapsack upper bound (UB3) incrementally in O(1) average amortized time per node.*

This was done using a 2D precomputed table of the fractional upper bound part of size (itemCnt x Capacity) which keeps the integer results of fractional computations for all

itemNum and possible remaining capacities. The code is in KnapsackInstance.java, precomputeFractionals().
The differences in performance vs the O(n) method were minimal in my tests although the O(1) method is just an array read at index. I think the explanation is that the O(n) method finishes very early on average and it is simple enough to be inlined. I also tried to skip the O(1) function call by making the 2D array public but didn't see much improvement even when going to 50-52 items.

b. *Avoid floating-point arithmetic and use integer arithmetic only in the Fractional Knapsack algorithm.*

   This was implemented in KnapsackInstance::Fractional(int itemNum, int remainingCap).
   I didn't see a noticeable difference by switching from:
   bestSum += (int)(Math.floor(rCap * GetItemValuePerWeight(i))); // to
   bestSum += ((rCap * values[i]) / weights[i]);

   This specific computation happens only on the last step, the fractional part so it is one per each Fractional() call so it is not a frequently executed statement.

c. *Any creative pruning techniques that you may come up with.*

   My optimized UB version uses the UB3 with O(1) fractional bound implementation and adds backtracking, inspired by the course material. It is faster than UB3 implementation because it combines B&B fractional upper bound with an overweight check which eliminates the fractional computation call if the solution is already infeasible.
   Based on the results this speeds up UB3 considerably which can be explained by UB3 exploring many already out-of-capacity partial solutions.
   Of course, this speed-up depends on many factors, but for the collected data, because of the relatively small knapsack capacity, adding a backtracking condition to UB3 reduces the running time considerably.
   The code for the UB3 Fast is in KnapsackBBSolver.java FindSolnsUBExtra() method.

How to compile and run

Unzip and go to the unzipped files folder and run:

```
% javac KnapSack.java
% java KnapSack 10
```