CSE6140/CX4140 Spring 2024 Project
Knapsack problem

April 15, 2024

1 Overview

The Knapsack problem is a well known NP-complete problem. In this project, we will treat the problem using different algorithms, evaluating their theoretical and experimental complexities on real world datasets.

2 Objectives

- Get hands-on experience in solving an intractable problem that is of practical importance
- Implement an exact branch-and-bound algorithm
- Implement approximate algorithms that run in reasonable time and provide high-quality solutions with guarantees
- Implement heuristic algorithms (without approximation guarantees)
- Develop your ability to conduct empirical analysis of algorithm performance on datasets, and to understand the trade-offs between accuracy, speed, etc., across different algorithms
- Develop teamwork skills while working with other students

3 Groups

You will be in a group of up to 4 students. We encourage you to form your own groups, and use Piazza as a resource. Afterwards, please designate one member per group to complete the assignment "Group and programming language submission for project" through Canvas by **Thursday, Mar. 28, 11:59pm**: In the text entry, put each student's last name and first name per line, followed by the programming language (Python, C/C++ or Java) your group will use. For instance:

```
Wang, RenChu
Cheng, Yuqi
Devaramani, Rajeshwari
Jiayi, Qian
Python
```

Note that if your team has less than 4 students, we may add students to your team. Also, rubrics will be the same for all teams, regardless of the number of members in each team.

4 Background

Given n items, where item i has a nonnegative weight w_i and a nonnegative value v_i, the Knapsack problem is to select a subset of items S such that the total value $\sum_{i \in S} v_i$ is maximized, and the total weight $\sum_{i \in S} w_i$ should not exceed a weight limit W.

For more details on the Knapsack problem, refer to Karp's original 21 NP-Complete problems `https://cgi.di.uoa.gr/~sgk/teaching/grad/handouts/karp.pdf`.

5 Algorithms

You will implement algorithms that fall into three categories:

1. exact, computing an optimal solution to the problem.

   Hint: The naive solution would be to enumerate all possible combinations (all possible fits) then verify all individual solutions. Is there a way to explore the sub-structures in a more efficient manner by bounding solutions? How do we derive a reasonable lower / upper bound?

2. construction heuristics some of which have approximation guarantees

   Hint: There's a simple modification to use the greedy approach instead of the pseudo-polynomial DP algorithm (all else being equal). Can you find what the greedy approach is and find the approximation ratio?

3. local search with no guarantees but usually much closer to optimal than construction heuristics.

   Hint: While local search does not guarantee the convergence of solutions, most of the time it should converge relatively quickly given a good search space. How do you formulate the search space such that the exploration is efficient and doesn't get stuck? This can have a night-and-day difference in simulated annealing or hill climbing algorithms.

In what follows, we present the high-level idea behind the algorithms you will implement

1. **Exact algorithm using Branch-and-Bound**. Implement the Branch-and-Bound algorithm as seen in class.

2. **Approximation algorithm with approximation guarantees**. Please implement an approximation algorithm for Knapsack with approximation guarantees as shown in class.

3. **Local Search**. There are many variants of local search (LS) and you are free to select which one you want to implement. Please implement 2 types/variants of local search. They can be in different families of LS such as Simulated Annealing vs Genetic Algorithms vs Hill Climbing, or they can be in the same general family but should differ by the neighborhood they are using, or by the perturbation strategy, etc.

6 Data

You will run the algorithms you implement on some real and random datasets, which will be available on Canvas: `DATA.zip`

This dataset comprises three sections. The first section is the test dataset, which includes problems alongside their optimal selections. These solutions serve as a benchmark to verify the correctness of your code. The remaining two sections consist of datasets of varying sizes: one small and one larger. Both of these datasets provide only the solutions with the optimal value, but without detailing the optimal selections themselves. Consequently, these datasets are intended to assess the accuracy of your code by comparing your solutions against these optimal values.

In each problem, the data is structured as follows: The first row indicates the number of items and the capacity of the knapsack. Subsequent rows provide the value and weight of each item. Therefore, the data format for problem in test set is as follows:

Table 1: Dataset Description

Set	Number of Problems	Number of Items	Names
Test	8	5-24	test , test_solution
Small Scale	10	4-23	small_scale, small_scale_solution
Large Scale	21	100-10000	large_scale, large_scale_solution

Example Input	Example Output
5 26	
24 12	0
13 7	1
23 11	1
15 8	1
16 9	0

7 Code

All your code files should include a top comment that explains what the given file does. Your algorithms should be well-commented and self-explanatory. Use a README file to explain the overall structure of your code, as well as how to run your executable.

Your executable should take as input i) the filename of a dataset and ii) the cut-off time (in seconds) iii) the method to use, iv) a random seed. If it is run with the same 4 input parameters, your code should produce the same output. The executable must conform with the following format:

```
exec -inst <filename> -alg [BnB|Approx|LS1|LS2] -time <cutoff in seconds> -seed <random seed>
```

Any run of your executable with any four inputs (filename, cut-off time, method and seed) must produce two types of output files:

1. Solution files:
   - File name: $< instance > \_ < method > \_ < cutoff > \_ < randSeed > *.sol$, e.g. $example\_BnB\_600.sol$, $example\_LS1\_600\_4.sol$ Note that as in the example above, randSeed is only applicable when the method of choice is randomized (e.g. local search). When the method is deterministic (e.g. branch-and-bound), randSeed is omitted from the solution file's name.
   - File format:
     (a) line 1: quality (the total value) of best solution found (integer)
     (b) lines 2: list of index of the numbers selected (comma-separated): $i_1, i_2, \cdots, i_n$
2. Solution trace files:
   - File name: $< instance > \_ < method > \_ < cutoff > \_ < randSeed > *.trace$, e.g. $example\_BnB\_600.trace$, $example\_LS1\_600\_4.trace$. Note that randSeed is used as in the solution files.
   - File format: each line has two values (comma-separated):
     (a) A timestamp in seconds (double)
     (b) Quality of the best found solution at that point in time. Note that to produce these lines, you should record every time a new improved solution is found.
     (c) Trace files are not required for approximation algorithms but are required for BnB and local search algorithms.
     An example:

     0.22, 50
     1.11, 69
     3.99, 71

8 Output

You should run all the algorithms you have implemented on all the instances we provide, and submit the output files generated by your executable, as explained in the Code section.

P.S.: DO NOT SUBMIT THE INPUT DATA FILES.

9 Report

9.1 Formatting

You will use the format of the Association for Computing Machinery (ACM) Proceedings to write your report. See `http://www.acm.org/publications/proceedings-template` for LaTeX and Word downloads. You can choose any template on this website.

9.2 Content

Your report should be written as if it were a research paper in submission to a conference or journal. A sample report outline looks like this:

- Introduction: a short summary of the problem, the approach and the results you have obtained.
- Problem definition: a formal definition of the problem.
- Related work: a short survey of existing work on the same problem, and important results in theory and practice.
- Algorithms: a detailed description of each algorithm you have implemented, with pseudo-code, approximation guarantee (if any), time and space complexities, etc. What are the potential strengths and weaknesses of each type of approach? Did you use any kind of automated tuning or configuration for your local search? Why and how you chose your local search approaches and their components? Please cite any sources of information that you used to inform your algorithm design.
- Empirical evaluation: a detailed description of your platform (CPU, RAM, language, compiler, etc.), experimental procedure, evaluation criteria and obtained results (plots, tables, etc.). What is the lower bound on the optimal solution quality that you can drive from the results of your approximation algorithm and how far is it from the true optimum? How about from your branch-and-bound?
- Discussion: a comparative analysis of how different algorithms perform with respect to your evaluation criteria, or expected time complexity, etc.
- Conclusion

9.3 Evaluation

We now describe how you will use the outputs produced by your code in order to evaluate the performance of the algorithms.

1. Comprehensive Table: Include a table with columns for each of your Knapsack algorithms as seen below. For all algorithms report the relative error with respect to the optimum solution quality provided to you in the Knapsack instance files. Relative error ($RelErr$) is computed as $(OPT - Alg)/OPT$. Round time and $RelErr$ to two significant digits beyond the decimal. For local search algorithms, your results for each cell should be the average of some number (at least 10) of runs with different random seeds for that dataset. You will fill in average time (seconds) and average vertex cover size. You can also report in any other information you feel is interesting. The comprehensive table should include results of all the instances of "large_scale" and "small_scale". It's optional to include instances from "test".

	Branch and Bound			Etc. (other algorithms)		
Dataset	Time (s)	total value	RelErr	Time (s)	total value	RelErr
instance	3.26	3400	0.0021			

Branch-and-Bound algorithms can take a long time on large instances. You can use the quality of your solution from the approximation algorithm as the initial upper bound, which is much tighter than a naive upper bound. You can set a time limit of 30 minutes or 1 hour which allows you to get some solution that is better than your initial upper bound.
The next three evaluation plots are applicable to local search algorithms only, and need only be presented for the instances "large_scale/large_1" and "large_scale/large_3". All the information you need to produce these plots is in your solution trace files.

1. Qualified Runtime for various solution qualities (QRTDs): A plot similar to those in Lecture on Empirical Analysis. The x-axis is the run-time in seconds, and the y-axis is the fraction of your algorithm runs that have 'solved' the problem. Note that 'solve' here is w.r.t. to some relative solution quality q^*. For instance, for $q^* = 0.8\%$, a point on this plot with x value 5 seconds and y value 0.6 means that in 60% of your runs of this algorithm, you were able to obtain a total value of at most the optimal value plus 0.8% of that. When you vary q^* for a few values, you obtain the points similar to those shown in the lecture.

2. Solution Quality Distributions for various run-times (SQDs): Instead of fixing the relative solution quality and varying the time, you now fix the time and vary the solution quality. The details are analogous to those of QRTDs.

3. Box plots for running times: Since your local search algorithms are randomized, there will be some variation in their running times. You will use box plots, as described in the 'Theory' section of this blog post: `http://informationandvisualization.de/blog/box-plot`. Read the blog post carefully and understand the purpose of this type of plots. You can use online box plot generators such as `http://boxplot.tyerslab.com/` to produce the plot automatically from your data.

10 Deliverables

Failure to abide by the file naming and folder structure as detailed here will result in penalties.
Only "one" student from each group should submit the following:

1. A PDF file of the report following the guidelines in section 9.

2. A zip file of the following files/folders:

   - Code: a folder named 'code' that contains all your code, the executable and a `README` file, as explained in section 7.
   - Output: a folder named 'output' that contains all output files, as explained in section 7.

11 Team Evaluation

Each student should also submit a *private* evaluation of their team members. For each team member (including self) include a score from 1 to 9 (scores for all team members should sum up to 10), outline the contributions that the member did to the project, and justification for the score.

References