

Coursework: Variable Neighbourhood Search for Bin Packing Problem

1. Introduction

Bin packing is one of the most studied combinatorial optimisation problems and has applications in logistics, space planning, production, cloud computing, etc. Bin packing is proven to be NP-Hard and the actual difficulties depend on both the size of the problem (i.e. the total number of items to be packed) and other factors like the distribution of item sizes in relation to the bin size as well as the number of distinct item sizes (different items may have a same size).

In this coursework, you are asked to write a C/C++/Python program to solve this problem using a variable neighbourhood search method. In addition to submitting source code, a written report (no more than 2000 words and 6 pages) is required to describe your algorithm (see Section 4 for detailed requirements). Both your program and report must be completed independently by yourself. The submitted documents must successfully pass a **plagiarism checker** before they can be marked. Once a plagiarism case is established, the academic misconduct policies shall be applied strictly.

This coursework carries 50% of the module marks. The rest of module marks comes from the final written exam.

2. Bin Packing Problem (BPP)

Given a set of n items, each item j has a size of a_j , BPP aims to pack all items in the minimum number of identical sized bins without violating the capacity of bins (V). The problem can be mathematically formulated as follow:

$$\begin{aligned}
& \text{minimize } B = \sum_{i=1}^n y_i \\
& \text{subject to } B \geq 1, \\
& \sum_{j=1}^n a_j x_{ij} \leq V y_i, \forall i \in \{1, \dots, n\} \\
& \sum_{i=1}^n x_{ij} = 1, \quad \forall j \in \{1, \dots, n\} \\
& y_i \in \{0, 1\}, \quad \forall i \in \{1, \dots, n\} \\
& x_{ij} \in \{0, 1\}, \quad \forall i \in \{1, \dots, n\} \forall j \in \{1, \dots, n\}
\end{aligned}$$

where $y_i = 1$ if bin i is used and $x_{ij} = 1$ if item j is put into bin i .

This mathematical formulation is generally NOT solvable by existing integer programming solvers like CPLEX, Gurobi, LPSolve, especially when the number of items n is large. In order to consistently solve the problem with good quality solutions, metaheuristics and hyper-heuristics are used, which is the task of this coursework.

3. Problem instances

Over the years, a large number of BPP instances have been introduced by various research. See <https://www.euro-online.org/websites/esicup/data-sets/> for a collection of different bin packing problem. In this coursework, we shall provide 3 instances files (binpack1.txt, binpack3.txt and binpack11.txt), respectively representing easy, medium and hard instances. From which 10 instances shall be selected for testing and evaluation of your algorithm in marking. For each test instance, only 1 run is conducted and its objective value is used for marking the performance component (see Section 5).

4. Experiments conditions and submission requirements

The following requirements should be satisfied by your program:

- (1) You are required to submit **two files exactly**. The first file should contain all your program source codes. The second file is a coursework report. Please do **NOT** compress the files.
- (2) Your source code should be properly commented.
- (3) Your report should include the followings:

- The main components of the algorithm, including solution encoding, fitness function, neighbourhoods as well as considerations regarding the intensification and diversification mechanisms.
- Statistical results (avg, best, worst of 5 runs) of the algorithm for all the problem instances, in comparison with the best published results (i.e. the absolute gap to the best results). **Note that although your report should include results for 5 runs but your final submission should only have one single run for each instance** (i.e. set global variable NUM_OF_RUNS=1 before you submit the code).
- A short discussion/reflection on results and performance of the algorithm.

(4) Name your program file after your student id. For example, if your student number is 2019560, name your program as 2019560.c (or 2019560.cpp, or 2019560.py).

(5) Your program should compile and run without errors on either **CSLinux** Server or a computer in the IAMET406 (the new computer lab). Therefore, please fully tested before submission. You may use one of the following commands (assuming your student id is 2019560 and your program is named after your id):

```
gcc -std=c99 -lm 2019560.c -o 2019560
```

or

```
g++ -std=c++11 -lm 2019560.cpp -o 2019560
```

For overseas students with poor connection to CSLinux, please contact GTAs (jiahuan.jin@nottingham.edu.cn or huayan.zhang@nottingham.edu.cn) for support. They'll be able to help test your program on CSLinux.

(6) After compilation, your program should be executable using the following command:

```
./2019560 -s data_file -o solution_file -t max_time
```

where 2019560 is the executable file of your program, data_file is one of problem instance files specified in Section 3. max_time is the maximum time permitted for a single run of your VNS algorithm. In this coursework, maximum of 30 seconds is permitted. soluton_file is the file for output the best solutions by your VNS algorithm. The format should be as follows:

```
# of problems
Instance_id1
obj=      objective_value      abs_gap
item_indx in bin0
item_indx in bin1
... ..
Instance_id2
obj=      objective_value      abs_gap
item_indx in bin0
item_indx in bin1
... ..
```

An example solution file for problem data file “**binpack1.txt**” is available on moodle.

- (7) The solution file output in (6) by your algorithm (`solution_file`) is expected to pass a solution checking test successfully using the following command:

```
./bpp_checker -s problem_file -c solution_file
```

where `problem_file` is one of problem data files in Section 3. If your solution file format is correct, you should get a command line message similar to: “**You total score out of 20 instances is: 80.**” If the solutions are infeasible for some instances, you would get error messages.

The solution checker can be downloaded from moodle page.

- (8) Your algorithm should run only ONCE for each problem instance and each run should take no more than 30 seconds.
- (9) Please carefully check the memory management in your program and test your algorithm with a full run on CSLinux (i.e. running multiple instances in one go). In the past, some submitted programs can run for 1-2 instances but then crashed because of *out-of-memory* error. This, if happens, will greatly affect your score.

5. Marking criteria

- The quality of the experimental results (40%). Your algorithm shall be tested for a file containing 10 instances chosen from the provided set of instances. The performance of your algorithm is evaluated by computing the absolute gap with the best known results using

$$abs_gap = your_average_objective - best_known_objective$$

Criteria	Mark
$abs_gap < 0$	New best results! Bonus: 3 extra marks for each best result.
$abs_gap = 0$	4 marks per instance
$0 < abs_gap \leq 1$	3 marks per instance
$1 < abs_gap \leq 2$	2 marks per instance
$2 < abs_gap \leq 3$	1 mark per instance

<ul style="list-style-type: none"> • <code>abs_gap > 4</code> or • infeasible solution or • fail to output solution within required time limit 	0 mark
--	--------

- The quality of codes (20%)
- Report (40%)

6. Submission deadline

4th May 2021, 4pm Beijing Time

7. How to submit

Submit via Moodle.

8. Practical Hints

- Solution encoding for bin packing is slightly more challenging compared with knapsack program because both the number of bins to be used and the number of items to be packed in each bin are parts of decisions to be optimised. Therefore, the data structure that is used to hold the packing information cannot be implemented via fixed-size arrays. You may consider to use **vector** from C++ STL (standard template library) which requires you to include `<vector.h>` as header file. If you prefer C style without classes, the following data type would be also acceptable:

```
struct bin_struct {
    std::vector<item_struct> packed_items;
    int cap_left;
};
struct solution_struct {
    struct problem_struct* prob; //maintain a shallow copy of problem data
    float objective;
    int feasibility; //indicate the feasibility of the solution
    std::vector<bin_struct> bins;
};
```

In this way, you could open/close bins and at the same time to add/remove items for a specific bin through API functions provided by the vector library.

- The search space of bin packing problem has a lot of plateaus that make the problem extremely difficult for simple neighbourhood methods. Therefore, multiple neighbourhood methods are more favoured but design and selection of a set of neighbourhood is critical to the performance of the algorithm. In addition to the common neighbourhood designs that can be found for TSP, knapsack, VRP problems, you are encouraged to use some of the neighbourhoods in the following paper (Section 4.2.2) <https://link.springer.com/article/10.1007/s10288-011-0182-8> to address the plateau search space issues.

- Although standard variable neighbourhood requires to sort neighbourhoods in increasing order of their sizes, in this application, the size of the neighbourhood may be not difficult to determine and the neighbourhood sequence can be decided through the experiments.
- Your algorithm must be runnable on CSLinux and/or computers on IAMET406. Therefore, you are not permitted to use external libraries designed specifically for optimisation.