Algorithms and Data Structures III (course 1DL481)
Uppsala University – Spring 2023
Report for Assignment 1 by Team 7

Anderson LEONG   Louis HENKEL

12th February 2023

Problem 2: Stochastic Local Search (SLS)

A lower bound on the number λ of shared elements of any pair among v subsets of size r drawn from a given set of b elements is given in [?]:

$$\mathrm{lb}(\lambda) = \left\lceil \frac{\left\lceil \frac{rv}{b} \right\rceil^2 ((rv) \bmod b) + \left\lfloor \frac{rv}{b} \right\rfloor^2 (b - ((rv) \bmod b)) - rv}{v(v-1)} \right\rceil \tag{1}$$

Task a: SLS Algorithm.

1. Representation. The current state of the local search is given by a $v \times b$ matrix, where each row v is a binary array of length b where there are always r elements that are 1.

2. Initial Assignment. It is made by randomly generating v binary arrays where r ones are randomly positioned.

3. Move. Each move we make is a switch in one row, making a permutation of a zero element with a one element.

4. Constraints. Each row must have exactly r elements which are one, which is a built in constrain which is never violated, because we only make swaps in the moves and each row gets initialised this way.

5. Neighbourhood. We create a move for permutation of a zero and a one within one row. The indexes to these ones and zeros are saved in 2 separate vectors to allow for easy retrieval and swaps. Hence, all possible permutations are easily retrievable or iterated through as they are all compiled in a list after initialisation. It allows us to make random moves by shuffling the possible moves in the vector to result in a different first improving neighbours that we select. As we can reach every feasible configuration for 1 row, we can also reach every feasible configuration of the matrix, but there will most likely be moves that increase cost between any given configuration and an optimal solution. The size of the neighbourhood is $v \cdot r \cdot (b - r)$.

6. Cost Function. Let v_i and v_j be the i-th and j-th row of a configuration. Then our cost function is given by:

$$\mathsf{cost} = \max_{i \neq j} \; v_i \cdot v_j$$

In practice, the cost is calculated by improvement. We calculate the difference in cost between the current configuration and the move we are probing. If improving moves have the same value, we choose the one with the initially higher cost as it is a better improvement in total.

7. Probing. We calculate the dot product of the row selected for swap, swap the elements we want to probe, and we get the largest product as output. This number is compared with the cost of the configuration before the swap, and should the cost decrease by 1 we have an improvement, If the cost increases, it means we are increasing the row cost, increasing the total cost by +1, or 0 implying no improvement. As the rows are represented in arrays, we can evaluate the cost difference from one element of the neighbourhood, we only need to evaluate the dot product of the new row and all the other rows, so we have a complexity of $O(v \cdot b)$

8. Heuristic. At initialisation, we create a list for each row, listing the possible swaps following the rules stated in Move that we can make for the given row. While probing we iterate of this list. If we find an improving move, following the cost, we commit it, and continue going through the list. Once we arrive at the end of the list we start again from the beginning, unless we do not find any improving move for an entire cycle through the list. Then we follow through with the rules listed in meta-heuristics

9. Optimality. As the Equation 1 gives a lower bound for the maximal cost (except of the input of $\langle 10, 8, 3\rangle$, we have found an optimal solution, if we reached the lower bound given by Equation 1, and can stop the algorithm. If the number of restarts set in the parameter β is reached, we also stop and return the best found configuration up until this point.

10. Meta-Heuristic. All moves made before arriving at the suboptimal local minima, will be commited and added to the tabu list for alpha iterations. This prevents the same moves from being made to escape from the local minima and hopefully enter a different neighbourhood.

11. Random Restarts. We can make random restarts when we the probing does not yield a better result. The number of restarts is given by the parameter β, with the tabu list we will no reach same local minimum again.

Task b: Implementation. We chose the high-performance programming language C++, for which a compiler is available on the Linux computers of the IT department. All source code is uploaded with this report. The compilation and running instructions are to have all the .cpp files, .h files, .cc file and the makefile in a folder with the terminal in the folder, enter the commands:

```
make
```

and run:

```
./InvDes v b r
```

to run the executable generated on the input parameters of choice v, b, r

```
./InvDes
```

to run the executable on the pre-input based on the java skeleton code provided.

The executable writes to the standard output a line followed by one line per row of the time of a run as well as the cost. If the argument p is written before the parameter v, b and r, then a space separated matrix representing the solution is also printed.

v	b	r	lb(λ)	$\langle\alpha,\beta\rangle = \langle 2,5\rangle$ time	steps	λ	$\langle\alpha,\beta\rangle = \langle 10,20\rangle$ time	steps	λ	exact
10	30	9	2	1.5285438	5	3	5.7164	5	3	$6.4787375 \cdot 10^{64}$
12	44	11	2	6.976366	5	3	19.19752	5	3	$1.239164 \cdot 10^{109}$
15	21	7	2	1.653098	5	3	5.189526	5	3	$4.4440315 \cdot 10^{63}$
16	16	6	2	0.9467204	5	3	62.2835612	5	3	$1.4439218 \cdot 10^{49}$
9	36	12	3	2.148164	5	4	64.446644	5	4	$9.6739127 \cdot 10^{75}$
11	22	10	4	61.01132	5	5	3.00055	5	5	$3.3944265 \cdot 10^{54}$
19	19	9	4	62.539938	5	5	125.586988	5	5	$2.9144089 \cdot 10^{75}$
10	37	14	2	3.79418	5	5	70.73318	5	5	$5.2415527 \cdot 10^{90}$
8	28	14	6	0.761037	5	7	2.896216	5	7	$2.1860830 \cdot 10^{56}$
10	100	30	6	46.0923	5	9	167.7912	5	9	$2.857551 \cdot 10^{246}$
6	50	25	10	2.465378	5	11	122.977204	5	11	$2.2987147 \cdot 10^{81}$
6	60	30	12	4.473836	5	13	68.85049	5	14	$8.4940661 \cdot 10^{98}$
11	150	50	7	257.1908	5	17	301.199	5	17	∞
9	70	35	11	17.96508	5	17	147.90506	5	17	$4.317829 \cdot 10^{173}$
10	350	100	4	306.8834	5	30	308.7114	5	30	∞
13	250	80	18	303.6696	5	27	302.5026	5	28	∞
10	325	100	21	303.907	5	33	305.1178	5	32	∞
15	350	100	19	309.9542	5	37	310.6142	5	36	∞
9	300	100	25	306.5702	5	37	303.6008	5	35	∞
12	200	75	17	301.304	5	30	302.142	5	28	∞
10	360	120	22	306.7936	5	46	307.01	5	46	∞

Table 1: Investment design: median runtime (in seconds), median number of steps, and median achieved λ, for two configurations of values for the local-search parameters α and β, over 5 independent runs per instance, with a timeout of 300.0 CPU seconds per run. The right-most column gives the number of candidate solutions the outlined exact algorithm has to examine per second in order to match the runtime performance of the seemingly best configuration of values for the local-search parameters, namely $\langle\alpha,\beta\rangle = \langle 2,5\rangle$, if the instance was solved to proven optimality, and 'n/a' for 'non-applicable' otherwise.

Task c: Experiments. All experiments were run under the the *gullviva.it.uu.se* server from the IT department, with Ubuntu 22.04 x86_64 architecture.

We attempted to have α be the number of iterations a move will be in the taboo list for, and β be the number of restarts from the original starting board to allow the search to escape the local minima, however, changing the parameters led to increased run time but with little to no improvement in performance. We suspect certain helper functions to be bugged that is causing our taboo search to not function as intended and hence its ability to arrive at the optimal solutions is low, where they only remain in the local minima regardless of the alpha beta parameters, hence, we chose to reduce these values to reduce run time.

Task d: Exact Algorithm. An exact algorithm could work as follows: It could perform brute force search, trying out all possible non-equivalent configurations. Two configurations are equivalent, if a permutations of the rows leads to both configurations to be the same. This would lower the size of the search space. For one row, the search space is given by $\binom{b}{r}$. As we have a total of b rows, there are in total $\binom{b}{r}^v$ possibilities. As the permutation of v elements yield a group of size $v!$, the number of equivalent classes is $b!$. Hence, the total search space is of size

$$\frac{\binom{b}{r}^v}{v!}$$

The number of candidate solutions this exact algorithm has to examine per second in order to match the runtime performance of the seemingly best configuration of values for the local-search parameters, according to Task c, of our stochastic local search algorithm is given in the right-most column of Table 1, for each instance solved to proven optimality.