Implementation of Memetic Algorithm for Multi-knapsack Problem

Zhangli WANG, 20028336

1 Algorithm Description

Algorithm 1: Memetic Algorithm for Multi-knapsack Problem

Require *Prob*: A problem structure
Require *ITEMS[n]*: A set of items ranged from 0 to *n*, which is known by *Prob*
Require *CAP[dim]*: A set of capacity ranged from 0 to *dim*, which is known by *Prob*

```
1.  start_time ← Current time of the system
2.  item_spent ← 0
3.  Initialize $P(t) \leftarrow \{S_1, \ldots, S_n\}$, $S_i \in \{0, 1\}^n$ /* Randomly initialize a population */
4.  While time_spent <= MAX_TIME Do
5.      PARENTS[POP_SIZE] ← $\Omega(P(t))$ /* $\Omega$ = Rollet Wheel Selection */
6.      Copy PARENTS to a newly created OFFSPRINGS
7.      For i := 0 To n
8.          Select $\{C_i, C_{i+1}\}$ from OFFSPRINGS;
9.          Crossover $\{C_i, C_{i+1}\} \leftarrow \Phi_c(C_i, C_{i+1})$ /* $\Phi_c$ = Reproduction: uniform crossover, including feasibility repair */
10.         i += 2
11.     Endfor
12.     For i := 0 To n
13.         Mutate $C_i \leftarrow \Phi_m(C_i)$ /* $\Phi_m$ = Reproduction: mutation, including feasibility repair */
14.         i++
15.     Endfor
16.     Coby and cover OFFSPRINGS to P(t)
17.     VNS_local_search(P(t), time) /* Local search will terminate if timeout */
18.     Update and record the solution with best objective value best_sln
19.     end_time ← Current time of system
20.     time_spent ← end_time-start_time
21.     Free in-loop memories
22. Endwhile
23. Free out-loop memories
24. Return best_sln
```

The pseudo codes of my memetic algorithm are shown above. Below is the detailed explanation and justification.

1.1 Initialization of the First Population

Algorithm 2: initialization of a random solution

Require *Prob*: A problem structure
Require *ITEMS[n]*: A set of items ranged from 0 to *n*, which is known by *Prob*
Require *CAP[dim]*: A set of capacity ranged from 0 to *dim,* which is known by *Prob*

```
1.  Create memory space for a solution pointer initialS
2.  CAP_TAKEN[dim] ← 0 /* Keep a record of capacities taken */
3.  For j := 0 To dim
4.      initialS->x[dim] = 0
5.  Endfor
6.  For i := 0 To n
7.      Random select an item $item_{select}$ from the item list
8.      If the item is unpacked Then
9.          If CAP_TAKEN[m]+$item_{select}$->size[m] <= CAP[m] for any m from 0 to
    dim-1 Then
10.             initialS->x[dim] = 1
11.             CAP_TAKEN[n] += $item_{select}$->size[n] for any n from 0 to dim-1
12.         Else
13.             i-- /* Avoid insufficient iteration */
14.             Continue
15.         Endif
16.     Endif
17. Endfor
18. Update capacities left of initialS
19. Free CAP_TAKEN
20. Return initialS
```

The initialization of solutions in the first population is adapted from the method provided by the method provided by Chu Beasley (1998). The method generates chromosome of solutions based on pure randomness. Therefore, to avoid generating infeasible solutions, following technique is adopted. Initially, all alleles are set to 0, then a random non-zero allele is selected, and the feasibility of the problem is evaluated if this allele would be set to 1. If the feasibility is still not violated, the allele will be set to 1. This procedure is repeated until the next change of allele will violate the feasibility.

1.2 Genetic Algorithm Operations

$$p_{select} = objective / \sum_{1}^{population\ size} objective$$

The while loop of the algorithm executes MA operations iteratively. In this section only GA operations are discussed. For selection, after entering the iteration, parents are selected from the previous population by rollete wheel selection, according to the formulae, the possibility of each solution is proportional to its objective value, which is the fitness value in this scenario.

For reproduction, firstly, a population offsprings is created copied from the selected parents. Secondly, every two solutions are selected iteratively from the offsprings and are performed uniform crossover. Finally, all solutions are performed mutation iteratively. After reproduction, all offsprings are copied to and cover the current population.

1.3 Feasibility Repair

Algorithm 3: Feasibility Repair

Require *sln*: A solution struct

```
1. Create memory space for a pointer to a list of item pointers items_sorted
2. Point item pointers of items_sorted to all items in the problem of sln
3. For i := 0 To n
4.     Calculate total size of item[i] in all dimensions, totalsize
5.     item[i]->unitprofit = item[i]->p/totalsize
6. Endfor
7. Sort items_sorted according to unitprofit in descent order
8. count←0
9. While sln is feasible && count<POPULATION_SIZE Do
10.    Drop item with lowest unitprofit /* Update chromosome, capacity left etc. */
11.    Update objective, feasibility of sln
12. Endwhile
13. count←0
14. While count<POPULATION_SIZE Do
15.    If sln->x[count] == 0 Do
16.        Check feasibility of sln if sln->x[count] is set to 1
17.        If sln is feasible Do
18.            Add item with highest unitprofit /* Update chromosome, capacity
   left etc. */
19.            Update objective, feasibility of sln
20.        Endif
21.    Endif
22. Endwhile
23. sln->feasibility = 1
24. Free items_sorted
25. Return sln
```

Feasibility repair should be executed to forcefully adjust an infeasible solution to feasible if infeasible situation occurs in reproduction. Referred to the feasibility repair operation introduced by Chu Beasley (1998), my operation is adapted and shown above.

1.4 Local Search

Algorithm 4: Variable Neighborhood Search

Require *sln*: A solution structure
Require *remained_time*: Remained time for the MA algorithm

```
1. neighbor_index := 0 /* Search from the closest the neighborhoods */
2. If remained_time > 0 Do /* if there is remained time for local search */
3.     While neighbor_index < 3 Do
4.         If neighbor_index == 1 Do
5.             Try to insert a new item into the pack until not feasible
6.         Elseif neighbor_index == 2 Do
7.             Execute best descent in the neighborhoods of 1-1 swap
8.         Elseif neighbor_index == 3 Do
9.             Execute best descent in the neighborhoods of 1-2 swap and 2-1 swap
10.        Endif
11.        Update the solution with best objective value best_neighb
12.        neighbor_index++ /* search next scale of neighborhoods */
13.    Endwhile
14.    sln := best_neighb
15. Endif
16. Free memories
17. Return sln
```

My algorithm adopts VNS best descent local search as the local search method for MA. This local search method exhaustively searches solutions in three complexity-increasing neighborhoods, which are adding items, singly swapping items, and swapping paired items, and updates the solution with best objective value in each search step. This method is time consuming but effective to search local optimal solution in limited iterations. Local search is carried out for each generation and will not be carried out if the maximum time for the MA is reached.

2 Parameter Tuning Process

Parameter tuning for this algorithm is carried out on Linux platform. However, the time counting was according to the real time rather than CPU time by mistake, so the output results may be worse than the actual results because the actual running time was less. Due to the time limit and restriction of Linux computability, I cannot perform systematical parameter tunings on Linux again with counting CPU time.

For problems of 100 items and 5 dimensions, objective values of all solutions are above 99% of the best objective values correspondingly. The following tables shows the results of more complex problems.

Psize	P_c	Uni P_c	P_m	Gap<1% Number	1%<Gap<2% Number	Gap>2% Number	Avg Gap%
10	0.9	0.5	0.03	18	9	3	1.092

10	0.9	0.5	0.07	14	11	5	1.293
10	0.9	0.5	0.09	16	11	3	1.189
10	0.9	0.5	0.01	17	8	5	1.131
10	0.8	0.5	0.01	18	10	2	1.076
10	0.7	0.5	0.01	18	9	3	1.087
10	0.8	0.5	0.01	17	8	5	1.133
10	0.8	0.6	0.01	16	11	3	1.095
10	0.8	0.5	0.01	18	11	1	1.024
30	0.8	0.5	0.01	15	6	9	1.393
60	0.8	0.5	0.01	17	11	2	1.015

Table 1 Parameter tuning results for problems of 250 items and 10 dimensions

Psize	P_c	Uni P_c	P_m	Gap<1%	1%<Gap<2%	Gap>2%	Avg Gap%
10	0.9	0.5	0.03	18	9	3	1.091
10	0.9	0.5	0.07	10	15	5	1.397
10	0.9	0.5	0.09	15	8	7	1.310
10	0.9	0.5	0.01	16	13	1	1.107
10	0.8	0.5	0.01	19	8	3	1.161
10	0.7	0.5	0.01	14	14	3	1.182
10	0.8	0.5	0.01	18	9	3	1.144
10	0.8	0.6	0.01	16	11	3	1.174
10	0.8	0.5	0.01	18	9	3	1.151
30	0.8	0.5	0.01	11	9	10	1.551
60	0.8	0.5	0.01	18	8	4	1.164

Table 2: Parameter tuning results for problems of 250 items and 30 dimensions

Psize = Size of the population
P_c = Probability of performing crossover for solutions
Uni P_c = Probability of performing crossover for alleles in a solution
P_m = Probability of performing mutation for solutions

The parameter tuning process was carried out to adjust the degree of convergency and dissociation of the algorithm to obtain higher global optimal solution. Through my tuning process, I found that tuning P_c can adjust the result macroscopically and tuning P_m can adjust the result microscopically. Increasing these values can increase dissociation, but if they exceed a threshold value, reproduction can generate many infeasible solutions to be repaired, which greatly disturbs the dissociation and weaken its influence.

Tuning Psize has many side effects. In my algorithm, I adopted best descent as the local search method, which means that local search is more time consuming compared to GA operations. If Psize is increased, this disparity is enlarged, so the computer spends more time for executing local search proportionally to GA operations in a time-limited running. Besides, large Psize can decrease the number of MA iterations. Tuning Psize is necessary for exploring global optimal solution when little improvement is made by tuning P_c and P_m.

Psize	P_c	Uni P_c	P_m
10	0.8	0.5	0.01

Table 3: My final parameters

3 My Memetic Algorithm Output

My best solution outputs are run on Windows platform with full computability utilized.

Item Types	Dimensions	Maximum Gap%	Minimum Gap%	Average Gap%
100	5	0.2133	0	0.0464
250	5	0.8293	0.0100	0.2364
500	5	0.8100	0.1135	0.3575
100	10	1.1898	0	0.3164
250	10	2.4366	0.2602	0.8827
500	10	1.8973	0.2866	0.8506
100	30	1.9418	0	0.5038
250	30	2.0410	0.5520	1.0570

Table 4: Results of running my algorithm with tuned parameters on all 8 problem types

4 Discussion of Genetic/Memetic Algorithm

4.1 Discussion of Advantages

1. Because GA and MA are population based, various kinds of solution schemes are considered to the problem. This means that they are more comprehensive for potential global optimal solutions, higher optimal values are always possible to be found if time is not limited.
2. Because GA and MA are population based, a group of good solutions can be generated.
3. Compared to GA, MA adopts an additional local search step, which means that MA is more likely to get convergence for each population. This can increase the speed of the search and produce better solutions in each generation steadily.
4. Through research, GA and MA are template algorithms, which means that they are adaptive for various kind of problems and they are suited for solving NP hard problems.

4.2 Discussion of Disadvantages

1. Because GA and MA are population based and selection is based on probability, which is highly relied on fitness function, the process of producing better solutions are slower than heuristic algorithms. Besides, the selection results may not be explicit if the population is too small. But if the population is too large on the contrary, the algorithm performs slow.
2. The algorithm is not heuristic which means that parameters are not improved on running. Parameters should be tuned manually for different types of problems, whose process is abstract for entry-level AI engineers.