Artificial Intelligence Methods Coursework Report

Memetic Algorithm for Multi-Knapsack Problem

1. Algorithm overview

To solve the multi-knapsack problem, Memetic Algorithm is implemented. Memetic Algorithm is a combination of population-based global search and individual-based local heuristic search. For the whole algorithm, it can be mainly divided into six parts: initialization, crossover, mutation, feasibility repair, first descent local search and replacement part.

2. Algorithm details and pseudo-code

2.1 Initialization part is implemented to initialize the population with the problem which read from the file and form a population which can be used for crossover and mutation. To achieve it, each loop the algorithm will pick one randomly and check if it can be added. After this operation, the algorithm will call evaluate_solution() function to double check its feasibility and calculate its objective value which will used in crossover function.

Algorithm 1 Initial Population

Require: $prob$: A problem struct which data from the data file, pop: A soulation struct array, N: the size of population, n: the number of items in the problem, dim: the number of dimensional

Ensure: a solutions struct array pop

function INIT POPULATION($prob, N, n, dim, pop$)
  for $p = 1 \rightarrow N$ **do**
    $pop[p].prob \leftarrow prob$
    for $j = 1 \rightarrow n$ **do**
      $pop[p][j] \leftarrow 0$
    end for
    for $i = 1 \rightarrow dim$ **do**
      $pop[p].cap\_left[i] \leftarrow prob->capacities[i]$
    end for
    randomly select j between 0 and $n-1$
  end for
  while $true$ **do**
    while $pop[p].x[j] = 1$ **do**
      $j \leftarrow randombetween0andn-1$
    end while
    $pop[p].x[j] \leftarrow 1$
    $can\_pack \leftarrow true$
    for $i = 1 \rightarrow dim$ **do**
      $pop[p].cap_left[i] \leftarrow pop[p].cap\_left[i] - prob->items[j].size[i]$
      if $pop[p].cap_left[i] < 0$ **then**
        $can\_pack \leftarrow false$
      end if
    end for
    if $can\_pack = false$ **then**
      $pop[p].x[j] \leftarrow 0$
      for $i = 1 \rightarrow dim$ **do**
        $pop[p].cap_left[i] \leftarrow pop[p].cap\_left[i] + prob->items[j].size[i]$
        break
      end for
    end if
  end while
  return pop
end function

2.2 Crossover function is used to select parents and make them crossover to generate new children., ternary tournament selection method is used to select parents as it can be implemented very efficiently and improve the accuracy of algorithm results. As the crossover part, one point crossover(1PTX) method is implemented if the crossover is happened. If the parents are not crossover, this algorithm will copy one of the parents

who has a higher objective value.

Algorithm 2 Select parent and Crossover

Require: $curt\_pop$: A soluation struct array which representing the current population
new_pop: A soluation struct array which representing the new population
Ensure: a new solutions struct array after crossover new_pop

```
function CROSS_OVER(curt_pop, new_pop)
    let N represents the size of pop, n represents the number of items in problem
    for (i = 0;i < N − 1;i + 2) do
        for r = 1 → 3 do
            if r = 0 then
                let parent1 equal to rand(0, N − 1)
                let parent2 equal to rand(0, N − 1)
            end if
            let curt1 equal to rand(0, N − 1)
            let curt2 equal to rand(0, N − 1)
            paren1 equal to Max(curt_pop[cur1].objective, curt_pop[parent1].objective)
            paren2 equal to Max(curt_pop[cur2].objective, curt_pop[parent2].objective)
        end for
        let crand = rand(0, 1)
        if (crand <= CROSSOVER_RATE) then
            let rd = rand(0, n − 1)
            for j = 1 → n do
                if j < rd then
                    new_pop[i].x[j] ← curt_pop[parent1].x[j]
                    new_pop[i + 1].x[j] ← curt_pop[parent2].x[j]
                else
                    new_pop[i].x[j] ← curt_pop[parent2].x[j]
                    new_pop[i + 1].x[j] ← curt_pop[parent1].x[j]
                end if
            end for
        else
            new_pop[i] ← curt_pop[parent1]
            new_pop[i] ← curt_pop[parent2]
        end if
    end for
    return new_pop
end function
```

2.3 In order to mimic the process of genetic evolution, mutation function is implemented. For this function, a random number between 0 and 1 will be generated for every gene and use to determine decide whether this gene will mutate. If this number is lower than the mutation rate, the corresponding gene will change from 1 to 0 or 0 to 1. On the contrary, there will be no change.

Algorithm 3 Mutation

Require: new_pop: A soluation struct array contains all the information which will be used in mutation
Ensure: a solution struct arraypop

```
function MUTATION(new_pop)
    let N represents the size of pop, n represents the number of items in problem
    for i = 1 → N do
        for j = 1 → n do
            crand = rand(0, 1)
            if crand < MUTATION_RATE then
                if pop[i].x[j] = 1 then
                    pop[i].x[j] ← 0
                else
                    pop[i].x[j] ← 1
                end if
            end if
        end for
    end for
    return new_pop
end function
```

2.4 After crossover and mutation, some solution may not meet the restrictions, to make sure all the solution is feasible, feasibility repair is implemented. This function uses greedy algorithm and it can be divided into two parts. The first part is checking and removing the items to make sure the solution is not feasible from the last to the first. The second step is check and add the items which can be added from the first to last.

Algorithm 4 Repair feasibility

Require: new_pop: A soluation struct array which may not meet the requirement
Ensure: solutions pop with all the elements is feasiable

function FEASIBILITY_REPAIR(pop)
  let N represents the size of pop, n represents the number of items in problem, dim represents the number of dimensional
  for $i = 1 \rightarrow N$ **do**
    for $d = 1 \rightarrow dim$ **do**
      set $sum \leftarrow 0$
      for $j = 1 \rightarrow n$ **do**
        $sum \leftarrow pop[i].prob- > items[j].size[d] * pop[i].x[j] + sum$
      end for
      for $j = n \rightarrow 1$ **do**
        if $pop[i].x[j] = 1$ and $sum > pop[i].prob- > capacities[d]$ **then**
          $pop[i].x[j] \leftarrow 0$
          $sum \leftarrow pop[i].prob- > sum - items[j].size[d]$
        end if
      end for
    end for
    for $d = 1 \rightarrow dim$ **do**
      set $sum \leftarrow 0$
      for $j = 1 \rightarrow n$ **do**
        $sum \leftarrow pop[i].prob- > items[j].size[d] * pop[i].x[j] + sum$
      end for
      $pop[i].capleft[d] \leftarrow pop[i].prob- > capacities[d] - sum$
    end for
    set $jude \leftarrow true$
    for $j = 1 \rightarrow n$ **do**
      if ($pop[i].x[j] = 0$ and $pop[i].capleft[d] >= pop[i].prob- > items[j].size[d]$) for ($\forall d \in dim$) **then**
        $jude \leftarrow true$
      else
        $jude \leftarrow false$
        break
      end if
      if ($jude$) **then**
        $pop[i].x[j] \leftarrow 1$
        for $d = 1 \rightarrow dim$ **do**
          $pop[i].capleft[d] \leftarrow pop[i].capleft[d] - pop[i].prob- > items[j].size[d]$
        end for
      end if
    end for
  end for
  return pop
end function

2.5 The Memetic Algorithm is based on the combination of global search and local search. The mutation and crossover belong to global search and the first descent local search method is implemented to achieve local search. This algorithm will find one added gene and another gene which can be added. Then compared these two genes and if the added gene's value is smaller than another, the algorithm will swap these two genes to make the solution better.

Algorithm 5 First Descent Local Search

Require: pop: A soluation struct array pop
Ensure: a solution struct arraypop after local search

function LOCAL_SEARCH_FIRST_DESCENT(pop)
  let N represents the size of pop, n represents the number of items in problem
  for $pz = 1 \rightarrow N$ **do**
    for $i = 1 \rightarrow n$ **do**
      set $check \leftarrow false$
      if $pop[pz].x[i] > 0$ **then**
        set $item \leftarrow i$
        for $j = 1 \rightarrow n$ **do**
          $ca \rightarrow true$
          if ($i \neq j$ and $pop[pz].x[j] = 0$ and $pop[pz].cap\_left[k] + pop[pz].prob \rightarrow items[item1].size[k] > pop[pz].prob \rightarrow items[j].size[k]$) and ($\forall d \in dim$) **then**
            $ca \leftarrow true$
            set $item2 \leftarrow j$
          end if
          if ca **then**
            if ($pop[pz].prob \rightarrow items[item2].p - pop[pz].prob \rightarrow items[item1].p > 0$) **then**
              $pop[pz].x[item1] \leftarrow 0$
              $pop[pz].x[item1] \leftarrow 1$
              $evaluate(pop[pz])$
              $check \leftarrow true$
              break
            end if
          end if
        end for
        if check **then**
          break
        end if
      end if
    end for
  end for
  return pop
end function

2.6 The replacement algorithm is used to selection the best n individuals and store it in the curt_pop for the next round. As there are two population in one generation, this function needs to traverse the two population n times where n is the size of the population to ensure that the size of the new population is consistent with the original one.

Algorithm 6 Replacement

Require: *curt_pop*: Last generation *new_pop*: The new generation, *rep_pop*: The population used to store the best n items between new_pop and curt_pop

Ensure: a new solutions struct array with best N items between new_pop and curt_pop

```
function CROSS_OVER(curt_pop, new_pop, rep_pop)
    let N represents the size of pop, n represents the number of items in problem
    for i = 1 → N do
        letcur1, cur2 ← 0
        for k = 1 → N do
            if curt_pop[k].objective > curt_pop[cur_max1].objective then
                cu_max1 ← k
            else
                cur_max2 ← k
            end if
            rep_pop[i] = Max(curt_pop[cur_max1].objective, new_pop[cur_max2].objective)
            Max(curt_pop[cur_max1].objective, new_pop[cur_max2].objective).objective ← 0
        end for
    end for
    cur_pop = rep_pop∀individual ∈ n
    return curt_pop
end function
```

3. Parameter tuning process

For Memetic Algorithm, there are serval parameter which can affect the accuracy of the result and the running time. The parameters are as follows: number of runs (NUM_OF_RUNS), population size (POP_SIZE), number of generations (NUM_OF_GEN), crossover rate (CROSSOVER_RATE) and mutation rate (MUTATION_RATE).

The Number of Runs:

This parameter is used to determine how many times does the Memetic Algorithm run for a same problem and it influences the running time and the quality of results. During the test, if other variables do not change, gap% to the best results is higher with this parameter becomes larger, but the result repeat rate of each MA becomes higher, and the running time will also be significantly longer for the same problem. To balance the quality of result and running time, the number of runs is set to 5 for the bigger instance and 3 for small instance.

Population Size:

Population size is a very important parameter in this algorithm. Both running time and the quality of results are deeply affected by this parameter. With other parameters do not changed, 100 is set to this parameter and added 100 each time to adjust it. With this parameter gets bigger, the result is getting better at first, as it comes 600 or larger, some of the problems result will not be changed or improve very little. However, the running time is steady growth with population size grow bigger. As a result, for small instance, 600 – 800 population size is used and for small instance, 300 – 500 population size is used for bigger problem.

The Number of Generations:

This parameter is used to determine the number of generations for a population in

Memetic Algorithm. In general, the larger the parameter, the longer the running time, the higher gap% to the optimal solution. The influence of the generation number is similar to the population size parameter. It also has a bigger influence on both the running time and quality of the results. During the test, keep other parameters unchanged, the running time becomes longer with larger number of generations. This parameter also has marginal effect. At the beginning of the test, 200 is set and 100 is added to this parameter each time. At first, the improvement of the result is significantly, after this parameter is bigger than 500, the improvement is very little. To balance the time and performance, 500 is set for small instance and 200 – 300 is set for larger instance.

Crossover Rate:

Crossover Rate indicates the probability of parents crossing to generate new children. This parameter only has influence on the gap% to the optimal result. To maximize the gap%, with other parameters are not changed, $0.6 - 0.95$ is tested during the test. As the result shows, the gap% increased form $0.6 - 0.85$, and then it decreased form 0.86 – 0.95. However, with other parameters changed, the range which gap increased will also change. After many tests, 0.85 ∓ 0.02 is used to get a better gap%.

Mutation Rate:

Mutation Rate is similar to the crossover rate and only affect the result's gap% to the optimal result. To simulate gene mutation process, this rate cannot set at a high number. During the test, both a high rate and too low number will lead the gap% decrease. At the beginning of the test, the rate is set to 0.01 and mines or add 0.001 each test. After many tests, 0.010 ∓ 0.003 is accepted to get a better gap% in this algorithm.

4. Comparison

The files which are used to compare with the optimal solution is mknapcb1.txt, mknapcb5.txt and mknapcb8.txt. The comparison results are as follows:

Mknapcb1.txt. There are 30 problems and each problem have 100 variables, 3 dims

Problem	C_Rate	M_Rate	Run	Generation	Pop	Time	Gap%	Ave_Gap%
1	0.83	0.015	5	500	700	64.2s	0.989	0.995
2	0.83	0.015	5	500	700	64.2s	0.997	0.995
3	0.83	0.015	5	500	700	64.2s	0.996	0.995
4	0.83	0.015	5	500	700	64.2s	0.992	0.995
27	0.83	0.015	5	500	700	64.2s	0.998	0.995
28	0.83	0.015	5	500	700	64.2s	0.998	0.995
29	0.83	0.015	5	500	700	64.2s	0.996	0.995
30	0.83	0.015	5	500	700	64.2s	0.999	0.995

Mknapcb5.txt. There are 30 problems and each problem have 250 variables, 10 dims

Problem	C_Rate	M_Rate	Run	Generation	Pop	Time	Gap%	Ave_Gap%
1	0.85	0.008	5	300	500	78.9s	0.986	0.984
2	0.85	0.008	5	300	500	78.9s	0.975	0.984
3	0.85	0.008	5	300	500	78.9s	0.969	0.984

4	0.85	0.008	5	300	500	78.9s	0.975	0.984
27	0.85	0.008	5	300	500	78.9s	0.986	0.984
28	0.85	0.008	5	300	500	78.9s	0.995	0.984
29	0.85	0.008	5	300	500	78.9s	0.993	0.984
30	0.85	0.008	5	300	500	78.9s	0.992	0.984

Mknapcb8.txt. There are 30 problems and each problem have 250 variables, 30 dims

Problem	C_Rate	M_Rate	Run	Genertion	Pop	Time	Gap%	Ave_Gap%
1	0.85	0.008	3	250	400	60.7s	0.967	0.976
2	0.85	0.008	3	250	400	60.7s	0.951	0.976
3	0.85	0.008	3	250	400	60.7s	0.973	0.976
4	0.85	0.008	3	250	400	60.7s	0.965	0.976
27	0.85	0.008	3	250	400	60.7s	0.990	0.976
28	0.85	0.008	3	250	400	60.7s	0.991	0.976
29	0.85	0.008	3	250	400	60.7s	0.986	0.976
30	0.85	0.008	3	250	400	60.7s	0.990	0.976

5. Analysis the GA/MA

The strength of GA/MA

Compared to the traditional optimization algorithm, GA can be applied to a wide range of problem areas as it is basically unlimited for the function to be searched. As MA is manipulating the encoding of decision variables, it is optimized to be better than traditional algorithms. By combining global search and local search, MA can get the global optimal solution while traditional optimization algorithm can only get the local optimal solution. MA starts parallel operation from many points, so it can effectively prevent the search process from converge to the local optimal solution. By using crossover and mutation operation, the optimization results are independent of initial conditions. What's more, MA also use efficient heuristic search instead of blindly exhaustive or completely random search, as a result, it usually can get satisfactory results.

As the different between GA and MA, in my opinion, MA is just like a frame which combine the global search and local search. For instance, in MA, it contains GA and a method of local search like First Descent Local Search algorithm or Simulate Anneal algorithm. By combining global search and local search, MA have a higher search efficiency and avoiding falling into local optimal solution.

The weakness of GA/MA:

With so many advantages, MA also has its own disadvantages. First, there are too many variables need to be controlled such as crossover rate, mutation rate, population size and so on. Each variable will affect the optimized results. To get the best result, a lot of tests need to be done and even after a lot of experimentation, it may not be the best combination. Another disadvantage is the potential capabilities of the parallelisms of the algorithm are not fully utilized which may waste a lot of time. Compared to MA, GA do not combine local search algorithm, which means it has poor local search ability.