PROBLEM OF FINDING THE SHORTEST PATH TO TAKE PACKAGES IN A STORAGE

Project of Optimization (IT3052E) course

Hanoi University of Science and Technology

Contents

DESCRIPTION

4. Lấy hàng trong kho

- Trong một kho có các kệ để hàng hóa 1, 2, .., M. Giả thiết kệ j đặt ở điểm j (j = 1,…,M).
- Có N loại sản phẩm được bày rải rác trên các kệ trong kho, mỗi loại sản phẩm i có thể được bày ở nhiều kệ với số lượng khác nhau.
- Biết rằng Q(i,j) là số lượng sản phẩm loại i được bày ở kệ j (i = 1,…,N và j = 1,…,M).
- Nhân viên kho, xuất phát ở cửa kho (điểm 0), cần vào kho lấy các sản phẩm cho 1 đơn hàng trong đó sản phẩm loại i cần lấy số lượng c
- Biết rằng d(i,j) là khoảng cách từ điểm i đến điểm j (i, j = 0,1,…,M)
- Hãy tính toán phương án lấy hàng cho nhân viên kho sao cho tổn quãng đường di chuyển là nhỏ nhất.

DESCRIPTION

- In a warehouse there are many racks for packages 1, 2, … , M. Suppose the rack is placed at point .
- There are N different types of packages placed all over the racks in the warehouse.
- The number of packages of type i placed at rack j is where and $i = 1, 2, \ldots, N$ and $j = 1, 2, \ldots, M$.
- A warehouse staff starts at the entrance of the warhouse (point 0), needs to pick up the packages for an order in which the number of packages of type i is $q(i)$.
- Given the distance from point i to point j is $d(i, j)$.
- Calculate the route for the staff so that the distance they have to travel is minimal.

VALUES

- M is the number of the racks in the storage
- N is the number of types of the package in the storage
- Matrix Q(i,j) provides information about the number of package i (i=1,..N) in the j rack (j=1,..M)
- Matrix d(i,j) gives information about the distance costs of moving from rack i to rack j (i,j=0,..M)

PRE-PROCESSING

- Because the matrix d[i][j] can be asymmetric or does not satisfy the triangle inequality. This means that the optimal path can goes through the same point more than once. To simplified the problem, we introduce the step of preprocessing the input.
- Preprocess: By pre-calculate the shortest path between any two racks in the storage, the solution can be transformed into a path of racking in which each racking is visited only once while taking into account all the racks visited. Here we used the Floyd-Warshall algorithm.

Backtracking

- Preprocess: By pre-calculate the shortest path between any two racks in the storage, the solution can be transformed into a path of racks in which each racking is visited only once while taking into account all the racks visited. Here we used the Floyd-Warshall algorithm.
- Backtracking: check all the permutations using M racks
- Time Complexity: O(NxM!)

Branch and bound

Cut all branches that exceed the upper bound:

F’ - min (d’[j] * max (0, q[j] - p[j] - Q[j][i])) ∀ package type j at current rack i

F’ : best result so far
d’[j] : minimum distance required to collected a unit of package j (pre-calculate)
q[j] : constraint – number of package type j demanded
p[j] : number of packages type j collected at current tour
Q[j][i] : number of packages type j at current rack i

Dynamic Programming

Using Dynamic Programming with memoization, the brute-force search running time can be significantly reduced

F = Min (F + d[x][y])

(racks visited,
last visit rack x,
Current collected packages,
remaining lacking
package types)

(racks visited + y,
an unvisited rack y,
update collected packages,
update # remaining
lacking package types)

Constraint Programming

Use OR-Tools to create a model for our problem as a constraint optimization problem.

Decision variables

$$x_{j,k} = \begin{cases} 1 \; if \; the \; j^{th} \; rack \; is \; at \; position \; k^{th} \; in \; the \; path, \\ 0 \; otherwise \end{cases}$$

Example:

```
x = [
      [1, 0, 0, 0],
      [0, 1, 0, 0],
      [0, 0, 0, 1],
      [1, 0, 0, 0],
]
```

coresponds with the path $0 \rightarrow 1 \rightarrow 3 \rightarrow 0$

Constraint Programming

Constraints:

$$\sum_{j=0}^{M} x_{j,k} = 1 \; \forall \; k = 0, 1, 2, \dots, k_0 \; ;$$

$$\sum_{k=1}^{k_0} x_{j,k} = 1 \; \forall \; j = 0, 1, 2, \dots, M \; ;$$

$$\sum_{k=1}^{k_0-1} \left(\sum_{j=1}^{M} Q(i,j) \times x_{j,k} \right) \geq q(i) \; \forall \; i = 1, 2, 3, \dots, N;$$

$$x_{0,0} = 1 \; ; x_{0,k_0} = 1 \; .$$

Objective function to minimize:

$$f = \sum_{k=0}^{k_0-1} \left(\sum_{i=0}^{M} \left(\sum_{j=0}^{M} (x_{i,k} \times x_{j,k+1}) \times d(i,j) \right) \right).$$

Greedy Algorithm 1

- Use search technique to find the nearest racking
- Repeat the process until taking enough of the number of types of packages
- Immediately come back to the entrance from the current location.

Greedy Algorithm 1

- The algorithm is good only when the number of packages of all types in all racks is closed to each other and the number of packages that need to be taken is small

Explanation:

- The algorithm only optimize the distance moving, not taking too much concern in the number of packages in the racks that we will visit.

Greedy Algorithm 2

- Use search technique to find the racking with the

highest ratio $\dfrac{\text{\# 'meaningful' packages}}{\text{distance from the current rack}}$

- Repeat the process until taking enough of the number of types of packages
- Immediately come back to the entrance from the current location.

Greedy Algorithm 2

- The algorithm is slightly worse than the first greedy algorithm in practical test cases.

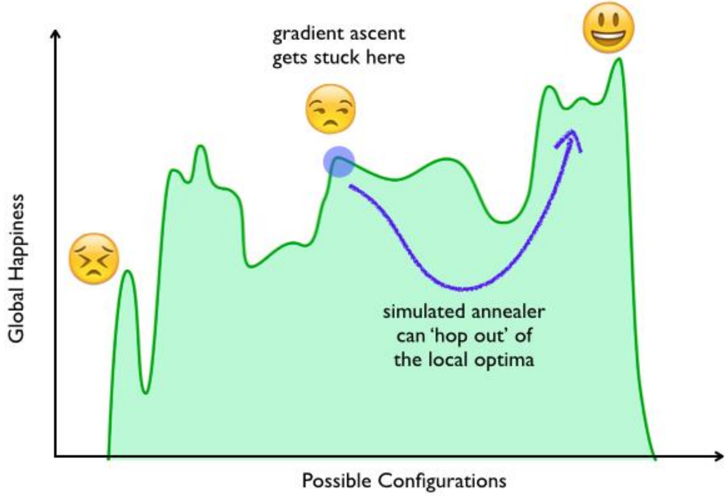
- However, it can be more stable and avoid produce too bad results since it take into account both distance and number of packages factor

Simulated Annealing

- A local search algorithm that can approximate global optimum without getting stuck in local optimum

- It takes into account three main factors: the initial and neighbor state, the cost function and cooling schedule

Graph Explanation of Simulated annealing:



Simulated Annealing

1. Searching Space (Initial State and Neighbor state)

- Initial State is a random permutation of (1,..M);
- Idea: A solution demands choice of racks visiting and order of visiting
- We obtain a neighbor state by a move that changed these two factors at minor enough rate - inversing a part of the list that is the previous state to get the current state.
  Choose an interval (l, r) (l is not necessarily smaller than r)

(l, r) = (2, 4)

R1 , R2 , R3 , R4 , R5 → R1 , R4 , R3 , R2 , R5

(l, r) = (5, 3)

R1 , R2 , R3 , R4 , R5 → R2 , R1 , R5 , R2 , R3

- Tabu list helps to prevent reversing the same interval (for a short term – M iterations).

Simulated Annealing

2. Cost function

- The fitness of a state is the total distance traveled in the route.
- We follow the solution until we pick up all the required packages
- We need to add the distance moving from the entrance to the first rack and the distance moving from the last rack to the entrance

$$cost\ function = \sum_{i=1}^{length} d(path[i-1], path[i]) + d(0, path[0]) + d(path[length], 0)$$

Simulated Annealing

3. Cooling Schedule

Temperature value (t) changing over running time results in different probabilities of accepting a worse neighbor state (P) at different stages of the SA algorithm

$$P = \begin{cases} 1 & if\,\Delta c \leq 0 \\ e^{-\Delta c / t} & if\,\Delta c > 0 \end{cases}$$

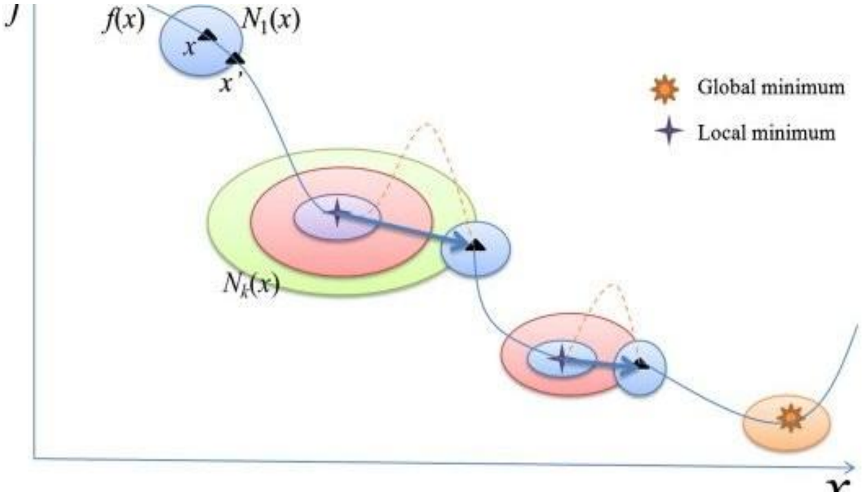
Geometrical cooling schedule

$$T_k = T_0 \times \alpha^k$$

$T_0 = 2000$
k: $1 \rightarrow 20000$
$\alpha = 0.0003$

Variable Neighborhood Search (VNS)

A variable neighborhood search combine with iterated local search improvement



Initial Solution

- Racks are added at random order to the path until the package constraint is met

Iterated Local Search:

- The path is usually short with respect to total #racks M
- Keep adding racks into consecutive initial solution to create different starting solutions.

constraint met | constraint met | constraint met

$R_1, R_2, R_3, R_4 \mid R_5, R_6, R_7, R_8, R_9, R_{10}, R_{11} \mid, R_{12}, \ldots, R_{M-3} \mid, R_{M-2}, R_{M-1}, R_M$

1st initial solution | 2nd initial solution

- These highly distinctive starting solutions help explore various regions of the searching space

Neighborhood

MOVE 1: CUT

R_1 , R_2 , R_3 , $\cancel{R_4}$, R_5 , R_6 , R_7

MOVE 2: SWITCH

R_1 , R_2 , R_3 , R_4 , R_5 , R_6 , R_7

MOVE 3: PUSH

R_1 , R_2 , R_3 , R_4 , R_5 ➡ R_2 , R_3 , R_1 , R_4 , R_5

MOVE 4: SWAP

R_1 , R_2 , R_3 , R_4 , R_5

R_8 R_{10}

R_6

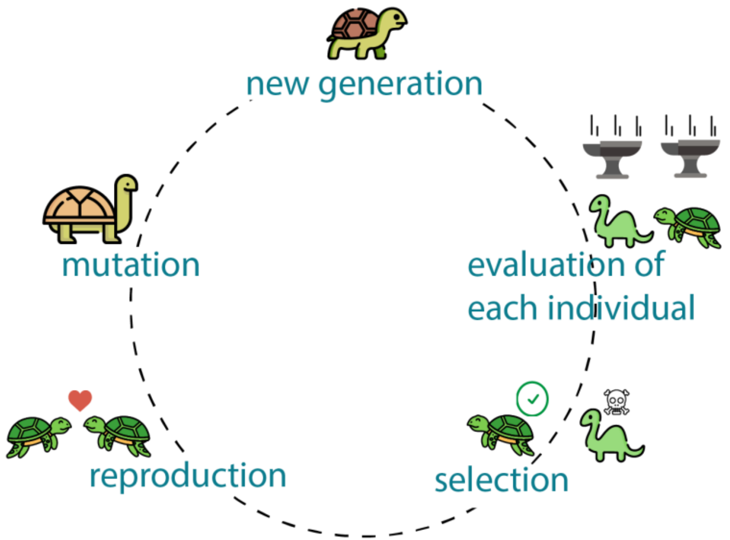
R_9 R_7

Variable Neighborhood Search (VNS)

- Choose the best move out of the 4 move types at each iteration (only constraint-satisfied moved allowed)
- Whenever getting stuck (cannot find a better solution within the neighborhood), expand the move to 2 and maximum 3 repeated times
- The 4th move takes the longest to run and can only run the 2nd and 3rd times with respectively small test cases.

Genetic Algorithm

- An algorithm based on the biological evolution.
- We present the path that we will visit to be a list that is the permutation of (1,...M)



Genetic Algorithm

Create the population:

- Set the population size pop_size = 200.
- Using random.shuffle function to shuffle list [1,…,M] to create individuals.
- An individual will present a path.
- Value pop_size, number of generation, crossover rate and mutation rate are parameters and can be changed based on the size of the test, sacrifice the running time to get better result and vice versa.

Genetic Algorithm

In a generation:

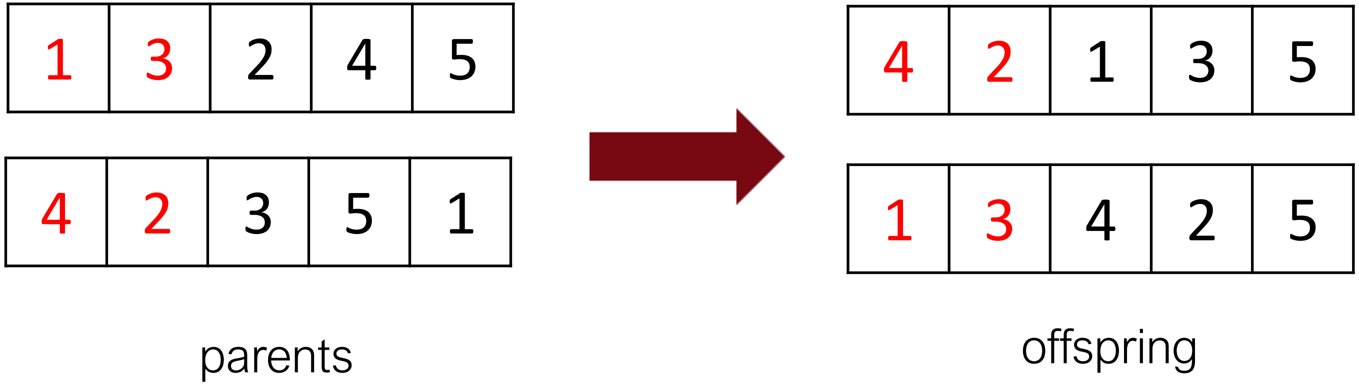
- Using the random.randint function to choose randomly 2 parents from the population, using the Single Point Crossover function to create 2 offspring, create a chance that happens mutation.
- Calculate the fitness of every individual in the population, sort population from the best to the worst. Keep 180 individual that have the best fitness and 20 that have the worst fitness.

Repeat 200 generations.

Genetic Algorithm

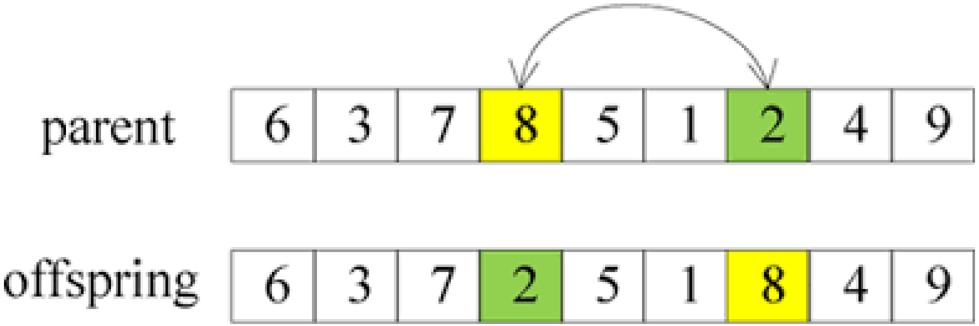
Single point crossover

Rate = 0.9



Swap mutation

Rate = 0.05



Comparing Results

Exact algorithms and greedy algorithms:

Data	Size	Backtracking		Dynamic Programming		Constraint Programming		Greedy 1	Greedy 2
		f	t(ms)	f	t(ms)	f	t(ms)	f	f
10x10data1	10x10	991	1012	991	119	991	22165	1273	1316
10x10data2	10x10	1086	1348	1086	76	1086	18950	1424	1968
10x10data3	10x10	1029	2581	1029	79	1029	12534	2913	1656
10x10data4	10x10	1083	2733	1083	87	1083	15040	1711	1526
10x10data5	10x10	1508	1134	1508	83	1508	36893	2234	1559

Comparing Results

Heuristic algorithms:

	Genetic Algorithm				
	f_min	f_max	f_avg	f_std	t_avg (ms)
10x10data1	991	1025	1018.2	13.6	5071
10x10data2	1086	1119	1099.2	16.1666323	5075
10x10data3	1029	1029	1029	0	5625
10x10data4	1083	1130	1103.2	18.03773822	5375
10x10data5	1508	1559	1527	18.78297101	5539
10x20data1	1230	1546	1377	124.9959999	9124
10x20data2	836	1004	932	69.09413868	8260
10x20data3	676	790	742	41.91419807	8806
10x20data4	705	961	858.2	85.25585024	8367
10x20data5	880	1146	1020.8	94.61162719	9153
100x50data1	1607	1922	1768.8	123.2808176	141733
100x50data2	1591	1838	1663.2	88.65979923	144452
100x50data3	1474	1621	1534.8	62.18488562	50374
100x50data4	1582	1842	1706	91.89776929	50331
100x50data5	1304	1511	1415.2	76.32404601	47400

Comparing Results

Heuristic algorithms:

	Simulated Anealing				
	f_min	f_max	f_avg	f_std	t_avg (ms)
10x10data1	1025	1146	1095.8	45.11053092	6291
10x10data2	1182	1318	1221.2	53.68947755	6391
10x10data3	1029	1284	1217.8	95.75050914	6980
10x10data4	1083	1155	1121.2	32.49861536	6727
10x10data5	1508	1656	1568.6	49.46352191	6923
10x20data1	1402	1691	1525.8	115.9161766	11544
10x20data2	1051	1170	1115.4	44.7821393	10471
10x20data3	778	903	853.4	41.14170633	11177
10x20data4	955	1194	1096.4	86.11294908	10564
10x20data5	1145	1328	1235.8	61.44070312	11596
100x50data1	2093	2214	2142.8	39.3822295	188428
100x50data2	1888	2177	2055.6	97.91138851	162168
100x50data3	1813	2270	1989.8	165.4054413	67339
100x50data4	1842	2064	1977.8	74.28701098	66544
100x50data5	1771	2115	1873.6	129.3701666	63794

Comparing Results

Heuristic algorithms:

	Variable Neighborhood Search				
	f_min	f_max	f_avg	f_std	t_avg (ms)
10x10data1	991	1161	1066.666667	70.64622346	365
10x10data2	1086	1242	1138	73.53910524	770
10x10data3	1029	1029	1029	0	825
10x10data4	1083	1130	1107.666667	19.25847577	600
10x10data5	1508	1601	1563.333333	29.14713632	532
10x20data1	1100	1272	1164.333333	76.61302471	38404
10x20data2	836	966	895.3333333	53.67391255	22062
10x20data3	661	730	689	29.63106478	17460
10x20data4	745	923	828	73.16192088	28254
10x20data5	844	952	880	50.91168825	40138
100x50data1	1164	1325	1230	68.85249935	30694
100x50data2	983	1113	1069	41.15823125	23084
100x50data3	938	1103	1027	67.98529253	21490
100x50data4	1153	1175	1160.666667	10.14341604	24359
100x50data5	870	1057	963.3333333	66.38440245	25534

Conclusion

Exact algorithms:

- Dynamic Programming algorithm appears to be the best exact algorithm with the fastest running time.
- The Backtracking algorithm with branch and bound techniques run faster than the Constraint Programming algorithm.

Greedy algorithms:

- Greedy algorithms produce surprisingly good results in much less running time in big test cases

Heuristic algorithms:

- For small tests, all 3 algorithms can reach the global optimum in 5 times testing.
- The VNS algorithm has stable running time and good results for all tests.
- The SA and GA appear to be bad when solving big test data (100x50 test size).
- For the GA test, the result is unstable for big data test and changing the parameter to sacrifice running time can return better optimum.

Assignment

Greedy Algorithm 1	Phan Duc Hung
Greedy Algorithm 2	Truong Gia Bach
Backtracking	Truong Gia Bach
Dynamic Programming	Truong Gia Bach
Constraint Programming	Tran Duong Chinh
Variable Neighborhood Search	Truong Gia Bach
Simulated Annealing	Truong Gia Bach
Genetic Algorithm	Phan Duc Hung
Data Collection and Analysis	Phan Duc Hung, Tran Duong Chinh
Slides	Phan Duc Hung, Tran Duong Chinh, Truong Gia Bach

KEY INSIGHTS ON

HOW TO END A PRESENTATION