Bài toán lấy hàng trong kho

Picking up packages in a warehouse problem

1 GROUP'S MEMBERS

Member's name	Student number	Email
Phan Duc Hung	20214903	hung.pd214903@sis.hust.edu.vn
Truong Gia Bach	20210087	bach.tg210087@sis.hust.edu.vn
Tran Duong Chinh	20210122	chinh.td210122@sis.hust.edu.vn

2 PROBLEM 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 q(i).
- 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 tổng quãng đường di chuyển là nhỏ nhất.

- In a warehouse there are many racks for packages $1, 2, \dots, M$. Suppose the j^{th} rack is placed at point $j, (j = 1, 2, \dots, M)$.
- 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 $Q(i,j)$ where $i = 1,2,\dots,N$ and $j = 1,2,\dots,M$.
- A warehouse staff starts at the entrance of the warehouse (point 0), need 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.

3 MODEL

Preprocessing: For this problem, we want to implement algorithms that can solve all datasets where the distance variable d might not satisfy the triangle inequality (d[a][b] + d[b][c] $<$ d[a][c]) or be asymmetric (d[x][y] $\neq$ d[y][x]). These kinds of dataset can lead to the solution(tour) where some racks are visited multiple times to optimize the tour length. This will create a much harder problem model. However, this issue can be easily solved using a preprocess part where the shortest distance between any two racks (travelling through some or none of other racks) is calculated. Here, we use the Floyd-Warshall algorithm for this preprocess part. It takes the complexity of $O(M^3)$, which is negligible with respect to further alogorithms' complexity.

We then proceed to solve the problem with this new distance data, this time each rack can only be visited atmost once in the optimal tour. The total length of the optimal solution is still conserved and we can also extend to the full tour of the "original" problem by keeping track of all the racks involved in the shortest path between any two racks. After this preprocess part, we build the following model for our problem.

Decision variables: have the form $x_{i,k} \in \{0,1\}$, where k represents the order of the rack in our route, and i represents the rack (i=0, 1, 2, ..., M). When $x_{i,k} = 1$, this means that the rack i is the kth rack that we visited on our route.

Constraints:

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

(each rack is only visited only once)

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

(each time only visit one rack)

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

(the total number of package of each type in the path the greater than the required amount)

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

(the path starts at point 0 and ends at point 0)

Objective function to minimize:

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

4 BACKTRACKING

Try all the tours possible formed by visiting all the racks or a partial set of racks in every order. Using backtracking, we generate all permutations of all sets of racks and save the shortest tour while checking the constraint of collecting enough packages of all types. The maximum complexity is $O(N \times M!)$. Since we will stop searching a branch if the current solution (or subtour) has already satisfied the constraint (i.e. collecting enough packages) as visiting more racks only increases the tour length, the actual running time is usually significantly faster in practical tests.

Branch and bound: We try to decrease the running time furthermore by maintaining an upper bound that equal to:

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

Here:

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

In our specific problem, by cutting all the branches that exceed this upper bound, all backtracking permutation paths can be much shorter, especially in tests while the number of packages in each rack is quite large with respect to the total packages demanded.

5 DYNAMIC PROGRAMMING

We implement another exhaustive search using the memoize dynamic programming algorithm, similar to the algorithm for the knapsack problem. The recurrence relation used for the dynamic programing algorithm is shown as follow:

$$F\left(\begin{array}{l}\text{racks visited,}\\ \text{last visit rack x,}\\ \text{current collected packages,}\\ \text{number of remaining}\\ \text{lacking package types}\end{array}\right) = \min_{\forall\ \text{unvisted rarck}} \left(F\left(\begin{array}{l}\text{racks visited + y,}\\ \text{an unvisited rack y,}\\ \text{update collected packages,}\\ \text{update number of remaining}\\ \text{lacking package types}\end{array}\right) + d[x][y]\right)$$

** From all the unvisted racks, return the minimum length of the function choosing each of those racks plus the distance from the last visited rack to the chosen one.*

Unlike the knapsack problem, our problem cares about the order of visiting (which results in tracking the last rack visited). Therefore, the memoization part (i.e. saving the best result at each state in order to access it immediately whenever a state with similar parameter values appears) is especially value since we will visit the same state multiple times providing that the previous partial tour is the same set of racks with different orders (the last rack visited is the same too, obviously).

6 CONSTRAINT PROGRAMMING

Our problem can be thought of as a constraint programming problem. Here we present a way to use OR-Tools to create the model for our problem and solve for the solution.

Our decision variables will have the form $x_{j,k} \in \{0, 1\}$ where k represents the order of the rack in our route, and i represents the rack $j = 0, 1, 2, \dots, M$). When $x_{j,k} = 1$, this means that the rack j is the k^{th} rack that we visited on our route.

Our pickup route starts at the gate (point 0), we interpret this as when $k = 0$, we are at point 0 (i.e $x_{0,0} = 1$). This is for convenience when inplementing the problem in code.

Because we visit each rack at most once and return to the the gate when we finish, k can have the values ranging from 0 to M. Let k_0 be the value of k when we return to the gate (i.e ending our route). We see that k_0 can have values ranging from 3 to M+1.

The constraints for our problem

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

(each rack is only visited only once)

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

(each time only visit one rack)

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

(the total number of package of each type in the path the greater than the required amount)

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

(the path starts at point 0 and ends at point 0)

We want to find the route that satisfies the above constraints while also being the shortest possible. Our objective function to minimize

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

The above constraints and objective function can be implemented in Python using OR-Tools' CP SAT solver.

7 GREEDY ALGORITHM 1

Using the searching technique to find the nearest racking. When we take enough packages of all types, immediately come back to the entrance from the current location.

```
path_cost += d(current_rack,0)
```

The case is good only when the number of packages of all types in all rackings is close to each other (not having a big gap between the number of packages of any type between 2 racks) and the number of packages that need to be taken is small. Because the algorithm only optimizes the distance cost, not taking too much concern in the number of packages needed to take.

8 GREEDY ALGORITHM 2

We also tried a different greedy idea for our problem. Unlike the previous greedy algorithm, instead of choosing the closest rack at each step, we choose the rack with the highest value of the index *(Number of packages) / (Distance from the current rack)*. It is worth noting that the number of packages here is the total number of meaningful packages of all types at that rack, in other words the number of packages that can be picked up until the demanded number is fulfilled (i.e. 0 for a package type if that package type is collected enough in previous racks). This index is taken using the following formula:

In practical testing, this greedy algorithm can be worse than the first greedy algorithm but can be more stable (i.e. return a "not too bad" result) as it takes into account all two factors (distance and number of packages) instead of only the distance factor in the first one.

9 GENETIC ALGORITHM

Because we only need to visit a racking one time to take all the packages on it, so in the worst case, we will visit all the racks, each rack is visited once.

I present the path in a list that is the permutation of (1, …, M) plus with the distance moving from the entrance to first racking that visit (D(0)), and the distance moving from the last racking that visit to the entrance (D(m+1))

Because in most cases we will not visit all the racks, so we only consider the list permutation of (1, …, M), taking 2 part D(0) and D(m+1) beside. The permutation (1, …, M) is the path and can end soon after having taken all the packages that are needed.

Create a population by using the `random.shuffle` function, the size of population is

```
pop_size = 200
```

In a generation:

- Using the `random.randint` function to choose randomly 2 individuals in the population to be the parents, using the Single Point Crossover to create 2 offspring from the parents.
- Repeat that 100 times. Create a possibility that is 10% that the cross fails.
- Create a possibility of 5% that happens the Single Swap Mutation, that in one random individual, 2 points will be swapped.
- Adding all the offspring to the population.
- Calculate the fitness (the optimize objective) of all the individuals in the population, sort the list from best to worst, keep 180 individuals that are the best and 20 that are the worst. Keep the `pop_size = 200`.

Repeat 200 generations to find the individual with best fitness is a local optimal.

Single Point Crossover:

- Choose a random partition in 2 individuals to be the crosspoint, fill the former part (the part that lasts from the beginning to the crosspoint) of parent 1 to offspring 2, and the former part of parent 2 to offspring 1.
- The latter part of offspring 1 will be the order of remaining number (from start to the end) of parent 1 that haven't been filled in the offspring 1 yet. We will place it from order until the offspring is fulfilled. The same goes for offspring 2 and parent 2.
- The rate that 2 parents participate in crossover is set at 0.9

Swap mutation:

- Choose 2 random positions in the offspring to be swapped
- The rate that Swap mutation is raised is set at 0.05

10 VARIABLE NEIGHBORHOOD SEARCH

We implement the variable neighborhood search to escape the local optimum in our heuristic search.

Initial solution: First we generate an initial solution which involves adding racks to the initial path at random to the optimal position until the package constraint is met. That is, every step, we choose a random rack that is not in the path yet and check all the positions in the current path (which initially only contain the gate as both the beginning and the end point, denoted by 2 indices 0) to find best position that increase the total tour length the least and add the rack there. We then remove the two 0s at two ends for easier handle later.

Iterated local search improvement: Usually when the initial solution is quite short with respect to M (i.e. usually less than half the number of racks are enough to fully collect the demanded number of packages in practical tests), we can extend the search by starting from more initial solutions by performing the above initial solution generation multiple times such that the next solution starts with the rack right after the ending rack of the previous solution. We continue to generate more initial solutions up until every rack has been included in at least one initial solution. We use this improvement to diversify the search as you will see in the later part that while choosing the neighbor state, we will prioritize removing existing racks in the current path rather than include racks that are not already there to optimize the total tour length.

Variable neighborhood: At each step, we explore the neighborhood that is determined by 4 different moves:

Move 1: Remove a rack from the current solution.

Move 2: Swap the position of a pair of racks of the current solution.

Move 3: Change the position of a rack without changing the order of other racks.

Move 4: Swap a rack in the current solution with a rack that is not there yet.

A move can only be accepted if the result solution also satisfies the constraint. Out of all acceptable moves, we will choose the best move (i.e. the shortest result tour). We apply the variable neighborhood search by extending the neighborhood by performing each move 1, 2 or maximum 3 times (of course the acceptance of

repeating a move 2 or 3 times is only checked after all). Notice that we only extend the neighborhood by repeating these moves in case we get stuck (i.e. cannot find any better solution repeating the current number of times these moves).

11 SIMULATED ANNEALING

The simulated annealing algorithm idea is that it is similar to the hill-climbing local search with an addition of the ability to accept worse neighbor states with a specific probability depending on each stage of the algorithm. There are 4 factors that we need to focus on: the initial state, the neighbor states, the objective function and the cooling schedule.

Initial State: A random permutation of M racks.

Neighbor States: An interval of the range from 1 to M is randomly selected. Notice that this interval can run from the back to the front of the permutation (e.g. the interval (8, 2) of a permutation of 10 racks contains 5 racks at indices 8, 9, 10, 1, 2). That interval is then reverse to form the potential new state.

* Tabu search improvement: We also use a tabu list to improve the performance of our simulated annealing algorithm. The tabu list, which consists M latest moves that are used to change the states, acts as a shorterm memory of the tabu search. It helps preventing the algorithm from revisiting a recent previous state and getting stuck in the same unappealing searching region.

Objective Function: Starting from the beginning of the current permutation of M racks, we take one rack at a time into our tour until the package constraint is met. The objective function is then the total length of that tour, plus the distance from the gate to the first rack and the distance from the last rack back to the gate.

Cooling Schedule: Beside from the initial state, the neighbor states and the objective function, the simulated annealing algorithm must have a cooling schedule which basically a function to change T during the algorithm that decide the acceptance probability of an worse neighbor state with respect to the objective function. This probability, which equal to …, must decrease over time (i.e. accept a wide range of solution at the early stage and mostly converge toward the local optimum later on), which mean that T also decrease over time. After some experiments, the following geometrical cooling schedule is chosen:

$$T_0 = 2000$$

$$T_k = T_0 \times \alpha^k \qquad k: 1 \rightarrow 20000$$

$$\alpha = 0.0003$$

12 DATA ANALYSIS

We put our algorithms to the test by running it with different data as input. In this part we summarize our findings and analize the results obtained.

Data	Size	Backtracking		Dynamic Programming		Constraint Programming	
		f	t(ms)	f	t(ms)	f	t(ms)
10x10data1	10x10	991	1012	991	119	991	22165
10x10data2	10x10	1086	1348	1086	76	1086	18950
10x10data3	10x10	1029	2581	1029	79	1029	12534
10x10data4	10x10	1083	2733	1083	87	1083	15040
10x10data5	10x10	1508	1134	1508	83	1508	36893

Table 12.1. The running time of exact algorithms with objective value f.

	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

Table 12.2. The analysis of running time and objective value f for Simulated Annealing.

	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

Table 12.3. The analysis of running time and objective value f for Variable Neighborhood Search

	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

Table 12.4. The analysis of running time and objective value f for Genetic Algorithm.

13 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 has faster running time than the Constraint Programming algorithm.
- The reason that Constraint Programming has slow running time is because we don't put in more constraints.

Greedy algorithms:

- Greedy algorithms produce surprisingly good results with much less running time in big test cases. The main reason is that the data that we generate for big test cases have the gap between the number of packages in 2 racks not so big, so by repeating many times choosing the nearest rack to visit, we can still take a good enough number of packages each type.

Heuristic algorithms:

- For small tests, all 3 algorithms can reach the global optimum in 5 times testing.
- The Variable Neighborhood Search algorithm has stable running time and good results for all tests.
- The Simulated Anealing appears to be bad when solving big test data (100x50 test size) when the results returned is not better than the result of Greedy algorithm. We believe the reason is that our neighbor states choosing method do not allow quick calculation of the objective function in asymmetric distance datasets (i.e. we have to calculate again the reversing interval). Therefore, in big testcases, the cooling rate can't be slow enough to escape the local optimum without exceeding the time limit.
- For the GA test, the result is unstable for big data tests, for some cases it can return very good optimum but for many cases the result can be worse than the result of greedy algorithms. Changing parameters such as pop_size, number of generations and mutation rate to sacrifice running time can return better optimum.

14 Possible Improvements

- Constraint Programming can be updated with more constraints that work for the brand and bound technique to reduce the running time.
- Simulated Annealing can be updated with better neighbor states choosing techniques that allow immediate value calculation of the new state.
- Genetic Algorithm can be built in with techniques to evaluate parameters that can be suitable for different tests.

15 Links and Related Information

The website of the project: https://github.com/hwnginsoict/Project-Optimization-Course-DSAI-HUST