Marking IDS:

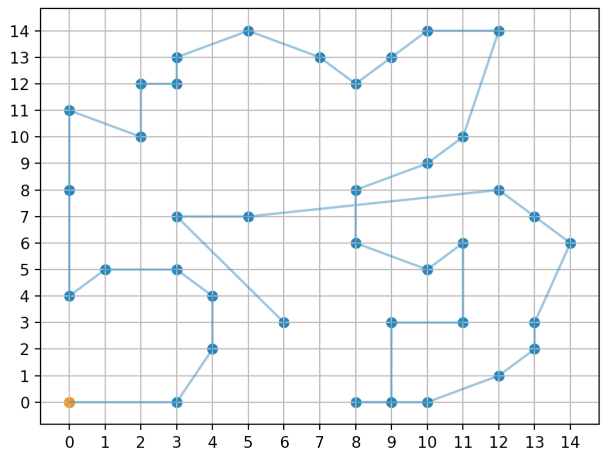
13123, 32149

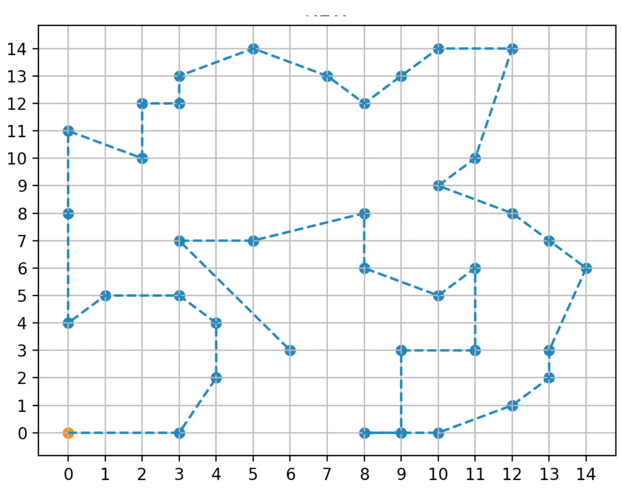
AI Principles and Techniques

15.12.2019

Travelling Salesman Problem

In the context of Vacuum World





Setting of the game

In the Vacuum World the cleaning agent has to clean the dirt in the most effective way. Organising the path in an effective way leads as to the Travelling Salesman Problem. So for the purpose of solving the TSP we are assuming that the agent knows where the dirt is.

To formalise the problem we consider the Vacuum World environment as an array of tuples of (d, x_i, y_j) type, with size $m x m$ (however the problem is equally solvable for $n x m$ size).

- d can have the value 0 or 1, where 0 is absence of dirt in the cell and 1 means dirt.
- x_i, y_j are the coordinates of the cell.

Using this notation we will first be able to sort the dirt cells which will be input in the search algorithms where the distance from one city to another will be calculated using the Manhattan distance ($= abs(x_a - x_d) + abs(y_a - y_d)$).

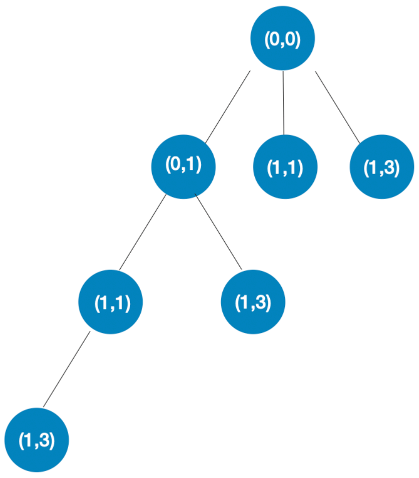
Search algorithms

The TSP is an NP-hard problem with $(n-1)!/2$ possible routes, where n is the total number of cities (in this case cells that need to be visited). In our relaxed problem we consider a route from point A to point B as having the same cost as from B to A (therefore the division by 2) and the route needs to start only from a specific point (the cell in which the agent is positioned initially).

We chose to test **Greedy** search and **Genetic** algorithm. However because Greedy search by itself gives sub-optimal results we decided to combine it with **2-OptSwap.**

The primary heuristic used is the **Manhattan distance** with which we calculate the distance between two cells and the overall cost of a route.

1. Further, the Greedy algorithm is following the **nearest neighbour method** by choosing the smaller cost path towards the next node. The search starts from agents' initial position, and calculates the distances towards all dirt cells, then selects the closest node as the next node, from where it again calculates the distance towards the remaining dirt cells and so on till it reaches the last cell.



As stated previously this algorithm gives sub-optimal results, and for this reason we are completing the search by using the **2-Opt heuristic**. In short terms, we are looping through the greedy resulted route and flipping sub-routes in the search of more cost effective paths. In order to avoid trying all the possible routes we are breaking the 2-opt loop when the result is not improving anymore. In this way obtaining a result in an optimal time period.

2. The Genetic algorithm implementation consists in the repetition of following stages:

1. Creating population (comprised of x randomly shuffled routes/individuals)
2. For n generations repeat:
    1. Ordering the population based on a score (1/cost)
    2. Creating the mating pool out of the best individuals
        1. Giving more chances to best routes (elite) to appear multiple times
    3. Breeding using cross-over method
        1. Keeping the elite
        2. Creating the offspring population

    4. Mutating and creating a new generation
        1. Keeping the elite
        2. Mutating the offsprings
3. End

The optimum parameters found after many iterations are the following:

- Elite size = 30
- Sample = 800
- Generations = 800
- Mutation rate = 0.4

However, these parameters should change and adapt based on the size of the route that needs to be calculated (number of cells/cities).

Table of comparison between the three approaches:

Greedy		Greedy & 2OptSwap		Genetic Algorithm	
Route size	Cost	Route size	Cost	Route size	Cost
6	9	6	9	6	9
8	25	8	21	8	21
8	25	8	25	8	20
10	27	10	27	10	26
11	21	11	21	11	18
12	39	12	33	12	33
13	20	13	20	13	19
15	42	15	38	15	38
18	48	18	42	18	45
20	60	20	52	20	58
21	54	21	45	21	45
22	70	22	66	22	96
36	101	36	90	36	184

The average time of runtime:

- For the first two search approaches is less than 1 second
- For the Genetic Algorithm is ~45 seconds
    - ~30 seconds for low number of cells (less than 20)

- ~ 70 seconds for bigger numbers of cells (30 - 40)
- This happens especially due to the numbers of generations/iterations set to 800 and the number of sample population set to 800.

It is easy to conclude that this implementation of Genetic Algorithm outperforms even Greedy+2OptSwap for small number of cities. However for bigger numbers of cells the current implementation of the GA can give results even worse than the Greedy search by itself.

The current implementation of the mutation changes the position of only two cells between themselves, which for a big number of cells could mean too little. In this way even if mutating we can be drawn to local convergence.

More advanced implementation of GA can be comprised of:

- Starting from an already manipulated sample of population (giving as input the greedy resulted route and creating shuffle samples out of it)
- Rank-scaled mutation rate
- Four permutation rules-based heuristic

The nature of the GA limits us from identifying the exact number of cells from which on it starts to give sub-optimal solutions. On an average we could estimate that this implementation of GA is highly likely to give sub-optimal solution for more than 20 cells.

To conclude neither of the implemented algorithms will be able to give the most effective route in term of cost for bigger numbers than 20 cells.

Extension - PDB implementation

The objective to have a pattern database is to have a complete exploration of the abstract state spaces. Most frequently PDB's correspond to external Breadth First Search with delayed duplicate detection. The construction of external pattern databases is especially suited to frontier search algorithms like Greedy and A-Star, as no solution path has to be reconstructed.

TSP and PDB

In order to limit the possibilities of the abstract state space, we've built a pattern database which is suitable for the following scenarios:

- The agent is always starting from the same position i.e. (0,0)
- The number dirt locations are constant.

In case of TSP it is not suggested to have a pattern database as there is no confirmed final goal state. The main goal is to travel all the points with the least possible cost. In such scenarios, greedy algorithms in combination with 2-OptSwap and genetic algorithms show much better results as shown in the previous sections.

Properties

Considering:

- the number of dirt locations are 10
- the biggest ‘m’ stored in our database is 5

The possible number of solutions with delayed duplicate detection can be given by

- nPr * rPr = 10P5 * 5P5 = 3628800.
- This is a faster process as we are not checking the duplicate nodes immediately and we are just storing them in the memory as and when a new node is generated.

The drawback of the above process is that it requires a lot of memory when the nodes are generated.

In case of TSP, if we are using a PDB we can have similar solutions for different partial problems. Hence, even if we are having duplicate solutions we need to store it for all possible combinations of a partial set of locations. The table below is an example of such situations.

starting_point	locations	path_solution	path_solution_cost
(0, 0)	((1, 0), (2, 0), (3, 0))	((0, 0), (1, 0), (2, 0), (3, 0))	3
(0, 0)	((1, 0), (3, 0), (2, 0))	((0, 0), (1, 0), (2, 0), (3, 0))	3
(0, 0)	((2, 0), (1, 0), (3, 0))	((0, 0), (1, 0), (2, 0), (3, 0))	3
(0, 0)	((2, 0), (3, 0), (1, 0))	((0, 0), (1, 0), (2, 0), (3, 0))	3
(0, 0)	((3, 0), (1, 0), (2, 0))	((0, 0), (1, 0), (2, 0), (3, 0))	3
(0, 0)	((3, 0), (2, 0), (1, 0))	((0, 0), (1, 0), (2, 0), (3, 0))	3

Process of building the PDB

Conditions:

- Grid Size → 5 x 5 ⇒ 25 cells
- Starting Point → (0,0)
- Number of dirt locations → (min. 2, max 12)

By default, based on the number of dirt locations, a pattern database of with m = floor[(number of dirt locations)/2] will be created.

- E.g. Number of Dirt Locations = 7
- m = floor[7/2] ⇒ m = 3

A breadth first search is performed by creating all possible permutations of ‘m’ dirt locations.

For each ‘m’, the path solution will be all permutations of the dirt locations with (0,0) as the starting location.

In the following table we see the solutions for dirt locations (1, 0), (2, 0) and (3, 0) is present. There are 6 possible solutions and for each solution the path cost is calculated by computing the pairwise manhattan distance.

starting_point	locations	path_solution	path_solution_cost
(0, 0)	((1, 0), (2, 0), (3, 0))	((0, 0), (1, 0), (2, 0), (3, 0))	3
(0, 0)	((1, 0), (2, 0), (3, 0))	((0, 0), (1, 0), (3, 0), (2, 0))	4
(0, 0)	((1, 0), (2, 0), (3, 0))	((0, 0), (2, 0), (1, 0), (3, 0))	5
(0, 0)	((1, 0), (2, 0), (3, 0))	((0, 0), (2, 0), (3, 0), (1, 0))	5
(0, 0)	((1, 0), (2, 0), (3, 0))	((0, 0), (3, 0), (1, 0), (2, 0))	6
(0, 0)	((1, 0), (2, 0), (3, 0))	((0, 0), (3, 0), (2, 0), (1, 0))	5

Timing

- **N** → Number of Dirt Locations present in the grid
- **M** → Maximum number of Dirt Locations in the Pattern Database

PDB Size (in MB) → The PDB is saved as a compressed pickle file in the hard disk. If the pickle file is loaded, it will be a dictionary with below keys:

- **'starting_point'** → Starting location of the agent. This will be always (0,0). As stated earlier, in order to contain the possibilities of the abstract state space we will always consider the starting location of the agent as (0,0)
- **'locations'** → A subset of the dirt locations, which is a tuple of tuples.
- **'path_solution'** → A solution to the partial problem i.e. subset of the dirt locations, obtained by a breadth first search, which is a tuple of tuples arranged in the order of visit.
- **'path_solution_cost'** → An integer value which is the cost of path solution obtained by the breadth first search.

Time → Time taken to form the PDB

n	m (less than 5 mins)	Time (HH:MM:SS:MS)	pdb size (in MB)
2	2	00:00:00.03	0.000314
3	3	00:00:00.04	0.0014
4	4	00:00:00.06	0.0235
5	5	00:00:00.51	0.606
6	6	00:00:18.49	23.74
7	6	00:02:10.28	169.17
8	5	00:00:27.15	36.13
9	5	00:01:05.24	81.38
10	5	00:02:01.64	162.85
11	5	00:04:02.35	298.63
12	5	00:04:57.13	512

The table shows the biggest 'M' such that constructing the PDB takes less than 5 minutes for each 'N' with starting location as (0,0).

References:

2-opt:

https://essay.utwente.nl/72060/1/Slootbeek_MA_EEMCS.pdf
https://ocw.mit.edu/courses/sloan-school-of-management/15-053-optimization-methods-in-management-science-spring-2013/lecture-notes/MIT15_053S13_lec17.pdf
http://pedrohfsd.com/2017/08/09/2opt-part1.html

Genetic Algorithm:

https://pastmike.com/traveling-salesman-genetic-algorithm/
https://www.tandfonline.com/doi/full/10.1080/25765299.2019.1615172
https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.259.7013&rep=rep1&type=pdf
https://www.tandfonline.com/doi/full/10.1080/25765299.2019.1615172

PDB:

https://www.aaai.org/Papers/JAIR/Vol22/JAIR-2209.pdf
https://onlinelibrary.wiley.com/doi/pdf/10.1111/0824-7935.00065
https://www.sciencedirect.com/topics/computer-science/pattern-database
https://www.ijcai.org/Proceedings/03/Papers/267.pdf