Inteligência Artificial

Projeto 1

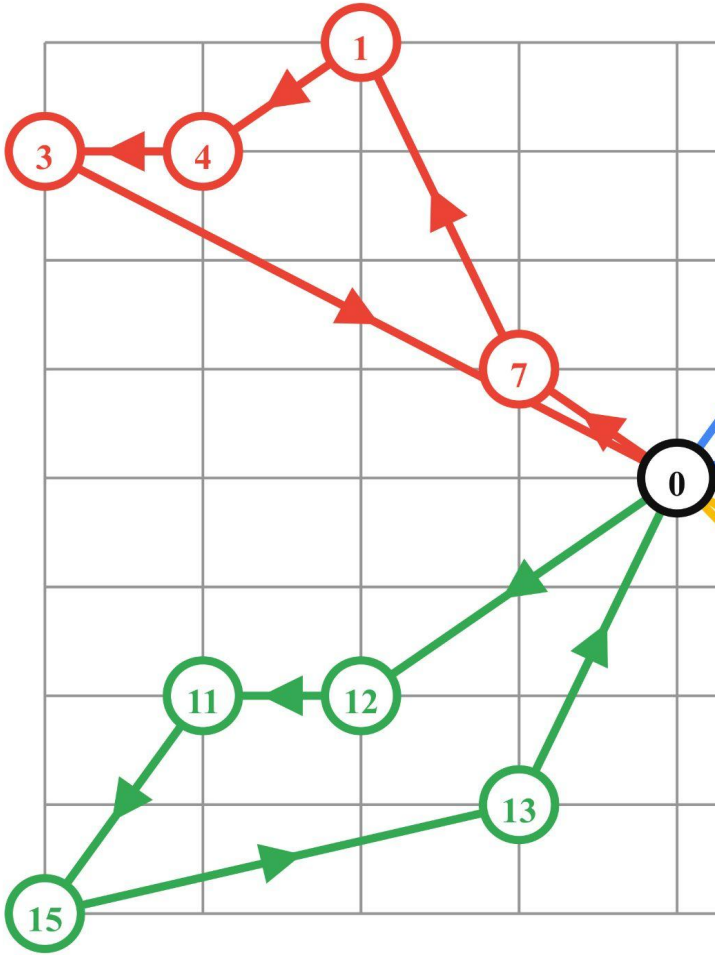
Especificação Problema

3A) Minimização do tempo de viagem

Versão simplificada do problema da ASAE de inspeção de estabelecimentos -> **Vehicle Routing Problem with Time Windows** - com restrição de horários (VRPTW)

Características:

- Conjunto finito de veículos (k) que têm de partir e chegar ao mesmo local (Depot)
  *k = floor(0.1 * nº_estabelecimentos) -> 10%*
- Partida às 9:00 AM de todos os veículos e o tempo de horas de trabalho é ilimitado
- A cada veículo é associada uma rota que contém determinados estabelecimentos de inspeção
- Cada estabelecimento só é inspecionado uma vez
- Cada estabelecimento só pode ser inspecionado se estiver aberto (senão tem de esperar). A partir do momento que o estabelecimento começa a ser inspecionado, a hora a que o estabelecimento fecha é irrelevante
- Objetivo: inspecionar todos os estabelecimentos no mínimo tempo possível (tempo = tempo viagem + espera + inspeção)



Dataset

“establishments.csv”
informação geral de 1000 estabelecimentos (id, nome, inspection time (minutos), horas em que o estabelecimento está aberto (vetor binário com tamanho=24), utilidade (importância para a sociedade), outros dados em princípio inúteis para o nosso problema)

“distances.csv”
distâncias (em segundos) entre todos os estabelecimentos (tabela 1000x1000, excluindo as linhas dos indexes)
Nota: linha (partida) e coluna (chegada) **??**

Output

Solução: para cada veículo, lista de estabelecimentos inspecionados (ID) por ordem

Métricas: tempo total de viagem, tempo total de espera, número de veículos utilizados, valor de utilidade para cada veículo, número de estabelecimentos inspecionados (deverá ser sempre todos)

Métricas de performance: tempo total de execução do algoritmo, tempo até atingir solução ótima, número de iterações até atingir solução ótima, número total de iterações

Bibliografia

Wikipedia

Introdução do problema Vehicle Routing Problem

- Generalização do TSP (Travelling Salesman)
- Solução ótima - NP HARD

(https://en.wikipedia.org/wiki/Vehicle_routing_problem)

Science Direct

Artigos científicos com informação prática e teórica sobre meta-heurísticas no Vehicle Routing Problem

(https://www.sciencedirect.com/topics/economics-econometrics-and-finance/vehicle-routing-problem)

Tese Mestrado Minho

Sobre Vehicle Routing Problem (inclui VRPTW) Informação, maioritariamente teórica, sobre o problema, possíveis restrições, algoritmos greedy, algoritmos exatos (branch and bound) e 3 meta-heurísticas (http://repositorium.sdum.uminho.pt/handle/1822/22289)

Capítulo Springer

Capítulo sobre VRPTW e algoritmos genéticos

(https://link.springer.com/chapter/10.1007/978-981-13-0761-4_52)

Formulação do Problema - *otimização*

Representação solução	Para cada veículo: lista com os estabelecimentos visitados (ID) por ordem
Vizinhança/Mutação/Crossover	Sempre tendo em conta as restrições: 1. Escolher um estabelecimento aleatório e trocá-lo para um veículo diferente (aleatório) 2. Trocar 2 estabelecimentos de diferentes veículos (escolha aleatória) 3. Trocar a ordem de 2 estabelecimentos dentro do mesmo veículo (Ainda pretendemos investigar mais)
Restrições:	• Cada veículo começa e acaba no mesmo sítio (Depot) • Máximo k veículos • Cada estabelecimento só é inspecionado 1 vez • Todos os estabelecimentos são inspecionados
Função avaliação:	1º - Cálculo do tempo total para cada veículo (tempo viagem + espera + inspeção) 2º - Cálculo do máximo desses valores (calculando assim o tempo que demora o carro que tem o trajeto mais longo, ou seja, chega de regresso depois de todos os outros)
Extra - Solução inicial:	Utilizar todos os veículos disponíveis, distribuir aleatoriamente todos os estabelecimentos e cada veículo começa e acaba no Depot

Implementação

Linguagem

Python

Bibliotecas

Pandas: importação de dados de csv

NumPy: processamento de grandes matrizes e funções matemáticas

Matplotlib: criação de gráficos

Meta-heurísticas

- Hill-climbing
- Simulated annealing
- Tabu search (evitar ciclos e soluções recentemente analisadas)
- Genetic algorithms

Visualização

Visualização gráfica (com informação extra em texto) da evolução da qualidade da solução

Interface

Interface em linha de comando para selecionar um algoritmo e os diferentes parâmetros de execução

Nota: uma vez que o problema que escolhemos era de otimização, decidimos esperar pela aula teórica sobre este assunto (2ª-feira) para iniciar o trabalho

Approach

- For evaluation, we created 2 functions: evaluate_solution_with_timewindow and evaluate_solution_without_timewindow. Both are responsible for evaluate our solution, returning the time that our solution would take to achieve the objective of the project. The difference between them is that the second one doesn't consider the time window (waiting time).

- The only heuristic that we used was the fitness of the solution, calculated by the functions above.

```python
def evaluate_solution_without_timewindow(solution: dict):
    values = []
    for cars in solution.values():
        time = 0
        for i in range(1, len(cars)):
            origin = cars[i-1]
            destiny = cars[i]
            time += (distances[destiny][origin])/60 #trip time (in minutes)
            if(destiny != 0): #dont inspect origin
                time += establishments[destiny][1] #inspection time (in minutes)
        values.append(time)

    # print("VALUES: ", values)
    return max(values)
```

```python
def evaluate_solution_with_timewindow(solution: dict):
    values = []
    for cars in solution.values():
        total_time = 0 #in minutes
        hour = 9 #in hours
        for i in range(1, len(cars)):
            origin = cars[i-1]
            destiny = cars[i]

            #trip time
            trip_time_min = (distances[destiny][origin])/60
            trip_time_hour = trip_time_min/60

            total_time += trip_time_min #trip time (in minutes)
            hour = (hour + trip_time_hour)%24

            #waiting time
            waiting_time_hour = calc_wainting(hour, destiny)
            waiting_time_min = waiting_time_hour * 60

            total_time += waiting_time_min #waiting time (in minutes)
            hour = (hour + waiting_time_hour)%24

            #inspection time
            if(destiny != 0): #dont inspect origin
                inspection_time_min = establishments[destiny][1]
                inspection_time_hour = inspection_time_min /60

                total_time += inspection_time_min #inspection time (in minutes)
                hour = (hour + inspection_time_hour)%24

        values.append(total_time)

    #print("VALUES: ", values)
    return max(values)
```

Algorithms

Hill Climbing

Starts with a random solution, then the algorithm calculates the score of a set of random neighbours and the best one is compared with the actual solution, choosing the best one.

Simulated Annealing

Starts with a random solution, then we have a temperature that is equal to 1000 and is decreased in each iteration. Then the algorithm calculates the score of a random neighbour: if is lower than the actual solution its is accepted, else it can be accepted with a probability of e^(fitness/temperature).

Tabu Search

Starts with a random solution, then generates a list of candidates out of current solution. Chooses the best one that is not in the tabu_list and makes it the current_solution and adds it to the tabu-list. Then, if it is lower than the best_solution, updates the best solution.

Genetic

Starts with the first element of the population as the best solution, and in each iteration, we select 2 members of the current population: one by tournament (best fitness) and the other by roulette (probability). We create 2 new children by making crossover of both and, with a given probability, the children can suffer mutation. Then we replace the 2 worst elements of the population by these new ones and we compare the members of the new population with the best solution, and if one is better that the best solution, we choose it

Results analysis

Total iterations - 1000, Total iterations without improvement - 500, 3 measurements for each value

Number of Establishments	Hill Climbing		Tabu search (tabu list=7, candidates=10)		Genetic (population=10)		Simulated Annealing	
	Time (s)	Solution (min)	Time (s)	Solution (min)	Time (s)	Solution (min)	Time (s)	Solution (min)
20	0.079	1040.077	1.167	958.613	0.898	1699.283	0.412	1563.260
100	0.279	1744.493	4.676	1494.723	4.279	2298.413	1.020	2674.350
500	1.228	2590.957	21.018	1941.95	18.225	3074.76	6.830	3084.210
1000	2.394	3033.637	37.093	2945.15	35.108	3140.04	17.560	3236.030

Conclusion and References

- In conclusion, the use of different optimization algorithms has been an effective approach to address optimization problems. Each algorithm has its strengths and limitations, and selecting the most suitable algorithms depends on the specific problem that we are dealing with.
- Hill climbing is a simple and fast algorithm, but can get stuck in local optima. Tabu search algorithm can overcome this limitation by avoiding revisiting previously visited solutions by keeping track of a tabu list. Simulated annealing introduces randomness into the search process to avoid being trapped in local optima and has the ability to escape from them, but it requires a cooling schedule to control the temperature parameter. Genetic algorithm uses the concept of natural selection and genetics to evolve a population, which can lead to a more diverse set of solutions, but is is computationally expensive.
- So, the combination of different algorithms showed promising results in improving the quality of the solutions and reducing the search time.

References

- https://www.geeksforgeeks.org/simulated-annealing/
- https://machinelearningmastery.com/simple-genetic-algorithm-from-scratch-in-python/
- https://smartmobilityalgorithms.github.io/book/content/TrajectoryAlgorithms/TabuSearch.html