ensia The National School of Artificial Intelligence المدرسة الوطنية العليا للذكاء الاصطناعي

Introduction to Artificial Intelligence

Lab 9 (Week 11) -Search Algorithms: Part 6 - Beyond classical search (Genetic Algorithm)

2023 - 2024

April 29^{th}, 2024

Objectives

- Implementation of a genetic algorithm for Job assignment problem on Py_search library.

Overview:

In LAB 9, you will delve into an interesting open-source library designed to handle problems using search strategies. It employs the same logic as we saw in previous labs, leveraging a general graph (or tree) search algorithm and building upon it the different uniformed and informed search strategies.
For the purposes of this lab, we will focus solely on one problem: the Job Assignment Problem. More specifically, we will explore a local version of this problem designed to accommodate local search strategies.

The Job Assignment Problem (JAP) is a combinatorial optimization problem concerned with efficiently assigning a set of tasks or jobs to a set of workers or resources. The objective is to minimize the overall cost or maximize the overall profit while satisfying certain constraints.
In its simplest form, the JAP involves assigning a set of jobs to a set of workers, each with their own associated costs or profits (Table 1). The goal is to find the assignment that minimizes the total cost or maximizes the total profit, subject to constraints such as job-worker compatibility and worker availability.

Table 1: Job Assignment Cost matrix

	$Taks1$	$Taks2$	...	$Taksk$
$Worker1$	Cost w_1t_k	Cost w_1t_2	...	Cost w_1t_k
$Worker2$	Cost w_2t_1	Cost w_2t_2	...	Cost w_2t_k
⋮	⋮	⋮	⋮	⋮
$Workerk$	Cost w_kt_1	Cost w_kt_2	...	Cost w_kt_k

You are provided with an attached Python notebook that contains the local search part of the Py_Search library, along with additional classes and functions to account for exploiting Genetic Algorithms to solve the Local Assignment Problem.

Links

To access the entire library, please refer to

- Py Search source code: `https://github.com/cmaclell/py_search`
- Py Search documentation: `http://py_search.readthedocs.org`

Your mission

Your mission in this lab primarily focuses on two tasks: understanding the business logic of the PySearch library (or at least the general flow of the provided components), and filling in the missing parts of the given methods within the GeneticAlgorithm class. The methods that require adjustment are as follows:

- `initialize_population`: to generate the initial population.
- `pmx_crossover`: stands for partial mapping crossover.
- `mutate`: to perform mutation on eligible individuals.
- `evolve_population`: to evolve to other generations.
- `run`: responsible for running the Genetic Algorithm.

```
def initialize_population(self):
    population = []
    for _ in range(self.population_size):
        individual = ... # Which problem method allows you to  create a random node
        population.append(individual)
    return population
```

```
def pmx_crossover(self, parent1, parent2):
    """Perform Partial Mapped Crossover (PMX) on two parent chromosomes."""
    # Choose two random crossover points checkpoint1 and checkpoint2 to create a
    crossover segment (in range of 0 and parent length)
        ...
    # Create copies of parent1 and parent2 to obtain offspring1 and offspring2.
        ...
    # Make sure checkpoin1 < checkpoin2
        ...
    # Copy the values from parent1 to offspring2 and from parent2 to offspring1 for the
    crossover segment (between checkpoint1 and checkpoint2)
        ...
    # Create a mapping that aligns the parents' values within the crossover segment
        ...
    # Update offspring1 and offspring2 by replacing duplicated values outside the
    crossover segment with mapped values
        # Use visited_indices set to avoid an infinite loop
        ...
    return offspring1, offspring2
```

```python
def mutate(self, individual):
    mutated_individual = individual
    if random.random() < self.mutation_rate:
        mutated_individual = ... # Which problem method allows you to perform mutation (
    swapping) ?
    return mutated_individual
```

```python
def evolve_population(self, population):
    ... # Select two parents (individuals) from the population
    if random.random() < self.crossover_rate:
        offspring1, offspring2 = self.crossover(parent1, parent2)
    else:
        ... # What should you do here ?
    ... # Mutate offspring1
    ... # Mutate offspring2
    population.extend([offspring1,offspring2])
    return population
```

```python
def run(self, num_generations):
    population = ... # Generate the initial population
    print(f"Initial population's best individual : {...}")
    for i in range(num_generations):
        print(f"*********Generation {i}**********")
        population = ... # Evolve !
        best_generation = ... # Get the best individual of this generation
        print(f"Generation {i}, Best Individual:{best_generation}")
    best_individual = ... # Get the final best individual
    solution_node = ... # Create a solution node out of best_individual
    yield solution_node
```

```python
""" Main section"""

    print("########################################")
    print("Genetic Algorithm")
    print("########################################")

    # ... Create your genetic algorithm object and run it.
```

Task 2

The **compare_searches** function is responsible for executing and comparing different strategies on the same problem to assess their performance.

- Incorporate the newly created Genetic Algorithm into the **compare_searches**.