Object Oriented Programming 2023/24

Optimal patrol allocation of planetary systems using Evolutionary Programming

MEEC – IST

1 Problem

The empire has a set of n patrols $P = \{p_1, \ldots, p_n\}$ responsible for maintaining order in m planetary systems $A = \{a_1, \ldots, a_m\}$ to counter the movements of the rebels. Each patrol p_i requires $c_{ij} \in \mathbb{N}^+$ units of time to pacify the system a_j. The time it takes for patrols to move from one planetary system to another is considered negligible (as the patrol ships travel through hyperspace).

The goal is to find an allocation of the planetary systems among the empire's various patrols that minimizes the time required to pacify the entire empire. This is due to the emperor's concern that if an area is not quickly patrolled, the rebels become too powerful to control. An allocation of the planetary systems is essentially a partition of $A = A_1 \cup \ldots \cup A_n$ where A_i corresponds to the planetary systems patrolled by p_i. The time t_i that a patrol p_i spends monitoring the systems in A_i is calculated as follows:

$$t_i = \sum_{j:a_j \in A_i} c_{ij}.$$

The objective is to find a distribution of the planetary systems $A = A_1 \cup \ldots \cup A_n$ that minimizes the total patrolling time t, computed as:

$$t = \max_{1 \leq i \leq n} t_i.$$

For instance, assume the empire has three patrols $P = \{p_1, p_2, p_3\}$ and controls 6 planetary systems $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$. The time required for each system to be patrolled by each patrol is represented in the following 3×6 matrix:

$$C = \begin{pmatrix} 1 & 2 & 1 & 1 & 2 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 \end{pmatrix}.$$

The objective is to identify an allocation of the planetary systems $A = A_1 \cup A_2 \cup A_3$ that minimizes the total time t required for patrolling the entire empire. In this scenario, one possible allocation is $A_1 = \{a_1, a_3, a_4, a_6\}$, $A_2 = \{a_2, a_5\}$, and $A_3 = \emptyset$, for which $t = \max\{1 + 1 + 1 + 1, 2 + 2, 0\} = 4$. It should be noted that there are alternative allocations that can patrol the empire within the same timeframe.

1.1 Police patrol as an open-shop scheduling problem

The Open Shop Scheduling Problem (OSSP) is a complex optimization challenge that arises in operations research and computer science, particularly within the context of manufacturing and production planning. The goal of the OSSP is to assign various tasks to machines or workstations, where each task requires processing on specific machines for certain durations and can be processed on any machine at any time, without a predefined sequence. The primary objective is to minimize the total completion time, known as the *makespan*, ensuring that no machine processes more than one task at a time and no task is processed by more than one machine simultaneously.

This problem is a notable example of NP-hardness, a classification in computational complexity theory that encompasses decision problems for which a solution can be verified in polynomial time, but no known algorithm can solve all instances of the problem in polynomial time. Specifically, the OSSP belongs to the subset of NP-complete problems, indicating that it is at least as hard as the most difficult problems in NP and that a polynomial-time algorithm solving one NP-complete problem could solve all NP problems.

Despite the relative simplicity of finding a solution for small instances of the OSSP, as in the given example, no polynomial-time algorithm is known for solving it in general. This lack of a polynomial-time solution is not just a minor inconvenience; it reflects a profound question in computer science related to the P vs NP problem, one of the seven Millennium Prize Problems. A solution to this problem, either by finding a polynomial-time algorithm for NP-complete problems or proving that no such algorithm exists, carries a reward of one million U.S. dollars.

Given the computational infeasibility of solving large instances of the OSSP exactly – where even a hypothetical 10GHz computer would take approximately 1200 centuries to find an optimal solution for a problem size of $n \times m = 75$ – researchers and practitioners often turn to heuristic and approximate methods. Among these, evolutionary programming stands out as a particularly effective approach.

Heuristic and metaheuristic approaches that can be applied to the OSSP include:

- Evolutionary Algorithms: Simulates the process of natural selection, wherein species that can adapt to environmental changes are able to survive and reproduce, thereby passing on successful adaptations to the next generation and progressively moving closer to an optimal solution. A relevant case of evolutionary algorithms is genetic algorithms, where the reproduction of individuals allows crossing over.

- Ant Colony Optimization: Mimics the foraging behavior of ants to find optimal paths through the solution space. This algorithm is based on the pheromone trail laid down by ants, which is used by other ants to find the path to food.

- Simulated Annealing: A probabilistic technique that explores the solution space by accepting both improvements and, with decreasing probability over time, certain degradations in solution quality. This method is inspired by the annealing process in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects.

- Tabu Search: Utilizes memory structures to avoid revisiting previously explored solutions, thereby encouraging the exploration of new areas of the solution space. This approach is particularly useful for solving optimization problems by guiding the search process away from less promising regions.

While these methods do not guarantee finding the optimal solution, especially for large and complex instances of the OSSP, they often yield sufficiently good solutions within a reasonable amount of computational time, making them practical for real-world applications.

1.2 Solving the OSSP through evolutionary programming

Evolutionary programming, inspired by the principles of biological evolution, employs iterative processes of mutation, selection, and reproduction to enhance a population of candidate solutions. In the context of challenges like the OSSP, evolutionary programming is adept at generating a variety of solutions across successive generations. This approach methodically advances towards more optimal schedules by effectively navigating through an extensive search space, a task that is considerably more efficient than methods based on exhaustive search.

One of the primary challenges in evolutionary programming is identifying the most fitting individual – or solution – within a potentially vast and complex search space. This difficulty arises from several factors:

- Vast Search Space: The search space in many problems, including the OSSP, can be exponentially large, making it impractical to evaluate every possible solution directly. Evolutionary programming aims to explore this space efficiently, but finding the absolute best solution without exhaustive search requires intelligent navigation strategies.
- Premature Convergence: There is a risk of the algorithm converging too early on suboptimal solutions. This happens when the population loses diversity too quickly, leading to stagnation where further iterations do not significantly improve the solutions.
- Balance between Exploration and Exploitation: Achieving the right balance between exploring new areas of the search space (exploration) and refining the best solutions found so far (exploitation) is critical. Too much exploration can lead to inefficiency and a lack of focus, while too much exploitation can cause premature convergence on suboptimal solutions.
- Evaluation Complexity: The fitness evaluation, which determines how well a solution meets the problem criteria, can be computationally expensive for complex problems. This makes the process of identifying the best-fitted individual time-consuming, especially when the population size is large or the number of generations is high.

1.3 Stochastic simulation

Stochastic simulation is a computational method used to model systems that have some degree of uncertainty or randomness. Unlike deterministic simulations, where the same set of initial conditions always produces the same result, stochastic simulations incorporate random variables and probabilistic behaviors to account for variability in outcomes. This approach is particularly valuable in analyzing and predicting the behavior of complex systems where uncertainty is inherent or where exact predictions are impossible due to the randomness involved.

Combining evolutionary programming with stochastic simulation offers a powerful approach to solving complex optimization and modeling problems. This synergy leverages the strengths of both methodologies, providing several advantages, such as handling uncertainty and variability, efficient exploration of complex search spaces, and improved solution quality and robustness.

Please refer to the lecture slides for the stochastic simulation of a toll highway.

2 Approach

This project aims to program a solution to the problem presented above in Java using evolutive programming modeled and implemented with objects.

2.1 Individual and population

The idea is to generate a **population** of ν individuals at instant zero and make it evolve until the final instant τ. An **individual** is a distribution of the planetary systems among the patrols of the empire.

For individuals of the initial population, the planetary systems are distributed randomly and uniformly among the patrols. The evolution of an individual modifies its distribution to find one that minimizes the time needed to police the whole empire.

2.2 Comfort and best-fitted individual

To each individual z it is associated a **comfort** $\varphi(z)$ given by:

$$\varphi(z) = \frac{t_{\min}}{t_z}$$

where:

- t_z is the time to patrol the empire given by the individual z;
- $t_{\min}$ it is a lower bound for the optimal time to patrol the empire, given by:

$$t_{\min} = \frac{\sum_{j=1}^{m} \min_{1 \le i \le n} c_{ij}}{n}.$$

Note that the comfort is a real value in (0,1] and that the most fitted individuals have a comfort value close to one or even one. The individual with greater comfort is termed as the **best-fitted individual** (in the event of existing more than one individual under these circumstances, it is considered one at random).

2.3 Random laws

Each individual z evolves according to its comfort by the following random mechanisms:

- **Death**, exponential variable with mean value $(1 - \log(1 - \varphi(z)))\mu$ for the death events.
- **Reproduction** is modeled as an exponential variable with a mean value of $(1 - \log(\varphi(z)))\rho$ between reproduction events. Following reproduction, a new individual is born, inheriting the distribution of planetary systems among the patrols from its parent, up to $\lfloor (1 - \varphi(z))m \rfloor$ systems. That is, the distribution of planetary systems in the offspring is determined as follows:
  (i) initially, it receives a distribution identical to that of the parent;
  (ii) from this initial distribution, $\lfloor (1 - \varphi(z))m \rfloor$ planetary systems are randomly and uniformly removed from the various patrols; and
  (iii) the planetary systems that were removed are then randomly and uniformly redistributed among the patrols.

- **Mutation**, exponential variable with mean value $(1 - \log(\varphi(z)))\delta$ between (mutation) events. The mutation of an individual involves randomly and uniformly removing a planetary system from one patrol, p_i, and then randomly and uniformly placing it into another patrol, p_j, where $i \neq j$.

2.4 Epidemics

The population evolves based on the individual evolution of its members and the occurrence of **epidemics**. An epidemic occurs whenever the number of individuals surpasses a maximum threshold, ν_{max}. Following an epidemic, only the five individuals with the highest comfort levels will survive. For each of the others, the probability of survival is given by $\frac{2}{3}\varphi(z)$.

2.5 Evolution

The evolution of the population is governed by discrete stochastic simulation, which is based on a pending event container. The simulation concludes under the following conditions:

- The time for the next event exceeds the final time τ.
- The population becomes extinct with no further events to simulate.
- An individual with a comfort level of one is identified.

3 Parameters and results

The program should accept the following parameters:

- Number n of patrols;
- Number m of planetary systems;
- Matrix C containing the times required for policing the planetary systems among the patrols;
- Final instant of evolution $\tau(> 0)$;
- Initial population ν;
- Maximum population $\nu_{max}(> \nu)$;
- Parameters μ, ρ, δ related to the events of death, reproduction, and mutation, respectively.

3.1 Example

The program can be invoked from the command line in two different ways. The first way is:

$$\texttt{java -jar project.jar -r}\ n\ m\ \tau\ \nu\ \nu_{max}\ \mu\ \rho\ \delta$$

This command does not contain the matrix C. Therefore, a random matrix must be generated, with the specified number of patrols n and planetary systems m.

The second way to invoke the program looks like this:

```
java -jar project.jar -f <infile>
```

where $< infile >$ is a text file (`.txt`) with all parameters needed for simulation, including the matrix C. The format of $< infile >$ is as follows:

$$\begin{array}{lllll} n\ m\ \tau\ \nu\ \nu_{max}\ \mu\ \rho\ \delta & & & & \\ c_{11} & c_{12} & c_{13} & \dots & c_{1m} \\ c_{21} & c_{22} & c_{23} & \dots & c_{2m} \\ \dots & \dots & \dots & \dots & \dots \\ c_{n1} & c_{n2} & c_{n3} & \dots & c_{nm} \end{array}$$

Therefore, for the example in Figure 1, we have an `input.txt` file on disk representing the simulation with:

```
3  6  10  10  100  10  1  1
1  2   1   1    2   1
2  2   2   2    2   2
3  3   3   3    3   3
```

During the parsing of the `input.txt` file, spaces and tabs should be ignored. If any other errors occur, the program should be aborted, and an explanation should be given to the user. Moreover, the `input.txt` file can be invoked from any location on the disk, using either a fixed or relative path with respect to the executable. In the previous example, the `input.txt` file was located in the same directory as the executable. However, the invocation should use a relative path if a user wishes to have a `TESTS` folder containing multiple test scenarios alongside the executable. For example:

```
java -jar project.jar -f ./TESTS/input.txt
```

or just:

```
java -jar project.jar -f TESTS/input.txt
```

Avoid hardcoding the location of files/directories in your project to ensure it can run on any computer. Hardcoding the file/directory location will result in a penalty to your project grade.

3.2 Results

During the simulation, the program should output the results of population observations to the terminal at intervals of $\tau/20$ time units. Each observation will include the current instant (*instant*), the number of events that have occurred (*events*), the current population size (*size*), the number of epidemics that have occurred (*epidemics*), the distribution of the best-fitted individual up to the current instant (*distribution*), the other candidate distributions stored (up to five, from *otherdist1* to *otherdist5*), the time required for policing the empire by the best-fitted individual up to the current instant (*time*), and the respective comfort level at the current instant (*comfort*), following this format:

```
Observation number:
                                   Present instant:                    instant
                                   Number of realized events:          events
                                   Population size:                    size
                                   Number of epidemics:                epidemics
                                   Best distribution of the patrols:   distribution
                                   Empire policing time:               time
                                   Comfort:                            comfort
                                   Other candidate distributions:      otherdist1 : time1 : comfort1
                                                                       otherdist2 : time2 : comfort2
                                                                       otherdist3 : time3 : comfort3
                                                                       otherdist4 : time4 : comfort4
                                                                       otherdist5 : time5 : comfort5
```

The observation *number* is determined by $\tau/20$, resulting in a total of 20 observations. The first observation corresponds to the instant $\tau/20$, and the last observation aligns with the time τ, indicating the end of the simulation. If the simulation concludes before reaching the final time (refer to Section 2.5), the program must still output a final observation. In such a scenario, either the *size* will be 0, or the *comfort* will be at 1.

The required *distribution* along with *otherdist1* to *otherdist5* should be formatted as follows, taking the example from Section 1:

```
{{1,3,4,6},{2,5},{}}
```

Any additional outputs to the terminal or deviation from this specified format will result in a penalty to the project grade.

4 Simulation

The simulator should perform the following steps:

1. Read the input parameters for the simulation and initialize or create the necessary values and objects.
2. Execute the simulation loop until one of the following conditions is met: (i) the final instant of evolution is reached; (ii) there are no more events to simulate; or (iii) an individual with a comfort level of one is identified. The population observations described in Section 3.2 should be output to the terminal during the simulation.

5 Examination and grading

5.1 Deadlines

The deadline for submitting the project on Fenix is (before) **June 1, 16:30**. The project accounts for 10 points of the final grade, which are distributed as follows:

1. **(2.5 points) UML**
   - The UML specification, including classes and packages (as detailed as possible), in `.pdf` format. Only `.pdf` format is accepted, with the file named `UML.pdf`.

2. **(7.5 points) Java**

   - The executable `project.jar` (with the respective source files `.java`, compiled classes `.class`). Source files are mandatory.
   - Javadoc documentation (generated by the Javadoc tool) of the application, placed inside a folder named `JDOC`.
   - At least five compelling examples with the corresponding input files and respective simulations. These examples should include something other than scenarios offered on the course webpage regarding this project and trivial variations thereof. The scenarios and their simulations should be placed in a folder named `SIM`, i.e., the `SIM` folder should contain the input files with the scenarios (for instance, `input1.txt`, `input2.txt`, etc) and the respective text files (`.txt`) with the results from the simulation (for instance, `simscenario1.txt`, `simscenario2.txt`, etc).

   The `UML.pdf`, the executable `project.jar`, the Javadoc documentation, and at least five input/simulation examples should be submitted via Fenix in a single file before **June 1, 16:30**. (The `.pdf` and `.jar` files, along with the folders `JDOC` and `SIM`, should be included in a single `.zip` file). Only files with a `.zip` extension will be accepted. Additionally, a **self-assessment form** via Forms and **registration for the project oral discussion** via an Excel file are required on Teams (**before June 3, 14:00**).

3. **Final discussion: 4-14 June 2024**
   The distribution of groups for the final discussion will be available **June 3, 16:00**. All group members must be present during the discussion. The final grade of the project will depend on this discussion, and it may not be the same for all group members. Regardless of the IDE used during project development, all students should be proficient in using Java on the command line.

5.2 Assessment

The assessment will be based on the following scale of cumulative functionality, where each level corresponds to a maximum grade. On a 10-point scale:

1. **(2.5 points)**: UML solution (0 points – non-existing, 0.5 points – bad, 1 point – sufficient, 1.5 points – good, 2 points – very good, and 2.5 points – excelllent).
2. **(7.5 points)**: Java implementation.
   (a) input, randomly generated matrix (0.5 point)
   (b) simulation (5.5 points)
   (c) output, finding the optimal patrol allocation (0.5 points)
   (d) Javadoc documentation (0.5 point)
   (e) examples of input/simulation (0.5 points)

The implementation of the requested features in Java, specific project requirements, and the quality of the oral discussion, are also important evaluation criteria, and the following discounts (on a 10-point scale) are pre-established:

1. **(-2 points):** OOP ingredients are not used or they are used incorrectly; this includes polymorphism, open-close principle, etc.

2. **(-1.5 points):** Java features are handled incorrectly; this includes incorrect manipulation of methods from `Object`, `Collection`, etc.
3. **(-1 points):** A non-executable JAR file or a JAR file without sources.
4. **(-0.5 points):** A submitted .zip outside of the requested format.
5. **(-0.5 points):** Hardcoded input file (as explained in Section 3.1).
6. **(-0.5 points):** Prints outside the format requested in Section 3.2.
7. **(-2 points):** Individual assessment of the student participation in the oral project discussion (on a per-student basis, rather than as a group).
8. **(-2 points):** Individual assessment of student ability to extract/build a JAR file, as well as compile/run the executable in Java from the command line (on a per-student basis, rather than as a group).

Projects submitted after the established deadline will incur a penalty. For each day of delay, a penalty of 2^{n-1} points will be deducted from the grade (on a 10-point scale), where n is the number of days delayed. For instance, projects submitted one day late will be penalized by $2^0 = 1$ point. Projects submitted two days late will be penalized by $2^1 = 2$ points, and so on. A day of delay is defined as a cycle of 24 hours from the specified submission date.