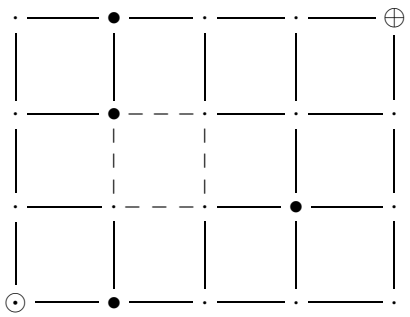
Object Oriented Programming Project 2024/25
The pathfinder problem

LEEC/MEEC – IST

1 Problem

Consider a grid $n \times m$. We want to find the best path, that is, the path with the lowest cost between an initial point, with coordinates (x_i, y_i), and a final point, with coordinates (x_f, y_f). Each path has an associated cost that is determined by the number of edges traversed. In general, the cost of each edge is 1, but there may be special areas where the cost is higher. There are n_{obst} obstacles at some points on the grid through which one can not proceed.

For instance, consider the following figure which represents a grid 5×4:



In this figure, the initial point, marked with $\odot$, has coordinates $(1, 1)$, the end point, marked with $\oplus$, has coordinates $(5, 4)$, the obstacles are marked with $\bullet$, and the edges of the special cost areas, indicated by dashed edges, have cost 4. In this case, the best path is $(1, 1), (1, 2), (2, 2), (3, 2), (3, 1), (4, 1), (5, 1), (5, 2), (5, 3), (5, 4)$ with cost 12. There are shorter paths, e.g.: $(1, 1), (1, 2), (2, 2), (3, 2), (3, 3), (4, 3), (5, 3), (5, 4)$, but its cost is higher (13).

1.1 The pathfinder problem

Path finding with obstacles is a fundamental problem in computer science and robotics, where the goal is to find a route from a starting point to a destination, avoiding any blocked or forbidden areas. One classic algorithm for this task is Dijkstra's algorithm, which efficiently finds the shortest path in a graph with non-negative edge weights by systematically expanding the lowest-cost paths. Variants of Dijkstra's algorithm include A* and algorithms adapted for dynamic or continuous environments.

However, adapting these algorithms to tackle multi-objective pathfinding – where one seeks to optimize several criteria simultaneously, such as minimizing distance and risk – is much more challenging. For example, finding all paths that are both as short as possible and as safe as possible between two points, or routing a robot through checkpoints most efficiently, often makes the problem NP-hard. This means there is no known algorithm that can solve all instances of the problem efficiently (in polynomial time).

Because of this computational complexity, approximate methods are often used. Some popular approaches include evolutionary programming (which mimics natural selection to evolve better solutions over time) and swarm programming (where simple agents collaborate to explore the space of possible paths). These methods can provide good-enough solutions in a reasonable time, even when exact algorithms become impractical.

1.2 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 multi-objective path finder, evolutionary programming is adept at generating various solutions across successive generations. This approach methodically advances towards optimal paths by effectively navigating through an extensive search space, a task that is considerably more efficient than methods based on exhaustive search.

The popularization of evolutionary programming was largely due to the famous case of the NASA antenna. In the mid-2000s, NASA engineers used evolutionary algorithms to design a communication antenna for the ST5 mission. One of the main challenges was that the antenna had to fit within the size constraints of a coin, which made it difficult for the engineers to design a standard antenna. The algorithm, inspired by natural evolution, produced a highly unconventional solution that was truly outside the box for the engineers involved (see https://www.jpl.nasa.gov/nmp/st5/TECHNOLOGY/antenna.html). This surprising result demonstrated how evolutionary methods can lead to innovative designs that might not be discovered through traditional engineering approaches.

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 multi-objective pathfinder, 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 to model systems with some 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

The goal of this project is to program a solution to the problem presented above in Java using evolutionary programming, modeled and implemented with objects.

The idea is to generate, at time zero, a population of ν individuals, all placed at the initial point, and allow them to evolve until the final instant τ. Each individual z contains a path – which, when completed, represents a potential solution to the pathfinder problem – and an associated comfort value computed as:

$$\varphi(z) = \left(1 - \frac{cost(z) - length(z) + 2}{(cmax - 1) \times length(z) + 3}\right)^k \left(1 - \frac{dist(z, (x_f, y_f))}{n + m + 1}\right)^k$$

where:

- $cmax$ is the maximum cost of an edge in the grid;
- $cost(z)$ is the cost of the path of the individual z;
- $length(z)$ is the length of that path – the number of edges traversed;
- $dist(z, (x_f, y_f))$ is the distance between the last point of the path and the final point – the fewer number of edges needed to be traversed until the final point is reached disregarding obstacles and special cost zones; each missing edge, from the current to the final point, values one to the overall sum.
- k is an input parameter of comfort sensitivity to small variations.

The comfort indicates how well the individual fits the problem. Each individual z evolves according to its comfort, by the following random mechanisms:

- **Death**: exponential variable with mean value $(1 - \log(1 - \varphi(z)))\mu$.

- **Reproduction**: exponential variable with mean value $(1 - \log(\varphi(z)))\rho$ between events. From reproduction, a new individual is born with. The path of the new individual is a prefix of the parent path. The length of that prefix is determined by the parent's comfort, always having 90% of the parent's path and a fraction $\varphi(z)$ of the remaining 10%.
- **Move** (usually called mutation in evolutive programming): exponential variable with mean value $(1 - \log(\varphi(z)))\delta$ between events. The move can occur equiprobably to an adjacent position on the grid not occupied by an obstacle. If the individual moves to a position that is already on its path the corresponding cycle should be eliminated.

The population evolves in function of the individual evolution of its elements and also by the occurrence of **epidemics**. Whenever the number of individuals exceed a maximum ν_{max}, an epidemic occurs. To the epidemic will always survive the five individuals with greater comfort. For each of the remaining, the survival probability is $\varphi(z)$.

The evolution of the population is ruled by discrete stochastic simulation, that is, based on a pending event container (PEC).

2.1 Scheduling events in the PEC

The progression of the simulation does not depend on the computer clock. Instead, it manages time using a PEC, where events are inserted and processed sequentially in order of their scheduled times. This event-driven approach ensures the simulation advances smoothly from one event to the next, continuing until the simulation time is reached or there are no more events left in the PEC.

Events are added to the PEC as follows. For each individual of the initial population, three new events must be added to the PEC:

- death, 1st move, and 1st reproduction.

In the simulation of the n-th move, one new event must be added to the PEC:

- $(n + 1)$-th move of the respective individual.

In the simulation of the n-th reproduction, four new events must be added to the PEC:

- $(n + 1)$-th reproduction of the respective individual.
- death, 1st move, and 1st reproduction of the child resulting from the reproduction being simulated.

Use the natural logarithm when a log function is required.

The time at which each individual will die is fixed upon their creation and remains constant throughout the simulation, irrespective of changes in their comfort.

When computing the move of an individual, it should occur equiprobably to any adjacent position on the grid that is not occupied by an obstacle. Therefore, you first need to identify the number of possible directions, say n, in which the individual can move. Next, draw a value from a uniform random variable between 0 and 1. If the value is less than or equal to $1/n$, the individual moves in the first direction; if it is greater than $1/n$ but less than or equal to $2/n$, the second direction; and so on. You should establish a fixed order for the directions: north

first, then proceeding clockwise. Note, however, that if the individual moves to a position that is already included in its current path, the resulting cycle should be eliminated.

Moreover, if an individual reaches the final point, it continues to move and does not stop until it dies – either from a death event or as a result of an epidemic. This is because, throughout its lifetime, following different paths may lead to improved comfort.

When computing the child's path in the reproduction, round up its size using `Math.ceil`.

Be careful not to add events that occur after an individual's death to the PEC. Additionally, ensure that no events occurring after the end of the simulation time are included.

2.2 Considerations regarding epidemics

The epidemic may be computationally demanding, forcing the garbage collector to clean up memory from the heap. During the epidemic, the five individuals with the highest comfort will always survive. For each of the remaining individuals, the probability of survival is given by $\varphi(z)$. Thus, for each individual, a value is drawn from a uniform random variable between 0 and 1. If the observed value is less than or equal to $\varphi(z)$, the individual survives the epidemic; if the value is greater than $\varphi(z)$, the individual dies.

3 Parameters and results

The program should receive the following data:

- the dimension n and m of the grid;
- the initial point (x_i, y_i) coordinates and final point (x_f, y_f) coordinates;
- a list of tuples (coordinate, coordinate, cost)

  $$\{((x_1, y_1), (x'_1, y'_1), c_1), \ldots, ((x_k, y_k), (x'_k, y'_k), c_k)\}$$

  where each tuple consists of two coordinates and a cost value. The first coordinate (x_i, y_i) represents the bottom-left corner of the rectangle, and the second coordinate (x'_i, y'_i) represents the top-right corner. The value c_i specifies the cost associated with traversing any edge in that area;
- the number of obstabcles n_{obst} and their coordinates (x_i, y_i), for $i = 1, \ldots, n_{obst}$;
- the final instant $\tau(> 0)$ of the evolution;
- the parameters $k, \nu, \nu_{max}, \mu, \delta, \rho$.

For simplicity, all program parameters may be assumed to be integers (greater than or equal to zero). Moreover, a special cost zone includes only the outline of the rectangle, not the entire area within it. If special cost zones overlap on the grid, edge by edge, consider only the maximum edge cost among the overlapping zones.

3.1 Example

Consider the example described in Section 1:

- grid dimension: $n = 5$ and $m = 4$;

- initial and final points: $(x_i, y_i) = (1, 1)$ and $(x_f, y_f) = (5, 4)$;
- special cost zone: $(x_1, y_1) = (2, 2)$, $(x'_1, y'_1) = (3, 3)$ and $c_1 = 4$;
- obstacles: $n_{obst} = 4$ with $(x_1, y_1) = (2, 1), (x_2, y_2) = (2, 3), (x_3, y_3) = (2, 4), (x_4, y_4) = (4, 2)$;
- evolution final instant: $\tau = 100$;
- initial population: $\nu = 10$;
- maximum population: $\nu_{max} = 100$;
- comfort sensitivity: $k = 3$;
- parameters related to the events: $\mu = 10, \delta = 1, \rho = 1$.

The initial point does not always need to be $(1, 1)$; however, the bottom-left point of the grid is always $(1, 1)$. Similarly, the final point does not always need to be at (n, m); however, the top-right point of the grid is always at (n, m).

3.2 Running your project in the command line and input format

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

`java -jar project.jar -r` $n\ m\ x_i\ y_i\ x_f\ y_f\ n_{scz}\ n_{obs}\ \tau\ \nu\ \nu_{max}\ k\ \mu\ \delta\ \rho$

This command does not include special cost zones or obstacles. Therefore, both random cost zones and obstacles need to be generated, given the specified values of n_{scz} and n_{obs} for each.

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 cost zones and obstacles. The format of $< infile >$ is as follows:

$n\ m\ x_i\ y_i\ x_f\ y_f\ n_{scz}\ n_{obs}\ \tau\ \nu\ \nu_{max}\ k\ \mu\ \delta\ \rho$
`special cost zones:`

x_1	y_1	x'_1	y'_1	c_1
x_2	y_2	x'_2	y'_2	c_2
...	...	...	...	...
$x_{n_{scz}}$	$y_{n_{scz}}$	$x'_{n_{scz}}$	$y'_{n_{scz}}$	$c_{n_{scz}}$

`obstacles:`

x_1	y_1
x_2	y_2
...	...
$x_{n_{obs}}$	$x_{n_{obs}}$

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

```
5  4  1  1  5  4  1  4  100  10  100  3  10  1  1
special cost zones:
2  2  3  3  4
obstacles:
2  1
2  3
2  4
4  2
```

In the `input.txt` file, the sections for special cost zones and obstacles (including their headers and corresponding values) are omitted if their numbers, as indicated in the first line of the file by n_{scz} and n_{obs}, are zero. 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.

3.3 Output

Recall that your program can be run with two options: `-r` and `-f`. At the start of each simulation, print all input parameters to the terminal in the same format as when the program is run with the `-f` option. This should include all parameters provided via the command line, as well as any parameters generated when using the `-r` option (formatted as if they were specified with the `-f`, including headers and corresponding values for special cost zones and obstacles). If you are running with the `-f` option, print the contents of `input.txt`. Next, add two line breaks.

During the simulation, the program should also print to the terminal the result of 21 observations of the population, realized from $\tau/20$ by $\tau/20$ time units (starting at time 0 with observation 0, and ending at time τ with observation 20). Each observation should include the present instant/time (*time*), the number of events already realized (*events*), the population size (*size*), the path of the best individual and its cost/comfort until the actual instant (*cost* if the final point has been hit and *comfort* otherwise), according to the following format:

Observation *number*:

Present time:	*time*
Number of realized events:	*events*
Population size:	*size*
Final point has been hit:	yes/no
Path of the best fit individual:	$[(x_1, y_1), \ldots, (x_j, y_j)]$
Cost/Comfort:	*cost/comfort*

The program should print to the terminal at the end of the simulation the path of the *best fit individual* over all the simulation. By the best fit individual, we mean:

- if there is an individual that reached the final point, the individual z with the lowest cost, independently of whether the individual z is alive (or not) at the end of the simulation;
- if none of the individuals reached the final point, the path of the individual z with the greatest comfort, independently of whether the individual z is alive (or not) at the end of the simulation.

The printed output should be in the following format:

`Best fit individual:` $[(x_1, y_1), \ldots, (x_j, y_j)]$ `with cost` *cost*

Any other printing to the terminal, or a print of this content out of this format, incurs a penalty in the project grade.

4 Simulation

The simulator should execute the following steps:

1. Read the input parameters for the simulation, print them to the terminal as requested in Section 3.3, and initialize or create the necessary values and objects in your project.
2. Run the simulation cycle until: (i) the evolution final instant is reached, or (ii) there are no more events to simulate. During the simulation, the population observations described in Section 3.3 must be printed to the terminal.
3. At the end of the simulation, the information requested in Section 3.3 must be printed to the terminal.

5 Examination and grading

5.1 Deadlines

The deadline for submitting the project on Fenix is (before) **June 12, 17:00**. 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 – 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** and **registration for the project oral discussion** are required on Teams (**before Monday, June 16, 14:00**).

3. **Final discussion: 16-20 June 2025 and 30-4 July 2025**
The distribution of groups for the final discussion will be available **June 16, 14:30**. 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 info (0.25 point)
   (b) simulation (6 points)
   (c) output, ~~finding the optimal patrol allocation~~ the optimal path (0.5 points)
   (d) Javadoc documentation (0.5 point)
   (e) examples of input/simulation (0.25 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. **(-3 points):** OOP ingredients and principles are not used or misused; this includes polymorphism, open-closed principle, loose coupling, design patterns, 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 3.2).

6. **(-0.5 points):** Prints outside the format requested in Section 3.3.

7. **(-3 points):** Individual assessment of the student participation in the oral project discussion (on a per-student basis, rather than as a group); including, 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.