Computational Intelligence WS24/25

Exercise Sheet 2 (Solution) — November 14th, 2024

Thomas Gabor, Maximilian Zorn, Claudia Linnhoff-Popien

1 Beaver goals and dreams

Consider a beaver in the final steps of building a dam. It just needs to add one more log at precisely the right angle. If it hits the right angle, the dam is now leak-proof and the water level (of the river flowing from behind the dam) rises. The beaver's behavior is thus given by a policy $\pi : \mathit{Level} \to \mathit{Angles}$ where $\mathit{Level} = [0; \infty) \subset \mathbb{R}$ is the water level in cm and $\mathit{Angles} = [0; 180] \times [0; 180] \subset \mathbb{R}^2$ are the horizontal and vertical angles in degrees at which the beaver tries to insert the final log into the dam.

(i) Assume that the environment is updated in discrete time steps and thus produces a sequence of states $\langle s_t \rangle_{1 \leq t \leq T}$ where $T \in \mathbb{N}$ is the episode length of the environment and $s_t \in \mathit{Level} \times \mathit{Angles}$ for all t. The beaver deems its work successful iff the water level keeps rising from some point in time on for at least three consecutive time steps and all future time steps afterwards. Give a goal predicate $\gamma : \langle \mathit{Level} \times \mathit{Angles} \rangle \to \mathbb{B}$ so that $\gamma(\langle s_t \rangle_{1 \leq t \leq T})$ holds iff the beaver deems its work successful.

$$\gamma\big(\langle (l_t, \_) \rangle_{1 \leq t \leq T}\big) \iff \exists t : \ \big(t \leq T - 2 \ \wedge \ \forall t' > t : l_{t'} > l_{t'-1} \big)$$

Now, further assume that a beaver's policy for putting logs into dams is encoded by a parameter vector $\theta \in \Theta = \{A, C, G, T\}^5$ where A, C, G, T are arbitrary fixed symbols. We are given a fitness function $\phi : \Theta \to \mathbb{R}$ so that $\phi(\theta)$ encodes the time it takes a beaver with policy π_θ to put a log into a dam successfully, i.e., lower values of $\phi(\theta)$ are better.

(ii) We initialize a population $X \in \Theta$ with population size $|X| = 10$. A nice *variation* function should

- construct 5 new individuals based on random candidates from the original population X,
- not always generate completely new individuals, but have them based on the given population X, and
- be able to reach every point in the search space through iterated application.

Give a complete definition for a nice *variation* function.

$$
\begin{aligned}
\mathit{variation}(\emptyset) &= \emptyset \\
\mathit{variation}(X) &= \{\mathit{mutate}(x)\} \uplus \mathit{variation}(X \setminus \{x, x'\}) \\
&\quad \text{where } X \neq \emptyset \\
&\quad \text{and } x, x' \sim X \\
\text{with } \mathit{mutate}(x) &= (x'_0, x'_1, x'_2, x'_3, x'_4) \\
&\quad \text{where } x'_i = \begin{cases} g \sim \{A, C, G, T\} & \text{if } i = j \\ x_i & \text{otherwise} \end{cases} \\
&\quad \text{and } j \sim \{0, ..., 4\}
\end{aligned}
$$

(iii) Consider the following two definitions of a *selection* function:

$$
\mathit{selection}_1(X) = \begin{cases} \mathit{selection}_1\big(X \setminus \{\arg\max_{x \in X} \phi(x)\}\big) & \text{if } |X| > 10, \\ X & \text{otherwise,} \end{cases}
$$

$$
\mathit{selection}_2(X) = \begin{cases} \mathit{selection}_2\big(X \setminus \{x\}\big) & \text{for } x \sim X \text{ if } |X| > 10, \\ X & \text{otherwise.} \end{cases}
$$

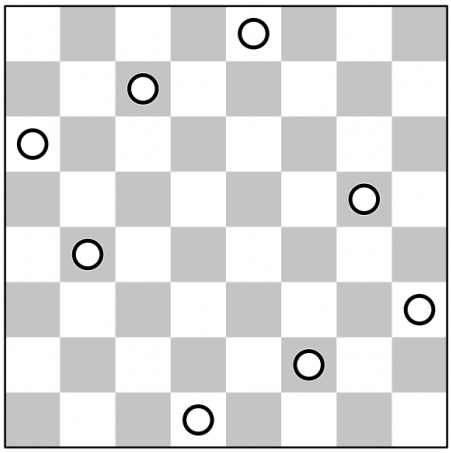
Briefly explain why both functions fulfill the definition for a *selection* function in the Evolutionary Algorithm as defined in the lecture? Which of these two functions is more directly useful in the context of optimization? State your argument.

Both functions reduce the given population (recursively) until $|X| = 10$ is reached. $\mathit{selection}_1$ exerts the stronger selection pressure as $\mathit{selection}_2$ exerts no directed pressure at all. Thus, only $\mathit{selection}_1$ is usable (thus useful) for optimization.

2 N-Queens Optimization

In this exercise we will consider the **N-Queens**[1] problem as one prominent example domain for the optimization methods we have covered in the lecture.

[1] `https://en.wikipedia.org/wiki/Eight_queens_puzzle`



(a) 8-Queens solution

```
function queens
Input  n: int, k: int, board: array of array of boolean);
Output boolean
begin
    if k ≥ n then return true;
    for i = 0 up to n–1 do begin
        board[i][k] ← true;
        if not board[i][j] ∀j : 0 ≤ j < k
        and not board[i – j][k – j] ∀j : 0 < j ≤ min(k, i)
        and not board[i + j][k – j] ∀ j : 0 < j ≤ min(k, n – i – 1)
        and queens (n, k + 1, board)
        then return true;
        board[i][k] ← false;
    end
    return false;
end
```

(b) N-Queens algorithm pseudocode (backtracking approach)

(i) Define the N-Queens problem as a single-sample optimization process and give one example for each of the elements in the tuple $D = (\mathcal{X}, \mathcal{T}, \tau, e, \langle x_u \rangle_{0 \leq u \leq t})$.

For example: For n queens we can let $\mathcal{X} = C^n$ where $C = [0; n) \subset \mathbb{N}$ (which might be interpreted as the queen's row position for each column). (Note that there needs to be exactly one queen in each column to be able to solve the problem.)

Let $\mathcal{T} \in \mathbb{N}$.

Let target function $\tau : \mathcal{X} \to \mathbb{N}$ be

$$\tau(x) = n - \sum_{i=0}^{n-1} \mathbb{1}[\texttt{check}(i, x)],$$

i.e., the sum of remaining, incorrectly positioned queens, where $\texttt{check}(i, x) \in \mathbb{B}$ could be a helper function to check if queen i is positioned correctly within solution candidate x. (Minimization of incorrect positions $\implies \tau(\_) \to 0$.)

One simple example for a step function e could vary the (row-)position of a random queen by ± 1 depending on a random value, e.g., the optimization e of a sample x_t at timestep t continues by

$$x_{t+1} = \begin{cases} x_t[r_1] + 1 \mod n, & \text{if } r_2 \leq 0.5 \\ x_t[r_1] - 1 \mod n, & \text{otherwise} \end{cases},$$

where $r_1 \sim [0; n) \subset \mathbb{N}$ and $r_2 \sim [0; 1] \subset \mathbb{R}$.

We can start with $t = 0$ and thus history $\langle x_0 \rangle$ with $x_0 \sim \mathcal{X}$.

Note: A smarter search space might be $\mathcal{X} = \mathfrak{S}(C)$ where $C = [0; n) \subset \mathbb{N}$ and $\mathfrak{S}(C)$ is the space of all permutations of C. However, then we also need a smarter step function e, e.g.,

$$x_{t+1} = \langle x_{t+1}[0], ..., x_{t+1}[n-1] \rangle$$
$$\text{where } x_{t+1}[i] = \begin{cases} x_t[r_2] & \text{if } i = r_1, \\ x_t[r_1] & \text{if } i = r_2, \\ x_t[i] & \text{otherwise,} \end{cases}$$

where $r_1 \sim C, r_2 \sim C \setminus \{r_1\}$ for each call of e.

(ii) Adjust your solution of the previous task and specify necessary parameters such that the *Simulated Annealing* (SA) Algorithm can be applied to the N-Queens Problem. Then, give a brief estimate on how SA would perform in comparison to other simple search algorithms like *Brute Force* and *Random Sampling* as the board size increases. How would the SA temperature parameter need to behave for increasingly larger boards?

For Simulated Annealing we would need:

- A valid initial position x_0, e.g. $[0, 1, 2, 3, 4, 5, 6, 7]$
- A decreasing, valid temperature function, e.g., $\kappa(t) = \max\{100 - t, 1\}$
- An optimization function utilizing the SA acceptance function A, e.g., the optimization process e_{SA} could continue via

$$x_{t+1} = \begin{cases} x'_t & \text{if } \tau(x'_t) \leq \tau(x_t) \quad \text{or} \quad r \leq A(\tau(x_t), \tau(x'_t), \kappa(t)), \\ x_t & \text{otherwise,} \end{cases}$$

  with $r \sim [0; 1] \in \mathbb{R}$, where we obtain x'_t via $e(x_t)$ above .

Evaluation: Better than Brute Force and Random Sampling for small n, better than Brute Force but only better than Random Sampling for follow up steps for large n (since the random starting position sample are equally likely to randomly hit valid positions).

Unintuitively, for larger n we can choose faster cooling κ temperature values, since more solutions allow us to converge (exploit) faster.

(iii) How would *Simulated Annealing* compare to *Weighted Random Sampling* for the N-Queens Problem? Is the *locality* property given here?

The question of *locality* can be argued both ways here: While *locality* of neighboring solutions gets *more likely* with the growing number of solutions (due to larger and larger spaces between valid solutions enabling smaller steps towards still – or now (more) – valid solutions), the other problem constraints still hold (one queen per row and column). Simple *vary row by ± 1 position* single-sample search will not get better with larger n, except (and only) if more than one queen is moved at once, since the probability of valid permutations also increases with larger n.

For small n, Simulated Annealing will therefore work better than Weighted Random Sampling, at least from every second step onwards, since – for large n – Weighted Random Sampling has no way of escaping the (many) local optima and is therefore ensured to converge in them (this can also be shown with other greedy algorithms e.g., Hill Climbing).

Let's now move on from single-sample search to population-based search – in our case the *Evolutionary Algorithm* – and explore the N-Queens problem further.

(iv) Research and assign the terms **phenotype, genotype**, and **fitness** to this problem and give an example of each. How do these terms map to the definition of the optimization process D from Task *(i)*?

*In genetics, a distinction is drawn between the **genotype** and the **phenotype**. The genotype refers to the genetic information hat is stored within the cells and copied during reproduction, e.g., while the phenotype refers to how that information is expressed (in an organism's body oder behavior, e.g.). The evaluation criteria according to which some genes propagate while others do not are called **fitness**. For biological systems a concise 'fitness value' is usually impossible to compute, but for artificial evolution fitness usually refers to a given target function (if available).*

An example specific to the N-Queens problem (with $n = 8$) could be :
Genotype: The 'encoding' of the problem as, e.g., array/genome of type C^8 or $\mathbb{N}^8$. For Example [4, 2, 0, 6, 1, 7, 5, 3] from Fig. 1a as C^n (see above, each row from top to bottom, offset to the right). Thereby genome $= x \in \mathcal{X}$.
Phenotype: The 'realization' of the genotype, in this case the genome arranged on the $n \times n$ chess field (as shown in Example 1a). Apart from the visual of the chess board, this has no direct connection to the process D.
Fitness: The 'goodness' of the solution/ evaluated genome. The evaluation function ϕ that determines this fitness is of course highly problem specific and can be as complex as desirable. Here $\tau(\cdot)$ could double as fitness funciton, i.e., $\tau([4, 2, 0, 6, 1, 7, 5, 3]) = 0$, as a valid solution would minimize the target (and fitness) function. Thereby (in some cases) $\phi(\cdot) \approx \tau(\cdot)$.

(v) Research and give one example each for generic **selection, mutation**, and **recombination** functions to optimize our N-Queens problem. You are free to choose your notation (e.g., code, pseudocode, formal notation etc.). Briefly explain your choices.

Let $X_0 \in \wp^*(C^8)$ of size $|X_0|$ be a randomly sampled initial population of *genome* : C^8 individuals (as defined above), assuming the 8-Queens problem.
Let **cutoff-selection** be a selection method of the form `sorted(X)[:m]`, that sorts the population by fitness and only keeps the m best individuals (with $m = |X_0|$). Occurs every round to maintain population size.
Let **one-point-mutation** be a mutation method of the form `X[i][l] = (X[i][l] + 1) % 8` for individual i that slightly changes exactly one locus l of the genome. Could happen with e.g., probability $p_{mut} = 0.1$.
Let **one-point-crossover** be a recombination method of the form `c = X[a][:4] + X[b][4:]` for randomly sampled partners $a, b \in X$. Could happen with e.g., probability $p_{rec} = 0.5$.

(vi) (Bonus 1) With the pseudocode algorithm given in Figure 1b), implement the backtracking solution algorithm to the N-Queens problem. The `queens` function is called with the number n of queens that defines the problem size, k = 0 indicating that the board should be filled starting from rank 0, and board being an $n \times n$ Boolean matrix that is initialized to false in all elements. If the function returns true, the problem can be solved. If the function returns false, the problem cannot be solved (the 3-Queens problem, for example, has no solution). (Hint: If you want to check the solutions, for smaller n the possible solutions are well documented.[2])

(vii) (Bonus 2) Instead of using the backtracking solution, try to optimize the N-Queens problem with an evolutionary algorithm approach. You could – for example – implement the evolutionary operators that you have defined above. How does the EA perform in comparison to the backtracking algorithm?

For both tasks (vi) and (vii), see solution code in `exercise_02_solution_code.zip`.

→ EA performs worse than backtracking solution for small n (on average), better for large n.

3 Running Example: Vacuum World

With the implementation of the *Vacuum World* example from Exercise 1, extend the code to include the following elements:

(i) Introduce a target function τ of your choice, which is able to evaluate a given policy π with respect to a metric of your choosing (e.g., number of times the robot changed the room, number of times the robot vacuumed a dirty room successfully, etc.). You can

[2] `https://en.wikipedia.org/wiki/Eight_queens_puzzle`

freely decide whether your τ is to be minimized or maximized.

(ii) In addition to the random policy π_{rand} from Exercise 1, now also prepare a policy π_{target}, that performs better than the random baseline policy with respect to your previously implemented target function τ. Depending on your target function, you might need to adjust the observation space to include the necessary information.

(iii) Collect again a fixed number of actions and observations for each π_{rand} and π_{target}, then evaluate both with your τ function and compare.