Olympiads in Informatics, 2025, Vol. 19, 115–127

DOI: 10.15388/ioi.2025.08

Olympiad Tasks in Changing Environment

Pavel S. PANKOV[1], Elena S. BUROVA[2], Elzat J. BAYALIEVA[3]
[1]*Institute of Mathematics, Kyrgyzstan*
[2]*AUCA, Kyrgyzstan*
[3]*J.Balasagyn Kyrgyz National University, Kyrgyzstan*
e-mail: pps5050@mail.ru, burova_e@auca.kg, elzat.bayalieva@gmail.com

Abstract. Traditionally, tasks in informatics are formulated within fixed (discrete) environment such as segments (1..N), rectangles (1..N × 1..M), or graphs containing various objects, obstacles, connections, and goals. However, in practice environment often change dynamically, and the information available is incomplete, which makes standard algorithmic solutions inadequate. Likewise, a living agent perceives only nearby elements of its environment. This paper explores a class of tasks that address these limitations. Some of these tasks were generated with the assistance of AI. The concepts of timexels (introduced by the authors in their prior work) and program time are used as foundational tools to describe changing discrete environments.

Keywords: Olympiad, informatics, task design, dynamic environment, motion, timexel, program time.

1. Introduction

In real-world applications, actions often occur with incomplete knowledge of a changing environment. While many problems in informatics focus on optimization and planning paths or motions for one or more agents, few explicitly account for environmental uncertainty and change. The aim of this paper is to introduce theoretical constructs, propose definitions, develop methodology, and create and solve illustrative problems reflecting such conditions.

Remark. There exist classical tasks involving visibility, such as *Seeing the Boundary* (IOI'2003 (along polygons)) and *Hermes* (IOI'2004 (along streets)), which are based on visibility along straight lines. However, these problems do not take distances into account, nor do they use visibility data for active decision-making.

Additionally, several well-known mathematical problems can be reinterpreted under the lens of dynamic environments or limited visibility (discussed in Section 3).

To incorporate a changing environment formally, we have introduced the concept of timexels (Pankov *et al.*, 2021).

General Task 1 (Formal Setup):

Let USt be a set of states St, with initial state Pt0 and desired final subset Ft1 $\subset$ USt. At each step, the state transitions – either deterministically or non-deterministically – according to a computable function W: USt $\rightarrow$ subset of USt. You are given partial or full information I(St) about the current state St. Your allowed action is to transition the system via a computable function Y: U $\rightarrow$ subset of USt. Your objective is to bring Pt0 to some Ft1 in the minimum number of steps.

The program must either:

- reach Ft1,
- output the minimum number of steps to Ft1, or
- compete against a specified (or jury-defined) algorithm W to validate its optimality.

Attempts were made to use AI to generate Olympiad-style tasks involving "to write a program in changing environment" but initial queries were too general. A more focused prompt, "write a program to catch a moving goal," yielded satisfactory tasks (see Section 4).

Remark. Informatics often presents dualities (Pankov, 1990). In virtual motion, one may interpret motion in two ways: either the space moves toward the observer, or the observer moves through space. Similarly, tasks may be described using permissions or prohibitions – passes or obstacles. We use the more convenient formulation.

Section 2 introduces *program time*, applies *timexels*, and proposes a formal definition of what it means to find an optimal computer program under dynamic conditions.

Section 3 defines tasks based on well-known mathematical problems as well as a *limited visibility* domain.

Section 4 contains new tasks on catching moving objects, including under conditions of limited visibility, with some of them generated by AI systems. Section 5 contains dual tasks involving temporal objects: one requires jumping onto objects only, while the other requires avoiding contact with them.

Notation:

- Z_0 := {…, -2, -1, 0, 1, 2, …};
- N_0 := {0, 1, 2, …};
- N_1 := {1, 2, 3, …};
- Directions: E, NE, N, NW, W, SW, S, SE (counterclockwise).

2. Program Time and Timexels

To represent dynamic processes, we extend the Eulerian perspective from continuum mechanics to discrete settings. We define space primitives as follows:

- 1D: dots
- 2D: pixels
- 3D: voxels

- Any similar objects could be called as “spacexels”.

By extending space primitive with a time dimension, we introduced “timexels” (space elements existing during one time step). Each timexel is indexed by its spatial coordinates, temporal index (a natural number indicating a moment of time-step), and optional attributes.

The concept of timexcel offers a framework for the approximate description of various processes.

Example (models red and green points converging into a black one, followed by rapid disappearance):

Represented as a set of 1D-timexcels (unordered):

(2, 0, red), (3, 1, red), (4, 2, black), (6, 0, green), (5, 1, green), (6, 3, black), (8, 4, black)

Same elements presented as spacexels with optional color attribute, ordered in a time dimension:

- time=0: (2, red), (6, green)
- time=1: (3, red), (5, green)
- time=2: (4, black)
- time=3: (6, black)
- time=4: (8, black)

We can identify that such time-indexed representation is limited in flexibility. It cannot, for example, model the dynamic placement of obstacles by an external agent (e.g. jury’s program for evaluation).

Definition 1: Program Time (p_time)

To formalize timexel-based programs, we define discrete *program time* p_time as follows:

- At the beginning, p_time := 0
- The program must include a clock procedure (subprogram or function) named Clo (or multiple procedures, such as Clo1, Clo2, etc.). Each execution of this procedure performs the operation p_time := p_time + 1
- All other operations in a program are instantaneous.

Example Task 2: Sorting Three Numbers (elements)

Define a Boolean function Clo(X, Y) = true if X < Y, false otherwise.

Comparisons of the elements in the program are allowed only via function Clo. Sample program code:

Program 2-1:

```
p_time := 0;
Input(U, V, W);
if (not Clo(U, V)) then Swap(U, V);
if (not Clo(V, W)) then Swap(V, W);
if (not Clo(U, V)) then Swap(U, V);
```

```
Output(U, V, W);
Output(p_time);
end.
```

After executing this algorithm, we will get result with p_time = 3.

Task 3: Optimal sorting problem

Using timexel model, the classical sorting problem becomes formalized in a following way.
Let:

- PP: a permutation of M elements
- P[M]: set of all permutations of M elements
- QQ: a program permutating PP with Clo function
- Q[M]: the (infinite) set of programs sorting via Clo function

Defining a function FF: $Q[M] \times P[M] \to N_1$ with the following conditions:

- If the program QQ with the initial data PP is completed with p_time < M^3 steps, then FF(QQ, PP) := p_time
- otherwise FF(QQ, PP) := M^3.

The goal is:

$$F(M) \coloneqq \min_{QQ \in Q[M]} \left\{ \max_{PP \in P[M]} F\,(QQ, PP) \right\} \tag{1}$$

Remark. Some authors refer to Swap as Clo2 (in our terminology), but distinguishing latent swaps within program code is challenging – particularly when assignment operators are disallowed.

Method 1. For small values of M, such problems are addressed as follows (according to our terminology):

First, it is formally proven that the inequality $F(M) < F_1$ is not possible. Subsequently, a program that executes in exactly F_1 steps is constructed.

In the case of Problem (1), the inequality $2^{F(M)} < M!$ does not hold.

This notation and methodological framework can be consistently applied to a broad range of problem types.

General Task 4: Adaptive Coin Weighing Problem

The study of such task was initiated by H. Steinhaus (H.Steinhaus, 1989, 1999). The task is defined as follows:

Given M indistinguishable coins, either (A) exactly one or (B) at most one of them is counterfeit. Additionally, it is known that (C) the counterfeit coin is either heavier or lighter than the genuine ones, or (D) the counterfeit coin differs in weight, but it is not known in which direction.

The objective is to determine the minimum number of weighings required to: (G1) identify the counterfeit coin, or (G2) in case (D), determine both the identity of the coun-

terfeit and whether it is heavier or lighter. Additional genuine coins, whose authenticity is known in advance, may be used in the process.

To formalize this task, the *Clo-weight function* is employed. This function accepts an even number of arguments, $2k$, representing coins. It outputs one of three possible relations: the first k coins are lighter (L), equal in weight (E), or heavier (H) than the second k coins. The relation H is defined as the inverse of L, i.e., H = −L.

For case (C), it has been shown that the inequality $3^{F(M)} < M$ does not hold. Similarly, in case (D), the inequality $3^{F(M)} < 2M$ is invalid.

Significant contributions to this problem were made by M. Kołodziejczyk (n. d.), who obtained the following results: in case (A)(C), it was shown that F(19) = 3; and in case (B)(D), assuming an unlimited number of additional genuine coins, F(40) = 4.

Task 5: Coin Swapping Scenario

As a novel contribution, we introduce Task 5, which, to the best of our knowledge, has not been previously studied. The task involves three coins, denoted *U*, *V*, and *W*, with the condition (A)(D): at most one coin is counterfeit, and it is known only that it differs in weight (heavier or lighter) from the genuine coins. After the first weighing, two of the coins are swapped. The objective (G2) is to identify the original counterfeit coin.

The following decision procedure is proposed:

Algorithm 5-1.

1. Initialize p_time := 0.
2. Perform the first weighing: Z1 := Clo-weight(U, V).
3. If Z1 = ‘E’, then conclude that coin *W* is counterfeit. Terminate. Set p_time := 1.
4. Otherwise (note: *U*, *V*, *W* refer to positions, not fixed coins), proceed:
    - Swap two coins and perform a second weighing: Z2 := Clo-weight(U, V).
    - If Z2 = Z1 (indicating that the counterfeit coin remained in place, and a genuine coin was swapped, making the new *W* genuine), then:
        - Perform a third weighing: Z3 := Clo-weight(U, W).
        - Analyze the outcome and conclude. Terminate. Set p_time := 3.
    - If Z2 = 'E' (indicating that the counterfeit coin – originally *U* or *V* – has moved to position *W*), then:
        - Perform a third weighing: Z3 := Clo-weight(U, V).
        - Analyze the outcome and conclude. Terminate. Set p_time := 3.

This algorithm demonstrates an efficient strategy to detect the original counterfeit coin under dynamic conditions involving positional changes after the initial measurement.

3. Tasks with a Horizon

We consider a class of problems involving partial information and spatial exploration, where the environment is incrementally revealed to the agent. This includes classical mathematical and algorithmic formulations as well as generalizations suited for practical robotics or agent navigation.

Task 6: Cantor's Spiral and Obstacle Detection

This task is inspired by Cantor's diagonal argument demonstrating the countability of pairs of natural numbers. Let the environment be defined as $\mathrm{Z}_{\geq 0} \times \mathrm{Z}_{\geq 0}$. The agent (denoted as A) begins at the origin (0, 0) and may attempt to move in one of the four cardinal directions: East (E), North (N), West (W), or South (S). Movement is governed by a Boolean function Clo(Z), where $\mathrm{Z} \in \{\mathrm{E}, \mathrm{N}, \mathrm{W}, \mathrm{S}\}$. If the move in direction Z is possible, then the agent is shifted accordingly and Clo(Z) = true; otherwise, the agent remains in place and Clo(Z) = false. An obstacle is present in the environment.

Objective of the task is to design an algorithm to detect the location of the obstacle.

Algorithm 6-1 (Spiral Search).

The agent performs a spiral motion originating from the starting point, thereby systematically exploring the grid and detecting obstacles via failed moves.

Remark. This formulation also implies the countability of rational numbers since every rational number can be represented as a pair of integers.

Task 7: General Graph Search with Local Visibility

Traditional formulations of labyrinth or maze-solving problems assume full knowledge of the environment, including coordinates and obstacle locations. In contrast, we present the problem in a setting that better reflects real-world constraints, where only local information is accessible to the agent.

The environment is an unknown connected undirected graph. The agent starts at an arbitrary vertex and may place markers on visited vertices. The following operations are available:

- Clo1(): Returns the number of adjacent vertices and indicates whether the goal vertex is among them.
- Clo2: Indicates whether a marker is present on the current vertex.
- Go(k): Moves the agent to the kth adjacent vertex (in a list returned by Clo1()).
- Goback(): Returns the agent to the previous vertex.

Design an algorithm to find the goal vertex.

General Task 8: Goal Search with Obstacles and Partial Visibility

We generalize the previous tasks to a scenario involving spatial navigation with a distant goal and partially observable obstacles.

The agent starts at an unknown location. The goal is far away and may or may not be directly visible. Some obstacles are present but can only be detected when in close proximity. Devise a strategy to reach the goal. This task can be formalized as follows:

Task 9

Let the environment be a 2D grid defined over 1..M × 1..M, with the origin in the southwest corner. The agent starts at position A(x_0, y_0), and the goal is located at Ft1(M, M). There is a single obstacle B_1. It is guaranteed that A, $Ft1$, and B_1 are distinct. The agent

may use the function Clo(Z) to attempt a move in direction $Z \in \{E, N, W, S\}$. The function behaves as in Task 6.

Objective: Determine the minimal program time p_time necessary to reach $Ft1$. Let the function F(M, x_0, y_0) denote the worst-case minimal steps required to reach the goal. Then:

$$F(M, x_0, y_0) = \min_{QQ \in Q[M]} \left\{ \max_{B_1} F\,(QQ, x_0, y_0, B_1) \right\} \tag{2}$$

Example. For M = 100, with initial position A(98, 100) and unknown obstacle at (99, 100):

- Clo(E) = false → agent remains at (98, 100)
- Clo(S) = true → (98, 99)
- Clo(E) = true → (99, 99)
- Clo(N) = true → (99, 100)
- Clo(E) = true → (100, 100)

Result: p_time = 5

Remark. In general, for most points (except those adjacent to the goal), the expression $F(M, x_0, y_0) = (M - x_0) + (M - y_0) + 3$ holds. However, calculating the exact value of $F(M)$ is difficult even for small values (e.g., $M = 3$) due to the infeasibility of brute-force methods over the infinite space of potential programs.

Algorithm 9-1: **Dynamic Programming Approach**

We present a backward dynamic programming approach based on the Manhattan distance.

Let:

Dist(x, y) = ($M - x$) + ($M - y$)

Initialize:

- F(99, 100) = 1
- F(100, 99) = 1
- F(98, 100) = 5
- F(99, 99) = F(100, 98) = 5

Then for increasing values of Dist(x, y), update:

$F(x, y)$ = max($F(x + 1, y) + 1$, Dist(x, y) + 3)

(e.g., for x = 97, y = 100)

Task 10: Generalization to Multiple Obstacles

Introduce B_1, B_2, …, B_K as obstacles. The prior guarantee that A, $Ft1$, and obstacles are distinct is no longer sufficient. It must additionally be guaranteed that a valid path from A to $Ft1$ exists.

Remark. Even for $K = 2$ obstacles, the task becomes computationally difficult. A direct extension of Algorithm 9-1 is not possible since multiple blocked directions may occur simultaneously (e.g., Clo(E) = false and Clo(S) = false).

Task 11: Search with Limited Field of View

Let the environment be $\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}$, and the agent starts at (0, 0). The agent can step in cardinal directions and sees only a 2 × 2 square centered at its current position (simulating a weak flashlight in a dark environment). A rectangle with even-length sides is placed somewhere in the environment. The task is to reach the center of the rectangle.

Algorithm 11-1.

1. **Stage I**: Perform spiral motion until the rectangle is detected.
2. **Stage II**: Traverse two adjacent sides of the rectangle while counting steps.
3. **Stage III**: Compute half-lengths of the sides.
4. **Stage IV**: Move to the center of one side, then to the center of the rectangle.

4. Tasks Involving Pursuit of a Moving Adversary

In this section, we present a series of algorithmic problems focused on capturing or immobilizing a moving target (referred to as a "virus" or "target") within a constrained environment. The initial formulations were proposed to AI systems (Gemini and *chat.deepseek.com*), and their responses have been unified and adapted into rigorous task definitions.

Task 12: The Elusive Target (Inspired by Gemini)

You are tasked with writing a program to control a virtual "catcher" that must intercept a moving "target" within a specified arena.

Environment: A toroidal grid shaped arena with dimensions $K \times L$, where coordinates wrap around at the boundaries (i.e., movement beyond one edge reappears at the opposite edge).

Initial Conditions:

- The **target** starts at position (x_t, y_t) and moves at a constant velocity vector (dx_t, dy_t).
- The **catcher** starts at position (x_c, y_c) and may move one step per time unit in any of the eight cardinal or diagonal directions.

Input (per time step):

- Grid dimensions: K, L
- Target's position: (x_t, y_t)
- Target's velocity: (dx_t, dy_t)
- Catcher's position: (x_c, y_c)

Output:

- One of the nine possible catcher moves: {N, NE, E, SE, S, SW, W, NW, STAY}

Goal: The catcher's goal is to reach the same grid cell as the target.

Constraints:

- The target may randomly change its velocity at unspecified intervals.
- The algorithm must operate within specified time and memory limits.

Scoring:

- The number of time steps taken to catch the target (lower is better).
- The algorithm is evaluated over a set of randomly generated target trajectories.

Example:
Input: 10 10 \ 5 5 \ 1 1 \ 2 2
Output: NE

Task 13: Dynamic Target Pursuit (Inspired by chat.deepseek.com)

You are tasked with writing a program to simulate an agent (e.g., a robot or drone) that must catch a moving target in a 2D grid-based environment. The target moves according to a predefined pattern, and the agent must determine the optimal path to intercept the target as quickly as possible.

Environment: A finite 2D grid of size $M \times M$, (1 ≤ M ≤ 100) with no wrap-around behavior.

Movement Rules:

- The **agent** moves in four directions: {N, E, S, W}.
- The **target** follows a repeating predefined movement sequence (e.g., E, N, W, S).
- The agent and target move simultaneously, one step at a time.
- The agent catches the target if they occupy the same cell at the same time.
- Both the agent and target cannot move outside the grid boundaries.

Input:

- Grid size M
- Initial positions of agent (x_a, y_a) and target (x_t, y_t)
- Target's repeating movement sequence

Output:

- Sequence of agent moves to intercept the target
- Total number of steps taken
- If interception is not possible, return "Target unreachable"

Scoring:

- 50%: Correctness
- 30%: Algorithmic efficiency
- 20%: Handling of edge cases (e.g., unreachable targets)

Example 13-1:
Input: $M = 5$; Agent: (0, 0); Target: (2, 2); Target Movement Sequence: (E, N, W, S).
Output: Agent moves (E, E, N, N).
Explanation:

At step 1: Agent moves E to (0, 1); Target moves E to (2, 3).
At step 2: Agent moves E to (0, 2); Target moves N to (1, 3).
At step 3: Agent moves N to (1, 2); Target moves W to (1, 2).
At step 4: Agent moves N to (2, 2); Target moves S to (2, 2).
Agent catches the target at step 4.

Remark. This task challenges participants to think about pathfinding, simulation, and optimization in a dynamic environment. It can be extended with additional complexities, such as obstacles in the grid or multiple targets, to increase the difficulty level for higher-tier Olympiads.

Extending this idea, we define the following generalized task:

General Task 14: Virus Immobilization on a Directed Graph

Environment: A directed graph. The adversary ("virus") starts at a known vertex and may move at each time step.

Variants:

- **(A)**: Virus moves arbitrarily along any outgoing edge.
- **(B)**: Virus follows a deterministic rule for choosing edges.

Objective: At each step, one or more vertices (excluding the virus's current location) may be disabled ("closed"). The goal is to immobilize the virus in the minimum number of steps.

Remark. In case (B), the problem may be interpreted as a pseudo-game due to the partially predictable behavior of the virus.

For example,

Task 15: Virus on a Line with Position Access (One-Dimensional)

Environment: Grid size 1..100, Virus starts at given position P_0 in {1, ..., 97}, and can move -2, -1, 1, 2 (arbitrarily).

Interaction Modes:

- P1 (Full Information): A numerical function Pos1() returns the current virus location.
- P2 (Partial Information): A Boolean function Pos2() returns true if the virus is at P; otherwise, false. This function may be used once per move.

Constraint: At each step, one position may be blocked with Clo-put() (excluding the current virus location).

Example 15-1.
Beginning of solution of the *Task 15-P1* for $P_0 = 50$.

- p_time := 0; Clo-put(52); [Virus moves]
- If Pos1() = 51 then Clo-put(53);
- If Pos1() = 49 then Clo-put(47);
- If Pos1() = 48 [Virus's optimal move] then Clo-put(46) [p_time = 2]

Example 15-2.
Beginning of solution of the *Task 15-P2* for $P_{0=}50$.

- p_time := 0; Clo-put(52); [Virus moves]
- If Pos2(51) then Clo-put(53) else Clo-put(47) [Virus moves]
- If Pos2(48) then Clo-put(46) else Clo-put(53) [Virus moves]

Task 16: Deterministic Virus in 2D Grid (for Task 14-B)

Environment: 100 × 100 grid. The virus attempts to move in priority order: East → North → West → South. If all are blocked, the virus stops. Agent can Clo-put an obstacle at each step.

Task: Given the virus's initial location, determine the minimum number of steps required to immobilize it by placing one obstacle per time step.

Example 16-1.

- p_time := 0; P_0 = (99, 100). Clo-put(100, 99) [Virus moves N];
- Clo-put(99, 100);
- Stop.

Result: Virus is immobilized and p_time = 2.

5. Tasks Involving Temporal Constraints

We now consider "timexcels" – time-constrained positions – as part of the environment.

Task 17: Robot Movement on Timed Platforms (1D)

Environment Setup:

- The robot starts at $P_{0\neq}100$
- It can wait or jump(K) where $|K| < 10$
- M timed platforms (coasters) are given: each specified by a position and time

Objective: Find the shortest command sequence to reach 100, using only valid coaster positions at the correct times.

Example 17-1:

- Input: $P_{0=}97$; Platforms: (98, 1), (95, 2), (91, 3), (100, 4)
- Sequence: Clo-wait; Clo-jump(-2) [R = 95, 2nd coaster]; Clo-jump(-3) [R = 92, 3rd coaster]; Clo-jump(8) [R = 100, goal]
- Output: WAIT, JUMP(-2), JUMP(-3), JUMP(8); p_time = 4

Task 18: Robot Navigation with Temporal Obstacles

Environment Setup:

- 1D grid 1..100
- The robot executes commands E and W
- M(1..1000) time-bound obstacles are specified by position and time (1D-timexcels)
- Guaranteed that there exists a sequence of commands to bring Robot to 100

Objective: Determine the shortest sequence of commands that navigates the robot from $P_0 \neq 100$ to 100 while avoiding obstacles.

Example 18-1:

- Input: $P_0 = 97$; Obstacles: (97, 2), (99, 2) [two obstacles appear at p_time = 2].
- Sequence: The command Clo-E cannot be continued because the sequences EE (to 99) and EW (to 97) cannot be executed.
- Output: Path WEEEE, p_time = 5.

Open Question: Can such tasks be solved more efficiently than via brute-force search?

6. Conclusion

This paper presents a range of computational tasks involving dynamic environments and limited observability. These tasks, often deceptively simple in description, reveal deep algorithmic complexity due to temporal dynamics and evolving state constraints. Many such problems are not solvable by brute force due to combinatorial explosion, highlighting the need for intelligent search strategies and heuristics. We propose that incorporating such "natural" tasks into Olympiads can better prepare participants for real-world algorithmic problem solving, as emphasized in Pankov (2008).

References

Steinhaus, H. (1989). *Kalejdoskop Matematyczny*. Wydawnictwa Szkolne i Pedagogiczne, Warszawa.

Steinhaus, H. (1999). *Mathematical Snapshots*. Dover Publs.

Kołodziejczyk, M. (n. d.). Two-pan balance and generalized counterfeit coin problem `https://www.mimuw.edu.pl/~andkom/Kule-en.pdf`

Pankov, P.S. (1990). Duality of control and observation parameters in obtaining guaranteed estimates. In: *Problems of Theoretical Cybernetics: Abstracts of Reports of the IX All-Union Conference* (September 1990). Volgograd, Part 1(2), p. 57.

Pankov, P.S., Imanaliev, T.M., Kenzhaliev, A.A. (2021). Automatic Makers as a Source for Olympiad Tasks. *Olympiads in Informatics*, 15, 75–82.

Pankov, P.S. (2008). Naturalness in Tasks for Olympiads in Informatics. *Olympiads in Informatics: Country Experiences and Developments*, 2**,** 16–23.

P.S. Pankov (1950), doctor of physics-mathematics sciences, prof., corr. member of Kyrgyzstan National Academy of Sciences (KR NAS), was the chairman of jury of Bishkek City OIs, 1985–2013, of Republican OIs, 1987–2012, participates in National OIs since 2020, was the leader of Kyrgyzstani teams at IOIs, 2002–2013, 2018–2023. Graduated from the Kyrgyz State University in 1969, is a head of laboratory of Institute of mathematics of KR NAS.

E.S. Burova (1986), Assistant Professor, Applied Mathematics and Informatics Program, American University of Central Asia.

E.J. Bayalieva (1984), Senior Lecturer in the Software Engineering program, Institute of Computer Technologies and Artificial Intelligence, J. Balasagyn National University.