Rules and tasks description

Gliwice, April 22 – 23, 2017

Contents

1. Editorial

Welcome to the finals of the ninth edition of the programming marathon Deadline24. As usual, the contest is organized by Future Processing and supported by universities and the Ministry of Digital Affairs. For the second time we are hosting you in Muzeum Śląskie in Katowice.

As in the previous years, we hand to you three tasks, traditionally related to the world of intriguing creatures – beetlejumpers. This time, you are given the opportunity to help them with farming potatoes (*A-grid-culture*) and starting space exploration (*Rocket Science*). What is more, you will take part in educating young generations and developing their spatial planning abilities (*B.E.S.T.O.W.*). What will be the result of your interference in the world of the beetlejumpers? We will find out soon.

We hope that the game will be exciting and all the contestants will be able to deal with the tension for the whole 24 hours. Let the best team win!

Deadline24 Team

2. Server communication

Receiving the current information about the virtual world and issuing commands is performed with TCP/IP protocol. Each team connects as a client to a proper competition server. The IP address and port which you use to connect are specified in *Servers* section of the tasks specification. Multiple connections can be established simultaneously, however, the total transfer for each computer is limited. The maximum number of connections and bandwidth limitation are provided in *Technical Arrangements*. Communication is conducted in text mode. A login and a password are required immediately after establishing a connection. Then, the session switches to command mode.

2.1. Logging in

Right after establishing a connection, the server sends a login request terminated with the end of line character: `LOGIN`. You should send your login first, and then the end of the line character. Once it is done, the server will ask for a password (`PASS`). You should send your password in a similar manner as the login. If the authorization succeeds, the server will respond with the `OK` message and will be then ready to receive commands. If the authorization fails, then you will receive the `FAILED 1 bad login or password` message and after that the connection will be closed.

Below you can find an exemplary record of the communication when logging in.

client → server	server → client
	`LOGIN`
`login1`	
	`PASS`
`secret`	
	`OK`

2.2. Commands

Each command consists of a command name, arguments (the number of which depends on the command type) and the end of line character. Parameters should be separated with at least one whitespace.

Server responds to each command with one of the following character strings:

- '`OK`' — if a command is accepted,
- '`FAILED` e msg' — in case of error; where e is the error code, and msg — error message.

Depending on the command, the server may optionally send or receive additional data. If the additional data are sent from a client to the server, then the latter will reply after receiving them in the above manner. Examples of communication records and the list of possible errors for each task are available in their descriptions.

Limitation on the number of commands There is a limit of the maximum number of commands issued during one turn for each server running each task. Reaching the limit will be signaled by the following error: `FAILED 6 commands limit reached, forced waiting activated`. When the error occurs, the server will send the additional message: `WAITING x` — where x ($x \in \mathbb{R}$) is the number of seconds left to wait, i.e., till the end of a current turn.

2.3. Conventions

If not stated otherwise, we assume that:

- Every line ends with a single character of ASCII code 10 ('\n'). Carriage return character ('\r'; ASCII code 13) which may appear in certain operating systems will be treated as a whitespace character.

- In case of data sent by the server, words and numbers will be separated with a single space character.
- In case of data sent by the client to the server (i.e., command parameters), any number of white characters (greater than zero) is allowed between the data and also at the beginning or at the end of the line.
- Whitespace characters are: space, carriage return ('\r') and tabulation sign ('\t').

3. Overall score

3.1. Contest speed up

K coefficient For each server, the score of a given team is additionally multiplied by the K coefficient – the coefficient of the contest speed. During the contest, the coefficient value is being increased exponentially, from 1 to 8. It means that a team may gain much more points in the last hours of the contest rather than at the beginning of the game. Therefore, it is important to keep improving your solutions, so as not to be outscored by other teams. The current value of K is available in the proper command of each task.

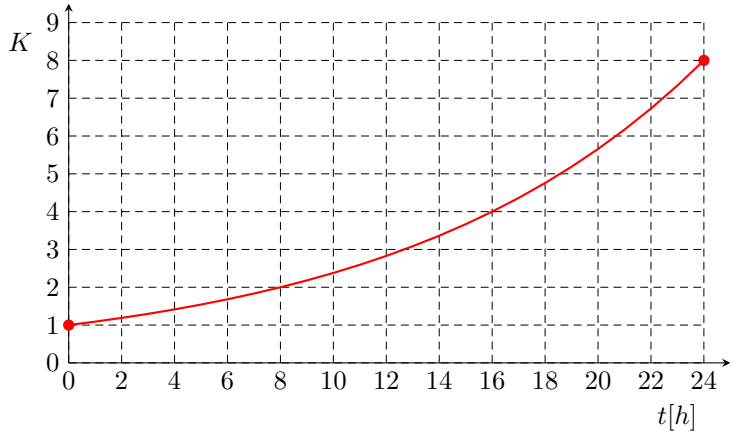

Figure 1: The change of the K coefficient in time.

3.2. Ranking points

Points Teams get points on each server as defined in a task specification. Points gained on each server will be converted into **ranking points** which are defined as a product of the value 100 and the ratio of the team's result (on a given server) and the arithmetic mean of the three best results (on a given server). Points given to a team for a specific task are the arithmetic mean of ranking points from all servers of a given task.

General ranking Competition raking is based on the sum of ranking points of all tasks.

4. Emergency situations

In case of situation where not every team is able to participate in the contest on the stated terms (e.g., because of the electricity outage, total or partial failure of local network, problems with contest hardware, etc.), and blame lies neither on those teams nor their equipment, the organisers will suspend the activity of the contest system. Additionally, the participants will be informed of a restart of the system by a local WWW service of the contest. Points will not be calculated during the breakdown. In that case all connections with the contest server might be lost.

5. B.E.S.T.O.W.

5.1. Introduction

Beetlejumpers were often accused of bringing nothing to the galactic community but relentless conquest, violence, and chaos. Those accusations are obviously absurd – beetlejumpers have a rich culture they eagerly share with all other civilizations – mainly by conquest, of course, but this is their culture and way of life.

One aspect of such culture is a simple game, used for educating beetlejumper larvae in spatial thinking. It is called Beetlejumper Expert Spatial Training Operation Workshop and its only mode worth an ambitious larva is the COM/COO (competitive / cooperative) version, intended for the proper training.

5.2. Problem

You – and your randomly assigned, anonymous partner – are given access to a predetermined, known to both of you, set of pieces in great variety of shapes, colors, prices, and values. You buy pieces (within certain limitations) and place them on your own playing space in an attempt to maximize its profit – both by placing high-value pieces and acquiring bonus for large adjoint cube.

To make things interesting, the piece is replicated and you can place it in another space as well – but the second space is shared with your game partner, who does exactly the same in his turn. In the shared space, your aim is to create large continuous shapes (as the shared space is flat) of single color.

5.3. Game model

The game takes place in 3-D and 2-D spaces in the form of turns of the same length. During every turn, the active player chooses a 3-D piece from those made available to him and places it in both his own space and in the shared space. The active player is changed in reaction to taken pieces.

The goals are different on different spaces:

On own space the goal is to create as large and adjoint cube as possible; pieces colors do not matter; what matters is the size of the largest cube a player built and the value of pieces he placed. This space is 3-D.

On shared space the goal is to create largest possible single–colored areas in player's preferred colors. This space is 2-D and there are additional mechanics of choosing preferred colors and placing 3-D pieces in 2-D.

5.4. Game time & space

5.4.1. Game space

The game space consists of two 3-D spaces – one for each player (own space) and one 2-D shared space. All spaces are completely empty in the beginning; their sizes can be different in different games. Own

space is divided into equal cubes and the shared space is divided into equal squares.

A player sees both his and his opponent's own space, as well as the shared space.

5.4.2. Game time

Game time is measured in discrete turns. In each turn, only the active player can influence the spaces. Active player is determined via **effort** track, where players' effort tokens denote how much effort the player exerted. Each piece has an associated effort that moves player's effort marker forward when the player takes that piece. The active player changes in two cases: when player's effort marker moves ahead of his opponent's marker as a result of acquiring effort from taking pieces or when the player did not take any of the pieces. In the latter case, the player's marker is automatically moved one step ahead of the opponent's marker (see 5.5.11.)

Effort track fulfills two additional roles. First, when both players' markers reach the 'max effort' position, the game ends and scores become final. Second, there are several equally-spaced thresholds; player who reaches a threshold before his opponent does, receives a benefit (see 5.5.13.).

5.5. Playing the game

5.5.1. Starting cash

Players are given the same amount of starting cash. Starting cash does not contribute to the score, it is the goal to convert it into as high score as possible.

5.5.2. Turns

Players' turns are not taken alternately – it is common that one player has several consecutive turns. This is decided via effort track. Each piece has an effort associated with it; effort moves the marker and if marker passes another player's marker, the active player changes. Therefore, it is possible to take several low–effort pieces in sequence.

5.5.3. Pieces

Each piece consists of at least one (usually far more) cubical blocks, connected to each other at their walls. The pieces are always adjoint – there are no free–floating blocks. Each block can be in any of the colors that are in play, this means that every piece can – and in most cases, will – be multi–colored. Each piece has associated **effort**, **price** and **value**. Players should minimise the costs (see 5.5.6.) and use the pieces' shapes, colors and values to increase scores of both own and shared spaces.

5.5.4. Taking pieces

During each turn players are able to take no more than one piece. Additionally, at any given moment, the player is able to hold one piece at most, therefore each time a piece is taken, the previous one is lost.

5.5.5. Vector of pieces

Pieces are arranged in a cyclical vector of N elements, numbered from 1 to N. At the start of the game, all pieces are considered available. Both players start with pointers pointing at the same piece. Either of you can access only few of the pieces at the moment, which ones, is determined by the players' range R (which is constant for the entire game) and the position of the pointer.

Players are able to reach a number of next available pieces that is less or equal to the range R (usually the number is exactly equal, but near the end of the game there may be less available pieces than the range allows to reach). Once a piece is taken, it is no longer available to any player. The player's pointer moves to the piece just taken. It is possible for choice areas of both players to overlap during the game (especially at the beginning), forcing them to compete for pieces.

Example The vector contains 20 pieces, numbered from 1 to 20. Piece range is 4. Both players' pointers are at 1, and both are allowed to take pieces 1, 2, 3, and 4. Player A takes piece 2, his pointer is moved to 2, changing the pieces available to him to 3, 4, 5, 6. After placing the piece and taking next one – for instance, 5, his pointer is moved to 5, and his in–range set is 6, 7, 8, 9. He ends his turn and turn passes to player B, who can choose from pieces 1, 3, 4, 6, as 2 and 5 have already been taken by A.

5.5.6. Cost of buying a piece

Cost of each piece is twofold: effort and price. Effort advances the effort marker of the player (see 5.5.2.). Price is subtracted either from the player's starting cash or own space profit. The starting cash is preferred when buying the piece, if the starting cash is not enough, rest of the price is subtracted from own space profit. The sum of starting cash and own space profit is the player's **piece purchasing power**. It is possible to have too little points to afford piece's price; in such case, a piece cannot be taken despite being in range.

5.5.7. Value of a piece

Each piece has a certain value which comes into play during own space profit calculation. Each piece placed at player's own space contributes to the overall own space profit. On the shared space, the piece value does not matter.

5.5.8. Colors of blocks

Each piece consists of cubical blocks of various colors. The colors come into play when placing pieces on the shared space. Placing the piece on the shared space creates areas of different colors.

5.5.9. Placing a piece

The player, after obtaining the piece, is able to place it on both own and shared space. However each obtained piece can be placed on each of these spaces only once. Players are also able to place a piece acquired in the previous turn, as long as they didn't acquire another piece nor they have already placed the piece in the previous turn.

In case of player placing a piece that was acquired in the previous turn, it is possible for the player to obtain and place a second piece in the same turn.

Placing a piece on own space Obtained piece can be rotated around any of the three (X, Y, Z) axes and can be placed anywhere on the target space. Targeting is handled via providing the position of the designated (one with (0,0,0) coordinates) block of the piece and rotations that are to be applied to it.

Placing the piece on own space increases the space profit. The amount that is added to overall own space profit is calculated from the piece value and the placement quality (see 5.5.14.).

On player's own space, colors of the blocks do not matter.

Placing a piece on the shared space As the shared space is 2-D, and the pieces are 3-D, they have to be turned into 2-D pieces to be placed. This is done via means of orthographic graphical projection – the piece can be rotated until the desired projection is visible on projection plane and then projected on the shared space, with visible cubes becoming squares. Colors of the cubes become colors of the squares after the projection.

Each adjoint area (squares connected by sides) of the same color has its own score. The bigger the area, the bigger its score. How much the score of a single area is worth for every player depends on how much the player prefers that color.

On the shared space, the piece value does not matter, only the colors and players' preferences do (see 5.5.12.).

Placing a piece – a practice So as the game is used for educating beetlejumpers, there are special operations available which give opportunities for practicing proper understanding of both rotation and orthographic projection. The practices are made on separated space and they do not influence the competition between rivals.

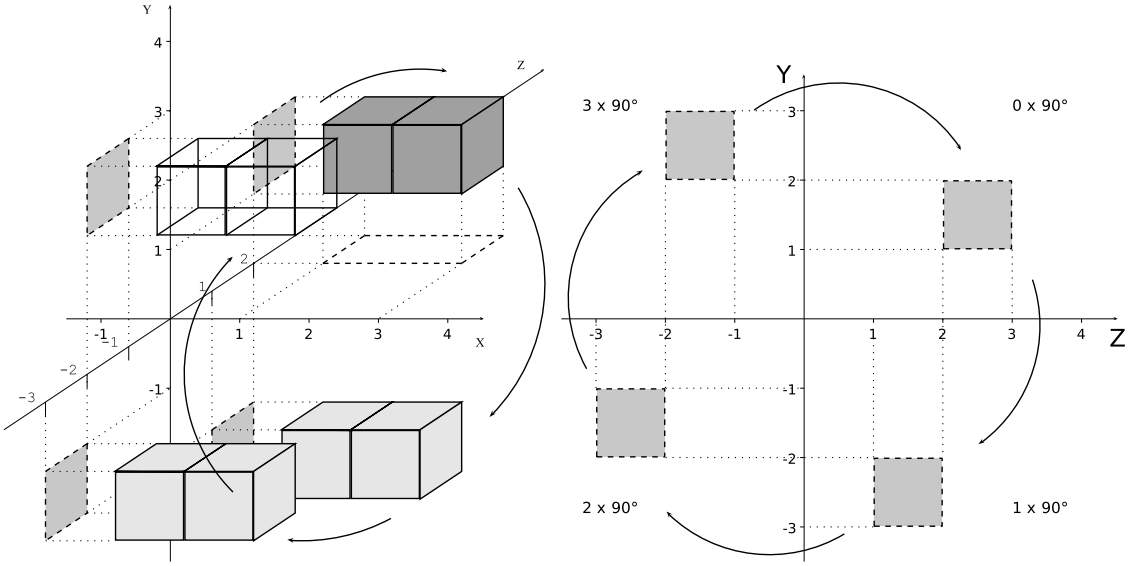

Figure 2: Exemplary piece rotation around X axis.

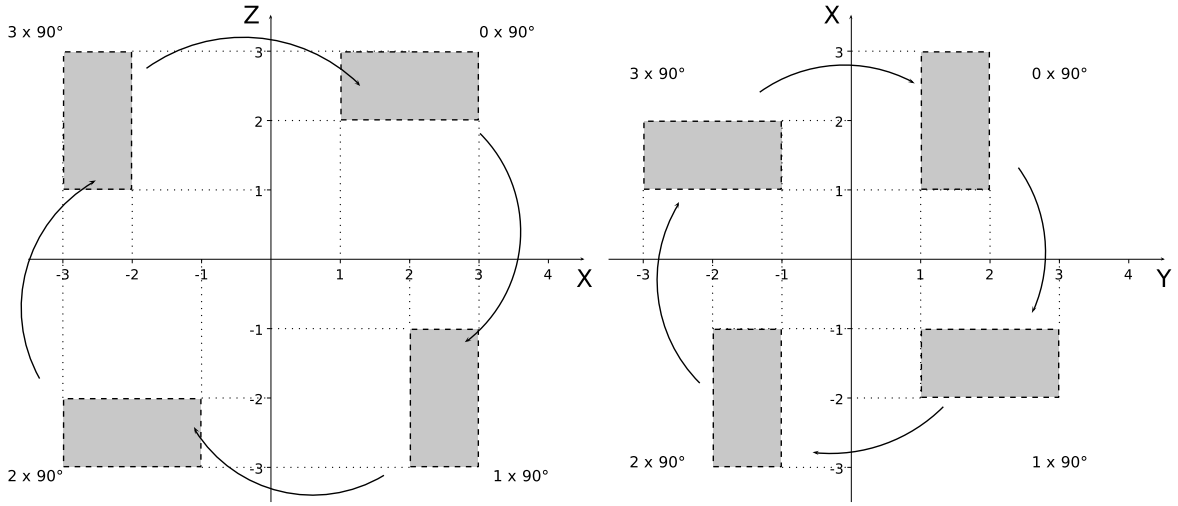

Figure 3: Exemplary piece rotation around Y axis (left) and around Z axis (right).

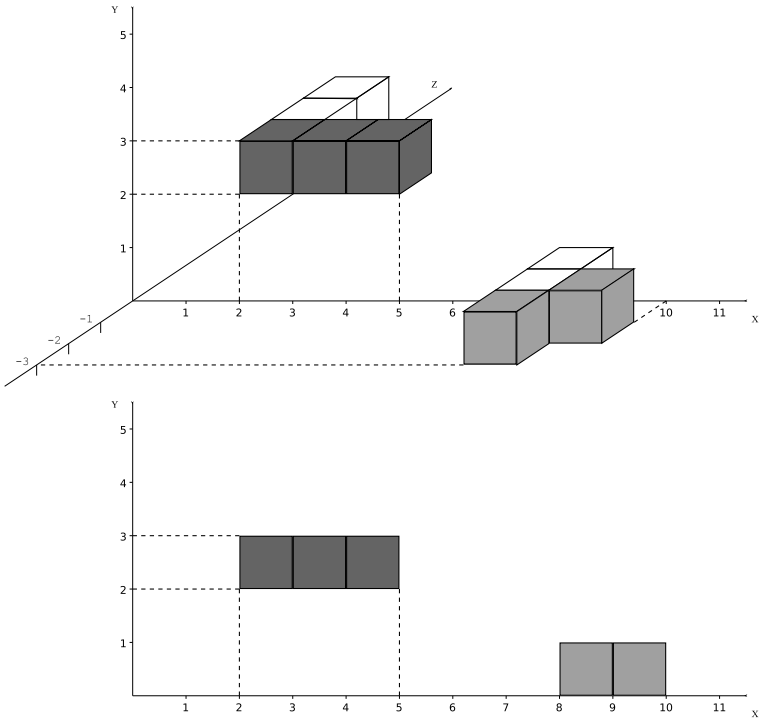

Figure 4: Exemplary pieces orthographic projection.

5.5.10. Turn progress

Each of the actions below can be executed by active player in any order, possibly many times.

- Active player chooses a piece from those available to him. The pointer on the vector containing the pieces is moved according to the rules described in 5.5.5. This action can be executed at most once during the turn.
- Active player places previously acquired piece in his own space, see 5.5.2.
- Active player places previously acquired piece in shared space, see 5.5.2.
- Active player determines his preference of colors (see 5.5.12.)

After limited time of the turn passes, the turn ends and the following takes place:

- If effort marker reached a threshold position on the track, and other player's effort marker didn't reach this threshold yet, the player receives a benefit (see 5.5.13.).
- If effort marker passed an endgame position on the track and other player's effort marker already passed it, the game ends.
- Active player is determined: If the effort marker of active player is ahead of the other player, the next player takes turn – the active player changes. If the marker is behind or on the equal position, the active player remains unchanged.

Example of active player determination process Both players start with effort marker on 0. Player A chooses a piece with effort 3 and plays it. His effort marker moves to 3. Player B has effort marker on 0, and as $0 < 3$, it's B's turn. He takes a piece with effort 2 and plays it, advancing his effort marker to 2. Player A has effort marker on 3, player B – on 2. As $2 < 3$, it is still B's turn. Player B takes a piece with effort 3 and plays it. This advances B's effort marker to 5. Player A has effort marker on 3, player B – on 5. As $3 < 5$, it is A's turn again. Player A takes a piece with effort 2 and plays it. This advances A's effort marker to 5. Both markers are on 5, but player A did not get ahead of B, thus it is A's turn.

5.5.11. Skip turn

Acquiring piece is optional. No matter whether a player did it, when assigned time of the turn runs out, the turn ends. If the player did not take the piece in the assigned time, he is considered to have skipped the turn. In such case, his effort marker moves one step ahead of his opponent. This is useful in case the player does not wish to make a move (for instance, to preserve resources) or cannot afford a price of any of the pieces in range.

Skipping turn is done automatically if player takes no piece in the limited time of his turn.

When skipping turn a small amount of cash is added to the player's starting cash. This increases player's piece purchasing power, but does not contribute to the score directly.

5.5.12. Shared space preferred colors

There are two crucial differences between own and shared space: importance of colors and number of dimensions. Areas of single color are scored for their size, but as the space is shared, both players receive the points. The catch – and the competition – is simple.

Each player, at any moment of his turn (this of course requires him to be the active player), can assign a score multiplier (chosen from a personal pool of [1, 3, 5]) to one of the colors. Once he does, he cannot change his mind for the duration of the game and he cannot assign the same multiplier to any other color, also he cannot assign any other multiplier to this color. Moreover, his opponent cannot assign his multiplier of equal value to the chosen color – he may assign another, though. Aside that limitation, given player's assignments do not affect his opponent.

All assignments are final – they cannot be changed until the end of the game.

Example There are 5 colors in the current game – red, green, blue, orange, white. Player A, at some point of the game, notes that there are some nice green shapes forming and chooses to assign $\times 5$ multiplier to green. Once he does, player B cannot assign his $\times 5$ to green. He chooses, after few further turns of observation, to assign $\times 3$ to green. A while later, player B notices that orange forms even better shapes and assigns his $\times 5$ to orange. Few more turns pass, and both players notice the most profitable color would be white – but none of them have free $\times 5$ multipliers, and player A assigns his $\times 3$ multiplier to white. Player B may assign $\times 1$ to white, or wait for further development.

Multipliers that were not assigned before the game ended are lost. All colors that the player did not assign a multiplier during the game are considered to have the multiplier of $\times 0$ for that player.

5.5.13. Single cube piece – an effort-based benefit

Beetlejumpers appreciate action and relentless march forward. Therefore, a player who reaches any effort threshold on the track before his opponent does, will receive a benefit – a single–use ability to fill a gap with a single cube piece of chosen color to be used in the shared space and a single cube piece with zero value to be used on his own space. The piece has no assigned effort and a zero price. The player may receive the benefit several times. They cumulate and might be used in further turns. If not used during the current game the benefits are lost.

5.5.14. Piece quality evaluation

Aside precision and building ability, beetlejumpers glorify style. Construction should be aesthetic and beautiful to behold – from the very beginning to the very end. Therefore, good placement of each piece is rewarded, while random, haphazard placement is punished. After a piece is placed, it's bounding box is calculated – the smallest cuboid on the player's space that contains the piece. Inside this cuboid, a count of two types of cubes is made:

1. cubes count that make the piece: V
2. filled cubes count inside bounding box: C_{bb}

The number of cubes in the bounding box that belong to pieces previously laid is important. The more of them are filled, the better. In essence, the better the piece fits into already existing construction, the better its quality score is.

Quality is calculated as follows:

$$Q = C_{bb} - V - Q_{min} \tag{1}$$

where Q_{min} is the minimum placement quality threshold.

If Q is above zero, the piece is considered well-placed, and carries a bonus P_b. Otherwise, the piece is placed poorly, and the Commission for Construction Style imposes a penalty P_p. Finally either the bonus or the penalty is applied to piece's value P_v.

Bonus is calculated via:

$$P_b = Q \times Q_{gi} \tag{2}$$

while penalty is:

$$P_p = Q \times Q_{pi} \tag{3}$$

Q_{gi} and Q_{pi} are values which remain constant during a single game.

Own space profit is increased by **final piece value** P_{vf}, which is calculated as follows (either bonus or penalty is equal to 0):

$$P_{vf} = P_v \cdot (1 + P_p + P_b) \tag{4}$$

Mind that quality bonus does contribute to player's piece purchasing power.

5.6. Start and the end

Initial state of the game All spaces are empty, the cyclical vector is filled with pieces, both pointers are on piece 1. Both effort markers are on 0 spot of the effort track. Both players have the same amount of starting cash for purchases. Data regarding effort thresholds and endgame point are available.

End of the game The game ends after both players' effort markers pass a predetermined position – the maximum effort point – on the effort track.

5.7. Scoring

Score is calculated based on state of both own and shared spaces. Both spaces are scored continuously and score from the end of the previous turn is available; final score is calculated at the end of the game. Starting cash serves only the purpose of providing initial purchasing power and is ignored in the final score.

5.7.1. Solid cube bonus

Both precision and ability to build are appreciated by beetlejumpers. Profit of the own space is multiplied by a factor C_b determined by the lenght of the edge of the largest adjoint cube in player's own space (D) and a constant I. The cube has to be continuous and solid, but it may have additional pieces attached that extend from the cube. Factor is calculated by formula:

$$C_b = I^D \tag{5}$$

5.7.2. Scoring of own space

The final score of the own space depends on own space profit and solid cube bonus (see 5.7.1.) and is equal to:

$$S_{mo} = S_{mp} \times C_b \tag{6}$$

5.7.3. Scoring of shared space

Each adjoint area of given color is given a score, based on its area A in squares:

$$S_c = A^{\,A_e} \tag{7}$$

The A_e exponent is constant in any given game and known from the beginning. Those S_c scores are multiplied by each player multipliers (see 5.5.12.), as they assigned them during the game, and added to player's score for that color.

Each player's shared space score is a sum of all colors' scores for that player:

$$S_{ms} = \sum_{i=1}^{C} S_c^i \times C_m^i \tag{8}$$

where C is the number of colors in use, S_c^i is the particular color score on shared space, and C_m^i is the player's multiplier for that color.

5.7.4. Combined score

Combined score S_{mc} of the player depends on both own and shared space score and is calculated as:

$$S_{mc} = S_{mo} + S_{ms} \times H \tag{9}$$

where S_{mo} is the score from own space, S_{ms} is the score on shared space, and H is the shared space coefficient – a game parameter specific for this particular game.

5.7.5. Victory points

The players combined scores are compared each turn and based on this comparison, the final scoring is produced.

The player with greater combined score is given 100 victory points while the game continues. When the game is finished this value is multiplied by 3, therefore victory is always worth 300 points. Defeated player is always given points which represent his combined score as percentage of combined score of the winning player (thus it is always below 100).

In case of a draw, both players receive 150 points. In rare case in which both players refuse to play the match (make no moves at all), the game is considered drawn, but scored at 0 for both players.

5.8. Commands

General guidelines on the communication protocol (connecting, logging in, commands, response format) are described in section *Server communication*. Below you can find commands for the problem B.E.S.T.O.W..

DESCRIBE_WORLD Returns parameters of the world and the value of the score scaling coefficient.

Parameters: none

Data (from server):

In the first and only line the server returns values separated with a single spaces:

- E_e ($E_e \in \mathbb{N}$, $100 \leqslant E_e \leqslant 5000$) — end of effort track – amount of effort that has to be reached to end the game,
- E_p ($E_p \in \mathbb{N}$, $10 \leqslant E_p \leqslant 1000$) — effort threshold period – amount of effort that separates effort thresholds (see 5.5.13.),
- C ($C \in \mathbb{N}$, $2 \leqslant C \leqslant 16$) — number of colors in use,
- B_s ($B_s \in \mathbb{N}$, $20 \leqslant B_s \leqslant 40$) — length of the edge of the shared space,
- B_o ($B_o \in \mathbb{N}$, $10 \leqslant B_o \leqslant 12$) — length of the edge of the own space,
- R ($R \in \mathbb{N}$, $1 \leqslant R \leqslant 40$) — maximum range the access pointer can reach from current position in vector of pieces,
- I ($I \in \mathbb{R}$, $1.0 \leqslant I \leqslant 2.0$) — impact of the own space's adjoint cube bonus on the score,
- S_p ($S_p \in \mathbb{N}$, $0 \leqslant S_p \leqslant 10000$) — starting amount of cash,
- H ($H \in \mathbb{R}$, $0.0002 \leqslant H \leqslant 5000$) — shared space score coefficient,
- Q_{min}($Q_{min} \in \mathbb{N}$, $0 \leqslant Q_{min} \leqslant 5$) — placement quality threshold,
- Q_{gi}($Q_{gi} \in \mathbb{R}$, $0.01 \leqslant Q_{gi} \leqslant 0.1$) — score impact factor of good (above or equal threshold) piece placement,
- Q_{pi}($Q_{pi} \in \mathbb{R}$, $0.05 \leqslant Q_{pi} \leqslant 0.2$) — score impact factor of poor (below threshold) piece placement,
- A_e($A_e \in \mathbb{R}$, $1.2 \leqslant A_e \leqslant 2$) — shared space area exponent,
- S_r ($S_r \in \mathbb{N}$, $1 \leqslant S_r \leqslant 50$) — amount of starting cash player recovers when skipping a turn,
- T ($T \in \mathbb{N}$, $1 \leqslant T \leqslant 3$) — duration of a single turn in seconds,
- L ($L \in \mathbb{N}$, $1 \leqslant L \leqslant 100$) — the maximum number of commands which can be issued in one turn,
- K ($K \in \mathbb{R}$, $1 \leqslant K \leqslant 8$) — value of the scaling coefficient.

GET_MATCH_INFO Provides match information – situation at the start of current turn.

Parameters: none

Data (from server):

In the first line, the server returns data regarding the player, separated with a single spaces:

- S_{mf} ($S_{mf} \in \mathbb{R}$, $0 \leqslant S_{mf} \leqslant 100.0$) — player's victory points,
- S_{mo} ($S_{mo} \in \mathbb{N}$, $0 \leqslant S_{mo}$) — player's score from his own space,
- S_{ms} ($S_{ms} \in \mathbb{N}$, $0 \leqslant S_{ms}$) — player's score from shared space,
- S_{mm} ($S_{mm} \in \mathbb{N}$, $0 \leqslant S_{mm}$) — player's piece purchasing power – cash that can be spent on pieces, sum of starting cash and S_{mp},
- S_{mp} ($S_{mp} \in \mathbb{N}$, $0 \leqslant S_{mp}$) — player's own space profit,
- E_m ($E_m \in \mathbb{N}$, $0 \leqslant E_m$) — player's effort,
- C values of M_m^i ($i \in \mathbb{N}$, $1 \leqslant i \leqslant C$, $M_m^i \in \{0, 1, 3, 5\}$) — player's color multipliers,
- B_{mo} ($B_{mo} \in \mathbb{N}$, $0 \leqslant B_{mo} \leqslant 20$) — number of available single-cube pieces for own space,

- B_{ms} ($B_{ms} \in \mathbb{N}$, $0 \leqslant B_{ms} \leqslant 20$) — number of available single-square pieces for shared space.

In the second line, the server returns data regarding the opponent, separated with a single spaces:

- S_{of} ($S_{of} \in \mathbb{R}$, $0 \leqslant S_{of} \leqslant 100.0$) — opponent's final score,
- S_{oo} ($S_{oo} \in \mathbb{N}$, $0 \leqslant S_{oo}$) — opponent's score from his own space,
- S_{os} ($S_{os} \in \mathbb{N}$, $0 \leqslant S_{os}$) — opponent's score from shared space,
- S_{oc} ($S_{oc} \in \mathbb{N}$, $0 \leqslant S_{oc}$) — opponent's piece purchasing power – cash that can be spent on pieces, sum of starting cash and S_{op},
- S_{op} ($S_{op} \in \mathbb{N}$, $0 \leqslant S_{op}$) — opponent's own space profit,
- E_o ($E_o \in \mathbb{N}$, $0 \leqslant E_o$) — opponent's effort,
- C values of M_o^i ($i \in \mathbb{N}$, $1 \leqslant i \leqslant C$, $M_o^i \in \{\texttt{0}, \texttt{1}, \texttt{3}, \texttt{5}\}$) — opponent's color multipliers,
- B_{oo} ($B_{oo} \in \mathbb{N}$, $0 \leqslant B_{oo} \leqslant 20$) — number of available single-cube pieces for own space,
- B_{os} ($B_{os} \in \mathbb{N}$, $0 \leqslant B_{os} \leqslant 20$) — number of available single-square pieces for shared space.

In the third line, the server returns data regarding the turn status:

- J, ($J \in \{\texttt{None}, \texttt{Me}, \texttt{Opponent}\}$) — who is the active player in this turn.

GET_ALL_PIECES Provides information about all pieces in the game.

Parameters: none

Data (from server):

The first line contains number of pieces:

- N ($N \in \mathbb{N}$, $10 \leqslant N \leqslant 1000$) — number of pieces in game,

Next N lines contain one piece each, in the form of values separated by a single spaces:

- ID ($ID \in \mathbb{N}$, $1 \leqslant ID \leqslant N$) — piece ID,
- C_n ($C_n \in \mathbb{N}$, $1 \leqslant C_n \leqslant 20$) — number of cubes the piece consists of,
- C_n values of $C_xC_yC_z$ ($C_x, C_y, C_z \in \mathbb{Z}$, $-10 \leqslant C_x, C_y, C_z \leqslant 10$) — coordinates of each of the cubes that make the piece,
- C_n values of C_c ($C_c \in \mathbb{N}$, $1 \leqslant C_c \leqslant C$) — color of each of the cubes that make the piece,
- P_c ($P_c \in \mathbb{N}$, $1 \leqslant P_c \leqslant 2000$) — price of the piece,
- P_v ($P_v \in \mathbb{N}$, $1 \leqslant P_v \leqslant 2000$) — point value of the piece,
- P_e ($P_e \in \mathbb{N}$, $1 \leqslant P_e \leqslant 5$) — effort of the piece.

After a player issues this command, he needs to wait either **100 turns** or until the end of the match (whichever happens first) before issuing it again.

GET_AVAILABLE_PIECES Provides list of all yet unused pieces in the game – situation at the start of this turn.

Parameters: none

Data (from server):

The first and only line contains, separated by single spaces:

- N_l ($N_l \in \mathbb{N}$) — number of pieces left available,
- N_l values of ID_i ($i \in \mathbb{N}$, $1 \leqslant i \leqslant N_l$, $1 \leqslant ID_i$) — IDs of available pieces.

GET_PIECES_IN_RANGE Provides list of all pieces that are in the range of the player – situation at the start of this turn.

Parameters: none

Data (from server):

The first and only line contains, separated by single spaces:

- N_r ($N_r \in \mathbb{N}$, $0 \leqslant N_r \leqslant R$) — number of pieces in range,
- N_r values of ID_i ($i \in \mathbb{N}$, $1 \leqslant i \leqslant N_r$, $1 \leqslant ID_i$) — IDs of pieces in range.

BUY_PIECE Buys a piece from pieces in range. The command is available only for an active player.

Parameters:

- P_i ($P_i \in \mathbb{N}$, $1 \leqslant N_r \leqslant R$) — position of piece identifier (starting from 1) on the list of pieces in range.

SET_MULTIPLIER Sets multiplier for the chosen color. The command is available only for an active player.

Parameters:

- C_m ($C_m \in \mathbb{N}$, $1 \leqslant C_m \leqslant C$) — chosen color ID,
- M_s ($M_s \in \{1, 3, 5\}$) — multiplier.

GET_MY_SPACE Lists team's own space elements – situation at the start of this turn.

Parameters: none

Data (from server):

In the first line server returns the number N.

In N^2 next lines server returns N blocks of N lines containing N values each. Each block contains N^2 units of space that share the same Z-axis position. Each line contains

- N values of F_i that share the same Y-axis position but differ by X-axis position ($F_i \in \{0, 1\}$).

GET_OPPONENT_SPACE Lists opponent's own space elements – situation at the start of this turn.

This command is identical in structure to GET_MY_SPACE.

GET_SHARED_SPACE Lists shared space elements – situation at the start of this turn.

Parameters: none

Data (from server):

In the first line server returns number N. Next N lines contains N values each. Each line contains

- N values of F_j that share the same Y-axis position but differ by X-axis position ($F_j \in \mathbb{N}$, $0 \leqslant F_j \leqslant C$). Value designates color of the specific field.

PROJECT_PIECE Projects a piece on the 2-D space for educational purpose. This is an auxiliary command which does not place the piece.

Parameters:

- C_n ($C_n \in \mathbb{N}$, $1 \leqslant C_n \leqslant 5$) — count of piece's cubes,
- C_n values of $C_xC_yC_z$ ($C_x, C_y, C_z \in \mathbb{Z}$, $-10 \leqslant C_x, C_y, C_z \leqslant 10$) — coordinates of each of the cubes that make the piece,
- R_x ($R_x \in \mathbb{Z}$) — number of times the piece should be rotated by 90 degrees around the X axis,
- R_y ($R_y \in \mathbb{Z}$) — number of times the piece should be rotated by 90 degrees around the Y axis,
- R_z ($R_z \in \mathbb{Z}$) — number of times the piece should be rotated by 90 degrees around the Z axis.

Data (from server):
The only line contains:

- C_{sn} ($C_{sn} \in \mathbb{N}$, $1 \leqslant C_{sn} \leqslant 5$) — count of piece's squares,
- C_{sn} values of C_xC_y ($C_x, C_y \in \mathbb{Z}$, $-10 \leqslant C_x, C_y \leqslant 10$) — coordinates of each of the squares that make the projected piece.

ROTATE_PIECE Rotates a piece in 3-D space for educational purpose. This is an auxiliary command which does not place the piece.

Parameters:

- C_n ($C_n \in \mathbb{N}$, $1 \leqslant C_n \leqslant 5$) — count of piece's cubes,
- C_n values of $C_xC_yC_z$ ($C_x, C_y, C_z \in \mathbb{Z}$, $-10 \leqslant C_x, C_y, C_z \leqslant 10$) — coordinates of each of the cubes that make the piece,
- R_x ($R_x \in \mathbb{Z}$) — number of times the piece should be rotated by 90 degrees around the X axis,
- R_y ($R_y \in \mathbb{Z}$) — number of times the piece should be rotated by 90 degrees around the Y axis,
- R_z ($R_z \in \mathbb{Z}$) — number of times the piece should be rotated by 90 degrees around the Z axis.

Data (from server):
The only line contains:

- C_{sn} ($C_{sn} \in \mathbb{N}$, $1 \leqslant C_{sn} \leqslant 5$) — count of piece's cubes,
- C_{sn} values of $C_xC_yC_z$ ($C_x, C_y, C_z \in \mathbb{Z}$, $-10 \leqslant C_x, C_y, C_z \leqslant 10$) — coordinates of each of the cubes that make the rotated piece.

PLACE_OWN_PIECE Places the piece on own space. The command is available only for an active player.

Parameters:

Parameters must be separated by single spaces:

- P_x ($P_x \in \mathbb{N}$) — position on the X axis of the [0,0,0] cube of the rotated piece,
- P_y ($P_y \in \mathbb{N}$) — position on the Y axis of the [0,0,0] cube of the rotated piece,
- P_z ($P_z \in \mathbb{N}$) — position on the Z axis of the [0,0,0] cube of the rotated piece,
- R_x ($R_x \in \mathbb{Z}$) — number of times the piece should be rotated by 90 degrees around the X axis,
- R_y ($R_y \in \mathbb{Z}$) — number of times the piece should be rotated by 90 degrees around the Y axis,
- R_z ($R_z \in \mathbb{Z}$) — number of times the piece should be rotated by 90 degrees around the Z axis.

PLACE_SHARED_PIECE Places the piece on shared space. The command is available only for an active player.

Parameters:

Parameters must be separated by single spaces:

- P_x ($P_x \in \mathbb{N}$) — position on the X axis of the [0,0] square of the rotated and projected piece,
- P_y ($P_y \in \mathbb{N}$) — position on the Y axis of the [0,0] square of the rotated and projected piece,
- R_x ($R_x \in \mathbb{Z}$) — number of times the piece should be rotated by 90 degrees around the X axis,
- R_y ($R_y \in \mathbb{Z}$) — number of times the piece should be rotated by 90 degrees around the Y axis,
- R_z ($R_z \in \mathbb{Z}$) — number of times the piece should be rotated by 90 degrees around the Z axis.

PLACE_SINGLE_CUBE_PIECE Places single cube piece on own space. The command is available only for an active player.

Parameters:

Parameters must be separated by single spaces:

- P_x ($P_x \in \mathbb{N}$, $0 \leqslant P_x \leqslant B_o$) — position of single cube of the piece on the X axis,
- P_y ($P_y \in \mathbb{N}$, $0 \leqslant P_y \leqslant B_o$) — position of single cube of the piece on the Y axis,
- P_z ($P_z \in \mathbb{N}$, $0 \leqslant P_z \leqslant B_o$) — position of single cube of the piece on the Z axis.

PLACE_SINGLE_SQUARE_PIECE Places single square piece on shared space. The command is available only for an active player.

Parameters:

Parameters must be separated by single spaces:

- P_x ($P_x \in \mathbb{N}$, $0 \leqslant P_x \leqslant B_s$) — position of single square of the piece on the X axis,
- P_y ($P_y \in \mathbb{N}$, $0 \leqslant P_y \leqslant B_s$) — position of single square of the piece on the Y axis,
- C_p ($C_p \in \mathbb{N}$, $1 \leqslant C_p \leqslant C$) — color of the square.

WAIT Waits till the next turn begins.

Parameters: none

Data (from server):

The server returns a single line of characters:

- WAITING S

where S ($S \in \mathbb{R}, S \geqslant 0$) stands for the number of seconds left to wait. Once the period is over, the server sends an additional line:

- OK

5.9. Errors

In case the command is incorrect, the server responds according to the description in section *Server communication* with the following message:

- '`FAILED` *e msg*',

where *e* is an error code, and *msg* — an error message. The table below consists of errors that may occur during communication process in problem B.E.S.T.O.W..

error code	error message
1	bad login or password
2	unknown command
3	bad format
4	too many arguments
5	internal error, sorry...
6	commands limit reached, forced waiting activated
101	specific command limit reached
102	piece not available
103	not your turn
104	piece out of space bounds
105	one of the elements placed on non empty space
106	you do not have a piece
107	cannot afford a piece
108	piece already placed on own space
109	piece already placed on shared space
110	match ended
111	color does not exist
112	invalid multiplier
113	equal multiplier already assigned to this color by opponent
114	piece already taken this turn
115	no single cube piece available
116	no single square piece available
117	you already picked a multiplier for this color
118	you already picked a different color for this multiplier

5.10. Servers

The games will be held on servers with a different types of pieces.

Table 1: Server addresses and parameters.

Name	**Address:Port**	**Number of colors**	**Piece shape difficulty**
Vanilla	`universum.dl24:20000`	medium	medium
Wicked	`universum.dl24:20001`	small	high
Flower	`universum.dl24:20002`	large	low

5.11. Example

Below you will find an exemplary record of the communication with the server.

client → server	server → client
	`LOGIN`
`login1`	
	`PASS`
`secret`	
	`OK`
`DESCRIBE_WORLD`	
	`OK`
	`10 5 5 10 10 3 1.410000 1500 1.000000 1`
	`↪ 0.010000 0.100000 1.500000 5 3600 30 1.056079`
`GET_MATCH_INFO`	
	`OK`
	`0.000000 0 0 1500 0 0 0 0 0 0 0 0 0`
	`0.000000 0 0 1500 0 0 0 0 0 0 0 0 0`
	`ME`
`GET_ALL_PIECES`	
	`OK`
	`10`
	`1 4 4 0 0 0 0 1 2 0 1 4 1 0 5 5 2 2 724 738 1`
	`2 4 1 1 1 1 2 1 2 1 1 2 2 1 4 4 4 3 1141 1203 4`
	`3 4 1 0 1 1 1 1 1 2 1 0 2 1 5 5 5 5 1131 1193 2`
	`4 4 0 2 0 1 2 0 1 3 0 1 3 1 1 5 5 1 926 926 1`
	`5 4 0 0 1 1 0 1 1 1 1 0 1 1 1 1 2 2 1444 1462 4`
	`6 4 0 2 0 1 2 0 1 3 0 1 3 1 3 1 1 3 1170 1185 4`
	`7 4 4 0 0 5 0 0 0 0 1 5 1 0 2 2 2 5 820 845 2`
	`8 4 4 0 0 0 0 1 2 0 1 4 1 0 1 1 1 1 875 898 5`
	`9 4 0 2 0 1 2 0 1 3 0 1 3 1 1 4 1 4 605 623 2`
	`10 4 0 0 1 1 0 1 1 1 1 0 -1 1 1 1 1 1 1159 1195 5`
`GET_AVAILABLE_PIECES`	
	`OK`
	`10 1 2 3 4 5 6 7 8 9 10`
`GET_PIECES_IN_RANGE`	
	`OK`
	`3 8 9 10`
`BUY_PIECE 3`	
	`OK`
`PLACE_OWN_PIECE 5 4 6 1 2 0`	
	`OK`
`PLACE_SHARED_PIECE 1 2 0 1 2`	
	`OK`
`SET_MULTIPLIER 1 3`	
	`OK`
`SET_MULTIPLIER 2 3`	
	`FAILED 118 you already picked a different`
	`↪ color for this multiplier`
`SET_MULTIPLIER 2 5`	
	`OK`

client → server	server → client
`WAIT`	
	`OK`
	`WAITING 0.765456`
	`OK`
`GET_MATCH_INFO`	
	`OK`
	`0.000000 0 0 1500 0 0 0 0 0 0 0 0 0`
	`100.000000 1515 25 1416 1075 5 3 5 0 0 0 1 1`
	`OPPONENT`
`WAIT`	
	`OK`
	`WAITING 0.963456`
	`OK`
`GET_MATCH_INFO`	
	`OK`
	`100.000000 1515 25 1416 1075 5 3 5 0 0 0 1 1`
	`84.025974 1294 0 1418 918 6 0 0 0 0 0 0 0`
	`ME`
`PLACE_SINGLE_CUBE_PIECE 1 1 6`	
	`OK`
`PLACE_SINGLE_SQUARE_PIECE 4 4 2`	
	`OK`
`WAIT`	
	`OK`
	`WAITING 0.344444`
	`OK`
`GET_MATCH_INFO`	
	`OK`
	`83.754045 1294 0 1418 918 6 0 0 0 0 0 0 0`
	`100.000000 1515 30 1421 1075 7 3 5 0 0 0 1 1`
	`OPPONENT`

client → server	server → client
`GET_MY_SPACE`	
	`OK`
	`10`
	`...[omitted 50 LINES]`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 1 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 1 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 1 1 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 1 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`...[omitted 20 LINES]`
`GET_SHARED_SPACE`	
	`OK`
	`10`
	`0 0 0 0 0 0 0 0 0 0`
	`1 0 0 0 0 0 0 0 0 0`
	`1 0 0 0 0 0 0 0 0 0`
	`1 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 2 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0`

6. A-grid-culture

6.1. Introduction

Beetlejumpers, aside from notable military might and significant – if a bit militarized – culture, boast about a stable and strong economy. Part of this economy lies in their agriculture. Of course, as most things in their proud society, beetlejumper agriculture is well-designed and based on merciless competition.

A great example of such organization is the agricultural world of Tuberus III, once a chaotic world mixing industry, animal herding, and numerous different areas. Now, after being conquered by beetlejumpers, it is a peaceful and orderly planet where some small companies grow potatoes – a vegetable of vital importance to beetlejumper war effort, used – in various stages of processing – for nourishment of soldiers and fuel for the machines. Companies tasked with growing of the potatoes constantly compete for limited area and harvest the precious plant. All rules regarding land are stated and executed by General Oversight Panel for Agriculture.

Tuberus III suffered a bit during the conquest and has been reconstructed by well-meaning engineering team with a horribly short deadline. Therefore, its geometry and surface are slightly different from what one would expect from a planet. Operators are advised to adapt.

6.2. Problem

The team leads one of the companies controlling agriculture on surface of Tuberus III. Your task is to acquire as much area as possible and make use of the land before competition snatches it away from you. Unfortunately, due to environmental constraints, you are given very few workers and no means of transportation, making the acquisition of territory rather tedious.

6.3. Game model

The game takes place on 2D board in turns of the same length. All teams play on the same board. Each turn, any team can give orders to their worker bots which can move around and place markers. These actions are made to seize area and then to order harvests which convert occupied area into produce. The one who accumulates the most produce at the end of the season, wins.

6.3.1. Game space

Game space consists of the entire planet surface. As rules of the Oversight Panel prohibit underground agriculture, the game area is 2-D. As mentioned before, the reconstruction of the planet went poorly due to time and budget constraints, resulting in a torus surface. The entire torus is represented as a grid of identical squares; those squares are fields where the potatoes grow. The fields are separated by Fielded Energy Node-Controlled Embankments – *FENCE* for short (these are sides of the squares). All FENCEs are controlled by Power Organization System Terminals – *POST*s for short – that are placed in equidistant manner on the entire planet surface (in squares vertices).

As the surface of the planet was reconstructed to resemble the original appearance, described by scouting forces as “somewhat hilly”, engineers put some hills on the surface. Those have proven to be problematic as locations for POSTs, and therefore contain none. Depending on the currently used worker

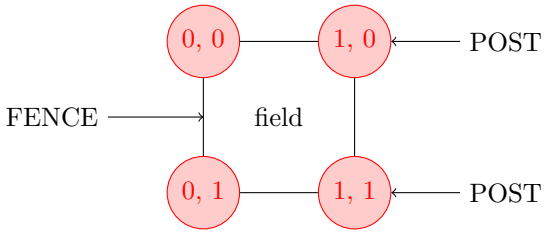

Figure 5: Basic field appearance

bot model, the hills might be an obstacle to worker bot movements (some vertices might be unavailable points for a given type of bots).

6.3.2. MaRKeRs, POSTs, FENCEs, and territory

Every POST can be controlled by any worker via use of Mathematically Refined Key Resources (*MaRKeR* for short). Each team has a unique **color** of MaRKeRs.

A POST with team's MaRKeR on it is controlled by the team. If two POSTs connected by a FENCE are both controlled by the same team (have the same color), so is the FENCE.

Entire area enclosed by FENCEs in one color is considered controlled by the team (with the exception of enclosures of other teams inside of that territory). Each largest connected area that can be surrounded by one color FENCEs (a set of FENCEs that constitute a simple closed polygonal chain) is called Big Limited Operation Breadth (*BLOB* for short). Harvesting may only be performed on a BLOB.

Depending on the current policies, the MaRKeRs either have very high lifespan (in essence, longer than operation time) or they expire quickly. Expiration time is known and remaining lifespan is provided with MaRKeR data.

Due to Panel's policy of transparency, all MaRKeRs placed on the board are public. The positions of all worker bots are also known by all teams.

6.3.3. Worker bots and their actions

Each team controls the same, constant number of worker bots. Worker bots move along the fence lines and access POSTs. Every worker bot is capable of taking control of a POST, but due to secrecy rules enforced by Beetlejumper Standard Norm 6979, they can do it only when they are alone at that location – not even a worker bot of the same team is allowed to be a witness of that operation.

This does not stop working bots from occupying the same point (still worker bots with company are forced to inaction). As the worker bots lack communication tools and weapons, they do not interact with each other. There is no way to influence another team's worker bot except by being present at the same POST.

Each worker bot has a storage of specific size which might be used to keep MaRKeRs of other teams. It can also create unlimited number of MaRKeRs in color of the team it belongs to.

Worker bot can only move from a POST to a neighboring POST along a connecting FENCE, and only once a turn. It may be also able to move from a POST to a nearby hill, but this is reserved only for special worker bot type – *mountain* model.

Worker bot actions Once a worker bot enters a POST, it can:

1. Create and place a new MaRKeR with team color.

   This action takes control over the POST. It requires no additional resources, except turn of time.

2. Place an opponent's MaRKeR from its storage.

   This grants the opponent control of the POST. It requires the worker bot to have opponent's MaRKeR collected before from another POST. The action removes the MaRKeR from storage, making storage space available again.

3. Delete own color MaRKeR present at the location.

Worker bots are not allowed to destroy any MaRKeR of other teams.

4. Ignore the POST, performing no action.

If the POST is already controlled by a team, the first action may also have more implications depending on who is in control:

- if the MaRKeR (and, thus, the POST) belongs to another team, worker bot must download and store it in its own storage first in order to abide the laws set by Oversight Panel,
- if the MaRKeR belongs to worker bot's team, it will be automatically deleted.

Please note that MaRKeR storage is limited. If the storage limit is reached, worker bot is no longer able to download and store other team MaRKeRs; this stops the worker bot from replacing opponent's markers with their own. To empty the storage, worker bot must place opponent's MaRKeRs on board – there is no other way to release storage space.

MaRKeRs in storage are not subject to expiration; once a MaRKeR from storage placed on board, its lifespan is equal to any freshly created MaRKeR.

6.3.4. Harvesting

Once a BLOB is secured, planting and harvesting may commence. Due to great advancements in agricultural technology the process is fast and flawless. To initiate the process, an order must be given. When this happens, the potatoes are planted and watered. Then time acceleration is applied, and harvesting bots arrive in order to retrieve grown potatoes. Amount of harvested produce is noted in the Oversight Panel's report and score is awarded to the team.

As there are different variants of potatoes, some of them may give larger harvest when special conditions are met (like shape of BLOB, different MaRKeRs inside of a BLOB, etc.). This is represented as harvest bonuses (see 6.6. for details).

As soon as the crops are calculated and noted, all MaRKeRs that delineated the recently harvested BLOB are removed. Depending on the current policy, the MaRKeRs that were inside the BLOB are either left as they were or removed as well – sometimes even with compensation to their owners.

There is a notable limitation – due to Oversight Panel policy against monopolization, no dimension of the harvest area can be larger than half of the planet size. The attempt to harvest will fail, if it is based on adjoint FENCEs with vertical or horizontal spread larger than $\left\lfloor \frac{size-1}{2} \right\rfloor$, where $size$ is a vertical or horizontal size of the map respectively.

Determining area in details Harvesting follows a certain, formalized procedure:

1. Team chooses a starting point – a POST with a MaRKeR color belonging to the harvesting team.
2. All possible MaRKeRs that can be reached by moving along controlled FENCEs are determined.

   The FENCEs create a connected set of segments. As the surface of a planet is a torus, the outermost fields of the map from both sides are considered to be neighboring fields. If vertical or horizontal spread of the created set of segments exceeds half of the planet size, the process is terminated and harvest is aborted.
3. All biggest BLOBs delineated by FENCEs and POSTs determined in previous step are taken into account and harvested.

   BLOBs that are connected by a single line of FENCE are harvested and scored separately – for the BLOB to be treated as one, it must consist of adjoining fields. If a BLOB contains another BLOB of the same team (inside), only the larger matters.
4. All MaRKeRs that directly delineated the harvested BLOBs are removed.

   Depending on current policy (see value of H in DESCRIBE_WORLD command), the MaRKeRs that were inside of the BLOBs are either left untouched, or they are removed. MaRKeRs that were not part of the immediate BLOB border are left on the board.

The examples present only MaRKeRs and do not involve bonuses – they are all assumed to be 0. Please also note that examples show only part of the game area and are not subject to size limitations.

Example of simple harvest This harvest is performed on field (4, 1). This BLOB contains 4 fields.

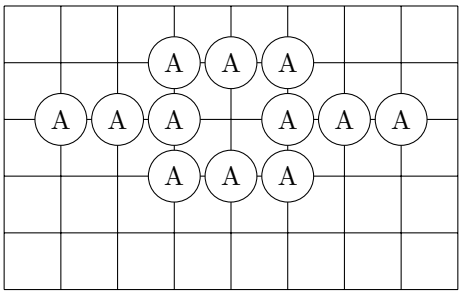

Figure 6: Example of simple harvest – before

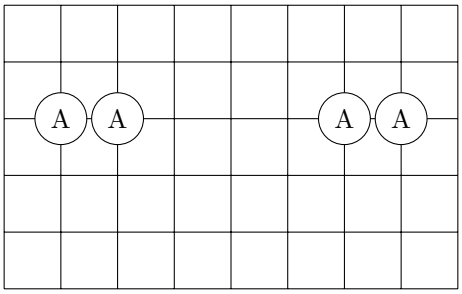

Figure 7: Example of simple harvest – after

Example of internal BLOBs This example contains several variants, but all start from the map depicted on fig. 8.

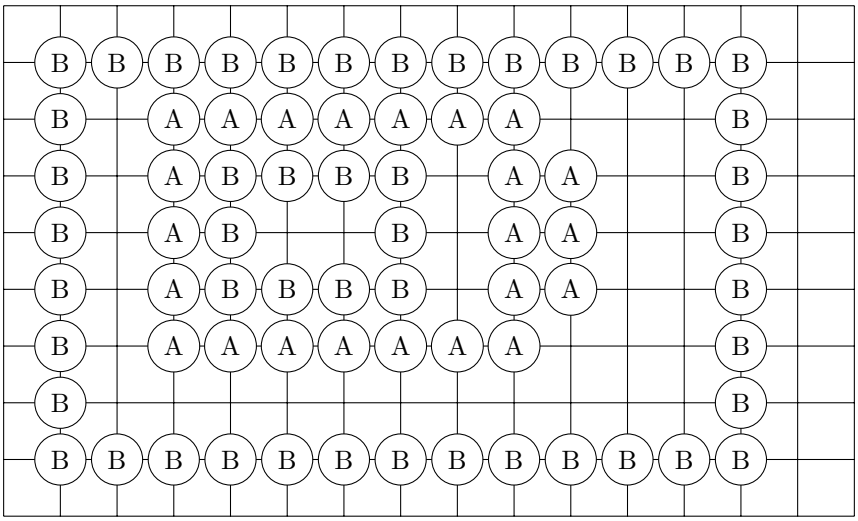

Figure 8: Example of internal blobs – before any harvest

Example of harvesting an encompassing BLOB Assuming the harvest would take place on (1, 1) with H equal to 0, it would be worth 58 fields for player of B color and board would appear like depicted on fig. 9. In case when H equals 1, the area score would remain the same, but the example board would be left empty.

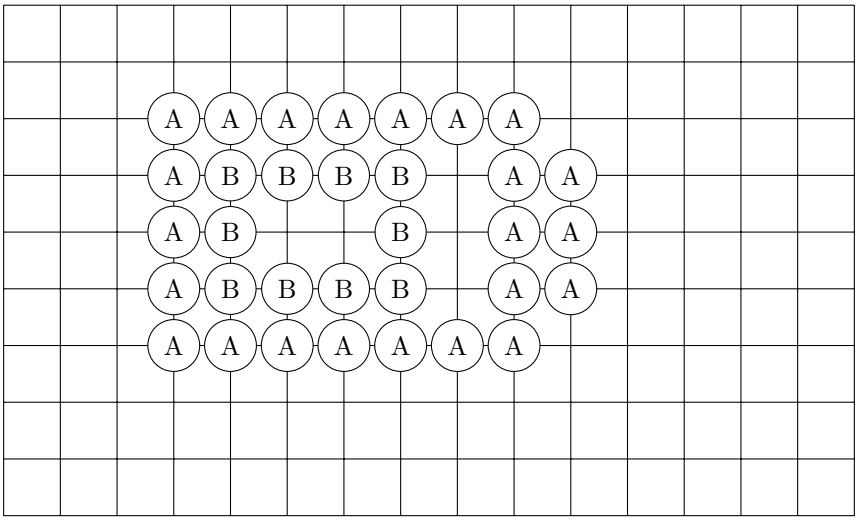

Figure 9: Example of internal blobs – after harvesting on (1, 1)

Example of harvesting an internal BLOB Assuming the harvest would take place on (4, 2) with H equal to 0, it would be worth 20 fields for player of A color and board would appear like depicted on fig. 10.

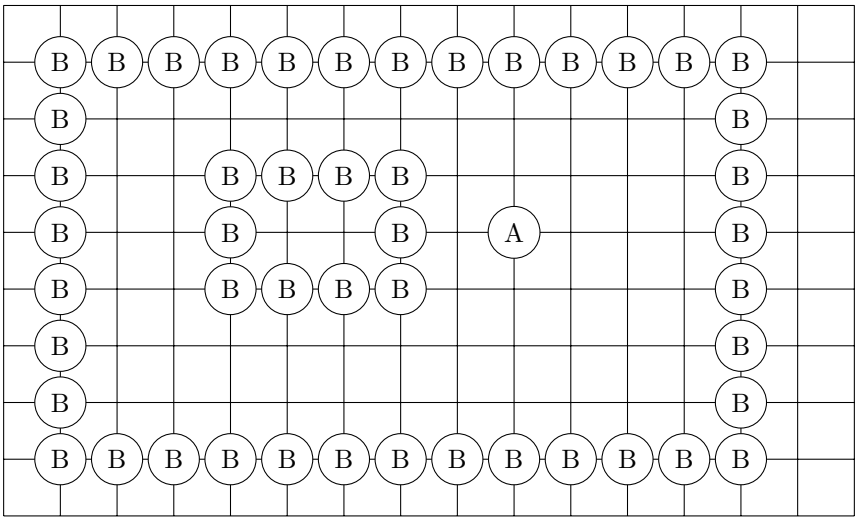

Figure 10: Example of internal blobs – harvest at (4, 2) with H of 0

Example of harvesting an internal BLOB Assuming the harvest would take place on (4, 2) with H equal to 1, it would be worth 20 fields for player of A color and board would appear like depicted on fig. 11.

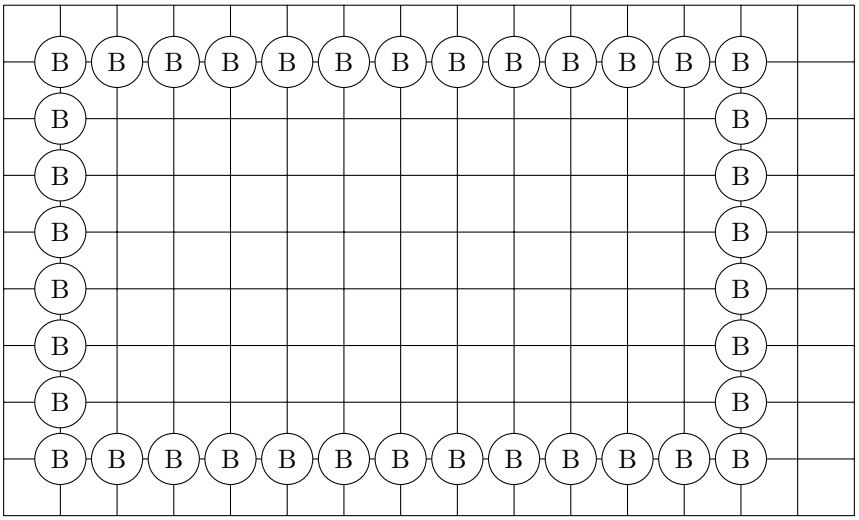

Figure 11: Example of internal blobs – harvest at (4, 2) with H of 1

Example of harvesting a deeply nested BLOB Assuming the harvest would take place on (6, 5), it would be worth 6 fields for player with B color and board would appear like depicted on fig. 12. Value of H has no impact in this case, as there are no further internal BLOBs in the nested BLOB.

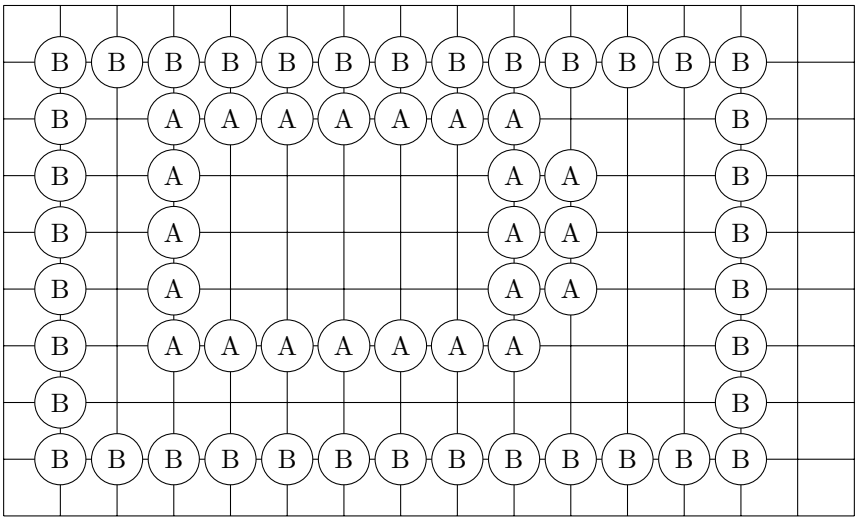

Figure 12: Example of internal blobs – harvest at (6, 5)

Example of harvesting bridged BLOBs Assuming the situation as on fig. 13., with the harvest taking place on (7, 2), it would be worth 8 fields in the first BLOB and 8 fields in the second BLOB (scored separatedly). Afterwards, the board would appear like depicted on fig. 14. Value of H has no impact in this case.

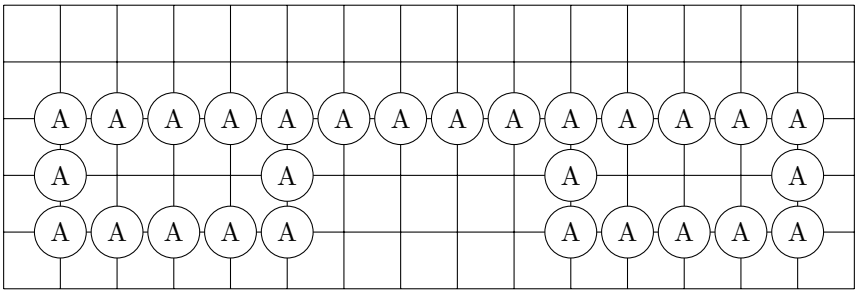

Figure 13: Example of bridged blobs – before harvest at (7, 2)

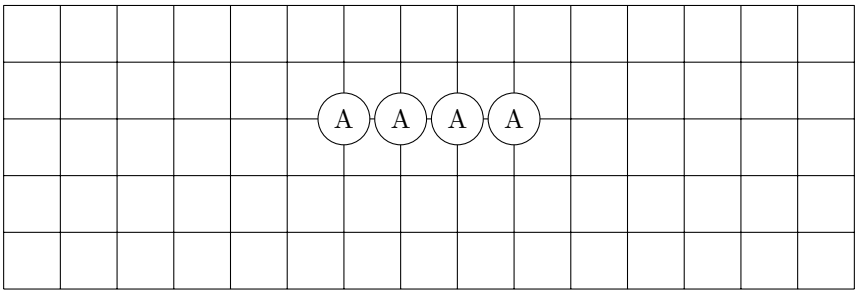

Figure 14: Example of bridged blobs – after harvest at (7, 2)

6.4. Auxiliary tools

Due to high complexity of territory determination process, Oversight Panel has provided a tool to simulate the harvests. Any team can provide a map of their own design, with MaRKeRs set on POSTs, and receive data about its value.

Due to the fact that calculations put certain strain on Panel's computers, otherwise used to monitor activity across the planet, access to this tool is limited in both space – it allows to perform simulation on 21×21 map only – and frequency – it can be called once per turn only. Invalid calls count to that limit – after all, Oversight Panel's resources are used to notify about the error.

6.5. Start and the end

Initial state of the game All worker bots are placed on board. There are no MaRKeRs on the board, thus no territory is taken.

End of the game The game ends after a known, predetermined amount of turn has passed – after all, it is known when harvesting season ends. Additionally, as Oversight Panel prefers active, dynamic operations, a prolonged period without any MaRKeRs being placed on board will result in premature closing of the operation. A countdown to such moment is provided along with information of the end of the harvest season.

6.6. Scoring

The process of harvesting, described in 6.3.4., yields revenue. As the larger areas are more efficient to harvest, larger BLOBs are worth more.

6.6.1. Bonuses

Oversight Panel has designed several incentives in order to increase crops and promote more organized approach to agriculture. The specific bonuses are:

Table 2: Harvest bonuses

Symbol	Given per...	Description
S_w	Harvest	At least two BLOBs with identical area were harvested at once
S_h	Harvest	At least three BLOBs with identical area were harvested at once
S_s	BLOB	The BLOB area is a square
Z	BLOB	More opponents' MaRKeRs than own ones inside the BLOB
W	BLOB	Like Z but MaRKeRs belong to at least two different opponents
T	BLOB	Inside of the BLOB is fully covered with MaRKeRs (no matter who they belong to)
N	Each MaRKeR	Each opponent MaRKeR inside the BLOB is given extra score
M	Each MaRKeR	Like N but the bonus goes to MaRKeR owner (opponent)

6.6.2. Detailed calculations

A value of particular BLOB is calculated via:

$$S_n = Q^D \cdot (1 + Z + W + T + S_s) \tag{10}$$

where Q is the count of team-controlled fields inside the BLOB (it excludes internal BLOBs controlled by competing teams), and D is a known, set for the current game, value. Z, W, T, and S_s represent bonus coefficients.

Total score for entire harvest is calculated via:

$$S_t = (\sum S_n) \cdot (1 + S_w + S_h) + (n \cdot N) \tag{11}$$

S_n represent scores for particular blobs, n is the total number of MaRKeRs inside the BLOBs (not counting the delineating MaRKeRs) that do not belong to the team. N, S_w and S_h represent bonus coefficients.

Opponent compensation Additionally, any opponent who did have MaRKeRs inside the harvested field receives following number of points:

$$S_e = n_e \cdot M \tag{12}$$

n_e is the total count of MaRKeRs in given team color inside the field, M represents bonus coefficient.

6.7. Commands

General guidelines on the communication protocol (connecting, logging in, commands, response format) are described in section *Server communication*. Below you can find commands for the problem A-grid-culture.

DESCRIBE_WORLD Returns parameters of the world and the value of the score scaling coefficient.

Parameters: none

Data (from server):

In the first line the server returns five values separated with a single spaces:

- A ($A \in \mathbb{N}$, $1 \leqslant A \leqslant 500$) — map size (both width and height) in POSTs,
- B ($B \in \{0, 1\}$) — if 0, worker bots cannot move to hills; if 1, they can move to hills, but still cannot place MaRKeRs on them,
- C ($C \in \{$`0-9A-Za-z`$\}$)) — representation of own team's color,
- D ($D \in \mathbb{R}$, $0.5 \leqslant D \leqslant 10.0$) — score coefficient,
- E ($E \in \mathbb{N}$, $1 \leqslant E \leqslant 50$) — amount of time between two allowed SHOW_MAP calls,
- F ($F \in \mathbb{N}$, $0 \leqslant F \leqslant 5000$) — lifespan of a MaRKeR,
- G ($G \in \mathbb{N}$, $0 \leqslant G \leqslant 5000$) — worker bot MaRKeR storage capacity,
- H ($H \in \{0, 1\}$) — if 0, SCORE command does not remove anything inside of harvested area; if 1, harvested area is purged clean,
- Z ($Z \in \mathbb{R}$, $0.0 \leqslant Z \leqslant 4.0$) — bonus for having more enemy MaRKeRs than own inside scored area,
- W ($W \in \mathbb{R}$, $0.0 \leqslant W \leqslant 4.0$) — bonus for having more enemy MaRKeRs than own inside scored area if the MaRKeRs belong to more than one opponent,
- T ($T \in \mathbb{R}$, $0.0 \leqslant T \leqslant 4.0$) — bonus for all POSTs inside scored area having MaRKeRs of any kind,
- S_w ($S_w \in \mathbb{R}$, $0.0 \leqslant S_w \leqslant 4.0$) — bonus for simultaneously scoring two or more territories with identical (considering own area only) surface area,
- S_h ($S_h \in \mathbb{R}$, $0.0 \leqslant S_h \leqslant 4.0$) — bonus for simultaneously scoring three or more territories with identical (considering own area only) surface area,
- S_s ($S_s \in \mathbb{R}$, $0 \leqslant S_s \leqslant 4.0$) — bonus for scoring territory in shape of a square,
- N ($N \in \mathbb{R}$, $0 \leqslant N \leqslant 30$) — bonus for each enemy MaRKeR inside the territory; it is given to the harvesting team,
- M ($M \in \mathbb{R}$, $0 \leqslant M \leqslant 30$) — bonus for each enemy MaRKeR inside the territory; it is given to the MaRKeR owner,
- I ($T \in \mathbb{N}$, $1 \leqslant I \leqslant 3$) — duration of a single turn in seconds,
- L ($L \in \mathbb{N}$, $1 \leqslant L \leqslant 100$) — the maximum number of commands which can be issued in one turn,
- K ($K \in \mathbb{R}$, $1 \leqslant K \leqslant 8$) — value of the scaling coefficient.

TIME_TO_END Provides number of turns to the end of the current game. 0 is displayed in the last turn.

Parameters: none

Data (from server):

A single line with single value:

- U ($U \in \mathbb{N}$) — turns left to the end of the current game due to predetermined game time,
- V ($V \in \mathbb{N}$) — turns left to the end of the current game due to lack of correct MaRKeR placements; resets on every correct PUT.

LIST_MY_WORKERS Lists all workers belonging to own team.

Parameters: none

Data (from server): First line of the response contains a single number:

- N_w ($N_w \in \mathbb{N}$, $1 \leqslant N_w \leqslant 10$) — number of workers,

Next N_w lines contain descriptions of specific workers, in form of values separated by single spaces:

- ID ($ID \in \mathbb{N}$, $1 \leqslant ID \leqslant 10000$) — worker ID,
- X ($X \in \mathbb{N}$, $0 \leqslant X < A$) — Position on the X axis,
- Y ($Y \in \mathbb{N}$, $0 \leqslant Y < A$) — Position on the Y axis,
- G_c ($G_c \in \mathbb{N}$) — number of MaRKeR colors that are in this worker bot's storage,
- G_c pairs of C_m, D_m ($D_m \in \mathbb{N}, C_m \in \{$`0-9A-Za-z`$\}$) — MaRKeRs in given color in worker bots' storage.

LIST_ENEMY_WORKERS Lists all workers belonging to other teams.

Parameters: none

Data (from server): First line of the response contains a single number:

- N_w ($N_w \in \mathbb{N}$, $1 \leqslant N_w \leqslant 500$) — number of workers,

Next N_w lines contain descriptions of specific workers, in form of values separated by single spaces:

- C ($C \in \{$`0-9A-Za-z`$\}$) — worker's team color,
- X ($X \in \mathbb{N}$, $0 \leqslant X < A$) — Position on the X axis,
- Y ($Y \in \mathbb{N}$, $0 \leqslant Y < A$) — Position on the Y axis.

SHOW_MAP Presents entire game map, including MaRKeRs (but without worker bots). Can be called once every E turns and returns state from the beginning of the current turn.

Parameters: none

Data (from server): First line of the response contains a single value:

- A ($A \in \mathbb{N}$, $1 \leqslant A \leqslant 500$) — map size in POSTs,

Next A lines contain A characters each, without separation. Every character represents a single POST. First line represents all POSTs with Y-axis coordinate of 0, second line represents all POSTs with Y-axis coordinate of 1, etc. A character can be:

- `.` — empty field,
- `#` — inaccessible field (a hill),
- `0-9A-Za-z` — field with a marker; specific symbol denotes a color.

Next A lines contain A values each, separated by single spaces. A value can be:

- `-` — there is no marker on the field,
- $V \in \mathbb{Z}$, $V \geqslant 0$ — amount of turns the MaRKeR will remain on the POST. 0 indicates the MaRKeR will be gone at the end of this turn.

SHOW_HISTORY Shows history of MaRKeR placements and harvests in the previous turn.

Parameters: none

Data (from server):

First line of the response contains information about MaRKeRs destroyed:

- M_d ($M_d \in \mathbb{N}$) — number of markers destroyed in the previous turn,

- M_c pairs of X, Y ($X, Y \in \mathbb{N}$, $0 \leqslant X < A$, $0 \leqslant Y < A$) — respectively, X-axis position, and Y-axis position of every marker destroyed last turn.

Second line of the response contains information about MaRKeRs placed:

- M_c ($M_c \in \mathbb{N}$) — number of markers placed in the previous turn,
- M_c triplets of X, Y, C ($C \in \{$`0-9A-Za-z`$\}; X, Y \in \mathbb{N}$, $0 \leqslant X < A$, $0 \leqslant Y < A$), describing, respectively, color, X-axis position, and Y-axis position of every marker placed last turn.

Third line of the response contains information about harvests:

- S_c ($S_c \in \mathbb{N}$) — number of SCORE commands successfully executed in the previous turn,
- S_c pairs of X, Y ($0 \leqslant X < A$, $0 \leqslant Y < A$) — respectively, X-axis position and Y-axis position of every SCORE command given last turn.

LAST_SCORE Provides summary of harvests that occurred last turn. Using this command in the first turn will provide information from previous game, if applicable.

Parameters: none

Data (from server):

Server returns a single line:

- S_{last} ($S_{last} \in \mathbb{N}$, $S_{last} \geqslant 0$) — number of teams who scored any point in the turn in question,
- S_{last} pairs of C, S ($C \in \mathbb{N}$, $C \in \{$`0-9A-Za-z`$\}$, $S \in \mathbb{R}$), where C is the color of the team, and S is the score they gained. If a given color is absent from this list, it means the score for this color is zero.

TEST_SCORE Simulates harvest on provided map. This command can be called once per turn.

Parameters: All four parameters must be provided in one line, separated by single spaces:

- S_{max} ($5 \leqslant S_{max} \leqslant 21$) — size of the test board,
- S_x ($0 \leqslant S_x < S_{max}$) — X-axis position for SCORE command,
- S_y ($0 \leqslant S_y < S_{max}$) — Y-axis position for SCORE command.
- $S_{max} \times S_{max}$ symbols of S_m ($S_m \in \{$`.#0-9A-Za-z`$\}$), representing POSTs; meaning of symbols is identical to one used in `SHOW_MAP`; first S_{max} symbols represent POSTs with S_y of 0, next S_{max} symbols represent POSTs with S_y of 1, etc. Those symbols must not be separated from each other.

Data (from server):
Server response to this command is identical to the response provided by LAST_SCORE command if the parameters are correct – a line containing `OK` and then a single line identical to one provided by LAST_SCORE, but preceded with another `OK`. In case they contain errors, server will answer with a line containing `ERR` and an error message.

MOVE Moves the worker bot. This command is executed at the end of the turn, after DESTROY, PUT and SCORE commands.

Parameters:

- ID ($ID \in \mathbb{N}$, $1 \leqslant ID \leqslant 10000$) — worker ID,
- M_x ($M_x \in \mathbb{Z}$, $-1 \leqslant M_x \leqslant 1$) — Movement along X axis,
- M_y ($M_y \in \mathbb{Z}$, $-1 \leqslant M_y \leqslant 1$) — Movement along Y axis.

DESTROY Destroys a marker on worker bot position. This command is executed at the end of the turn, before PUT, SCORE, and MOVE commands.

Parameters:

- ID ($ID \in \mathbb{N}$, $1 \leqslant ID \leqslant 10000$) — worker ID,

PUT Puts a marker of choice. This command is executed at the end of the turn, after DESTROY commands and before SCORE and MOVE commands.

Parameters:

- ID ($ID \in \mathbb{N}$, $1 \leqslant ID \leqslant 10000$) — worker ID,
- M_c ($M_c \in \{$`0-9A-Za-z`$\}$) — color of the dropped MaRKeR; please note that if this is different from C, worker bot must have appropriate color in its storage or the command will fail.

SCORE Harvests and scores an area. This command is executed at the end of the turn, after DESTROY and PUT, but before MOVE commands.

Parameters:

- X, ($X \in \mathbb{N}$, $0 \leqslant X < A$) — X-axis position of one of the POSTs delineating scored area,
- Y, ($Y \in \mathbb{N}$, $0 \leqslant Y < A$) — Y-axis position of one of the POSTs delineating scored area.

WAIT Waits till the next turn begins.

Parameters: none

Data (from server):

The server returns a single line of characters:

- WAITING S

where S ($S \in \mathbb{R}, S \geqslant 0$) stands for the number of seconds left to wait. Once the period is over, the server sends an additional line:

- OK

6.8. Errors

In case the command is incorrect, the server responds according to the description in section *Server communication* with the following message:

- '`FAILED` *e msg*',

where *e* is an error code, and *msg* — an error message. The table below consists of errors that may occur during communication process in problem A-grid-culture.

error code	error message
1	bad login or password
2	unknown command
3	bad format
4	too many arguments
5	internal error, sorry...
6	commands limit reached, forced waiting activated
100	show map command limit reached
101	invalid worker id
102	worker already has move scheduled
103	invalid move offset
104	cannot move here
105	worker already has put scheduled
106	cannot put here
107	invalid color
108	cannot put in crowd
109	worker does not carry specified marker
110	worker already has destroy scheduled
111	cannot destroy here
112	cannot destroy in crowd
113	invalid position
114	test score command limit reached
115	invalid test map size
116	invalid test map description
117	no marker at specified position
118	too large territory

6.9. Servers

The games will be held on multiple servers.

Table 3: Server addresses and parameters.

Name	**Address:Port**	**Model of worker bot**	**Score policy**	**Long-term MaRKeRs**	**Storage space**
Grid 1	`universum.dl24:20003`	Normal	Destroy internal	Yes	Average
Grid 2	`universum.dl24:20004`	Mountain	Keep internal	No	Average
Grid 3	`universum.dl24:20005`	Normal	Keep internal	Yes	Small

6.10. Example

Below you will find an exemplary record of the communication with the server.

client → server	server → client
	LOGIN
login1	
	PASS
secret	
	OK
DESCRIBE_WORLD	
	OK
	5 1 g 2.000000 30 50 10 0 0.200000 0.200000 0.000000 0.000000
	↪ 1.000000 1.000000 0.030000 0.000000 1 20 1.039187
TIME_TO_END	
	OK
	3594 895
LIST_MY_WORKERS	
	OK
	4
	1 1 1 0
	2 1 2 0
	3 2 2 0
	4 2 0 0
MOVE 4 0 1	
	OK
WAIT	
	OK
	WAITING 0.309000
	OK
LIST_MY_WORKERS	
	OK
	4
	1 1 1 0
	2 1 2 0
	3 2 2 0
	4 2 1 0
PUT 1 g	
	OK
MOVE 1 1 0	
	OK
PUT 2 g	
	OK
PUT 3 g	
	OK
PUT 4 g	
	OK
WAIT	
	OK
	WAITING 0.309000
	OK

```
SHOW_HISTORY
                                        OK
                                        0
                                        4 1 1 g 1 2 g 2 2 g 2 1 g
                                        0
PUT 1 g
                                        FAILED 108 cannot put in crowd
SCORE 1 1
                                        OK
WAIT
                                        OK
                                        WAITING 0.309000
                                        OK
SHOW_HISTORY
                                        OK
                                        0
                                        0
                                        1 1 1
LAST_SCORE
                                        OK
                                        1 g 2.000000
PUT 2 g
                                        OK
WAIT
                                        OK
                                        WAITING 0.143000
                                        OK
SHOW_MAP
                                        OK
                                        5
                                        #....
                                        .....
                                        .g...
                                        .....
                                        .....
                                        - - - - -
                                        - - - - -
                                        - 49 - - -
                                        - - - - -
                                        - - - - -
TEST_SCORE 5 1 1 #.....xx...xx........abc.
                                        OK
                                        OK 1 x 2.0
TEST_SCORE 5 0 0 #.....xx...xx........abc.
                                        OK
                                        ERR 117 no marker at specified position
```

7. Rocket Science

7.1. Introduction

Contrary to widespread belief, beetlejumpers were not always space-faring species. In times immemorial, they were confined to a single planet – their homeland. Of course, being aware of threats associated with such a situation – and dwindling resources of their planet – they chose to leave. First few state-sponsored attempts failed horribly, causing untold destruction. The government chose then to privatize the space sector and make the entire venture an exercise in competition. And this is where you come in. Gather money for your venture, start investment, attract shareholders, avoid lawyers, build the first rocket, and launch it for the glory of the future beetlejumper generations!

7.2. Problem

Your team's mission is to collect money and launch a rocket that will carry one of the first spaceships ever created by beetlejumpers. You have a base in one of the cities, few cars which can help you arrange monetary means, an idea how to earn them, and a grand vision of reaching space.

If you succeed in building a rocket not only your glory will be remembered, but also your wallet will expand enormously. The government will reward several trailblazing projects. Successful rocket launching will multiply all your investment expenses and make them return to your account.

7.3. Game model

The game takes place in the form of turns of the same length. During every turn teams communicate with the server and give commands to control their cars and investments.

Game map The game map is represented by a connected, weighted graph. Cities are located in nodes, and the roads are represented as edges. Edge's weight is the **base cost** of traversing the edge (measured in fuel units). As this is the beetlejumper's homeland, it has been thoroughly mapped – the maps are provided ahead of time, as files available for download. Details regarding the data provided by beetlejumper cartographers are available in 7.9.

7.4. Earning money

Traveling around the game map – from one city to another – will make beetlejumpers gain experience and knowledge. These two will directly translate to your cash profit.

Cars earn money by completing cycles – the longer the cycle, the more money is earned by completing it. Each cycle must begin and end at the team's base. A cycle that contains only unique roads produces more revenue than a cycle in which some roads were re-used – after all, new roads bring new opportunities.

7.4.1. Fuel

Cars use up fuel which limits their range. Not all roads are equal – some are harder to traverse and thus require more fuel. If a car has insufficient fuel to traverse the specific road, it must be pushed through the entire length of the road, greatly prolonging the journey. A pushed car uses no fuel, so on easier roads it may resume traveling under its own power (with engine working and using remaining fuel).

7.4.2. Travel time

Traveling along a single road takes specific value of time which is independent from fuel usage. Default travel time for all roads (using a car engine) is **2** turns. This value might be decreased. If the car has an upgraded engine (see 7.4.5.), then going from one city to the neighboring one takes only **1** turn.

If the driver is forced to push the car instead of driving it, the road takes **10** turns for the first 5 pushes. Each next push takes **180** turns, due to driver exhaustion. Whenever the car comes back to the home city, the driver is getting rest and is able to make new "quick" pushes.

7.4.3. Road degeneration and adjustment

Additionally, deregulation of transport standards recently introduced by Beetlejumper Commissariat of Wheeled Transport caused certain chaos in automotive industry. In result of this, every team has different wheel layout, resulting in different patterns of road damage.

The most obvious consequence of this situation is simple – every time team's car drives on a given road, the road becomes easier and cheaper for them to traverse in the future, but harder and more expensive for their opponents. This is represented by two values for each road: road's global **handicap** (H_r – increased each time any car passed the road) and your team **road bonus** (B_r^t – increased when your car goes through). These values does not change the travel time for a road, but do change the amount of fuel used for traveling – after all, greater effort is required.

The effective fuel units used for taking a specific road (F_r) are calculated as follows:

$$F_r = max(3, C_r + H_r - B_r^t), \tag{13}$$

where C_r is the base cost of driving through the road (given together with the game map).

7.4.4. Revenue

Some cities around the given world have greater importance then other ones. This also influences the money might be earned while traveling. Once the car returns to the base city, several factors are taken into account:

- number of roads traversed, represented as n,
- whether any of the roads were traversed more than once,
- greatest city importance from all cities visited in given cycle, represented as M_{max},
- constant value of profit from traversed road, represented as R.

If all roads in the cycle were unique, i.e. no road was traversed more than once (cities can be visited more than once) final revenue (S) from the cycle is calculated as:

$$S = n \cdot R \cdot 1.04^n \cdot M_{max} \tag{14}$$

Otherwise, final revenue is calculated as:

$$S = n \cdot R \tag{15}$$

S is then rounded up to nearest integer and value is added as cash to your team account.

7.4.5. Cars

Every team has several cars that they can put to use immediately from the begging of game. Each car might be used for two purposes: visiting cities to earn profits and – at the same time – transporting previously earned money from your home (base city) to your investment – rocket launch site.

There is no way to increase the number of available vehicles. But there is a way to upgrade them. Each car can be fitted with two upgrades – a better engine, and a secured trunk.

A **powerful engine** halves the travel time, assuming the car travels under its own power (a good engine means nothing if it is not operational).

Local regulations will allow transportation only of a limited amount of cash money. This value might be increased if the car was equipped with a **secured trunk**.

None of the modifications affect revenue from the cycle. Any car can have the upgrades, they are purchased for every car separately and they do not interfere – you may have a car with both a powerful engine and a better trunk. An upgrade can be made only when the car stops in the home city.

Cars are automatically refueled in the home city, assuming the team can afford it.

7.5. Investments – building a rocket

The ultimate goal of your team is to build a rocket and launch it, ushering a new era of the beetlejumper civilization. To achieve this you must set aside a part of your revenue as the funds for building a rocket.

7.5.1. Starting an investment

To start building a rocket, the team must declare one of the cities as a launch site. This cannot be a city that serves as the home base – theirs or anyone else's. A failure of launching from home could lead to a catastrophic loss of beetlejumpers' base resources which is unacceptable. Due to similar safety considerations and need for diversification, a launch site may not be set up in a city where a launch site already exists. Any other city is fair game.

Additionally, the founder has to pay taxes and license tariffs – a fixed amount of money deducted from team account and non-refundable under any circumstances. This payment is taken directly from your cash stored in home city (does not need to be delivered anywhere) as soon as one of your cars reaches your desired city to become a launch site and you give the proper command to found rocket investment at that place.

7.5.2. Attracting shareholders

Once the investment is started, resources must be delivered from home base to launch site via cars. As rocket science may consume a lot of effort, you may need help with collecting enough money to get things ready. Everyone who delivers even minimal amount of money to the launch site becomes a **shareholder**.

As being the founder it is your responsibility to attract other teams to invest money at your launch site. First of all, the location of your investment might be important for them so as to deliver goods without huge effort. Moreover, you declare how much of the profit from the successful launch will go to the shareholders and how large share is necessary to have a measure of control over the rocket.

7.5.3. Controlling a launch site

Each shareholder may count on a great money return in case of successful launch. However, initially they have very little to see and say in the investment, unless they cross the share threshold – determined by launch site owner at founding of the site – and become a **controlling shareholder**.

If that happens the team will obtain access to full statistics about the investment, a possibility to use legal support (see 7.6.2.) and to *press the launch button*. Once sufficient amount of money has been transferred to the site a controlling shareholder may give a command to launch the rocket. Of course, doing so too early will not give any profits.

When the investment is already started and ongoing, a site founder is given the same rules as other investors. To be in control, the founder has to reach the share threshold previously set up. The individual investor is never allowed to withdraw the money already given to the launch site. Money return to team's account is only possible on successful rocket launch.

7.5.4. Mission completed

A successful launch means huge returns – the government multiplies all money collected for building a rocket and the revenue is divided among shareholders.

Every single investor that has any shares participates in the profits. The percentage of profits that was initially (at founding the launch site) declared as going to shareholders is divided among them according to their shares. The rest of the profit goes to the founder of the launch site.

7.6. Lawyering

Building a rocket is a complicated industry, it demands dedication of money, knowledge and time. Taking shortcuts is a temptation: to reduce the resources needed, sometimes the investors use knowledge from illegal sources; the founder may also be accused of mismanagement, and so on. For all such cases a launch site may be assisted by lawyers.

Each team may employ one lawyer to protect legal issues of a given investment. Lawyers are always contractors – you hire (pay) them to support a certain launch site for a certain period of time. If there are no pending court cases for your lawyer, you may change at any point the launch site the employer is assigned to. Each time you pay the contract is extended for 30 turns.

Each lawyer has own **motivation** value. Beetlejumper legal practitioners tend to have poor memory due to being overworked. They only remember being paid for latest 30 turns, so any payments made before that time become forgotten. A lawyer motivation is the fraction (in percent) of the money the person was paid to the whole amount of cash the person could be paid for all the period remembered (30 turns). In other words, lawyers have to be paid every turn to keep their motivation high.

After being employed and sufficiently motivated, the lawyer can be used to protect chosen investment from being sued, or to sue opponents' launch site.

7.6.1. Investment protection

A lawyer assigned to launch site influences two properties of that investment: legal **protection level** and **highest motivation**.

The protection level P is calculated via formula:

$$P = min(\left\lceil \frac{10 \cdot L_c}{T_c} \right\rceil, 10) \tag{16}$$

where L_c is the number of lawyers assigned to the launch site, and T_c is the number of controlling shareholders. Please note that P may not exceed 10. If there are more lawyers than controlling shareholders, legal competences start to overlap and protection stays at 10. If there are no controlling shareholders, P equals 0.

The second parameter important for protection is the highest motivation (H) of all present lawyers within the given investment.

7.6.2. Suing

A controlling shareholder may start a lawsuit on behalf of the investment against different launch site. To do so the amount of cash already collected within the investment needs to be greater or equal to 10% of the sued investment total deposit. That money is not spent, but it must be available for the trial process to begin.

To sue another launch site the controlling shareholder is also obligated to employ a lawyer in the controlled investment. Based on this lawyer assignment the suing party is defined (the one which claims something from the sued party).

Once suit is placed, following applies until it ends:

- Neither sued nor suing launch sites can accept money,
- Suing lawyer may not be reassigned from suing investment,
- Neither sued party nor suing party can be sued by anyone else until current legal battle is resolved.

The last point does not protect other investments even if they have common investors to the ones taking part in the lawsuit. Being sued also does not stop any rocket from being launched.

Before the court The suit takes 10 turns to resolve. This is a time to mount up a defense and strengthen offense – employ lawyers and increase their motivation. After that time, if any of the following applies, the suit is won by the suing party:

- $P_{att} \geqslant P_{def} + 3$,
- $P_{att} \geqslant P_{def}$ and $H_{att} \geqslant H_{def} + 10$,

where P_{att} is the legal protection level of the suing party, P_{def} is the legal protection level of the sued party. H_{att} is the highest motivation of the suing party, H_{def} is the highest motivation of the sued party.

Consequences of victory If the suit is successful, the suing party receives 50% of the sued investment total deposit. The money to cover the transfer is provided from the following sources, in the following order:

1. Investments of the team which started the lawsuit (if any amount available).
2. Investments of the remaining shareholders (including owner), in amounts proportional to their shares.

As far as the victorious site (suing party) is concerned, the money is distributed among all shareholders, according to their shares.

Consequences of loss If the suit is lost, the suing site will pay 20% of the money they were suing for (which means 10% of the sued site total deposit). The rules which order sources of the money transfer are same like for previous case (first, the deposit of the team which started the lawsuit, then the others – proportionally to their shares). This could drive some investors' shares to zero, but will not cause them to pay what they do not have.

The money is only taken to cover court administration expenses, thus it is not given to any team or investment.

Lawyers availability In some parts of the planet lawyers might not be available. That means no team can employ a lawyer and no lawsuits can be started.

7.7. Start and the end

Initial state of the game All cars are placed in their home bases and are fully tanked up, the roads are undamaged (all handicaps and bonuses equal 0), and there are no active investments. Each team has the same (small) amount of money.

End of the game The game ends after either 3 rockets are launched (immediately), 30 turns pass from the first launch, or predetermined, known, number of turns have passed.

7.8. Scoring

Score is based on amount of money accumulated by each team. Following resources are taken into account:

- Money in the base account.
- Money on the road, transported in cars.
- Money deposit in yet unfinished investments.

Money spent on fuel, car upgrades, taxes, and lawyers does not count to the score – it has been spent as a cost to obtain the profits. If none of the cars which a team controls did any cycle around the map, the score equals 0.

7.9. Cartographic data

This chapter contains data provided by Beetlejumper Unified Cartographic Establishment – an organization that provides the planetary maps used in company operations. The maps are delivered in specific format, described below. All values are integers.

In the first line, there are three values:

- *CityCount* — number of cities (nodes) in game
- *RoadCount* — number of roads (edges) in game
- *BaseCount* — number of home bases

In the next line, there are *BaseCount* values, separated by single spaces:

- *BaseLocation* — ID of the cities containing home bases

In the third line, there are *CityCount* values, separated by single spaces:

- $VisitingBonus$ — city visiting bonus, used to determine value of M_{max} (see 7.4.4.)

Next *RoadCount* lines contain three values each, separated by single spaces:

- $RoadFrom$ — ID of source city for the road,
- *RoadTo* — ID of target city for the road,
- *BaseCost* — Base, unmodified cost of traversing the road.

Name of the file currently in use is provided in `DESCRIBE_WORLD` command, under $Name$.

7.10. Commands

General guidelines on the communication protocol (connecting, logging in, commands, response format) are described in section *Server communication*. Below you can find commands for the problem Rocket Science.

DESCRIBE_WORLD Returns parameters of the world and the value of the score scaling coefficient.

Parameters: none

Data (from server):

In the first line the server returns values separated with a single spaces:

- M ($M \in \mathbb{N}$, $1 \leqslant M \leqslant 10000$) — number of cities,
- S_r ($S_r \in \mathbb{N}$, $0 \leqslant S_r \leqslant 500000$) — amount of money that has to be deposited on the site before rocket launch is allowed,
- S_f ($S_f \in \mathbb{N}$, $0 \leqslant S_f \leqslant 500000$) — amount of money for taxes and tariffs, paid at investment creation,
- S_{ue} ($S_{ue} \in \mathbb{N}$, $0 \leqslant S_{ue} \leqslant 50000$) — cost of car engine upgrade,
- S_{ut} ($S_{ut} \in \mathbb{N}$, $0 \leqslant S_{ut} \leqslant 50000$) — cost of car trunk upgrade,
- N_{max} ($N_{max} \in \mathbb{N}$, $0 \leqslant N_{max} \leqslant 50000$) — maximum money that a typical car can carry,
- U_{max} ($U_{max} \in \mathbb{N}$, $0 \leqslant U_{max} \leqslant 50000$) — maximum money that can carry a car with upgraded trunk,
- S_g ($S_g \in \mathbb{N}$, $0 \leqslant S_g \leqslant 50000$) — cost of an unit of car fuel,
- R ($R \in \mathbb{R}$, $0.0 \leqslant R \leqslant 50000.0$) — base road revenue,
- Q ($Q \in \mathbb{N}$, $0 \leqslant Q \leqslant 100$) — profit multiplier for investments after rocket launch,
- S_{lmin} ($S_{lmin} \in \mathbb{N}$, $0 \leqslant S_{lmin} \leqslant 5000$) — minimal lawyer fee,
- S_{lmax} ($S_{lmax} \in \mathbb{N}$, $0 \leqslant S_{lmax} \leqslant 5000$) — maximal lawyer fee,
- T ($T \in \mathbb{N}$, $1 \leqslant T \leqslant 3$) — duration of a single turn in seconds,
- L ($L \in \mathbb{N}$, $1 \leqslant L \leqslant 100$) — the maximum number of commands which can be issued in one turn,
- K ($K \in \mathbb{R}$, $1 \leqslant K \leqslant 8$) — value of the scaling coefficient,
- $Name$ (a string value without spaces) — name of the world map file.

In the second line, server returns single value:

- M_h ($M_h \in \mathbb{N}$, $1 \leqslant M_h \leqslant 30$) — number of home bases.

In the third line, server returns values separated with a single spaces:

- M_h values of ID_h ($ID_h \in \mathbb{N}$, $1 \leqslant ID_h \leqslant M$) — ID of cities containing home bases of teams.

HOME Describes your team home base.

Parameters: none

Data (from server):

Server returns five one line with values separated with a single spaces:

- ID_h ($ID_h \in \mathbb{N}$, $1 \leqslant ID_h \leqslant M$) — ID of your home city,
- C_h ($C_h \in \mathbb{N}$) — amount of money stored in the home city,
- ID_l ($ID_l \in \mathbb{N} \cup \{\texttt{NONE}\}$, $1 \leqslant ID_l \leqslant M$) — ID of the city which lawyer is protecting,
- L_m ($L_m \in \mathbb{N}$, $0 \leqslant L_m \leqslant 100$) — lawyer motivation.

ROADS_STATUS Returns current descriptions of roads known to your team. A road is known if one of your cars is currently placed at one of road ends.

Parameters: none

Data (from server):

First line contains one value:

- R_c ($R_c \in \mathbb{N}$) — number of roads seen by from your cars,

Next R_c lines contain following values separated by single spaces:

- ID_{rs} ($ID_{rs} \in \mathbb{N}$, $1 \leqslant ID_{rs} \leqslant M$) — ID of one of the ends of the road,
- ID_{re} ($ID_{re} \in \mathbb{N}$, $1 \leqslant ID_{re} \leqslant M$) — ID of the other end of the road,
- B_r ($B_r \in \mathbb{R}$) — team travel bonus for the road,
- R_h ($R_h \in \mathbb{R}$) — total cost of traversing the road (sum of base cost and current road handicap),

CARS Lists cars available to your team.

Parameters: none

Data (from server):

First line of the response contains one value:

- C ($C \in \mathbb{N}$) — number of cars your team controls.

Next C lines contain following values separated by single spaces:

- ID_c ($ID_c \in \mathbb{N}$) — Car ID,
- ID_m ($ID_m \in \mathbb{N}$, $1 \leqslant ID_m \leqslant M$) — ID of the city the car is currently in,
- C_f ($C_f \in \mathbb{N}$, $0 \leqslant C_f \leqslant 100$) — car's fuel level,
- C_e ($C_e \in \{\texttt{NORMAL}, \texttt{FAST}\}$) — car's engine type,
- C_t ($C_t \in \{\texttt{NORMAL}, \texttt{BIG}\}$) — car's capacity state,
- R_r ($R_r \in \{\texttt{U}, \texttt{R}\}$) — flag indicating whether any of the roads passed in current cycle has repeated itself; `U` indicates all-unique route, `R` is used otherwise,
- R_c ($R_c \in \mathbb{N}$) — number of roads visited since leaving home city,
- C_b ($C_b \in \mathbb{N}$, $0 \leqslant C_b \leqslant 10$) — number of turns the car will still be on the road and thus unable to receive orders,
- C_m ($C_m \in \mathbb{N}$) — money currently carried by the car.

MOVE Orders a car to traverse a distance.

Parameters:

- ID_c ($ID_c \in \mathbb{N}$) — ID of the car,
- ID_m ($ID_m \in \mathbb{N}$, $1 \leqslant ID_m \leqslant M$) — ID of the target city.

UPGRADE_CAR Upgrades a given car in designated way. A car has to located in your home city.

Parameters:

- ID_c ($ID_c \in \mathbb{N}$) — ID of the car,
- U ($U \in \{\texttt{ENGINE}, \texttt{TRUNK}\}$) — upgrade type.

FOUND_ROCKET Starts a new rocket investment. Requires caller to have S_f money in the accounts and a car at the target location.

Parameters:

- ID_c ($ID_c \in \mathbb{N}$) — ID of the car initiating new launch site,

- S_g ($S_g \in \mathbb{N}$, $0 \leqslant S_g \leqslant 100$) — intended shareholder gain, in percentage of final profit,
- S_t ($S_t \in \mathbb{N}$, $0 \leqslant S_t \leqslant 20$) — share percentage threshold required to have control.

ROCKET_INFO Gives report about all existing investments and their rockets.

Parameters: none

First line of the response contains one value:

- C_r ($C_r \in \mathbb{N}$) — number of launch sites available.

Next C_r lines contain following values separated by single spaces, each line representing single launch site:

- ID_m ($ID_m \in \mathbb{N}$, $1 \leqslant ID_m \leqslant M$) — ID of the city where the investment is located,
- S_g ($S_g \in \mathbb{N}$, $0 \leqslant S_g \leqslant 100$) — shareholder gain (in percentage of final profit),
- D_o ($D_o \in \mathbb{Z}$, $D_o \geqslant 0$) — your team's deposit,
- D_t ($D_t \in \mathbb{Z}$, $D_t \geqslant 0$) — total deposit,
- F_d ($F_d \in \{N, Y\}$) — `Y` if rocket is already launched, `N` if not,
- F_c ($F_c \in \{N, Y\}$) — `Y` if your team has control over the investment, `N` otherwise,
- F_l ($F_l \in \mathbb{N}$) — number of turns till the end of the current lawsuit,
- F_o ($F_o \in \mathbb{N} \cup \{\texttt{NONE}\}$) — ID of the city where suing investment is located, `NONE` if no lawsuit,
- S_s ($S_s \in \mathbb{N}$, $0 \leqslant S_s \leqslant 30$) — number of shareholders,
- S_t ($S_t \in \mathbb{N}$, $0 \leqslant S_t \leqslant 20$) — share percentage threshold required to have control,
- L_c ($L_c \in \mathbb{N} \cup \{\texttt{NONE}\}$, $0 \leqslant L_c \leqslant 30$) — number of lawyers protecting the investment (information available to controlling shareholders),
- S_c ($S_c \in \mathbb{N} \cup \{\texttt{NONE}\}$, $0 \leqslant S_c \leqslant S_s$) — number of controlling shareholders (information available to controlling shareholders),
- P ($P \in \mathbb{N}$, $0 \leqslant P \leqslant 10$) — legal protection level,
- L_m ($L_m \in \mathbb{N} \cup \{\texttt{NONE}\}$, $0 \leqslant L_m \leqslant 100$) — highest lawyer motivation (information available to controlling shareholders).

TAKE Transfers money from home account to the car. The car has to be placed in base city.

Parameters:

- ID_c ($ID_c \in \mathbb{N}$, $1 \leqslant ID_c \leqslant N_c$) — ID of the car,
- S_t ($S_t \in \mathbb{N}$, $1 \leqslant S_t \leqslant S_{max}$) — amount of money to transfer from team account to the car,

GIVE Transfers money from a car to the investment. Car has to be placed in a city with a launch site.

Parameters:

- ID_c ($ID_c \in \mathbb{N}$, $1 \leqslant ID_c \leqslant N_c$) — ID of the car,
- S_t ($S_t \in \mathbb{N}$, $1 \leqslant S_t \leqslant S_{max}$) — amount of money to transfer from the car to the investment.

LAUNCH_ROCKET Orders rocket launch. Successful only if deposit gathered in the investment reached a specified level (see DESCRIBE_WORLD S_r).

Parameters:

- ID_r ($ID_r \in \mathbb{N}$, $1 \leqslant ID_r \leqslant M$) — ID of the city containing the launch site.

LAWYER_PAY Pays the lawyer to start a new contract of legal protection.

Parameters:

- S_l ($S_l \in \mathbb{N}$, $S_{lmin} \leqslant S_l \leqslant S_{lmax}$) — amount of money paid to the lawyer,

- ID_r ($ID_r \in \mathbb{N}$, $1 \leqslant ID_r \leqslant M$) — ID of the city containing the investment the lawyer is supposed to be assigned to.

LAWYER_SUE Sues another investment. The lawsuit is started on behalf of the investment your lawyer is assigned to and only if you are a controlling shareholder of it.

Parameters:

- ID_r ($ID_r \in \mathbb{N}$, $1 \leqslant ID_r \leqslant M$) — ID of the city containing the target launch site (sued party).

TRANSFERS Lists past payments that resulted from legal cases.

Parameters: none

First line contains one value:

- L_c ($L_c \in \mathbb{N}$) — number of legal cases finished in most recent 10 turns,

Next R lines contain following values separated by single spaces:

- ID_p ($ID_p \in \mathbb{N}$, $1 \leqslant ID_{rs} \leqslant 30$) — ID of the payer (city ID of the investment that lost the lawsuit),
- ID_r ($ID_r \in \mathbb{N} \cup$ `COURT`) — ID of the receiver; `COURT` means this transfer is a result of a failed suit,
- L_a ($L_a \in \mathbb{N}$) — amount paid.

TIME_TO_END Provides the number of turns to the end of the current game.

Parameters: none

Returned line contains two values:

- O ($O \in \mathbb{N}$) — the number of already launched rockets (launched by any team),
- E ($E \in \mathbb{N}$) — the number of turns till the end of the game.

WAIT Waits till the next turn begins.

Parameters: none

Data (from server):

The server returns a single line of characters:

- WAITING S

where S ($S \in \mathbb{R}, S \geqslant 0$) stands for the number of seconds left to wait. Once the period is over, the server sends an additional line:

- OK

7.11. Errors

In case the command is incorrect, the server responds according to the description in section *Server communication* with the following message:

- '`FAILED` *e msg*',

where *e* is an error code, and *msg* — an error message. The table below consists of errors that may occur during communication process in problem Rocket Science.

error code	error message
1	bad login or password
2	unknown command
3	bad format
4	too many arguments
5	internal error, sorry...
6	commands limit reached, forced waiting activated
100	car not found
101	car is busy
102	road does not exist
103	car is not in base
104	upgrade already completed
105	invalid upgrade type
106	not enough money
107	requested city is already occupied
108	invalid shareholder profit
109	invalid shareholder threshold
110	launch site not found
111	you do not control the rocket
112	rocket is not ready
113	there is no one to get the money
114	rocket was already launched
115	invalid amount
116	lawyer has been paid already
117	prosecuting lawyer is not allowed to change assignment
118	you have no active lawyer
119	investment during a lawsuit
120	unable to sue yourself
121	cannot take more money
122	lawyer is not available

7.12. Servers

The games will be held on servers with different parameters.

Table 4: Server addresses and parameters.

Name	**Address:Port**	**Lawyering available**
Rockets 1	`universum.dl24:20006`	Yes
Rockets 2	`universum.dl24:20007`	No

7.13. Example

Below you will find an exemplary record of the communication with the server.

client → server	server → client
	`LOGIN`
`login1`	
	`PASS`
`secret`	
	`OK`
`DESCRIBE_WORLD`	
	`OK`
	`100 10000 1000 2000 500 200 500 1 5.000000 15 100 500`
	`↪ 1 40 2.56829 world1`
	`5`
	`1 43 66 24 64`
`HOME`	
	`OK`
	`1 3593 NONE 0`
`CARS`	
	`OK`
	`5`
	`4321 1 100 NORMAL NORMAL U 0 0 0`
	`4322 1 100 NORMAL NORMAL U 0 0 0`
	`4323 1 100 NORMAL NORMAL U 0 0 0`
	`4324 1 100 NORMAL NORMAL U 0 0 0`
	`4325 1 100 NORMAL NORMAL U 0 0 0`
`ROADS_STATUS`	
	`OK`
	`4`
	`1 2 0 2`
	`1 7 0 1`
	`1 17 0 3`
	`6 1 0 2`
`MOVE 4321 17`	
	`OK`
`WAIT`	
	`OK`
	`WAITING 0.149000`
	`OK`
`WAIT`	
	`OK`
	`WAITING 0.217000`
	`OK`
`FOUND_ROCKET 4321 70 20`	
	`OK`
`TAKE 4325 200`	
	`OK`

```
MOVE 4325 17
                OK
WAIT
                OK
                WAITING 0.925000
                OK
WAIT
                OK
                WAITING 0.649000
                OK
GIVE 4325 200
                OK
ROCKET_INFO
                OK
                1
                17 70 200 200 N Y 0 NONE 1 20 0 1 0 0
MOVE 4325 1
                OK
WAIT
                OK
                WAITING 0.546000
                OK
WAIT
                OK
                WAITING 0.706000
                OK
HOME
                OK
                1 2402 NONE 0
```