# DEVELOPMENT OF SEARCH-BASED AGENTS FOR THE PHYSICS-BASED SIMULATION GAME GEOMETRY FRIENDS

Daniel Fischer

Master Thesis DKE 15-03

Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Faculty of Humanities and Sciences
of Maastricht University

Thesis committee:

Dr. M. H. M. Winands
Dr. ir. J. W. H. M. Uiterwijk

Maastricht University
Department of Knowledge Engineering
Maastricht, The Netherlands
March 30, 2015

# Summary

Geometry Friends is a platform video game like the first Donkey Kong game. It has simulated physics and takes place in a 2D environment. In the rectangular track of the competition a rectangle is controlled, which can move sideways and resize. The objective of the game is to collect all diamonds in the environment in the least amount of time. In 2013 and 2014 there were Geometry Friends competitions at the IEEE Conference on Computational Intelligence and Games. This master thesis focuses on the development of agents for the game Geometry Friends. The thesis gives first an overview of the properties of physics-based simulation games and requirements of the agents for these games. The search techniques Monte-Carlo Tree Search (MCTS) and A* are investigated for Geometry Friends. MCTS is able to find the most promising action with random samples. A* searches the minimum-cost path. To find a route in a level collecting all diamonds, the search techniques need an abstraction of this level. The abstraction is based on a directed graph with nodes and edges. For example, nodes are the ends of obstacles and diamonds in the level. Edges represent the shape in which the rectangle reaches a node from another. Four A* versions, MCTS, and a combination of MCTS and A* are investigated as search techniques to calculate a route of nodes to collect the diamonds of a level. Subgoal A* is one of these A* versions. It searches for a minimum-cost route where all diamonds can be collected. If a time limit is reached or no route is found, the number of diamonds to collect is reduced by one. The calculated route is used by the driver. The driver is rule-based and follows the route node by node. It returns the actions to perform. The different search techniques with different properties are compared to each other in the levels for the 2013 and 2014 competitions. The techniques are also compared to the best agent of the competitions. Also 10 new difficult levels are created for a larger and more challenging test set. It is discussed why and which search technique is either more or less effective than the others. The search technique Subgoal A* was concluded as best search technique for Geometry Friends with a 81% and 32% higher score for the 2013 and 2014 competitions, respectively, than the best agent for the 2013 and 2014 competitions.

# Contents

# Chapter 1

# Introduction

In Artificial Intelligence (AI), games are important since its origin. Games are a challenging benchmark to test the performance of different techniques. This master thesis is about how to develop an intelligent agent for the game Geometry Friends. Geometry Friends is a platform video game like the first Donkey Kong game, it has simulated physics and takes place in a 2D environment. The objective of the game is to collect all diamonds in the environment in the least amount of time. One of two search techniques is Monte-Carlo Tree Search (MCTS), which is used because it gets more and more interesting in the last years because of its successful application. The second search technique is A*. This thesis explains how the abstraction is performed, how MCTS and A* are implemented, and how the player is controlled through the level. It describes what are the problems and how they can be solved with which improvements. At the end, different methods and improvements are compared to each other to demonstrate, which are best for Geometry Friends. The next sections explain the meaning of AI in games (1.1), what are the properties of physics-based simulation games (1.2), MCTS (1.3.1), A* (1.3.2), the abstraction in general (1.3.3), the problem statement and research questions (1.4), and an outline of the thesis (1.5) are described.

## 1.1 AI in Games

Game AI started with classic board games. In 1950, as computers became available, Shannon (1950) and Turing (1953) proposed ideas for chess programs. Samuel (1959) of IBM developed a checkers-playing program, which was characterized as tricky but beatable by strong amateur opponents (Barto, 1998). The program learned by playing against itself. To choose chess or checkers was a logical choice because these are games of perfect information so that the computer has access to the entire state of the game. Furthermore, the rule set is limited and the game is deterministic. With faster computer hardware and better search techniques even expert players were beaten. The machine Deep Blue developed by IBM won the second six-game match against world chess champion Garry Kasparov in 1997.

In the past decades, video games started with simple 2D games like Pong and Pac-Man and now are 3D real-time games with sophisticated graphics and physics. For video games, which getting more and more complex, also designing a game-playing agent is more challenging. The complexity of these games offers a range of excellent testbeds for AI research. In the 1980s role-playing games and turn-based strategy games were developed but they had not a sophisticated AI, usually it was a simple rule-based system. In 1992, with the game Dune II, a new genre of computer games was introduced: real-time strategy (RTS). This RTS genre requires the use of efficient AI techniques because a player can perform an action anytime. Until today, new genres and features of games are one reason for the development of new AI techniques or enhance existing techniques. The next section continues with AI in games with looking at physics-based simulation games in combination with the real-time property.

## 1.2 Physics-Based Simulation Games

In the past few years, physics-based simulation games such as Angry Birds or Computational Pool have become popular. The agent has to be adapted to handle the physics, otherwise the player loses interest because the agent's performance is insufficient to be a competitive opponent. All game objects of the

game world have physical properties. The physics simulator of these games has complete information about these properties. The physics simulator simulates and displays each action and its consequences. A result of the physics and real-time property is that there is a very large or potentially infinite number of possible actions, which is difficult to handle for an agent. To predict an action is quite difficult because noise is added to the simulator. The actions are non-deterministic. If the exact physical properties or the behavior of the simulator are unknown, it is even more challenging. This can be handled by learning through observation. An intelligent agent should be able to (1) determine or at least approximate the outcome of actions, (2) plan the best action or actions in sequence in a given situation, (3) handle uncertainty, (4) and combine reasoning and simulation. There are several AI competitions of physics-based simulation games, such as Angry Birds, the Physical Traveling Salesman Problem and Geometry Friends, to foster research in this area.

## 1.3 Search

To find the most promising route to solve a level of Geometry Friends a search technique is required. Two search techniques are selected, MCTS and A*. This section explains how MCTS and A* work and which properties they have. To use a search technique an abstraction of the game is required. This section describes which properties are important for an abstraction and why the abstraction can be difficult.

### 1.3.1 Monte-Carlo Tree Search

In 2006, Monte-Carlo Tree Search (MCTS) had its breakthrough with its good performance in the game Go (Kocsis and Szepesvári, 2006; Coulom, 2007). Since that time, MCTS has become more and more in focus of Game AI research (Browne *et al.*, 2012). With random samples the algorithm is able to find the most promising action in a given domain. In general, MCTS can be divided into four steps (Chaslot *et al.*, 2008). In the first step, selection, a special selection policy is used to selected nodes of the built tree until it reaches a leaf node. If it is the first time, simply the root node is selected. If a leaf node of the tree is selected, the play-out step starts. In the play-out step a random or special play-out policy is used to determine an action. After this action a next action is selected until a given depth or the end state is reached. Action means some kind of action, which leads from one state to another. In the expansion step the node reached by the first action of the play-out is selected and added to the tree. In the last step the play-out result is backpropagated through the nodes selected in the selection step and the new node. This means statistics are updated. The play-out is discarded. This process with all four steps is performed for a certain amount of time. With the statistics of the nodes the most promising action can be determined.

MCTS has two properties that makes it quite interesting. There is no domain knowledge required and the algorithm can be stopped any time and the result can be used. The second property is important in a real-time game like Geometry Friends.

The Physical Traveling Salesman Problem (PTSP) is a real-time game where MCTS performs successfully (Perez *et al.*, 2013). In the PTSP the environment is a maze with waypoints scattered around. The player governs a spaceship and has to collect all waypoints in the least amount of time. It consists of a route planner, which computes the order of collecting the waypoints before the game starts. Because this computation has a time limit, it is not possible to compute the complete route. The driver controls the spaceship in real-time and has a time limit of 40 ms to response an action. If the precomputed route ends, the driver computes the waypoint to reach next. In the route planner and in the driver MCTS is used.

### 1.3.2 A*

A* is selected as second search technique, which is a widely used pathfinding algorithm to find the shortest route (Hart, Nilsson, and Raphael, 1968). It finds the minimum-cost path given a start node and an end node. A* follows a route of the lowest expected total costs while it traverses the route. To select the most promising node, a function $f(x)$ is required, which computes the expected costs of a known node. The costs are computed by Equation 1.1.

$$f(x) = g(x) + h(x) \tag{1.1}$$

$g(x)$ are the true costs from the initial node to node $x$ and $h(x)$ are the estimated costs from node $x$ to the goal node. $h(x)$ is computed by an admissible heuristic.

Starting from the initial node, for all successor nodes the costs are computed and added to a list. Next the node with the lowest expected costs of the list is selected and visited next. Again the costs of all successor nodes are computed and added to the list. This is performed until A* finds the shortest route. The single steps are not explained in detail in this section but it gives a brief introduction.

### 1.3.3 Abstraction

Abstraction is a method to represent the level of the game so that search can be used. At first a general explanation about abstraction is described and then a brief introduction of the selected approach is given.

One of two general approaches is to group together actions. The result of grouping together more or less similar actions is that the branching factor decreases and therefore the complexity of the search tree decreases. The abstraction of the upper tree should be smarter and maybe more time-consuming than the lower part of the three because in the upper part a wrong decision can lead to a non-optimal action. In the lower part, the play-out, a high number of simulations is wanted so that the abstraction is less sophisticated. Weinstein, Mansley, and Littman (2010) use an extension called Hierarchical Optimistic Optimization applied to Trees (HOOT) to overcome the discrete actions limitation of UCT with replacing the action selection policy. Bellemare, Veness, and Bowling (2013) proposed an interesting approach with image-based factored environment recognition for a collection of 20 Atari 2600 games. Jiang, Singh, and Lewis (2014) showed an approach finding local abstractions in layered directed acyclic graphs in Markov decision processes and local and approximate homomorphisms, which increases the performance of MCTS in Othello.

In the second approach a sequence of actions is abstracted to a high-level task. With fewer actions the depth of the tree decreases and this increases the forward planning of MCTS. These sequences of actions are called *macro-actions* and are successfully used in Ms Pac-Man (Pepels and Winands, 2012) and in the Physical Traveling Salesman Problem (Perez *et al.*, 2013).

A quite interesting abstraction for Geometry Friends is given by Kim, Yoon, and Kim (2014). A directed graph is built with nodes and one/two-way edges. For example, nodes are at the ends of an obstacle, at a narrow alley or at the point where the player will fall down. The edges describe how node $B$ can be reached by node $A$ if there is an arrow from node $A$ to $B$. Within this graph building process physical constraints are taken into account but the physical properties of Geometry Friends are unknown. This approach can be assigned to the second abstraction approach because the edges abstract a sequence of actions to one action. In addition it is a domain dependent approach for Geometry Friends and not directly applicable to other games.

## 1.4 Problem Statement & Research Questions

Geometry Friends is a real-time physics-based simulation game (Rocha, 2009). The aim of the thesis is to develop agents, which can solve the competition levels of this game. The representation of the levels is done by an abstraction, which cannot represent the world without missing information. For the rectangle and circle players the abstraction has to be different because they have different actions. There is no information about the physics, which makes the abstraction more inaccurate. A* is a common search technique to find the shortest route in a relatively short amount of time, but in Geometry Friends the shortest route is not the best route. The best route is the one where all the diamonds can be collected. Physical conditions are the reason why the shortest route is not the best route. MCTS has the properties that it can be stopped any time and no domain knowledge is necessary. It can be that MCTS is not optimal because of short calculation periods or problems with action or state space. The part which controls the player has to take into account the physics and it can have problems because the abstraction is not perfect.

As a result the problem statement is as follows:

- How can we develop an agent for Geometry Friends?

The following four research questions can be derived:

- How can the game world be abstracted by taking into account physics?

  A graph is built for each game level with nodes for obstacles and diamonds. The physics conditions are taken into account in the edges between the nodes.

- How can A* be incorporated in Geometry Friends?

  A* uses the abstraction to calculate a route. There are several A* versions proposed in this thesis, Greedy Goal A*, Y-Heuristic A*, Permutation A* and Subgoal A*.

- How can MCTS be integrated in Geometry Friends?

  MCTS also uses the abstraction to calculate a route. In addition, a combination of MCTS and A* is presented.

- How can the driver take into account the physics and missing information due to abstraction?

  The driver is rule-based and follows the route node by node. The physics is taken into account by the driver, which accelerates, decelerates or does not move. It is also taken into account in the route because of the abstraction. Missing information is not taken into account.

## 1.5 Thesis Outline

This section provides a brief overview of all following chapters of the thesis. First, Chapter 2 is about the game Geometry Friends where the game environment, possible actions of the player, the development environment, and uncertainty of the game are explained. Chapter 3 describes the abstraction, what is represented by the nodes and edges, and how they are created. In Chapter 4 the search techniques A* and MCTS, and a combination of MCTS and A* are explained in detail. For A* there are four A* versions with different approaches. Chapter 5 explains the rule-based driver. All rules are described and in what situations which rule is applied. All experiments and results are shown in Chapter 6. The setup of the experiments, the performances of the best agent in the Geometry Friends 2013/2014 Competition, the abstraction computation and all different search approaches are presented. In the discussion, it is explained why an approach is either more or less effective than others. Finally, Chapter 7 concludes the thesis. The thesis is summarized, the research questions are answered and an outlook for possible future extensions is given.

# Chapter 2

# Geometry Friends

The physics-based simulation game Geometry Friends was developed originally for the Wii in 2009. It is the result of a master thesis that had as goal to create a collaborative gameplay experience (Rocha, 2009). The vision is that human and agent players play in cooperation. Since 2013 there is an official Geometry Friends Competition of the IEEE Conference on Computational Intelligence in Games (CIG). In August 2015 the competition will run again. The goal in this platform game is to collect all diamonds in the environment in the least amount of time. To show the challenge of the goal in this chapter the game environment (2.1), the different players and actions (2.2), the development environment (2.3), and uncertainty in the game (2.4) are described.

## 2.1 Game Environment

Geometry Friends takes place in a 2D environment with simulated physics, gravity and friction. The physics model is unknown (i.e., there is no forward model). There are 30 different levels in the 2013 competition. 10 levels for each track with different properties, 5 public levels and 5 private levels. The 2014 competition has the same public levels but some private levels changed. There is the rectangle track for the rectangle player, the circle track for the circle player, and the cooperation track for both players. The differences of the players and their actions are described in the next section. In Figure 2.1 the rectangle public level 5 is shown. The green block is the rectangle player. The black objects are simple obstacles. The purple objects are the diamonds to collect. Sometimes a diamond is also called collectible. The layout of the obstacles, the initial position of players and the place of diamonds is static but different in each level. Figure 2.2 shows a more complex level of the cooperation track. A special yellow obstacle can be seen, which is an obstacle for the green rectangle player but not for the yellow circle player. This obstacle exists also in green where it is an obstacle for the yellow player but not for the green one.

In the competition there is a time limit, a completion bonus and a collectible unit value, which vary for each level for the 2013 competition. The 2014 competition has the same completion bonus and collectible unit value for each level. The formula of the score is shown in Equation 2.1. The score is calculated by $V_{Completed}$, which is the completion bonus, multiplied by $maxTime$ minus $agentTime$, which is the time limit and the time the agent needed, divided by $maxTime$. At least $V_{Collect}$, the collectible unit value, is multiplied by $N_{Collect}$, the number of collected diamonds, and added to the score.

$$ScoreRun = V_{Completed} \times \frac{(maxTime - agentTime)}{maxTime} + (V_{Collect} \times N_{Collect}) \tag{2.1}$$

## 2.2 Players and Actions

Geometry Friends has two types of players, the green rectangle and the yellow circle with different abilities. In Figure 2.3 the different actions can be seen. The green rectangle can slide sideways, resize from a square to a vertical or horizontal rectangle with no mass change. In any case, the surface is always the same for any shape. The yellow circle can roll sideways, jump and resize the radius/mass. When the square or the circle player moves sideways they accelerate and decelerate. For example, if the player

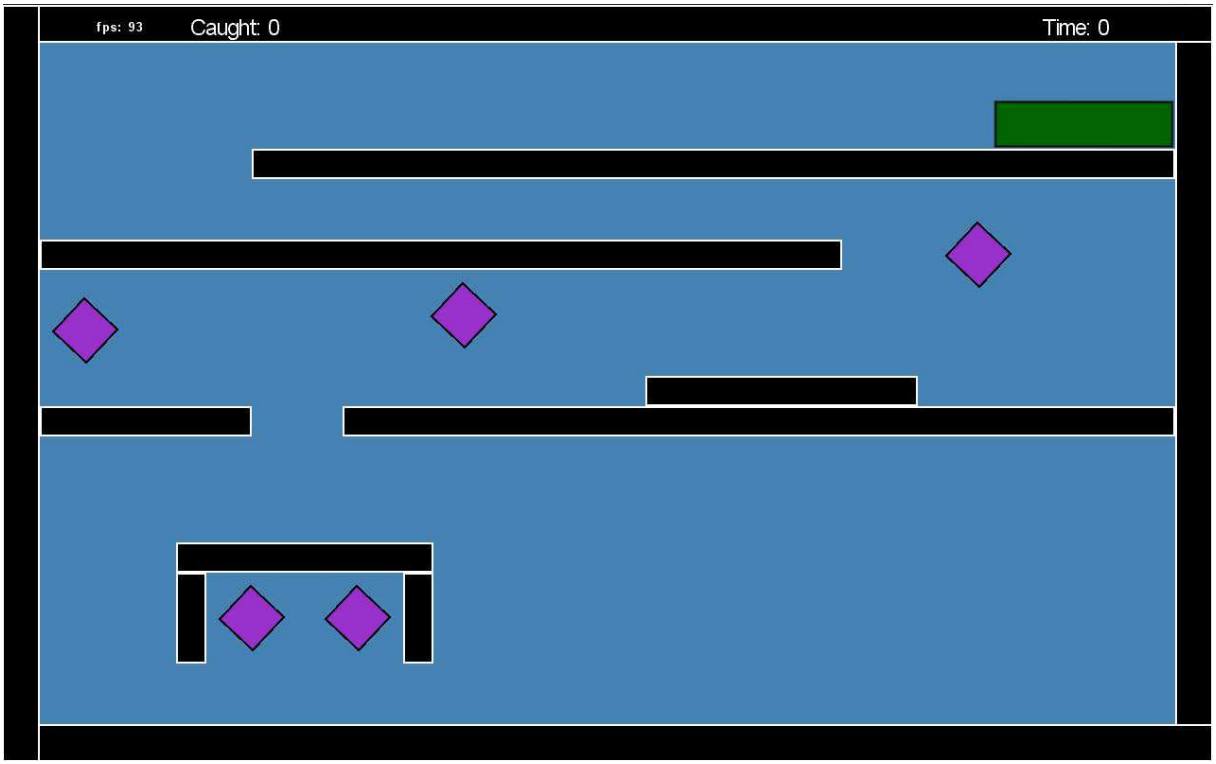


Figure 2.1: Rectangle Level 5

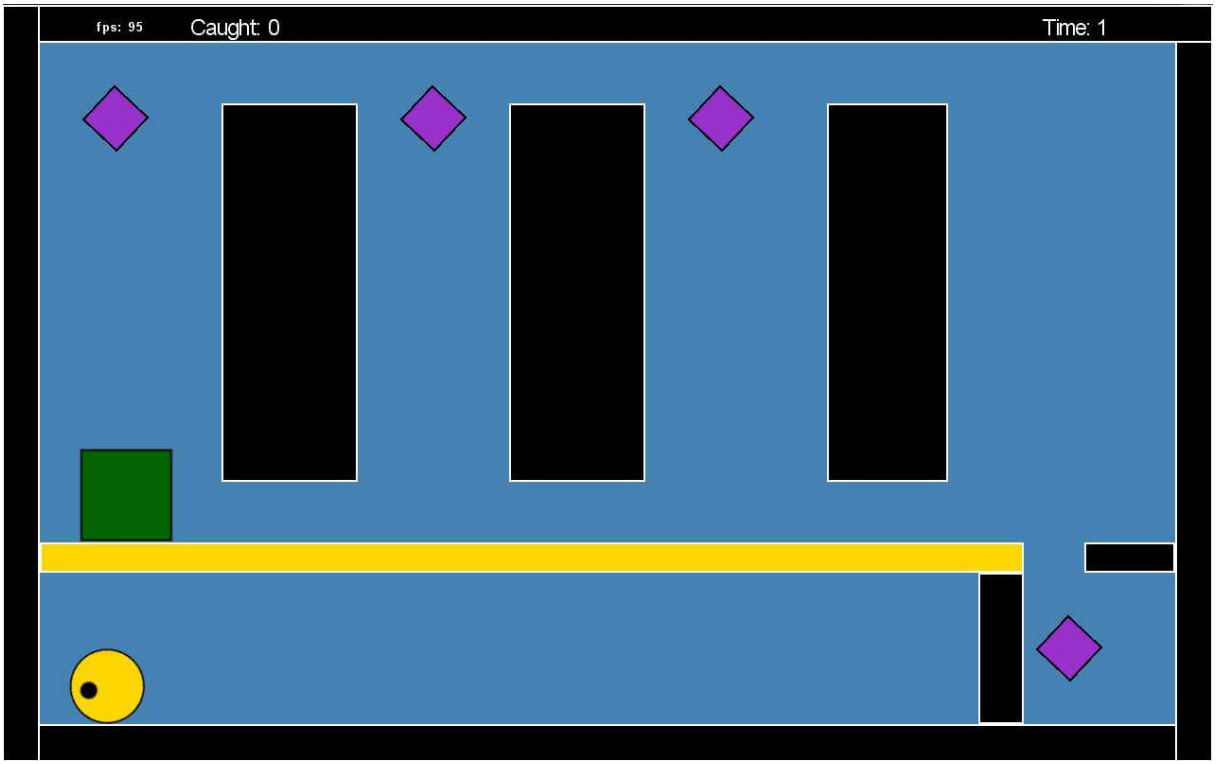


Figure 2.2: Cooperation Level 8

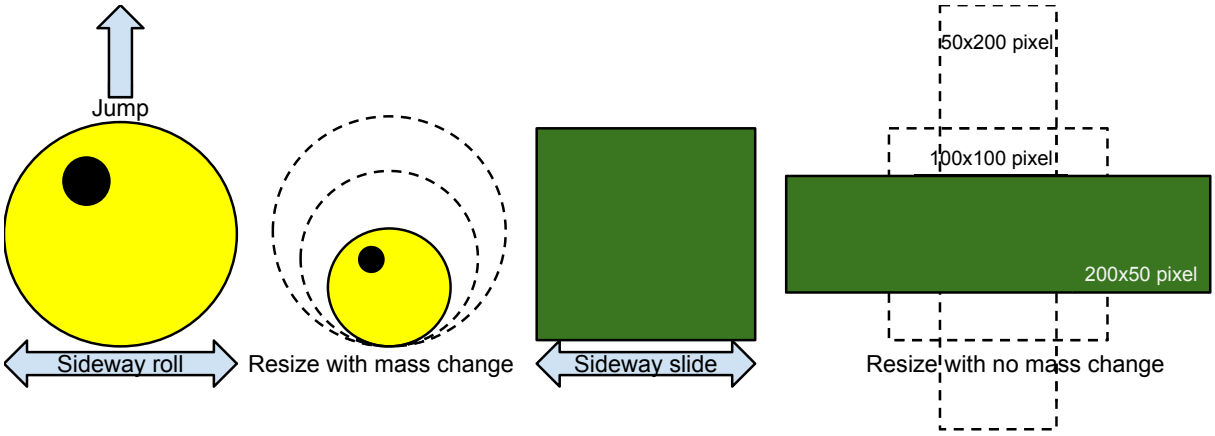


Figure 2.3: Actions of the players

moves left and the move left action is performed, the player accelerates, if the player moves left and the move right action is performed, the player decelerates.

The levels have a different walkthrough for each player. For the rectangle player it is a challenge in puzzle solving. Platform construction and the order of collecting diamonds are difficult. The circle player also has some puzzles but the main part is the actuation. Here, puzzles require precise skills and timing. The cooperation track combines both challenges.

## 2.3 Development Environment

Geometry Friends is developed in C#, Microsoft's Visual Studio is therefore required. On the wiki page[1] of the project it is written how the project has to be configured and for each player a sample agent is described, which is already implemented. All agent files are kept in the main project folder *GeometryFriendsAgents* with the sample agent classes *BallAgent* and *SquareAgent*. Before running the executable file to start the game, the project has to be built. The game automatically loads the agent files once a specific level is picked in the level overview.

There are three methods given, which are important for development of an agent. The *Setup* method is called once a level is started and gives the following information:

1. Number of obstacles, green obstacles, yellow obstacles and collectibles
2. Square $x/y$ position, $x/y$ velocity and height
3. Circle $x/y$ position, $x/y$ velocity and radius
4. Of all the obstacles $x/y$ position, height and width
5. Collectible $x/y$ position

Position means the center point of an object. The full level screen has a width of 1280 pixels and a height of 800 pixels. The *UpdateSensors* method updates the number of collectibles, square information, circle information and collectible coordinates. At each time step (∼10ms depending on the hardware) an action can be executed in the *Update* method with *SetAction*.

## 2.4 Uncertainty

Geometry Friends has some noise and bugs. For example, the first noise factor is that the player runs into a hidden pixel on a flat obstacle and jumps in the air or it gets interrupted by a hidden pixel at an edge of an obstacle. Another noise factor is that the rectangle player information $x/y$ position, velocity and height have noise of about plus/minus one pixel.

[1]Link to the wiki page: `http://gaips.inesc-id.pt:8081/geometryfriends-wiki/doku.php`

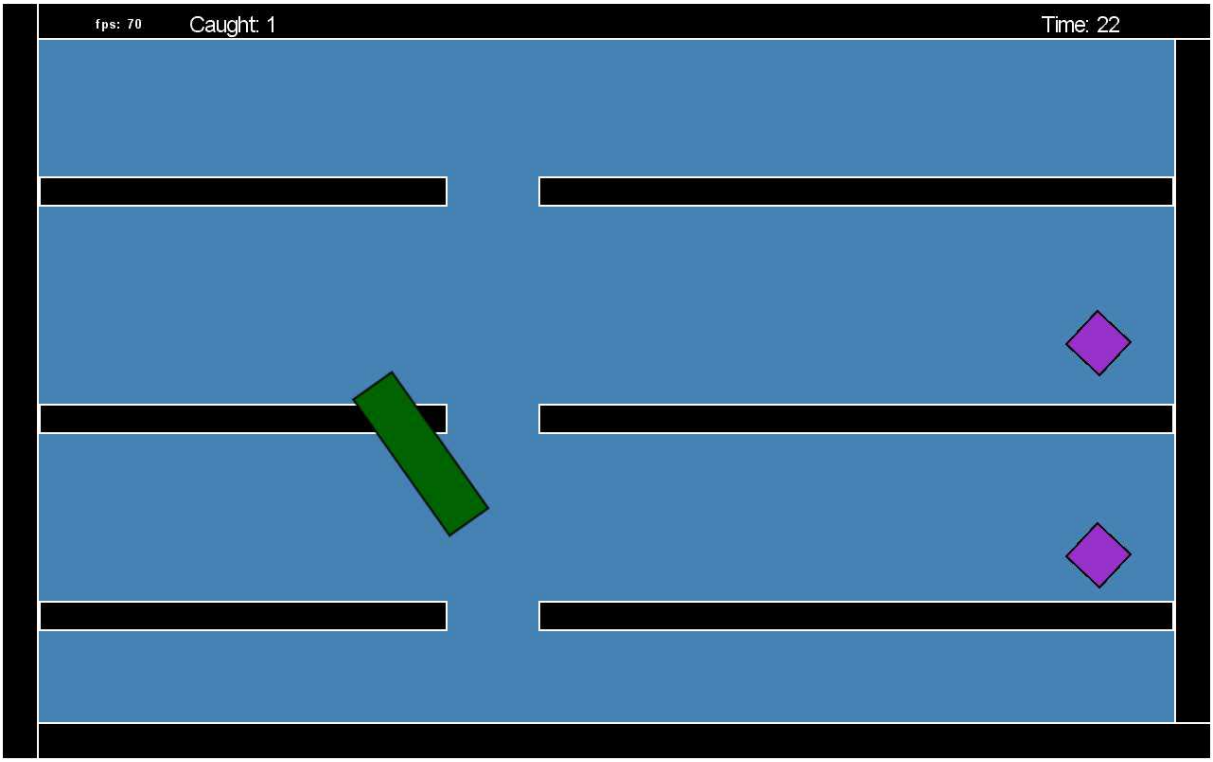


Figure 2.4: Rectangle player gets stuck in an obstacle

Now the bugs of the game are described. The given rectangle player information about the height is reversed with the width if the height of the rectangle is lower than 158 pixels and the rectangle overturns on another obstacle. If this happens and the rectangle morphs up or down, the second bug is triggered. The rectangle gets stuck in the obstacle and cannot move anymore. The player can also get stuck in an obstacle during morphing in a gap. Figure 2.4 shows the rectangle player in an obstacle.

The game has a timer, which starts when the player can move and stops if all obstacles are collected. The third bug is that the game and also the timer freeze during calculations. That means the timer does not take into account different calculation times. The time is a factor of the competition score so this timer bug is quite a problem. The noise and the bugs have to be taken into account during the development of the agents.

# Chapter 3

# Abstraction

The abstraction represents the structure of a level and possible actions of Geometry Friends. This is the foundation to compute routes with both search techniques, A* and MCTS. This abstraction takes into account the rectangle player and its abilities. For the circle player the basics could be the same but the movement and thus the physical conditions are different. The approach of the abstraction is based on the following two articles. The article by Kim *et al.* (2014) describes a directed graph abstraction for Geometry Friends. The graph consists of nodes and one/two-way edges. For example, nodes are at the ends of an obstacle, at a narrow alley or at the point where the player falls down. The edges describe how node $B$ can be reached by node $A$ if there is an arrow from node $A$ to $B$. For instance, the rectangle player has three different edges for a "Square", "Horizontal rectangle" and "Vertical rectangle" action. Within this graph building process physical constraints are taken into account but the paper does not explain it and the physical properties of the game Geometry Friends are unknown. Next, the article by Perez *et al.* (2013) describes an abstraction for the Physical Traveling Salesman Problem, which uses only the *collectables*. The approach of an own abstraction is to combine everything, the nodes given by the obstacles, create nodes for the *collectables* and take into account the physics of the game.

The abstraction is computed when a level starts. Looking back at Section 2.3 it means the implementation is in the class *SquareAgent*. The abstraction is performed at the first call of the *Update* method. At this step all information is given. A first simple representation of the level is required to compute the nodes and edges of the abstraction. This representation is built by the given obstacle information. In a *boolean* matrix all obstacles are set as *true* and the free space is set as *false*. With this matrix the nodes (3.1) and edges (3.2) can be calculated in the next sections.

## 3.1 Nodes

Nodes are key pixels in the level to follow a route. They have special attributes which are:

- $x$ coordinate
- $y$ coordinate
- Attribute to know if the node is a diamond or not
- Attribute to know if the node leads to a fall-down node
- Attribute to know if it is a pseudo node

The $x$ and $y$ coordinates and whether the object is a diamond are obvious attributes. The other two attributes are important and explained further in Chapter 5. The nodes are based on the initial position of the player, obstacles and diamonds. All nodes are one pixel above an obstacle except high diamond nodes, which are at the center of the diamond.

Figure 3.1 shows the nodes of rectangle level 5 to explain all types of nodes. The initial position node is the center point of the player and can be seen in Figure 3.1 labeled as node 1. An ordinary obstacle has two nodes at the left and right corner, each one pixel above and one pixel to the left/right (similar to Kim *et al.* (2014))(e.g. node 2 and 3). If the obstacle is next to another obstacle, which obstructs

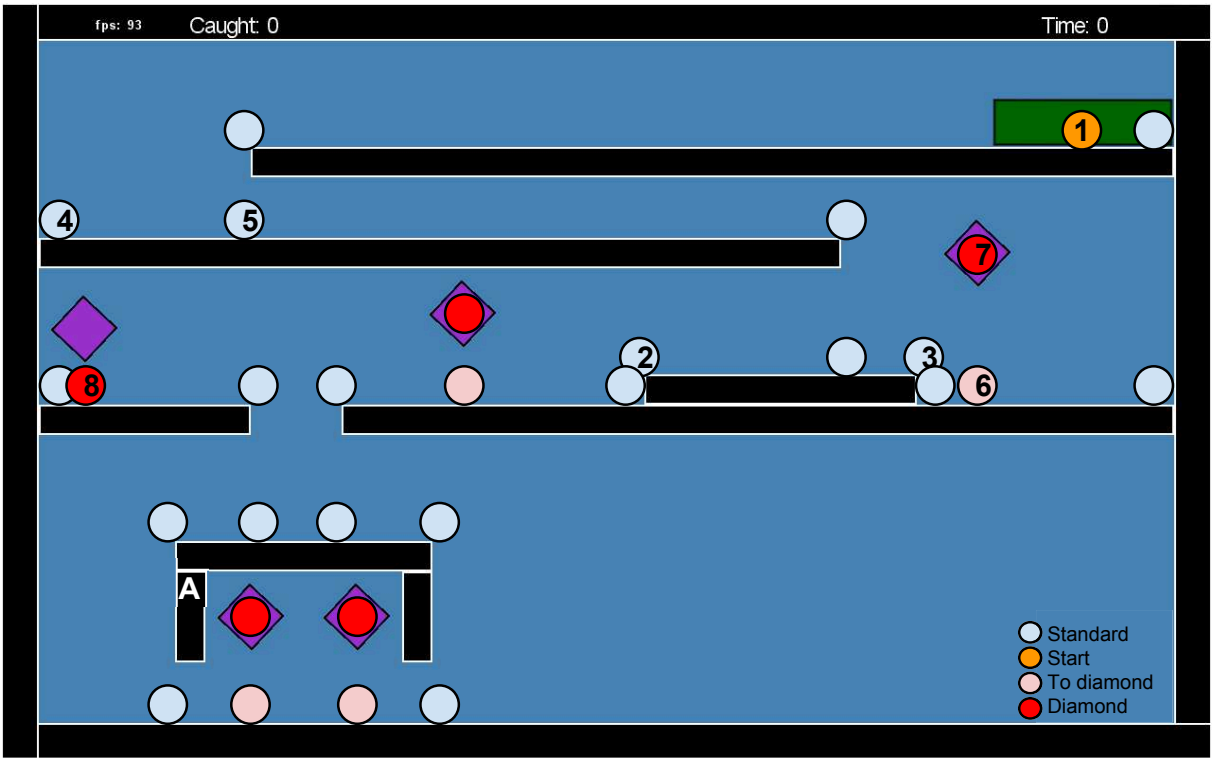


Figure 3.1: Nodes of rectangle level 5

the direction, the node is placed in front of the other obstacle (e.g. node 4). If the obstacle is next to an obstacle with the same height level there is no node. If there is an obstacle completely on another obstacle, the lower obstacle has no nodes (e.g. obstacle $A$). If there is free space next to the obstacle, a fall-down node is created at the ground one pixel to the left/right and straight down until another obstacle is reached (similar to Kim *et al.* (2014))(e.g. node 5). For each diamond a straight fall-down node is created (e.g. node 6). If the diamond is more than 80 pixels above the obstacle below, the center of the diamond is a node and marked as diamond (e.g. node 7). If the diamond is below the threshold of 80 pixels, the fall-down node of the diamond is marked as diamond and no other node is created (e.g. node 8).

## 3.2 Edges

The edges between two nodes describe the action the player has to perform to get from one node to another one. The edges are represented in an adjacency matrix with numbers from 0 to 3. The number 1 implies an action as a square, 2 an action as a horizontal rectangle, and 3 an action as a vertical rectangle. The number 0 implies there is no edge between the nodes. Another important piece of information is the direction the player has to follow to reach the goal node using the given action of the adjacency matrix. The direction is also stored in a matrix with the same size as the adjacency matrix. There are eight different directions from 0 to 7, from right to upper right, clockwise (Figure 3.2). The last piece of information, for later calculations, are the distances between nodes with an edge. The distances are also stored in a matrix. To compute the distance the Euclidean distance is used.

The edge process starts with comparing each node to all the other nodes. The direction and the distance are computed with delta $x$ and delta $y$. If one or both nodes are diamond nodes and the direction between the nodes is 1 (lower right), 3 (lower left), 5 (upper left) or 7 (upper right), this edge is not taken into account. These directions are not taken into account because diagonal lines are more prone to error than driving over vertical or horizontal obstacles. If an error occurs while collecting a diamond, for instance, in direction 1 (lower right), this is much worse than not reaching a non-diamond node. Because of this edges incident to one or two diamond nodes are only allowed for straight vertical

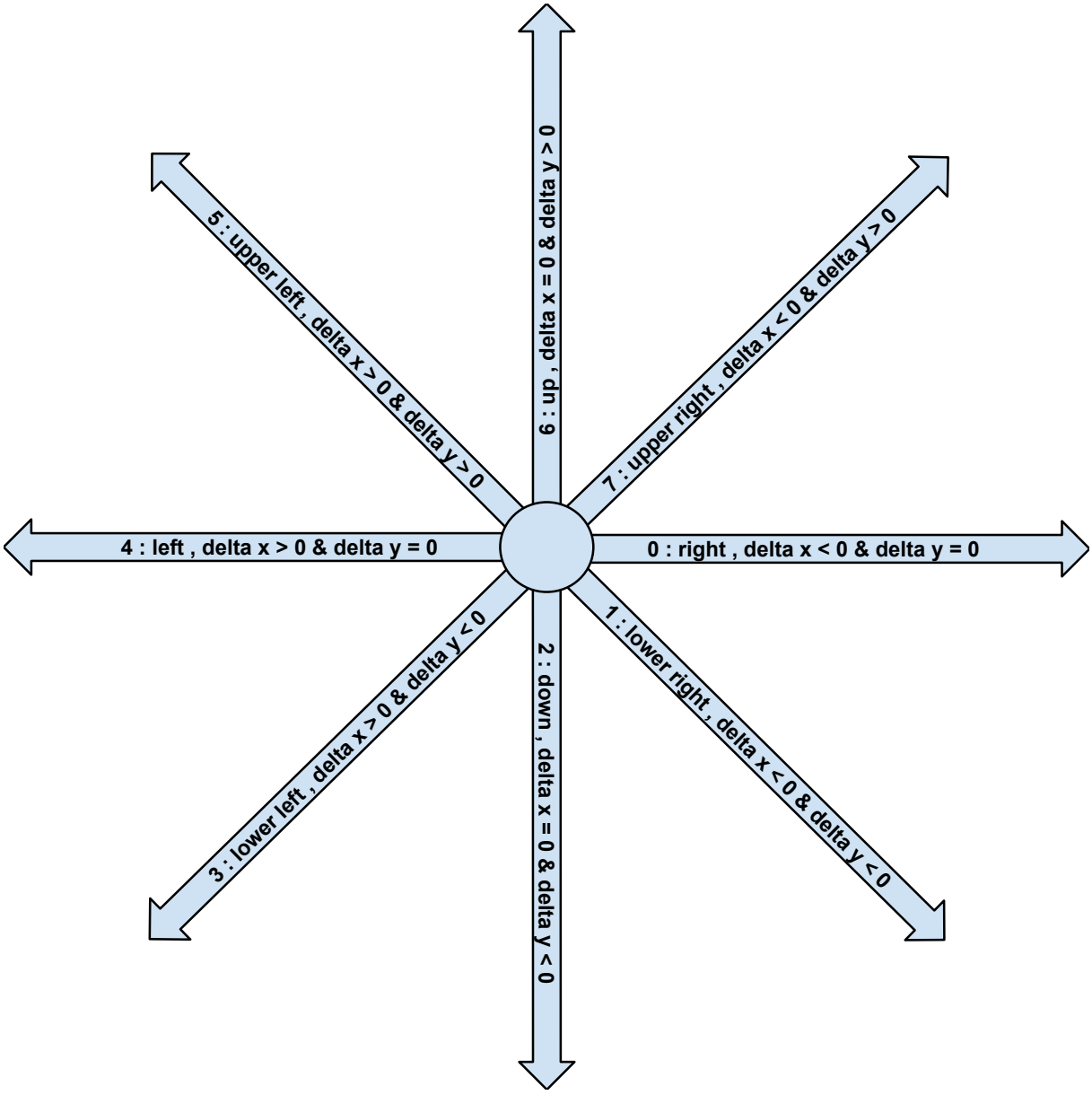


Figure 3.2: Definition of all directions

or horizontal lines and not for diagonal lines. In general, direction 5 (upper left) and 7 (upper right) are not taken into account to calculate the edges because it is not necessary. If one of the following three conditions holds, there is no edge between two nodes and the corresponding entry is set to 0 in the adjacency matrix:

- $\Delta y \geq 200$ & $\Delta x = 0$, the rectangle cannot jump and its maximum height is 200 pixels
- $\Delta y \geq 50$ & $\Delta x \neq 0$, the targeted platform is too high
- $direction = 6$ & ($\Delta y$ <200 & >75) & $node2 \neq diamond$, it is not necessary to continue in up direction if the second node is not a diamond because the rectangle player could not move along, the node is too high

Otherwise, the algorithm continues with shifting the three different shapes of the rectangle (square, horizontal rectangle and vertical rectangle) pixel by pixel along the straight vertical, horizontal or diagonal line, depending on the direction.

The straight vertical or horizontal lines are straightforward to identify but for the diagonal lines Bresenham's line algorithm is used (Bresenham, 1965). The algorithm is developed in the field of computer graphics and determines the points of a $n$-dimensional raster that should be selected in order to form a close approximation to a straight line between two points. Algorithm 1 shows the implemented Bresenham's line algorithm. The start point and goal point coordinates $x0$, $y0$ and $x1$, $y1$ are required to start the algorithm. Also the matrix *obstacleOpenSpace* with all obstacle and open space pixels was computed before. At the beginning delta $x$ and delta $y$ are computed and $sx$ and $sy$ define the working octant. In Bresenham's algorithm the $xy$-symmetry is used for the octants. There are also two error variables $err$ and $e2$. A while loop starts and if the current point ($x0$, $y0$) is an obstacle, null is returned. Else the point is added to the line variable. The line variable is a list of all points of the straight diagonal line between the nodes. If the start point is the goal point, the algorithm is finished and the complete line is returned. Else $e2$ is defined as $2 \times err$. If $e2$ is higher than delta $y$, delta $y$ is added to $err$ and $x0$ is shifted $\pm 1$ pixel in $x$ direction, depending on the octant ($sx$). If $e2$ is lower than delta $x$, delta $x$ is added to $err$ and $y0$ is shifted $\pm 1$ pixel in $y$ direction, depending on the octant ($sy$). With the new $x0$ and $y0$ values the next iteration starts. Summarizing Bresenham's line algorithm shifts the start point pixel by pixel in $x$ and $y$ direction depending on the octant and error value in the goal point direction.

If in any direction a selected pixel of the shifting process of the rectangle is an obstacle pixel, the entry in the adjacency matrix is set to 0. It is also set to 0 if in direction 0 (right) or 4 (left) there is no obstacle in between the nodes for a distance of 200 pixels or more because the rectangle would fall down. In Figure 3.3 the nodes and edges of rectangle level 5 can be seen and Figure 3.3 helps to explain the different types of edges. In direction 0 (right) or 4 (left) the shifting is from the start node to the end node in $x$ direction and from the start node, the height of the rectangle along the $y$ direction. An edge example of a square edge in direction 0 (right) or 4 (left) is labeled as edge 1 in Figure 3.3. Edge 2 of Figure 3.3 is a horizontal rectangle edge in direction 0 (right) or 4 (left). For direction 2 (down) and 6 (up) the shifting is from the start node to the end node in $y$ direction and to left and right the width of the rectangle along the $x$ direction. At least one of both $x$ directions has to be successful. In Figure 3.3 a square edge in direction 2 (down) is edge 3, a vertical rectangle edge in direction 2 (down) is edge 4 and a vertical rectangle edge in direction 6 (up) is edge 5. The shifting of direction 1 (lower right) and 3 (lower left) is different. For each pixel of the diagonal line from top to down, the shifting starts from the pixel of the diagonal line, the width of the rectangle along the $x$ direction to the right. If it gets stuck and the shifting is not finished the rectangle is shifted the missing width from the pixel of the diagonal line to the left. An example of a square edge in direction 1 (lower right) is edge 6.

After the calculation of the edges there are two more steps. The first step is executed if an edge exists between two nodes with the following three properties:

- Both nodes lead to a fall-down node
- Direction 1 (lower right) or 3 (lower left)
- Distance lower than 150 pixels

Then the edge is set to 0 because the rectangle could get stuck in a gap. The second step is performed if there is a diamond without any edge so that it is unreachable. If this is the case, the edge calculation

**Algorithm 1** Bresenham's line algorithm

**Require:** Start point with $x0$ and $y0$, goal point with $x1$ and $y1$, *obstacleOpenSpace* is given

```
List pixels ← ⊘
int dx = abs(x1 - x0)
int sx = x0 < x1 ? 1 : -1
int dy = abs(y1 - y0)
int sy = y0 < y1 ? 1 : -1
int err = dx + dy;
int e2 ← 0;
while true do
    if obstacleOpenSpace[y0,x0] = true then
        return null
    end if
    pixels.Add(Array ← x0, y0)
    if x0 = x1 and y0 = y1 then
        break
    end if
    e2 = 2 × err
    if e2 > dy then
        err += dy;
        x0 += sx;
    end if
    if e2 < dx then
        err += dx;
        y0 += sy;
    end if
end while
return pixels
```

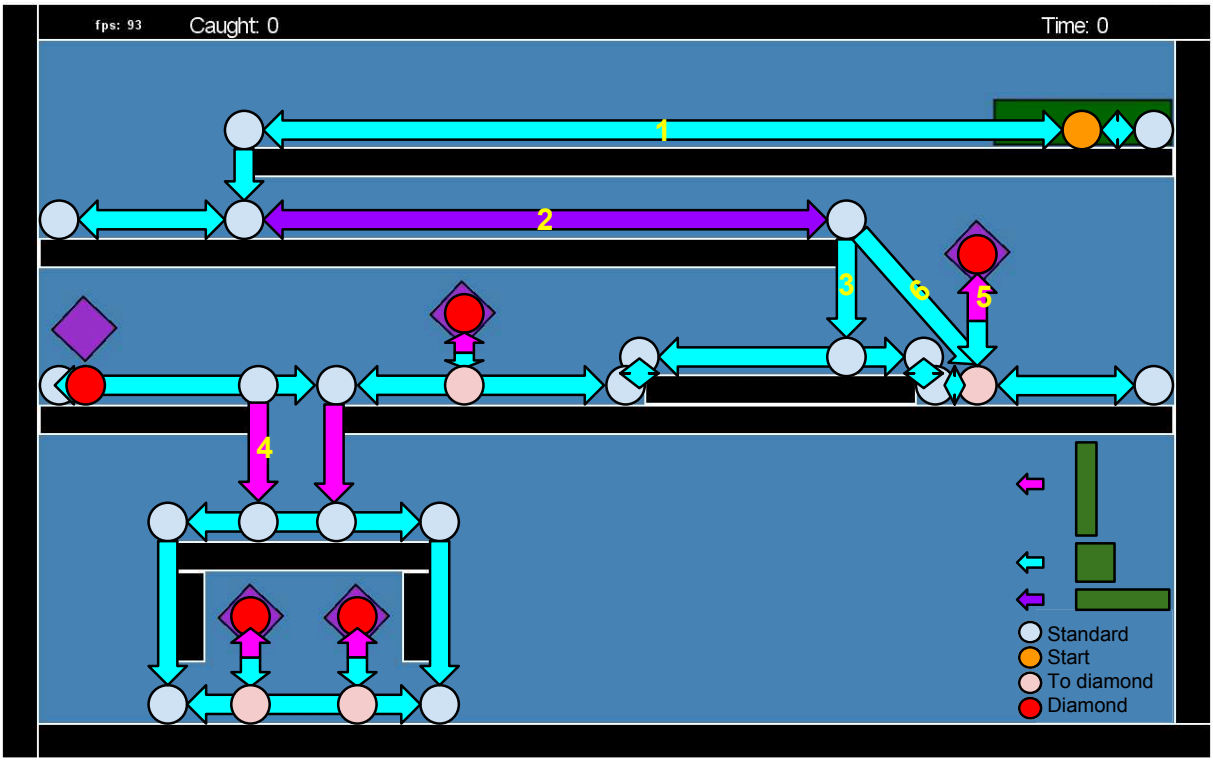


Figure 3.3: Nodes and edges of rectangle level 5

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 2 | 0 | 0 | 3 |
| 3 | 1 | 0 | 0 |

(a) Actions

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 7 | 0 |
| 2 | 3 | 0 | 2 |
| 3 | 4 | 6 | 0 |

(b) Directions

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 484 | 63 |
| 2 | 484 | 0 | 480 |
| 3 | 63 | 480 | 0 |

(c) Distances

Figure 3.4: Examples of three matrices

is performed again but the diamond is set to an ordinary node for the calculation process so that more edges are calculated this time.

Algorithm 2 shows the edge calculation. Algorithm 3 shows a method performed in Algorithm 2 called *CheckEdge*, which calculates the directions, distances and shifts different rectangle sizes.

**Algorithm 2** Calculation of edges

```
1: 2D Array adjacencyMatrix ← ∅
2: 2D Array directionMap ← ∅
3: 2D Array distanceMap ← ∅
4: for all n1 ∈ Nodes do
5:   for all n2 ∈ Nodes do
6:     if n1 = n2 then
7:       continue
8:     end if
9:     Array actionDirectionDistance = CheckEdge(n1, n2)
10:    adjacencyMatrix[n1 index, n2 index] = actionDirectionDistance[0]
11:    directionMap[n1 index, n2 index] = actionDirectionDistance[1]
12:    distanceMap[n1 index, n2 index] = actionDirectionDistance[2]
13:    if actionDirectionDistance[1] = Direction.Down and actionDirectionDistance[0] ≠ 0 then
14:      n1.setLeadsToFallDown(true)
15:    end if
16:  end for
17: end for
```

A simple example of the result of the calculation is given with Figure 3.4, which shows three matrices. In general, the node number of real Geometry Friends levels is between 10 and 30 nodes so that the matrices have a dimension of 10×10 to 30×30. This example has only three nodes. Figure 3.4 (a) is the adjacency matrix with all actions, Figure 3.4 (b) is the direction matrix and Figure 3.4 (c) shows the distance matrix. All diagonal fields are set to 0 because the node is not compared with itself. Field 3:1 in Figure (a) has value 1, which means the square player can reach node 3 from node 1. Field 3:1 in Figure (b) has value 0, which means the square has to go in direction right. Field 3:1 in Figure (c) has value 63, which means the distance between the nodes is 63 pixels. For Field 1:3 this is vice versa.

In Field 3:2 of Figure (a) the value is 3, which means the vertical rectangle player can reach node 3 from node 2. Field 3:2 in Figure (b) has value 2, which means the vertical rectangle has to go in direction down. Field 3:2 in Figure (c) has value 480, which means the distance between the nodes is 480 pixels. In this case it is not vice versa. There is no edge from node 3 to node 2. Direction 6 (up) and the distance of 480 pixels imply why. Because of the physical constraints it is not possible.

**Algorithm 3** Calculation of directions, distances and shifts different rectangle sizes: CheckEdge(Node n1, Node n2)

```
1: int Δx = n1.getX() - n2.getX()
2: int Δy = n1.getY() - n2.getY()
3: int edge ← 0
4: int direction ← 0
5: int distance ← 0
6: if Δx < 0 and Δy = 0 then
7:    direction = Direction.Right
8:    distance = Δx × -1
9: end if
10: if Δx < 0 and Δy < 0 and not(n1.isDiamond() or n2.isDiamond()) then
11:    direction = Direction.RightDown
12:    distance = √((Δx)² + (Δy)²)
13: end if
14: For all other directions the if cases are similar and skipped
15: if Δy ≥ 200 and Δx = 0 or Δy ≥ 50 and Δx ≠ 0 or direction = 6 and (Δy < 200 and > 75) and
    not(n2.isDiamond()) then
16:    bool obstacle = checkSquareSize(squareSize, direction, n1, n2)
17:    if not(obstacle) then
18:       return Array ← 1, direction, distance
19:    else
20:       obstacle = checkSquareSize(horizontalRectangleSize, direction, n1, n2)
21:    end if
22:    if not(obstacle) then
23:       return Array ← 2, direction, distance
24:    else
25:       obstacle = checkSquareSize(verticalRectangleSize, direction, n1, n2)
26:    end if
27:    if not(obstacle) then
28:       return Array ← 3, direction, distance
29:    end if
30: end if
31: return Array ← edge, direction, distance
```

# Chapter 4

# Search

After building the abstraction a search technique is used to calculate the most sophisticated route. This includes to collect all diamonds of a level in the least amount of time. It is more important collecting all diamonds than make a mistake while collecting because of driving too fast. The order of collecting is important because of the physics. If a diamond is at a high obstacle and the player starts at this obstacle, it has to collect this diamond first because if it falls down, it cannot reach the high diamond anymore. There are two different search techniques. A* and MCTS have been introduced in Section 1.3 and are now explained in detail. There is also an approach of combining both search techniques. The first section of this chapter (4.1) is about A* with different adaptations. It is explained how A* works in detail and which different node orders and heuristics are used. In Section 4.2 MCTS is explained in detail with its policies. Section 4.3 describes how and why both techniques are combined.

## 4.1 A* Versions

A* is able to find the minimum-cost route given a start and a goal node. In Geometry Friends the number of diamonds is between two and five so that there is not only one start and one goal node. Also the order of collecting the diamonds is important. Each adapted A* approach uses a different order of diamonds but the A* algorithm to calculate the route for a node pair is the same. Therefore A* is explained in detail at the beginning. Then the different A* versions are described.

### 4.1.1 A*

A* uses the node list and the distance map of the abstraction. It starts with a calculation of the $f$ score of the given start node. The $f$ score is calculated by the $g$ score, the true costs, from the initial node to the current node and adding the heuristic value $h$, the estimated costs, from the current node to the goal node. The heuristic, which is admissible, is computed by the Euclidean distance. The triple of node index, $f$ score and $g$ score is stored in a so-called open list. The open list includes all triples, which could be part of the shortest route while searching for the minimal route and have to be checked. The new current node is the node of the open list with the lowest $f$ score. The node with the lowest $f$ score is added to the closed list, which means that there exists no shorter route for this node. Next all neighbors of the current node are selected. If the neighbor is already in the closed list, the next neighbor is selected. Then the $g$ score of the neighbor is calculated by the $g$ score of the current node and the distance between the current node and the neighbor. A* finds the distance in the distance map. If the neighbor exists already in the open list, this instance of the neighbor is also selected. If the neighbor does not exist in the open list or the existing instance has a higher $g$ score, there are more steps. It is stored that the neighbor was reached by the current node, the neighbor's $f$ score is calculated and the triple of the neighbor is stored in the open list or updated if it already exists. This all is performed until the open list is empty or the selected node of the open list with the lowest $f$ score is the goal node. The segment of selecting the first entry of the open list until it is empty or the goal node is reached is shown in Algorithm 4. The complete route can be rebuilt by searching for each reached by node starting with the goal node. A* has the property to calculate the same route for the same level and A* is not optimal. It searches for

the minimum-cost route but this is not the best route to collect all diamonds in a level. There are levels where it is not possible to collect all diamonds with the shortest route between the diamonds.

**Algorithm 4** A* part of selecting the node with the lowest $f$ score and calculating new $f$ scores to determine the route

**Require:** start and goal node is set and start node with $f$ and $g$ score is added to open list

```
1: while openList.Count > 0 do
2:   Array current = LowestFScore();
3:   if current[0] = goal then
4:     return Route(current[0])
5:   end if
6:   closedList.Add(current)
7:   List neighbors = Neighbors(current[0])
8:   for all neighbor ∈ neighbors do
9:     if GetOfList(closedList, neighbor) ≠ null then
10:      continue
11:    end if
12:    int gScore = current[2] + distanceMap[current[0], neighbor]
13:    Array neighborInList = GetOfList(openlist, neighbor)
14:    if neighborInList = null or gScore < neighborInList[2] then
15:      cameFrom[neighbor] = current[0] {cameFrom is an array with the size of the number of nodes}
16:      int fScore = gScore + HeuristicValue(neighbor, goal)
17:      if neighborInList = null then
18:        openList.Add(Array ← neighbor, fScore, gScore)
19:      else
20:        ReplaceInOpenList(neighbor, fScore, gScore)
21:      end if
22:    end if
23:  end for
24: end while
```

### 4.1.2 Greedy Goal A*

Greedy Goal A* searches the shortest route for one diamond each time. This means to start from the initial node and select the shortest reachable diamond. Then from this diamond again search for the shortest reachable diamond until all diamonds are selected. The problem with this approach is that this greedy route is not always the best route where all diamonds can be collected. Figure 4.1 shows an example where Greedy Goal A* cannot collect the last diamond labeled as 3 because it starts with diamond 1 at the bottom and then the rectangle cannot get up to diamond 3 anymore. Algorithm 5 shows the Greedy Goal A*. It starts with adding all diamond indices to a list. While this list is not empty, A* is used to calculate the route between the start index, which is 0 at the beginning, and all other diamond nodes. Each route and its complete distance are stored. Then the shortest of these routes is selected and added to a list of the complete route. The start index is set to the goal node of the selected shortest route, the collected diamond is removed from the diamond indices list. The while loop continues if the diamond indices list is not empty. After the while loop, the individual routes are merged to one route. If there exists no route, an empty route is returned.

### 4.1.3 Y-Heuristic A*

To handle the issue of the Greedy Goal A* a heuristic is used, which takes the physics into account. The diamond with the highest $y$ coordinate is the first diamond to collect. Next are those diamonds with the next highest $y$ coordinates. The $x$ coordinate is not taken into account. Because of the $y$ coordinate heuristic this adapted A* is called Y-Heuristic A*. It is a reasonable heuristic and approach because the rectangle player has to start at the highest reachable diamond else he cannot reach it anymore. Having this diamond order, A* calculates the shortest route between two nodes pair by pair and all results are

**Algorithm 5** Greedy Goal A*

```
List diamondIndex ← ⊘
for int n = 0 to nodes.Count n++ do
    if nodes[n].isDiamond() then
        diamondIndex.Add(n)
    end if
end for
int startIndex ← 0
List queueList ← ⊘
while diamondIndex.Count > 0 do
    List queueListTemp ← ⊘
    List queueDistanceListTemp ← ⊘
    AStar astar
    for int i = 0 to diamondIndex.Count , i++ do
        astar ← AStar(startIndex, diamondIndex[i])
        queueListTemp.Add(astar.Run())
        queueDistanceListTemp.Add(astar.GetCompleteDistance())
    end for
    int shortest ← queueDistanceListTemp[0]
    for int i = 1 to queueDistanceListTemp.Count , i++ do
        if queueDistanceListTemp[i] < shortest then
            shortest = queueDistanceListTemp[i];
        end if
    end for
    if shortest ≠ int.MaxValue then
        int shortestIndex ← queueDistanceListTemp.IndexOf(shortest)
        queueList.Add(queueListTemp[shortestIndex])
        startIndex = diamondIndex[shortestIndex]
        diamondIndex.Remove(startIndex)
    end if
end while
if queueList.Count = 0 or queueList[0] = null then
    return  Queue ← ⊘
end if
List completeList ← ⊘
completeList.AddRange(queueList[0].ToList())
for int i = 1 to queueList.Count , i++ do
    List temp ← queueList[i].ToList()
    temp.RemoveAt(0)
    completeList.AddRange(temp)
end for
Queue completeQueue ← ⊘
for int i = 0 to completeList.Count , i++ do
    completeQueue.Enqueue(completeList[i])
end for
return  completeQueue
```

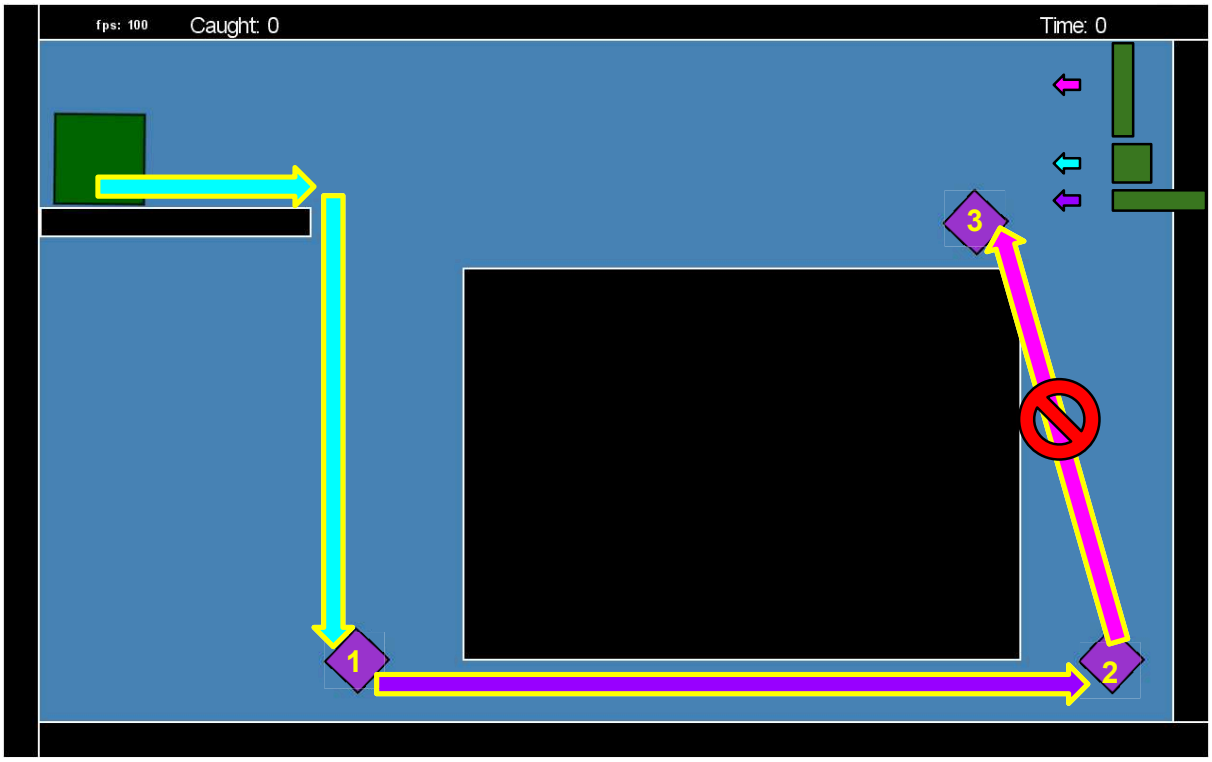


Figure 4.1: Rectangle level 7 with Greedy Goal A*

merged together. Figure 4.2 shows the calculated route of Y-Heuristic A* in rectangle level 5. Y-Heuristic A* has a similar problem like Greedy Goal A*. There may be levels, which the heuristic cannot solve. The algorithm of Y-Heuristic A* is not shown because it is similar to Greedy Goal A* except the calculation of the diamond order in descending $y$ coordinate order. For these diamonds A* calculates the route for each two diamonds in a row. At the end all routes are merged together. If there exists no route for a node pair this route is skipped. If there exists no complete route an empty route is returned.

### 4.1.4 Permutation A*

Another approach is to search the shortest route where the most diamonds can be collected over all partial permutations without repetition of the diamonds. This approach is called Permutation A*. For instance, if there are 5 diamonds in the level and all permutations are calculated, first the routes for all permutations with 5 diamonds are calculated with A*. Then the shortest route of all calculated routes is selected. It is possible that there are no routes to collect all 5 diamonds. Then the algorithm continues with all permutations with 4 diamonds and searches also for the shortest route if at least 1 route with 4 diamonds can be found. Otherwise, the algorithm continues with 3 diamonds. This approach ensures that the most diamonds with the shortest route are collected if the calculation time is not limited. Equation 4.1 shows the number of permutations for 5 diamonds.

$$\sum_{k=1}^{5} \frac{n!}{(n-k)!} = \frac{5!}{(5-1)!} + \frac{5!}{(5-2)!} + \frac{5!}{(5-3)!} + \frac{5!}{(5-4)!} + \frac{5!}{(5-5)!} = 325 \tag{4.1}$$

For 5 diamonds with $k$=*1* to $k$=*5* and $n$=*5* the result is 325 permutations. For the current maximum number of 5 diamonds the calculation time is acceptable but with more diamonds the calculation time would be insufficient.

### 4.1.5 Subgoal A*

The last approach of A* does not use any predefined diamond order. It is an adapted A* algorithm where the A* node has a new property and the heuristic is set to 0. The property stores the collected

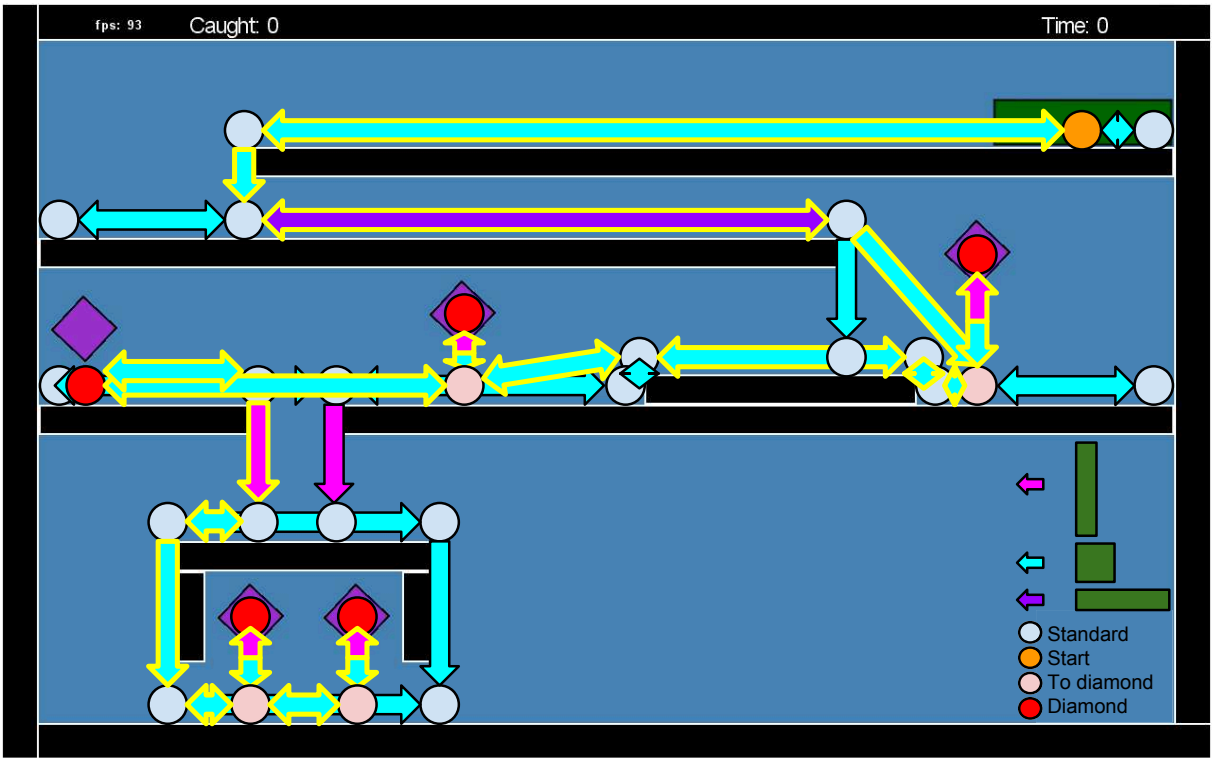


Figure 4.2: Calculated route of Y-Heuristic A* in yellow

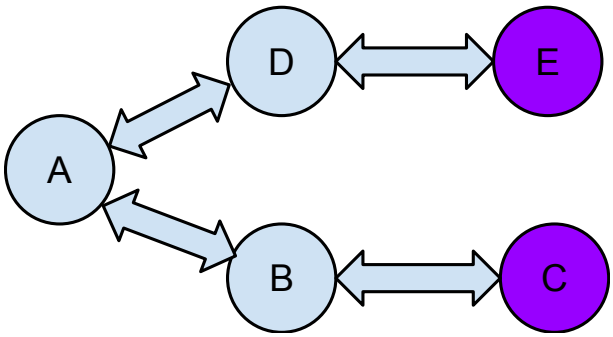


Figure 4.3: Subgoal A* graph example

diamonds and the order of the diamonds for each node. It is used to search for all diamonds with the ordinary A* pathfinding. An example is given with Figure 4.3. There are five nodes and two of them are diamond nodes. Without the new property A* cannot find both diamonds and would return only the route $A$, $B$, $C$ or $A$, $D$, $E$, depending on which diamond will be collected last. With the new property there are different states of $B$, for instance, $B$ without a diamond. $B$ with diamond collected at $C$. $B$ with diamond collected at $E$. These collected diamonds are stored in the new property. This means, if A* searches from $A$ to $B$ to $C$, A* will not stop but searches again back to $B$ because the first $B$ state does not contain the diamond collected at $C$ and in the current state the diamond was collected at $C$. This is equal for all other states so that A* will lead to the diamond at $E$ and returns a complete route from $A$ to $B$ to $C$ to $B$ to $A$ to $D$ to $E$. For complex levels it is possible that the search time is high so that the given number of diamonds to collect is decreased by one for instance every two seconds. It is also possible that in less than two seconds A* already finishes the search for all diamonds but could not find a route. Then the number of diamonds is also decreased by one and the algorithm starts again. Because this algorithm allows more than one goal node it is called Subgoal A*.

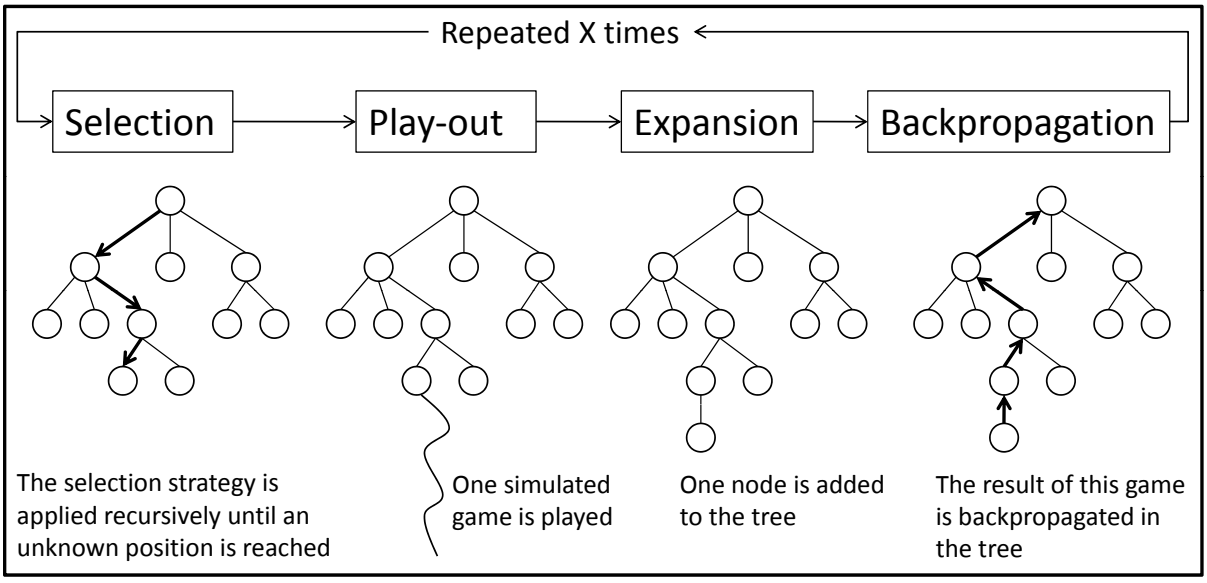


Figure 4.4: Four steps of Monte-Carlo Tree Search

## 4.2 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) (Coulom, 2007) determines the most promising action in four steps, which were explained in general in Subsection 1.3.1 and are shown in Figure 4.4.

MCTS uses the node list, the adjacency matrix with all actions and the number of diamonds of the level. A MCTS node has the following attributes:

- Array with its children
- Its parent node
- Attribute to count how often the node was visited
- A quality measurement value of the node
- The number of collected diamonds so far
- Attribute to know if the node is a diamond or not
- Attribute to know if the node is a terminal state or not

The algorithm starts from the root node and uses the selection policy until a leaf node is reached. Every time the child, which has the highest UCT value, is taken.

UCT means Upper Confidence Bound for Trees, which is the most popular selection policy algorithm (Kocsis and Szepesvári, 2006). The formula of UCT can be seen in Equation 4.2.

$$UCT = \overline{X}_c + C\sqrt{\frac{ln(n_p)}{n_c}} \tag{4.2}$$

In the equation a child $c$ is selected to maximize the value. $n_p$ describes the number of times the parent has been visited, $n_c$ is the number of times the child has been visited and $C$ is a constant. The second term describes the exploration. $\overline{X}_c$ is the average reward of child $c$, it can have a range of $[0, 1]$ and is the exploitation part.

Equation 4.3 shows the implemented and adapted UCT algorithm of the selection policy. The problem of the selection is to find a good ratio of exploration and exploitation. The algorithm ensures that all unvisited children are at least visited once before any child is investigated further.

$$UCT = \frac{AccRewardVal_c}{n_c + \epsilon} + \sqrt{\frac{ln(n_p + 1)}{n_c + \epsilon}} + (random(1) \times \epsilon) \tag{4.3}$$

$AccRewardVal_c$ is the current accumulated reward value of the child, which is given by the backpropagation, $n_c$ is the number of times the child was visited and $n_p$ is the number of times the parent was visited. The random value is between zero and one, and $\epsilon$ is used to add a random property for the selection of new children. It is also used to avoid divisions by zero visits at new children. $\epsilon$ has a value of 0.01. $\overline{X}_c$ of the general UCT equation is replaced by the accumulated reward value of the child divided by the sum of the visits of the child and $\epsilon$. The constant $C$ of the general UCT equation is set to 1. The exploitation part $\frac{AccRewardVal_c}{n_c+\epsilon}$ in the adapted UCT equation is not between 0 and 1 but this is not necessary.

Now back to MCTS and the selected leaf node. If the leaf node is a terminal state the algorithm stops. Otherwise the leaf node is expanded. The expansion means to create all children of the leaf node by using the actions of the adjacency matrix of the abstraction. For every child it is set whether it is a diamond and a terminal state. If it is a diamond, the number of collected diamonds is increased. Then one of the new children is selected by the play-out policy and the play-out starts. This policy chooses actions randomly. In the play-out actions are selected according to this policy. This selection is performed until a terminal state is reached or the length of the path in the play-out is higher than 10 times the number of nodes of the level. At the end of the play-out the value is calculated by Equation 4.4 (Kim *et al.*, 2014).

$$\frac{(N_{Collect} + 0.01) \times V_{Collect} \times avg(V_{Completed})}{length} \tag{4.4}$$

$N_{Collect}$ is the number of collected diamonds. $V_{Collect}$ describes the points given for each collected diamond. $V_{Completed}$ is the bonus for collecting all diamonds of a level. The length is the length of the path in the play-out, which is used to penalize a route with a longer path.

After calculating the value it is checked whether a new best route was found. Best route means a route where the most diamonds are collected or when the diamond number not changed and the calculated value is higher. For this a *bestCollected* and *bestValue* attribute was set to 0 at the beginning of MCTS. If the state at the end of the play-out has a higher number of collected diamonds than *bestCollected*, or a higher value than *bestValue* was calculated if the number of collected diamonds is equal to *bestCollected*, a new best route is found. The best route is calculated by selecting always the parent of each node until there is no parent. Also both attributes are set if a new best route is found. The last step is to backpropagate the calculated value and to increase the number of visits by one for all visited nodes except those of the play-out step. The complete process from the selection policy at the root node to the backpropagation is performed for a predetermined time. Algorithms 6 to 9 show the main process, the selection, the expansion and the play-out.

**Algorithm 6** MCTS main algorithm

```
1: LinkedList visited ← ⊘
2: MCTSNode current ← this
3: visited.addLast(this)
4: while not current.isLeaf() do
5:    current = current.select()
6:    visited.addLast(current)
7: end while
8: if current.endState then
9:    return
10: end if
11: current.expand()
12: MCTSNode newNode = current.select()
13: visited.addLast(newNode)
14: double value = playOut(newNode)
15: for all node ∈ visited do
16:    node.updateStats(value)
17: end for
```

Figure 4.5 shows the calculated route of MCTS in rectangle level 5. It is not the complete route of MCTS, the route is truncated. MCTS returns a lot more nodes than needed. A method called *ClearRoute* truncates all parts of the route where in between the same node there is no diamond (Algorithm 10). It

**Algorithm 7** MCTS selection process: select()

```
1: MCTSNode selected ← ⊘
2: double bestValue = -∞
3: for all child ∈ children do
4:    double uctValue = Equation 4.3
5:    if uctValue > bestValue then
6:       selected = child
7:       bestValue = uctValue
8:    end if
9: end for
10: return selected
```

**Algorithm 8** MCTS expansion process: expand()

```
1: List childIndex ← ⊘
2: for int i = 0 to Nodes.Count , i++ do
3:    if adjacencyMatrix[this, i] ≠ 0 then
4:       childIndex.add(i);
5:    end if
6: end for
7: Array children ← ⊘
8: for int i = 0 to childIndex.Count , i++ do
9:    MCTSNode child = ← ⊘
10:   child.diamond = Nodes[child].isDiamond()
11:   child.collected = collected
12:   if child.isDiamond() then
13:      if not visited.Contains(child) then
14:         child.collected++
15:         if child.collected = number of all diamonds then
16:            child.endState = true;
17:         end if
18:      end if
19:   end if
20:   children[i] = child
21: end for
```

**Algorithm 9** MCTS play-out process: playOut(MCTS node)

```
1: double value ← 0
2: int counter ← 1
3: while not node.endState and counter < Nodes.Count × 10 do
4:    node.expand()
5:    MCTSNode randomNode = node.select()
6:    node = randomNode
7:    counter++
8: end while
9: value = Equation 4.4
10: if MCTS.bestCollected < node.collected or (MCTS.bestValue < value and MCTS.bestCollected =
    node.collected) then
11:   MCTS.SetBestRoute(node);
12:   MCTS.bestCollected = node.collected;
13:   MCTS.bestValue = value;
14: end if
15: return value
```

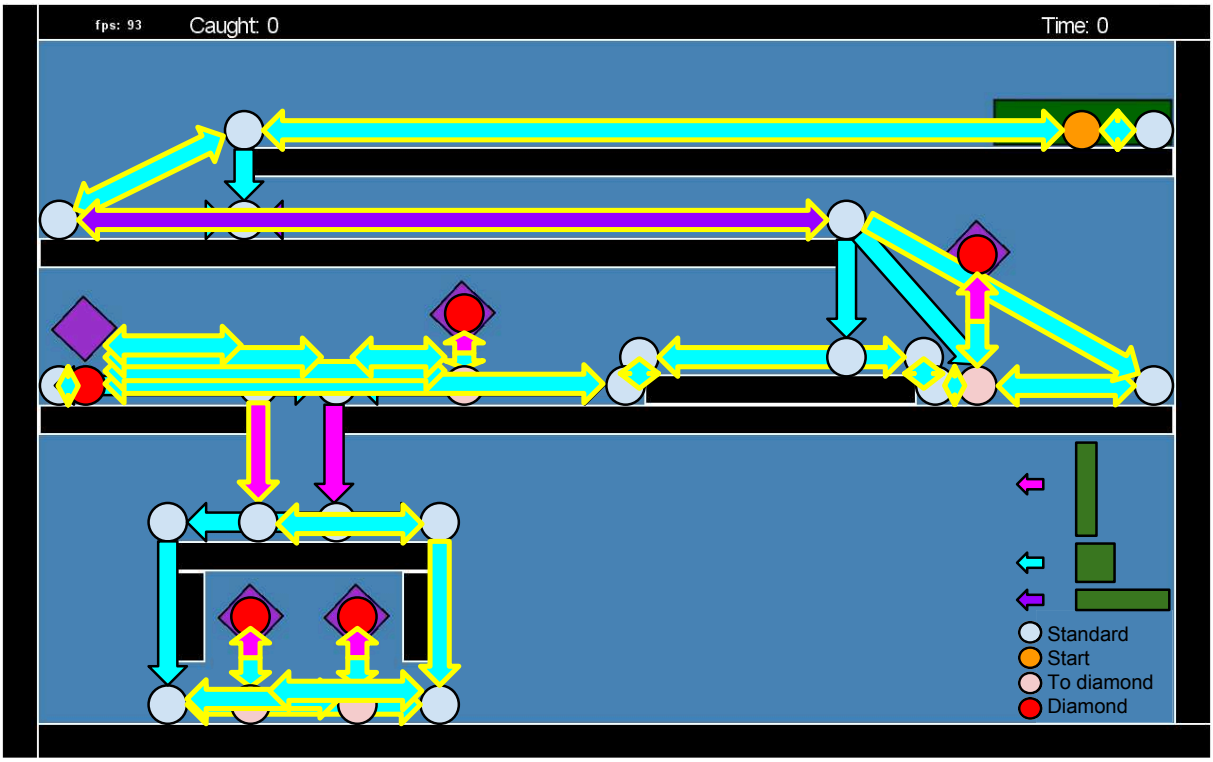


Figure 4.5: Calculated route of MCTS in yellow

is a simple but effective method to reduce the nodes of the route, for instance, in a complex level from 148 to 32. The benefit is different for each level and for every MCTS run. MCTS has the property to calculate similar but different routes for the same level.

**Algorithm 10** Truncate method for MCTS: ClearRoute(routeNodes)

```
1:  Queue shortRoute ← ⊘
2:  List visitedDiamond ← ⊘
3:  for int i = 0 to routeNodes.Length , i++ do
4:      shortRoute.Enqueue(routeNodes[i])
5:      int index ← 0
6:      for int j ← i+1 to routeNodes.Length , j++ do
7:          if routeNodes[j].isDiamond() and not visitedDiamond.Contains(routeNodes[j]) then
8:              break
9:          end if
10:         if routeNodes[i].Equals(routeNodes[j]) then
11:             index = j
12:         end if
13:     end for
14:     if index ≠ 0 then
15:         i = index
16:     end if
17: end for
18: return shortRoute
```

## 4.3 Monte-Carlo Tree Search & A*

While A* has the property to calculate the same route for the same level, which can be positive if it is a efficient route but really bad if it is a weak route. A weak route means a route where not all diamonds can be collected. MCTS has the property to calculate a similar but different route for the same level, which is positive in the way that it is unlikely to calculate a weak route every time but it is also unlikely to calculate a perfect route every time. This means MCTS is robust. Also MCTS has the problem of long routes, which can be solved by A*. Because of this situation when playing a level several times MCTS calculates the route, which guarantees that the order of diamonds takes into account physics with a possible different order and A* calculates the path between the diamonds, which resolves the problem of long routes. Algorithm 11 shows the combination of MCTS and A* and in Figure 4.6 the calculated route of MCTS & A* can be seen. The algorithm starts with performing MCTS and getting the long route. Then *ClearRoute* is used to truncate the long route into a shorter one. The route is converted from a *Queue* to a *List* to use a loop. A *List* is created to store the diamond nodes. For each node of the route it is checked whether it is a diamond. If the node is a diamond it is added to the diamond node list. At the end of the loop the diamond node list has all diamond nodes in the calculated order by MCTS. With A* a new route is calculated with the diamond node list, node pair by node pair. The route is calculated like in the A* adaptations with a given node ordering.

**Algorithm 11** Combination of MCTS & A*

```
1: MCTS mcts = MCTS(nodes, direction, distance)
2: Queue route = mcts.Run()
3: route = ClearRoute(route)
4: Array routeAsArray = route.ToArray()
5: List diamondNodes ← ∅
6: for int n = 0 to routeAsArray.Length n++ do
7:    if routeAsArray[n].isDiamond() then
8:       diamondNodes.add(routeAsArray[n])
9:    end if
10: end for
11: route = calcShortestRouteWithDiamondOrderAStar(diamondNodes)
```

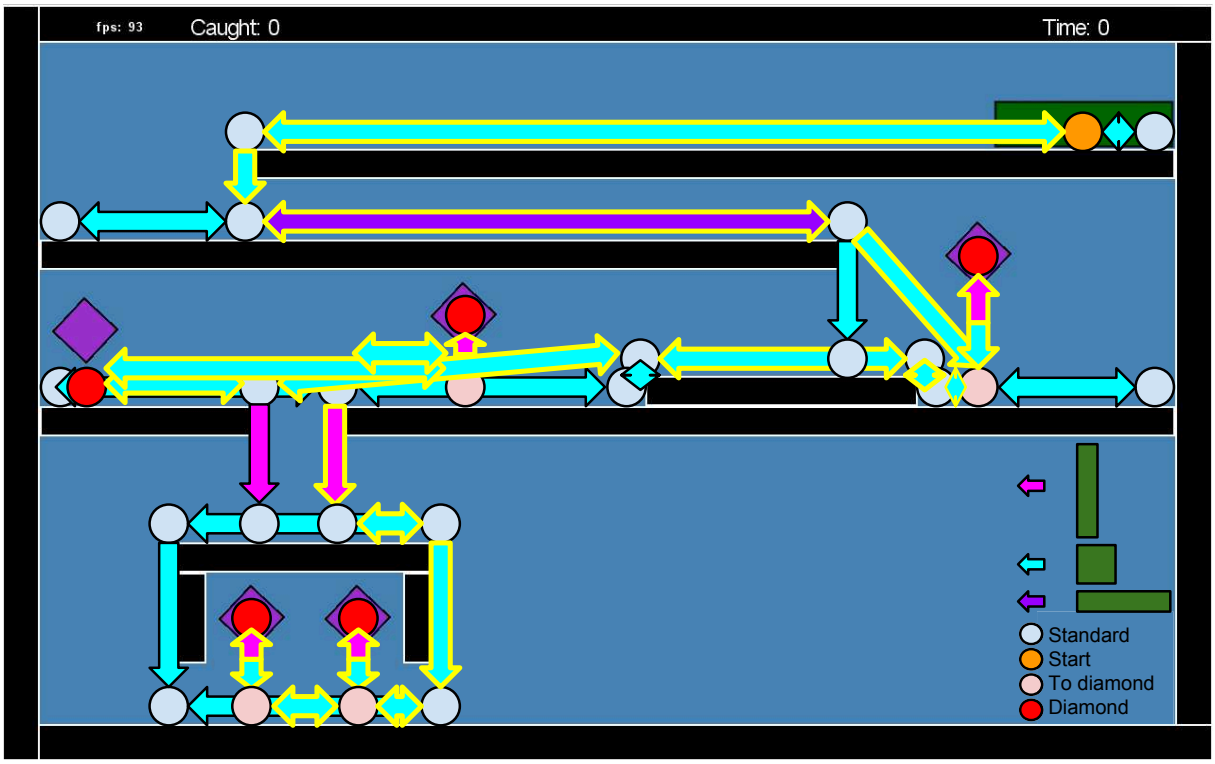


Figure 4.6: Calculated route of MCTS A* in yellow

# Chapter 5

# Driver

After calculating the route with the search algorithm the driver is required to follow this route in the level. The driver is created after the abstraction calculation and search. It needs the nodes, adjacency matrix, direction map of the abstraction and the route of the search technique. Looking back at Section 2.3, the *GetAction* method of the driver is performed in the *Update* method in the *SquareAgent* class. The rectangle executes the actions, which the driver returns. *GetAction* needs the current position of the rectangle and is called each time step (40 ms). Rules are used to check, which action should be executed next or whether a next node of the route is reached. The current position of the rectangle, the previous and next nodes, the previous and next actions, the previous and next directions, and the distance are variables to check which rule is applicable. In the next sections the general procedure (Section 5.1) and all rules of the driver (Section 5.2) are explained.

## 5.1 General Procedure

Before describing the procedure of the driver the following attributes of the driver are shown:

- *Previous node*, the node the driver started from
- *Next node*, the node the driver tries to reach
- *Next node 2*, the node the driver tries to reach after next node
- *Previous direction*, the direction, which was used to reach the previous node
- *Direction*, the direction to reach the next node
- *Direction 2*, the direction to reach the next node 2 from next node
- *Previous action*, the action, which was used to reach the previous node
- *Action*, the action to reach the next node
- *Action 2*, the action to reach the next node 2 from next node
- *Distance*, the distance between the current position of the rectangle and the next node
- *Distance list*, stores up to the last 40 distances
- *Run algorithm*, between 0 and 4 and determines, which algorithm is used for the recalculation of the route
- *Height of rectangle*
- *Width of rectangle*
- *Bottom y coordinate of rectangle*
- *Center x coordinate of rectangle*

- *Velocity in y direction*
- *Velocity in x direction*

When the driver is created it starts with setting the initial nodes, actions and directions attributes. The main method *GetAction* starts with setting the width, height, $x$ and $y$ coordinate, and $x$ and $y$ velocity of the rectangle. Next, the distance is calculated by the Euclidean distance and added to the distance list. If the distance list has a length of 40 and the first entry of the list is the same as the last entry of the list, the driver has gotten stuck somehow for 1.6 seconds. If the driver has gotten stuck, it recalculates the complete route by one of the following search techniques:

- 0 = MCTS and A*
- 1 = MCTS
- 2 = Y-Heuristic A*
- 3 = Greedy Goal A*
- 4 = Permutation A*
- 5 = Subgoal A*

The *run algorithm* attribute is set before starting the game. To recalculate the route the driver performs the same methods like described in Chapter 4 but beforehand the new initial rectangle position is set and the collected diamonds are set to non-diamond nodes. With the new route the new initial nodes, actions and directions attributes are set. The distance list is cleared if the length is 40.

The next step is to check whether the next node is reached. This is the case if:

- The distance is lower than the half width and the next node is not a pseudo node or
- The direction is 6 (up) and the distance is lower than the half height or
- The direction is 1 (lower right), 2 (down) or 3 (lower left) and the rectangle is at the $y$ coordinate of the next node and the distance is lower than 3 times the width or
- The next node is a pseudo node and the distance is lower than 3

If any of these conditions hold, the nodes, actions, directions and distance attributes are updated. If the new next node does not exist, the route is recalculated. If the new direction is 2 (down), this is a special case. The problem is that there is no sophisticated way to fall down through a gap with the given abstraction. At each obstacle's edge of the gap a node leads to a fall-down node in direction 2 (down). With the given abstraction the rectangle can only cross the gap and falls down somehow. The best way to fall down through a gap is to drive in the middle of the gap and then to resize. In the abstraction there is no node in the middle of a gap so a pseudo node is created in the middle of the gap. If the rectangle reaches the pseudo node, it resizes to a vertical rectangle and falls down. In order to prevent the rectangle to fall down before reaching the pseudo node, action 2 (horizontal rectangle) is used to reach the pseudo node.

After checking if the next node is reached there are different rules for different circumstances, which return a matching action to execute. These rules are explained in the next section.

## 5.2 Rules

A rule-based approach is used to return a matching action. This means there are many if else cases where different combinations of attributes hold and one of the rectangle's action is returned. The following rules describe which combination of attributes must hold and which action is executed then. It is also explained why a rule is used:

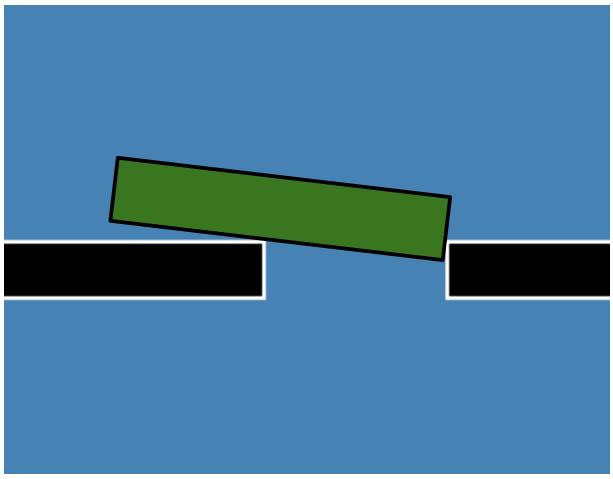

Figure 5.1: Get stuck at a gap

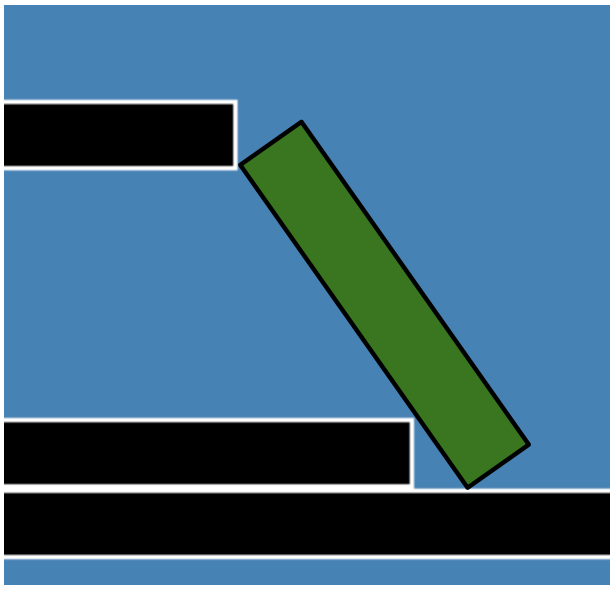

Figure 5.2: Get stuck in a diagonal position

- Rule 1 of the driver is to let the rectangle drive back if it gets stuck while driving to the pseudo node to fall down through a gap. Figure 5.1 shows the rectangle, which gets stuck. Rule 1 is used if the next node is a pseudo node and the first entry of the distance list and tenth entry of the distance list is the same. If the direction is to the right and the distance is lower than 200, to accelerate in direction left is returned. If the direction is to the left and the distance is lower than 200, to accelerate in direction right is returned.
- Rule 2 is to accelerate randomly to the left or right if the rectangle get stuck for 600 ms in a diagonal position. An example is shown in Figure 5.2. The diagonal position is checked by the method *IsDiagonalOrientation*.
- Rule 3 is to morph up the rectangle if it is in front of a higher obstacle. Then the rectangle can overturn to the higher obstacle if it drives in the direction of the higher obstacle (Figure 5.3). This rule is used if the previous direction is up, the direction is right or left and the difference between the $y$ coordinate of the previous node and the rectangle's $y$ coordinate is higher than 4, which means the rectangle is not on the higher obstacle. To morph up is returned if the height is lower than 160 and morphing up is possible without morph into an obstacle. Otherwise, if the direction is right, to accelerate in direction right is returned or if the direction is left, to accelerate in direction

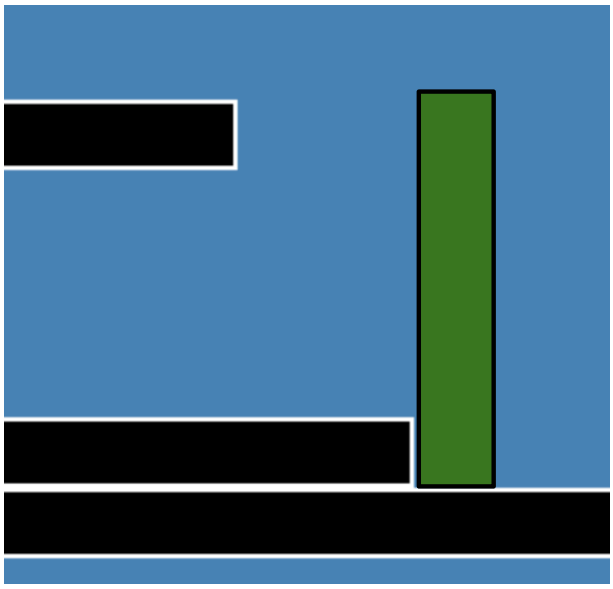

Figure 5.3: Overturn to a higher obstacle

left is returned.

- Rule 4 returns to morph down if the action is 1 (square), the height is higher than 102 and the direction is not up.
- Rule 5 returns to morph up if the action is 1 (square), the height is lower than 98 and it is possible to morph up.
- Rule 6 returns to morph down if the action is 2 (horizontal rectangle) and the height is higher than 52.
- Rule 7 holds if the action is 3 (vertical rectangle), the direction is not up, the height is lower than 194, morph up is possible, the rectangle is not in a diagonal position and the velocity in $y$ direction is lower than 5. It returns to morph up.
- Rule 8 is to accelerate the rectangle in direction right. It decelerates if it is too fast. It also takes into account the pseudo node to stop at the pseudo node. At first the direction has to be right, upper right or lower right. Then it returns to accelerate in the left direction to decelerate if the distance is lower than 110 and the velocity in $x$ direction is higher than 50 and direction2 is not right (drives not in right direction after reaching the next node). It also decelerates if the velocity in $x$ direction is higher than 200. Otherwise, it returns to accelerate in direction left if the next node is a pseudo node, the distance is lower than 12 and the velocity in $x$ direction is higher than 2. If the next node is a pseudo node, the distance is lower than 6 and the velocity in $x$ direction is between 2 and -2, the rule returns to do nothing. If both pseudo node cases are false, it is returned to accelerate in direction right.
- Rule 9 is the opposite of Rule 8 but for the left direction.
- Rule 10 is for the up direction. If the height is lower than 194 and morph up is possible, to morph up is returned. Otherwise, if the difference of the $x$ coordinate of the next node and the rectangle's $x$ coordinate is higher than the half width of the rectangle, the rule returns to accelerate in direction right if the difference is lower than 0. If the difference is equal to 0 or higher, to accelerate in direction left is returned. It is possible that the rectangle morphs up and does not reach the next node because the rectangle is too far away from the node. If this is the case, the rectangle has to drive in the direction of the next node.
- Rule 11. After the rectangle drives to the pseudo node and morphs up it falls down. It is possible that there is another gap under the first gap. Rule 11 ensures that the rectangle will not fall through

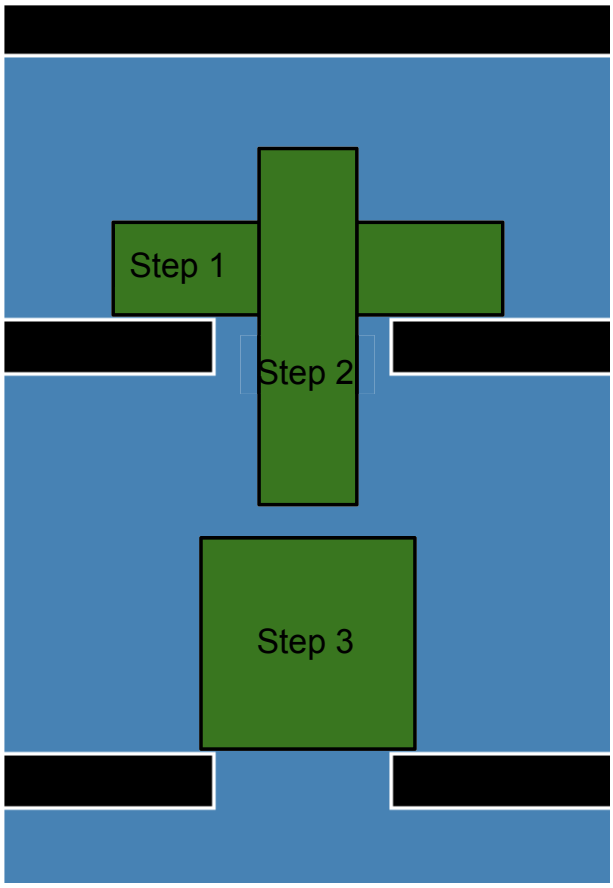


Figure 5.4: Morph down before fall through a second gap

the second gap because the rectangle morphs down if it is above the next node (Figure 5.4). In detail, morph down is returned if the direction is down, the next node leads to a fall-down node, the difference between the rectangle's $y$ coordinate and the previous node's $y$ coordinate is higher than width - 15, and the difference of the previous node's $y$ coordinate and the next node's $y$ coordinate is higher than 200. It also holds if the difference between the rectangle's $y$ coordinate and the previous node's $y$ coordinate is higher than width - 45, and the difference of the previous node's $y$ coordinate and the next node's $y$ coordinate is lower or equal 200. The difference is that for a longer fall down the rectangle morphs down later and for a short fall down the rectangle morphs down earlier.

- Rule 12 is to drive slowly right or left if the rectangle gets stuck at an edge of an obstacle. This can happen if the rectangle, for instance, drives to the left to an edge, reaches the next node and then decelerates. The next direction is down but the rectangle does not drive to the left anymore (Figure 5.5). The slowly driving is also used if the rectangle cannot reach the next node because he cannot morph up (Figure 5.5). In detail, rule 12 holds if the previous node is not a pseudo node, direction is down, previous direction is right or left, and the difference between the $y$ coordinate of the rectangle and the $y$ coordinate of the previous node is lower than 5. It also holds if the previous direction is right or left and morph up is not possible. Then if the previous direction is right and the velocity in $x$ direction is lower than 40, accelerate in direction right is returned. If the previous direction is left and the velocity in $x$ direction is lower than -40, accelerate in direction left is returned.

- Rule 13 is to prevent the rectangle to get stuck in a small gap between an edge of an obstacle and a wall (high obstacle) at the other side (Figure 5.6). Rule 13 holds if the previous node leads to a fall-down node, the previous node is not a pseudo node, the direction is lower right or down or lower left, morph up is possible, and the velocity in $x$ direction is higher than 50 or lower than -50.

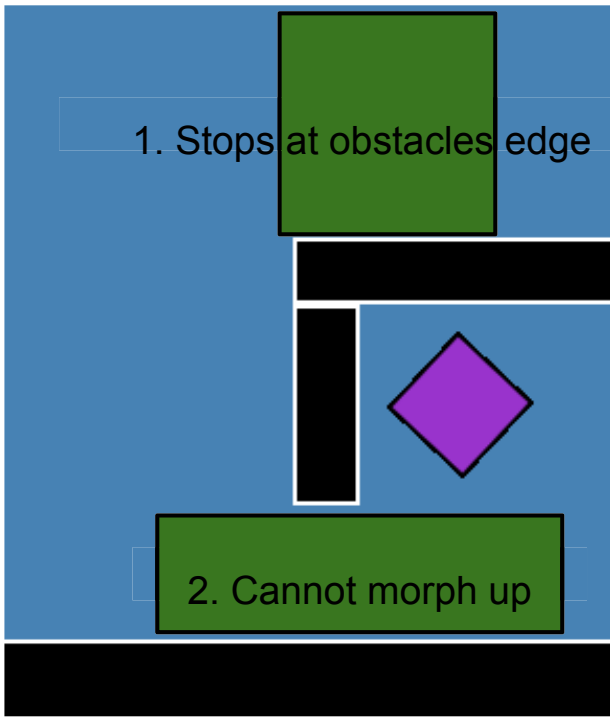


Figure 5.5: Two situations where the driver continued driving slowly

Then if the distance between the previous node and the wall is lower or equal to 125, morph up is returned. This leads to switching between morph up and down, which result is that the rectangle not gets stuck.

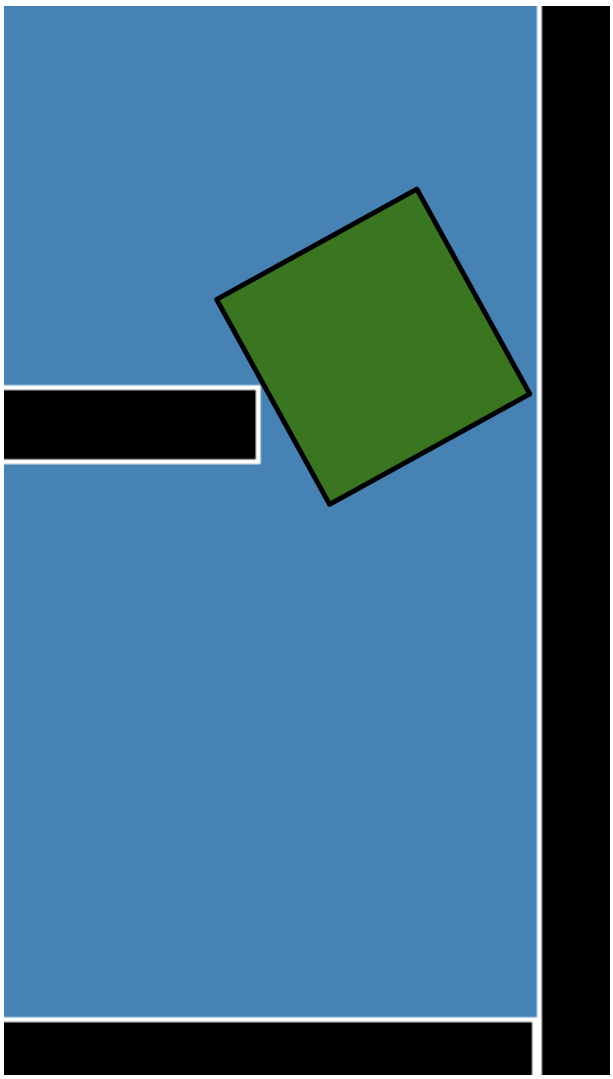

Figure 5.6: Get stuck between an edge of an obstacle and a wall

# Chapter 6

# Experiments and Results

After explaining all the techniques this chapter is about the experiments and their results. All techniques are tested in the given levels of the 2013 and 2014 competitions. Also new levels are created (See Appendix A) to have a more complex test set and for a better differentiation. The new levels are to challenge the weak spots of the different techniques. First the setup of the experiments is described (6.1). Then the results of the best agent of the 2013 and 2014 competitions (6.2.1), the results of the abstraction calculation (6.2.2) and the results of all techniques are explained (6.2.3). Finally, a discussion is provided (6.3).

## 6.1 Setup

The setup describes which techniques are used with which properties in which test set. For the experiments the following techniques are used and once more briefly summarized:

- **Greedy Goal A***: A* is used to search for the nearest diamond and then from this diamond to the next nearest diamond step by step until all diamonds are found. The route of each node pair is merged together to one route.
- **Y-Heuristic A***: The diamond order is given by the decreasing $y$ coordinate and A* searches for the route of each node pair. The route is merged together.
- **Permutation A***: All permutations are calculated and it starts with searching for the shortest route of all permutations with all diamonds. If this route does not exist, it continues with the permutations with the diamond number decreased by one. For each route of a permutation A* is used for each node pair and the complete route is merged together.
- **Subgoal A***: An extra property is added to the A* node to store the number and order of collected diamonds. This technique uses A* to search for the complete route with all diamonds or decreases the number of diamonds to find a route with fewer diamonds.
- **MCTS**: Uses Monte-Carlo Tree Search to find the best route.
- **MCTS & A***: Uses Monte-Carlo Tree Search to have the diamond order and calculates the route for each node pair with A* and merges the route together.

The test set consists of 22 levels. 10 levels of the Geometry Friends 2013 Competition, 2 new levels of the Geometry Friends 2014 Competition and 10 new levels. For the 2013 and 2014 competitions a formula (Chapter 2.1, Equation 2.1) is given to calculate a value, which describes the quality of a technique. In 2013 the completion bonus and the bonus for each collected diamond is different for each level. In more complex levels each diamond and the completion bonus is a higher value. In 2014 the completion bonus is 1000 and the bonus for each diamond is 100 for all levels. For the new levels only the collected diamonds and the run time is compared without calculating a quality measurement value. The new levels are created by taking into account the different weaknesses of the different techniques.

For MCTS and all four A* techniques the search time of a route is set to 2 seconds. MCTS stops the computation after 2 seconds but is also tested with 1000 iterations without a time limit. A* decreases

the number of diamonds to collect by one every 2 seconds. For Greedy Goal A*, Y-Heuristic A* and Permutation A* that means there is a time limit for each A* search but there is no time limit to find an order of the diamonds.

A problem is that while the calculation of the abstraction and route is running the timer of the game freezes and therefore the calculation time is not included in the time given by the game. So the calculation time of the abstraction and the search time to determine the route is measured separately. The actual idea was to create automatic test runs and a script was created but the given console start of the game has a different game behavior. Two levels could not be solved in several runs with the console start. With the start in Visual Studio there are no problems and the levels can be solved every time. But the console run has also the problem that it does not store how many diamonds are collected if not all are collected. For these reasons it is tested manually in Visual Studio and all levels are tested five times. Manual testing was rather time consuming and took 30 hours.

## 6.2 Performance

After the explanation of the setup this section is about the performance. The results of the best agent of the 2013 and 2014 competitions are described and the score of other participants in the 2014 competition are shown to see how well the best 2013/2014 competition agent performs. Then the focus is on the abstraction. The time of the calculation, the number of nodes and the number of diamonds are described for each level. At last, the results of all different search techniques are explained. The completed runs, the collected diamonds, the time to solve the level, the search time and the score of the competitions are shown for each competition level. For the new levels the completed runs, the collected diamonds, the time to solve a level and the search time are shown. Furthermore, an overview of all search techniques in all levels can be seen.

### 6.2.1 Best Agent 2013/2014

Now the results of the best agent (CIBot) in the 2013 and 2014 competitions are described to give an overview of its performance. In Table 6.1 the results of the 2013 competition of CIBot are shown. It was the only participant in 2013 and can solve the first 5 levels. It cannot solve the last 5 levels once. CIBot reaches a total score of 5741 in the Geometry Friends 2013 Competition.

Table 6.1: Best 2013 (CIBot) results for all 2013 competition levels

| Level | Completed Runs | Diamonds (Max) | Time (Limit) - sec | Score 2013 |
|---:|---:|---:|---:|---:|
| 1 | 10 | 2.0(2) | 11.4(40) | 686 |
| 2 | 10 | 2.0(2) | 8.9(25) | 411 |
| 3 | 8 | 2.6(3) | 35.7(80) | 1705 |
| 4 | 7 | 1.4(2) | 12.4(20) | 108 |
| 5 | 7 | 4.4(5) | 51.4(90) | 1636 |
| 6 | 0 | 2.0(3) | 60.0(60) | 400 |
| 7 | 0 | 1.8(3) | 35.0(35) | 198 |
| 8 | 0 | 0.2(3) | 65.0(65) | 64 |
| 9 | 0 | 0.0(2) | 60.0(60) | 0 |
| 10 | 0 | 1.3(4) | 85.0(85) | 533 |
| Total Score 2013 | | | | 5741 |

The CIBot team is also a participant in the 2014 competition. The improved 2014 CIBot version wins the competition of 2014 against two other participants. The 2014 agent can solve the first 6 and the last level successfully. 3 levels it cannot solve even once (Table 6.2). This time it reaches a score of 6466 but with other completion and diamond bonus so that it is not comparable to 2013.

Table 6.3 shows all three participants of the 2014 competition, CIBot, OPU_SCOM and KUAS-IS Lab. CIBot has 69 complete runs with an collecting average of 2.3 out of 2.9 diamonds. The number

Table 6.2: Best 2014 (CIBot) results for all 2014 competition levels

| Level | Completed Runs | Diamonds (Max) | Time (Limit) - sec | Score 2014 |
|---|---|---|---|---|
| 1 | 10 | 2.0(2) | 12.5(40) | 889 |
| 2 | 10 | 2.0(2) | 10.1(25) | 798 |
| 3 | 9 | 2.9(3) | 32.8(80) | 880 |
| 4 | 10 | 2.0(2) | 9.1(20) | 747 |
| 5 | 10 | 5.0(5) | 41.6(90) | 1037 |
| 6 | 10 | 3.0(3) | 22.0(60) | 934 |
| 7 | 0 | 2.0(3) | 35.0(35) | 200 |
| 8 | 0 | 0.0(3) | 35.0(35) | 0 |
| 11 | 0 | 1.0(2) | 40.0(40) | 100 |
| 12 | 10 | 3.0(3) | 20.9(85) | 881 |
| Total Score 2014 | | | | 6466 |

2.9 describes the maximum average number of diamonds to collect in the 2014 competition[1]. The total score is 6466. Compared to OPU_SCOM the difference is sufficient with 60 complete runs, 2 out of 2.9 diamonds and a total score of 5475. KUAS-IS Lab is by far behind the other two agents with 24 complete runs, obtaining 1.4 out of 2.9 diamonds and a total score of 2526.

Table 6.3: Overview of all 2014 agents in the 2014 competition

| Agent | Completed Runs | Diamonds (Max 2.9) | Score 2014 |
|---|---|---|---|
| CIBot | 69 | 2.3 | 6466 |
| OPU_SCOM | 60 | 2.0 | 5475 |
| KUAS-IS Lab | 24 | 1.4 | 2526 |

### 6.2.2 Abstraction

After showing the results of the previous competitions this subsection gives an overview about the abstraction performance with the abstraction calculation time, number of nodes and number of diamonds for each level. Tables 6.4 and 6.5 show the average abstraction calculation time in milliseconds, the number of nodes including the diamonds and only the number of diamonds. The calculation time for the competition levels is 50 ms to 252 ms with 10 to 30 nodes and 2 to 5 diamonds. The calculation time for the new levels is 53 ms to 241 ms with 16 to 43 nodes and 2 to 9 diamonds. Salient levels are the competition level 5 with 30 nodes and 5 diamonds, the competition level 10 with a calculation time of 252 ms, the new level 3 with a calculation time of 228 ms and 34 nodes, the new level 4 with a calculation time of 241 ms, 43 nodes and 9 diamonds, and the new level 8 with 37 nodes and 9 diamonds. The average abstraction calculation time is 123.6 ms and therefore is quite fast.

### 6.2.3 Search Techniques

The results of Greedy Goal A* can be seen in Tables 6.6 and 6.7. The level number, the completed runs, the collected diamonds and the maximum number of diamonds, the needed time of the completed runs and the time limit, the search time to determine the route in milliseconds, the 2013 competition score and the 2014 competition score can be seen for all competition levels in every first table of each technique. The second table shows only level number, the completed runs, the collected diamonds and the maximum number of diamonds, the needed time of the completed runs and the search time in milliseconds for all new levels. For 12 levels of the competitions Greedy Goal A* can solve 10 levels completely 5 out of 5 times and 2 levels not once. The search time is 3 ms for each competition level. For the 2013 competition the total score is 10063 and for the 2014 competition the total score is 7815. Greedy Goal A* can solve 6

[1]Link to the competition results: `http://gaips.inesc-id.pt:8081/geometryfriends/?page_id=364`

Table 6.4: Abstraction overview for all competition levels

| Level | Abstraction Calculation - ms | Nodes | Diamonds |
|---|---|---|---|
| 1 | 106 | 10 | 2 |
| 2 | 50 | 13 | 2 |
| 3 | 167 | 15 | 3 |
| 4 | 83 | 12 | 2 |
| 5 | 158 | 30 | 5 |
| 6 | 148 | 16 | 3 |
| 7 | 53 | 11 | 3 |
| 8 | 118 | 23 | 3 |
| 9 | 90 | 19 | 2 |
| 10 | 252 | 18 | 3 |
| 11 | 158 | 16 | 3 |
| 12 | 43 | 17 | 3 |

Table 6.5: Abstraction overview for all new levels

| Level | Abstraction Calculation - ms | Nodes | Diamonds |
|---|---|---|---|
| 1 | 91 | 24 | 4 |
| 2 | 75 | 16 | 4 |
| 3 | 228 | 34 | 4 |
| 4 | 241 | 43 | 9 |
| 5 | 95 | 19 | 3 |
| 6 | 175 | 25 | 3 |
| 7 | 53 | 16 | 3 |
| 8 | 127 | 37 | 9 |
| 9 | 93 | 21 | 3 |
| 10 | 115 | 21 | 2 |

of the new levels completely and 4 levels not once. The search time is 3 ms or 4 ms for each new level. It can collected 29 diamonds of all 44 diamonds in all new levels. For these 44 diamonds it is only possible to collect a maximum of 42 diamonds. There are levels where it is not possible to collect all diamonds but there is for instance a route to collect 3 instead of 1 diamond out of all 4 diamonds in the level.

Y-Heuristic A* (Tables 6.8 and 6.9) can solve 11 competition levels completely 5 out of 5 times. It cannot solve level 9 once. The search time is 2 ms or 3 ms. The 2013 competition score is 10469 and the 2014 competition score is 8585. The new levels also have a search time of 2 ms or 3 ms. Here it can solve 3 levels completely 5 out of 5 times and none of the others. In all new levels Y-Heuristic A* collects 33 diamonds.

Permutation A* (Tables 6.10 and 6.11) can solve 11 competition levels completely 5 out of 5 times. It cannot solve level 9 once. The search time is 3 ms or 4 ms and for competition level 5 11 ms. The 2013 competition score is 10382 and the 2014 competition score is 8565. The new levels have a search time between 3 ms and 6 ms except level 4 with 26935 ms and level 8 with 32477 ms. It can solve 7 levels completely 5 out of 5 times and none of the others. Permutation A* collects 39 diamonds in all new levels.

Subgoal A* (Tables 6.12 and 6.13) can solve 11 competition levels completely 5 out of 5 times. It cannot solve level 9 once. The search time is 2 ms and for competition level 5, 6 ms. The 2013 competition score is 10413 and the 2014 competition score is 8541. The new levels have a search time between 2 ms and 3 ms except level 4 with 2314 ms and level 8 with 4063 ms. It can solve 5 levels completely 5 out of 5 times, one more level completely 3 out of 5 times and none of the others. Subgoal A* collects 37.2 diamonds in all new levels.

MCTS with 2000 ms search time (Tables 6.14 and 6.15) can solve 3 competition levels completely 5 out of 5 times, 6 levels completely 1 to 4 out of 5 times. It cannot solve 3 levels once. Level 8 it can solve

Table 6.6: Greedy Goal A* results for all competition levels

| Level | Completed Runs | Diamonds (Max) | Time (Limit) - sec | Search Time - ms | Score 2013 | Score 2014 |
|---|---|---|---|---|---|---|
| 1 | 5 | 2.0(2) | 13(40) | 3 | 670 | 875 |
| 2 | 5 | 2.0(2) | 7(25) | 3 | 430 | 920 |
| 3 | 5 | 3.0(3) | 27(80) | 3 | 2040 | 963 |
| 4 | 5 | 2.0(2) | 11(20) | 3 | 145 | 650 |
| 5 | 5 | 5.0(5) | 47(90) | 3 | 1847 | 978 |
| 6 | 5 | 3.0(3) | 27(60) | 3 | 941 | 850 |
| 7 | 0 | 2.0(3) | 35(35) | 3 | 220 | 200 |
| 8 | 5 | 3.0(3) | 27(65/35) | 3 | 1515 | 529 |
| 9 | 0 | 0.6(2) | 60(60) | 3 | 186 | |
| 10 | 5 | 3.0(3) | 27(85) | 3 | 2069 | |
| 11 | 5 | 3.0(3) | 20(40) | 3 | | 900 |
| 12 | 5 | 3.0(3) | 21(50) | 3 | | 950 |
| Total Score 2013 | | | | | 10063 | |
| Total Score 2014 | | | | | | 7815 |

Table 6.7: Greedy Goal A* results for all new levels

| Level | Completed Runs | Diamonds (Max) | Time - sec | Search Time - ms |
|---|---|---|---|---|
| 1 | 5 | 4(4) | 41 | 3 |
| 2 | 0 | 1(3/4) | | 3 |
| 3 | 0 | 1(4) | | 3 |
| 4 | 0 | 1(9) | | 4 |
| 5 | 5 | 3(3) | 29 | 3 |
| 6 | 5 | 3(3) | 27 | 4 |
| 7 | 0 | 2(2/3) | | 3 |
| 8 | 5 | 9(9) | 31 | 4 |
| 9 | 5 | 3(3) | 16 | 3 |
| 10 | 5 | 2(2) | 25 | 3 |
| Collected Diamonds | | 29(42/44) | | |

completely 4 out of 5 times with the time limit of 65 seconds of the 2013 competition. With the time limit of 35 seconds of the 2014 competition it can solve level 8, twice. The route calculation is always 2000 ms. The 2013 competition score is 5598 and the 2014 competition score is 4505. It can solve 1 of the new levels completely 5 out of 5 times, 4 more levels 2 to 3 out of 5 times and none of the others. MCTS with 2000 ms search time collects 31.6 diamonds in all new levels.

MCTS with 1000 iterations (Tables 6.16 and 6.17) can solve 4 competition levels completely 5 out of 5 times, 5 levels completely 2 to 4 out of 5 times. It cannot solve 3 levels once. Level 8 it can solve completely 3 out of 5 times with the time limit of 65 seconds of the 2013 competition. With the time limit of 35 seconds of the 2014 competition it cannot solve level 8 once. The route calculation is between 135 ms and 887 ms except level 10 with 21027 ms. The 2013 competition score is 5520 and the 2014 competition score is 5178. It can solve 2 of the new levels completely 5 out of 5 times, 3 more levels 1 to 3 out of 5 times and none of the others. The new levels have a search time between 338 ms and 22139 ms. MCTS with 1000 iterations collects 30.2 diamonds in all new levels.

MCTS & A* with 2000 ms search time (Tables 6.18 and 6.19) can solve 7 competition levels completely 5 out of 5 times, 4 levels completely 1 to 4 out of 5 times. It cannot solve level 9 once. The route calculation is between 2006 ms and 2014 ms. The 2013 competition score is 9017 and the 2014 competition score is 7300. It can solve 2 of the new levels completely 5 out of 5 times, 5 more levels 1 to 4 out of 5 times and

Table 6.8: Y-Heuristic A* results for all competition levels

| Level | Completed Runs | Diamonds (Max) | Time (Limit) - sec | Search Time - ms | Score 2013 | Score 2014 |
|---|---|---|---|---|---|---|
| 1 | 5 | 2.0(2) | 13.0(40) | 2 | 670 | 875 |
| 2 | 5 | 2.0(2) | 7.0(25) | 2 | 430 | 920 |
| 3 | 5 | 3.0(3) | 27.0(80) | 2 | 2040 | 963 |
| 4 | 5 | 2.0(2) | 11.0(20) | 2 | 145 | 650 |
| 5 | 5 | 5.0(5) | 45.8(90) | 3 | 1864 | 991 |
| 6 | 5 | 3.0(3) | 27.0(60) | 3 | 941 | 850 |
| 7 | 5 | 3.0(3) | 12.0(35) | 2 | 547 | 957 |
| 8 | 5 | 3.0(3) | 27.0(65/35) | 2 | 1515 | 529 |
| 9 | 0 | 0.8(2) | 60.0(60) | 3 | 248 | |
| 10 | 5 | 3.0(3) | 27.0(85) | 3 | 2069 | |
| 11 | 5 | 3.0(3) | 20.0(40) | 2 | | 900 |
| 12 | 5 | 3.0(3) | 21.0(50) | 2 | | 950 |
| Total Score 2013 | | | | | 10469 | |
| Total Score 2014 | | | | | | 8585 |

Table 6.9: Y-Heuristic A* results for all new levels

| Level | Completed Runs | Diamonds (Max) | Time - sec | Search Time - ms |
|---|---|---|---|---|
| 1 | 0 | 2(4) | | 3 |
| 2 | 0 | 1(3/4) | | 2 |
| 3 | 0 | 3(4) | | 3 |
| 4 | 0 | 7(9) | | 3 |
| 5 | 5 | 3(3) | 29 | 2 |
| 6 | 0 | 2(3) | | 3 |
| 7 | 0 | 2(2/3) | | 2 |
| 8 | 5 | 9(9) | 26 | 3 |
| 9 | 0 | 2(3) | | 2 |
| 10 | 5 | 2(2) | 25 | 2 |
| Collected Diamonds | | 33(42/44) | | |

none of the others. MCTS & A* with 2000 ms search time collects 34 diamonds in all new levels.

MCTS & A* with 1000 iterations (Tables 6.20 and 6.21) can solve 7 competition levels completely 5 out of 5 times, 3 levels completely 2 to 4 out of 5 times. It cannot solve level 6 and 9 once. The route calculation is between 136 ms and 884 ms except level 10 with 20737 ms. The 2013 competition score is 8288 and the 2014 competition score is 7329. It can solve 2 of the new levels completely 5 out of 5 times, 4 more levels 2 to 4 out of 5 times and none of the others. The new levels have a search time between 624 ms and 21324 ms. MCTS & A* with 1000 iterations collects 33.6 diamonds in all new levels.

Table 6.22 gives an overview of all different techniques with their property, the total 2013 and 2014 competition scores, the collected diamonds of the new levels and the average search time in milliseconds for all levels. Y-Heuristic A*, Subgoal A* and Permutation A* have all high and similar competition scores. For the new levels the Permutation A* and the Subgoal A* are better than the Y-Heuristic A*. Next best technique is Greedy Goal A* and then MCTS & A*. The best agent of the 2013/2014 competition (CIBot) and MCTS do not show a strong performance.

Table 6.10: Permutation A* results for all competition levels

| Level | Completed Runs | Diamonds (Max) | Time (Limit) - sec | Search Time - ms | Score 2013 | Score 2014 |
|---|---|---|---|---|---|---|
| 1 | 5 | 2.0(2) | 13.0(40) | 3 | 670 | 875 |
| 2 | 5 | 2.0(2) | 7.0(25) | 3 | 430 | 920 |
| 3 | 5 | 3.0(3) | 27.0(80) | 4 | 2040 | 963 |
| 4 | 5 | 2.0(2) | 11.0(20) | 3 | 145 | 650 |
| 5 | 5 | 5.0(5) | 47.6(90) | 11 | 1839 | 971 |
| 6 | 5 | 3.0(3) | 27.0(60) | 4 | 941 | 850 |
| 7 | 5 | 3.0(3) | 12.0(35) | 4 | 547 | 957 |
| 8 | 5 | 3.0(3) | 27.0(65/35) | 4 | 1515 | 529 |
| 9 | 0 | 0.6(2) | 60.0(60) | 3 | 186 | |
| 10 | 5 | 3.0(3) | 27.0(85) | 4 | 2069 | |
| 11 | 5 | 3.0(3) | 20.0(40) | 3 | | 900 |
| 12 | 5 | 3.0(3) | 21.0(50) | 4 | | 950 |
| Total Score 2013 | | | | | 10382 | |
| Total Score 2014 | | | | | | 8565 |

Table 6.11: Permutation A* results for all new levels

| Level | Completed Runs | Diamonds (Max) | Time - sec | Search Time - ms |
|---|---|---|---|---|
| 1 | 5 | 4(4) | 41 | 4 |
| 2 | 0 | 1(3/4) | | 4 |
| 3 | 0 | 3(4) | | 5 |
| 4 | 5 | 9(9) | 51 | 26935 |
| 5 | 5 | 3(3) | 29 | 3 |
| 6 | 5 | 3(3) | 27 | 6 |
| 7 | 0 | 2(2/3) | | 4 |
| 8 | 5 | 9(9) | 22 | 32477 |
| 9 | 5 | 3(3) | 16 | 4 |
| 10 | 5 | 2(2) | 25 | 3 |
| Collected Diamonds | | 39(42/44) | | |

## 6.3 Discussion

The discussion starts with the abstraction calculation time and the number of nodes and diamonds for each level (Tables 6.4 and 6.5). Although the number of nodes and diamonds are, for instance, in the new levels 4 and 8 much higher than in the other levels, the abstraction calculation time does not increase significantly. That means a high number of nodes and diamonds does not imply the level is complex. Simple levels with fewer nodes and diamond can be more complex. This can be seen for the competition level 10 with 252 ms calculation time, 18 nodes and 3 diamonds, and the new level 8 with 127 ms calculation time, 37 nodes and 9 diamonds. The calculation complexity of a level depends on the possible edges between the nodes. The 12 competition levels are given by the competition and they have different difficulty levels. The 10 new levels are created to challenge the different implemented techniques. For instance, there are levels where following the shortest route means that not all diamonds can be collected or that a lower diamond have to be collected first to collect more diamonds. There are also levels where not all diamonds can be collected but there is one path where, for instance, only one diamond can be collected and another path where three diamonds can be collected. Two levels have 9 diamonds, which are to distinguish between efficient and inefficient techniques. So for each technique there are levels to challenge their weak spots.

Table 6.12: Subgoal A* 2000ms results for all competition levels

| Level | Completed Runs | Diamonds (Max) | Time (Limit) - sec | Search Time - ms | Score 2013 | Score 2014 |
|---|---|---|---|---|---|---|
| 1 | 5 | 2.0(2) | 13.0(40) | 2 | 670 | 875 |
| 2 | 5 | 2.0(2) | 7.0(25) | 2 | 430 | 920 |
| 3 | 5 | 3.0(3) | 27.0(80) | 2 | 2040 | 963 |
| 4 | 5 | 2.0(2) | 11.0(20) | 2 | 145 | 650 |
| 5 | 5 | 5.0(5) | 49.8(90) | 6 | 1808 | 947 |
| 6 | 5 | 3.0(3) | 27.0(60) | 2 | 941 | 850 |
| 7 | 5 | 3.0(3) | 12.0(35) | 2 | 547 | 957 |
| 8 | 5 | 3.0(3) | 27.0(65/35) | 2 | 1515 | 529 |
| 9 | 0 | 0.8(2) | 60.0(60) | 2 | 248 | |
| 10 | 5 | 3.0(3) | 27.0(85) | 2 | 2069 | |
| 11 | 5 | 3.0(3) | 20.0(40) | 2 | | 900 |
| 12 | 5 | 3.0(3) | 21.0(50) | 2 | | 950 |
| Total Score 2013 | | | | | 10413 | |
| Total Score 2014 | | | | | | 8541 |

Table 6.13: Subgoal A* 2000ms results for all new levels

| Level | Completed Runs | Diamonds (Max) | Time - sec | Search Time - ms |
|---|---|---|---|---|
| 1 | 5 | 4(4) | 41.0 | 2 |
| 2 | 0 | 1(3/4) | | 3 |
| 3 | 0 | 3(4) | | 3 |
| 4 | 3 | 8.2(9) | 55.7 | 2314 |
| 5 | 0 | 2(3) | | 2 |
| 6 | 5 | 3(3) | 27.0 | 2 |
| 7 | 0 | 2(2/3) | | 2 |
| 8 | 5 | 9(9) | 26.8 | 4063 |
| 9 | 5 | 3(3) | 16 | 2 |
| 10 | 5 | 2(2) | 25 | 2 |
| Collected Diamonds | | 37.2(42/44) | | |

In the 2013 competition CIBot can solve the first five levels well but is not able to solve the last five levels even once (Table 6.1). It reaches a total score of 5741 and wins the competition. In the 2014 competition the same but improved agent can win again with 6466 points with a different score calculation (Table 6.2). The improvements are significant, it can solve 7 out of 10 levels nearly perfectly. The remaining 3 levels it cannot solve completely a single time. CIBot is more sophisticated than the other two participants of the 2014 competition (Table 6.3).

The Greedy Goal A* has good results, 10 out of 12 competition levels it can solve completely 5 out of 5 times (Table 6.6). Level 7 it cannot solve completely because it searches for the shortest route where it jumps down from a platform and it cannot reach a diamond anymore, which it has to collect first. In level 9 the driver is not able to fall down through a gap because there is another platform above the gap at the right side so that to morph up at the middle of the gap is not possible. This is a driver issue so that it occurs at each technique. With a total score of 10063 for the 2013 competition and 7815 for the 2014 competition Greedy Goal A* is significantly better than the CIBot agent. In the new levels it can solve 6 out of 10 levels completely 5 out of 5 times (Table 6.7). In level 2, 3 and 4 the technique has problems because of selecting the shortest route. With a longer route it is possible to collect more diamonds. In level 7 it collects the maximum number of reachable diamonds. Greedy Goal A* collects 29 diamonds, which is not a good result compared to the other techniques. The search time is very low

Table 6.14: MCTS 2000ms results for all competition levels

| Level | Completed Runs | Diamonds (Max) | Time (Limit) - sec | Search Time - ms | Score 2013 | Score 2014 |
|---|---|---|---|---|---|---|
| 1 | 5 | 2.0(2) | 20.0(40) | 2000 | 600 | 700 |
| 2 | 5 | 2.0(2) | 14.4(25) | 2000 | 356 | 624 |
| 3 | 2 | 2.2(3) | 48.5(80) | 2000 | 1069 | 378 |
| 4 | 5 | 2.0(2) | 14.4(20) | 2000 | 128 | 480 |
| 5 | 2 | 3.8(5) | 61(90) | 2000 | 1111 | 509 |
| 6 | 0 | 1.0(3) | 60.0(60) | 2000 | 200 | 100 |
| 7 | 1 | 2.0(3) | 15.0(35) | 2000 | 258 | 314 |
| 8 | 4/2 | 2.8/2.2(3) | 37.5/29.3(65/35) | 2000 | 1218 | 285 |
| 9 | 0 | 0.8(2) | 60.0(60) | 2000 | 248 | |
| 10 | 0 | 1.0(3) | 85.0(85) | 2000 | 410 | |
| 11 | 3 | 2.6(3) | 31.0(40) | 2000 | | 395 |
| 12 | 4 | 2.8(3) | 22.5(50) | 2000 | | 720 |
| Total Score 2013 | | | | | 5598 | |
| Total Score 2014 | | | | | | 4505 |

Table 6.15: MCTS 2000ms results for all new levels

| Level | Completed Runs | Diamonds (Max) | Time - sec | Search Time - ms |
|---|---|---|---|---|
| 1 | 0 | 2.0(4) | | 2000 |
| 2 | 0 | 2.6(3/4) | | 2000 |
| 3 | 0 | 1.8(4) | | 2000 |
| 4 | 3 | 8.4(9) | 78.0 | 2000 |
| 5 | 0 | 2.0(3) | | 2000 |
| 6 | 2 | 2.2(3) | 28.0 | 2000 |
| 7 | 0 | 1.0(2/3) | | 2000 |
| 8 | 2 | 7.4(9) | 46.5 | 2000 |
| 9 | 3 | 2.2(3) | 30.7 | 2000 |
| 10 | 5 | 2.0(2) | 25.6 | 2000 |
| Collected Diamonds | | 31.6(42/44) | | |

for all levels with 3 ms to 4 ms.

Y-Heuristic A* has the best results with a score of 10469 in the 2013 competition and 8585 in the 2014 competition (Table 6.8). It has also the fastest search time with 2 ms to 3 ms. It can solve all competition levels completely 5 out of 5 times except level 9 because of the driver issue explained before. It can only solve 3 of the new levels completely 5 out of 5 times but the number of all collected diamonds is 33 and higher than with Greedy Goal A* (Table 6.9). In general Y-Heuristic A* performs better than Greedy Goal A* in the new levels but, for instance, in level 4 or 9 it collects not all diamonds because of the $y$ coordinate order of diamonds.

Permutation A* has a competition score as good as Y-Heuristic A*. The small differences can be explained due to noise of the game environment. It has a 2013 score of 10382 and a 2014 score of 8565 (Table 6.10). It scores best in the new levels with collecting 39 diamonds and solving 7 levels completely 5 out of 5 times (Table 6.11). This is reasonable because Permutation A* searches for the shortest route to collect the most diamonds over all possibilities. It cannot find the best route in level 2 of the new levels. Permutation A* has a problem, which can be seen in level 4 and 8 of the new levels. With a high number of diamonds the search time explodes to 26935 ms and 32477 ms and is not acceptable. With the number of diamonds of the competition levels the search time is good with 3 ms to 11 ms.

Subgoal A* with a time limit of 2000 ms lies between Y-Heuristic A* and Permutation A*. Subgoal

Table 6.16: MCTS 1000iter. results for all competition levels

| Level | Completed Runs | Diamonds (Max) | Time (Limit) - sec | Search Time - ms | Score 2013 | Score 2014 |
|---|---|---|---|---|---|---|
| 1 | 5 | 2.0(2) | 20.8(40) | 413 | 592 | 680 |
| 2 | 5 | 2.0(2) | 13.2(25) | 135 | 368 | 672 |
| 3 | 2 | 1.6(3) | 33(80) | 763 | 922 | 395 |
| 4 | 5 | 2.0(2) | 13.2(20) | 294 | 134 | 540 |
| 5 | 3 | 3.8(5) | 52.0(90) | 760 | 1267 | 633 |
| 6 | 0 | 1.2(3) | 60.0(60) | 660 | 240 | 120 |
| 7 | 4 | 2.8(3) | 18.8(35) | 802 | 430 | 650 |
| 8 | 3/0 | 2.2/1.6(3) | 48.7/35(65/35) | 725 | 847 | 160 |
| 9 | 0 | 1.0(2) | 60.0(60) | 236 | 310 | |
| 10 | 0 | 1.0(3) | 85.0(85) | 21027 | 410 | |
| 11 | 5 | 3.0(3) | 27.8(40) | 668 | | 605 |
| 12 | 4 | 2.8(3) | 22.3(50) | 887 | | 723 |
| Total Score 2013 | | | | | 5520 | |
| Total Score 2014 | | | | | | 5178 |

Table 6.17: MCTS 1000iter. results for all new levels

| Level | Completed Runs | Diamonds (Max) | Time - sec | Search Time - ms |
|---|---|---|---|---|
| 1 | 0 | 2.0(4) | | 3706 |
| 2 | 0 | 3.0(3/4) | | 871 |
| 3 | 0 | 1.0(4) | | 21723 |
| 4 | 1 | 6.6(9) | 67.0 | 2358 |
| 5 | 1 | 2.2(3) | 35.0 | 22139 |
| 6 | 5 | 3.0(3) | 30.0 | 1104 |
| 7 | 0 | 1.0(2/3) | | 9757 |
| 8 | 3 | 8.0(9) | 83.0 | 1199 |
| 9 | 0 | 1.4(3) | | 338 |
| 10 | 5 | 2.0(2) | 24.8 | 981 |
| Collected Diamonds | | 30.2(42/44) | | |

A* reaches a 2013 competition score of 10413 and a 2014 competition score of 8541 (Table 6.12). For the competition levels the search time is good with 6 ms for level 5 and 2 ms for all others. Subgoal A* 2000 ms collects 37.2 diamonds in the new levels and has the second best result (Table 6.13). It solves 5 levels completely 5 out of 5 times and level 4 3 out of 5 times. The search time for the new level 4 is 2314 ms and 4063 ms for level 8. It is a higher search time but acceptable for a high number of diamonds and a good result. The search time for the other levels is 2 ms to 3 ms.

MCTS with 2000 ms search time has the worst results with a score of 5598 in the 2013 competition and 4505 in the 2014 competition (Table 6.14). It can solve 3 levels of the competition completely 5 out of 5 times and 6 levels 1 to 4 out of 5 times. The general problem of MCTS is a longer route, which is more time consuming and it is, for instance, more likely that the rectangle get stuck in a gap. In level 8 the time limit is reduced to 35 seconds for the 2014 competition and therefore 2 complete runs are excluded. For the search time of 2000 ms compared to the A* search time the results are insufficient. For the new levels the result is not as bad as the result for the competitions, it is better than the Greedy Goal A* with 31.6 diamonds (Table 6.15). It can only solve level 10 completely 5 out of 5 times and 4 other levels 2 to 3 out of 5 times. But it is noticeable that it can solve level 2 with 2.6 diamonds and all other A* techniques have only 1 diamond. These results show the advantage and the disadvantage of MCTS. MCTS can find both the best and the worst routes. A* is deterministic and computes the same

Table 6.18: MCTS & A* 2000ms results for all competition levels

| Level | Completed Runs | Diamonds (Max) | Time (Limit) - sec | Search Time - ms | Score 2013 | Score 2014 |
|---|---|---|---|---|---|---|
| 1 | 5 | 2.0(2) | 13.0(40) | 2006 | 670 | 875 |
| 2 | 5 | 2.0(2) | 8.8(25) | 2006 | 412 | 848 |
| 3 | 5 | 3.0(3) | 27.0(80) | 2006 | 2040 | 963 |
| 4 | 5 | 2.0(2) | 11.0(20) | 2006 | 145 | 650 |
| 5 | 5 | 5.0(5) | 53.2(90) | 2007 | 1761 | 909 |
| 6 | 1 | 1.4(3) | 27.0(60) | 2006 | 348 | 250 |
| 7 | 4 | 2.8(3) | 14.0(35) | 2006 | 466 | 760 |
| 8 | 5 | 3.0(3) | 27.0(65/35) | 2006 | 1515 | 529 |
| 9 | 0 | 0.8(2) | 60.0(60) | 2006 | 248 | |
| 10 | 3 | 2.2(3) | 26.3(85) | 2014 | 1412 | |
| 11 | 5 | 3.0(3) | 20.8(40) | 2007 | | 780 |
| 12 | 4 | 2.8(3) | 21.5(50) | 2009 | | 736 |
| Total Score 2013 | | | | | 9017 | |
| Total Score 2014 | | | | | | 7300 |

Table 6.19: MCTS & A* 2000ms results for all new levels

| Level | Completed Runs | Diamonds (Max) | Time - sec | Search Time - ms |
|---|---|---|---|---|
| 1 | 2 | 2.8(4) | 33.0 | 2009 |
| 2 | 0 | 1.8(3/4) | | 2008 |
| 3 | 0 | 2.8(4) | | 2007 |
| 4 | 4 | 8.4(9) | 51.3 | 2010 |
| 5 | 5 | 3.0(3) | 29.0 | 2016 |
| 6 | 2 | 2.4(3) | 26.0 | 2006 |
| 7 | 0 | 1.0(2/3) | | 2009 |
| 8 | 3 | 8.2(9) | 64.3 | 2008 |
| 9 | 1 | 1.6(3) | 43.0 | 2006 |
| 10 | 5 | 2.0(2) | 21.0 | 2008 |
| Collected Diamonds | | 34.0(42/44) | | |

route every time.

MCTS with fixed 1000 iterations is as bad as MCTS with 2000 ms but with a higher 2014 competition score of 5178. The score of 2014 is much higher because the score of level 7 and 11 are much higher. This is not because of the technique differences, it is simple luck. The 2013 competition score is 5520 (Table 6.16). It can solve 4 of the competition levels completely 5 out of 5 times and 5 levels 2 to 4 out of 5 times. The search time is between 135 ms and 887 ms except level 10 with 21027 ms. MCTS 2000 ms should have better results because it has in general more search time with more iterations but this cannot be seen in the results. Level 10 has an unacceptable search time with 21027 ms, the other search times are acceptable with 135 ms to 887 ms but are much higher than A* times, and MCTS has worse results. For the new levels the search times are worse with 338 ms to 22139 ms. But the score of the new levels is 30.2 so better than Greedy Goal A* but worse than MCTS 2000 ms (6.17). It solves 2 of the new levels completely 5 out of 5 times and 3 levels 1 to 3 out of 5 times. It has the highest score of level 2 with 3 diamonds.

MCTS & A* with 2000 ms reduces the route length of MCTS by A* and can improve the results of MCTS so that the 2013 competition score is 9017 and the 2014 competition score is 7300 (Table 6.18). The scores are between the MCTS results and Greedy Goal A*. In general, A* helps MCTS to reduce the time to solve a level so that the level gives a higher score and it is less likely to get stuck somehow.

Table 6.20: MCTS & A* 1000iter. results for all competition levels

| Level | Completed Runs | Diamonds (Max) | Time (Limit) - sec | Search Time - ms | Score 2013 | Score 2014 |
|---|---|---|---|---|---|---|
| 1 | 5 | 2.0(2) | 13.0(40) | 270 | 670 | 875 |
| 2 | 5 | 2.0(2) | 8.8(25) | 136 | 412 | 848 |
| 3 | 4 | 2.8(3) | 27.0(80) | 624 | 1756 | 810 |
| 4 | 5 | 2.0(2) | 10.6(20) | 204 | 147 | 670 |
| 5 | 5 | 5.0(5) | 48.4(90) | 884 | 1828 | 962 |
| 6 | 0 | 1.0(3) | 60.0(60) | 679 | 200 | 100 |
| 7 | 4 | 2.8(3) | 13.5(35) | 812 | 470 | 771 |
| 8 | 5 | 3.0(3) | 27.0(65/35) | 718 | 1515 | 529 |
| 9 | 0 | 1.0(2) | 60.0(60) | 425 | 310 | |
| 10 | 2 | 1.8(3) | 26.0(85) | 20737 | 1080 | |
| 11 | 5 | 3.0(3) | 20.0(40) | 246 | | 900 |
| 12 | 5 | 3.0(3) | 21.8(50) | 819 | | 864 |
| Total Score 2013 | | | | | 8288 | |
| Total Score 2014 | | | | | | 7329 |

Table 6.21: MCTS & A* 1000iter. results for all new levels

| Level | Completed Runs | Diamonds (Max) | Time - sec | Search Time - ms |
|---|---|---|---|---|
| 1 | 0 | 2.2(4) | | 12946 |
| 2 | 0 | 2.6(3/4) | | 884 |
| 3 | 0 | 2.2(4) | | 17028 |
| 4 | 4 | 8.4(9) | 56.0 | 2387 |
| 5 | 5 | 3.0(3) | 25.0 | 21324 |
| 6 | 3 | 2.6(3) | 26.3 | 1065 |
| 7 | 0 | 0.8(2/3) | | 772 |
| 8 | 2 | 7.2(9) | 63.5 | 1272 |
| 9 | 4 | 2.6(3) | 32.0 | 624 |
| 10 | 5 | 2.0(2) | 21.0 | 1062 |
| Collected Diamonds | | 33.6(42/44) | | |

MCTS & A* with 2000 ms solves 7 competition levels completely 5 out of 5 times and 4 levels 1 to 4 out of 5 times. The search time of all levels is between 2006 ms and 2016 ms. In all new levels it can collect 34 diamonds, which is the third best score behind Subgoal A* and Permutation A* (Table 6.19). MCTS & A* with 2000 ms can solve 2 of the new levels completely 5 out of 5 times and 5 levels 1 to 4 out of 5 times.

MCTS & A* with fixed 1000 iterations for MCTS has a 2013 competition score of 8288 and a 2014 competition score of 7329 (Table 6.20). The 2014 score is similar to MCTS & A* 2000 ms but the 2013 score is much lower. The lower score is again because of simple bad luck. MCTS & A* 1000 iterations can solve 7 competition levels 5 out of 5 times and 3 levels 2 to 4 out of 5 times. The search time is between 136 ms and 884 ms except level 10 with 20737 ms. The search time for level 10 is too long but the other search times are acceptable even if the A* times are much faster. For the new levels the total result of 33.6 diamonds is similar to MCTS & A* with 2000 ms. MCTS & A* 1000 iterations can solve 2 of the new levels completely 5 out of 5 times and 4 level s 2 to 4 out of 5 times (Table 6.21). The search time is rather long for 3 levels with 12946 ms to 21324 ms and acceptable for the others with 624 ms to 2387 ms.

In summary Subgoal A*, Permutation A* and Y-Heuristic A* are the best techniques in Geometry Friends. Hereafter Greedy Goal A* and MCTS & A* follow. The best agent of the 2013/2014 competition

Table 6.22: Overview of all techniques for 2013/2014 competition and new levels

| **Technique** | **Property** | **Score 2013** | **Score 2014** | **New Levels (Max 42/44)** | **Average Search Time - ms** |
|---|---|---|---|---|---|
| Best 2013/14 (CIBot) | | 5741 | 6466 | | |
| Greedy Goal A* | 2000 ms | 10063 | 7815 | 29.0 | 3 |
| Y-Heuristic A* | 2000 ms | **10469** | **8585** | 33.0 | **2** |
| Permutation A* | 2000 ms | 10382 | 8565 | **39.0** | 2704 |
| Subgoal A* | 2000 ms | 10413 | 8541 | 37.2 | 292 |
| MCTS | 1000 iter. | 5520 | 5178 | 30.2 | 4161 |
| MCTS | 2000 ms | 5598 | 4505 | 31.6 | 2000 |
| MCTS & A* | 1000 iter. | 8388 | 7329 | 33.6 | 3905 |
| MCTS & A* | 2000 ms | 9017 | 7300 | 34.0 | 2008 |

and MCTS do not show a strong performance. MCTS & A* has not a fixed route but sometimes seems not to have sufficient time to compute the best route. Greedy Goal A* has the problem to follow the shortest route, which is not always the best route to collect all diamonds. Y-Heuristic A* has the problem to follow the diamonds in descending $y$ coordinate order, which is not always the best route. In the new levels Y-Heuristic A* has only a score of 33 diamonds because there are new levels where it is not sensible to follow the descending $y$ coordinate order. Permutation A* has the best score of 39 diamonds in the new levels but the search time can explode in levels with many diamonds. Subgoal A* has most of the time a low search time. In levels with many diamonds a good compromise is to reduce the number of diamonds to collect after 2000 ms. It has the second best score with 37.2 diamonds in the new levels. With a score of 10413 for the 2013 competition and a score of 8541 for the 2014 competition the results of Subgoal A* are 81% and 32% higher, respectively, than the results of the best agent (CIBot) for the 2013 and 2014 competition. It may be concluded that overall Subgoal A* is the best technique.

# Chapter 7

# Conclusions

This chapter is about the conclusion (7.1) and future work (7.2). In the conclusion section the thesis is summarized, the research questions are answered and a final conclusion is given. In the section future work ideas for improvements of existing techniques or new approaches are described.

## 7.1 Conclusion

The conclusion starts with a summary of the thesis. In this thesis a fast domain dependent abstraction for the rectangle player of Geometry Friends has been proposed. The abstraction is based on a directed graph with nodes and edges. For example, nodes are the ends of obstacles and diamonds in the level. Edges represent the shape in which the rectangle reaches a node from another node. Four A* versions, MCTS and a combination of MCTS and A* have been developed as search techniques for Geometry Friends. They use the abstraction to compute a route to collect the diamonds of a level. Greedy Goal A* uses A* to search for the nearest diamond and then from this diamond to the next nearest diamond step by step until all diamonds are found. The route of each node pair is merged together to one route. Y-Heuristic A* collects the diamonds in decreasing $y$ coordinate order and A* searches for the route of each node pair. The route is merged together. In Permutation A* all permutations are calculated. It starts with searching for the shortest route of all permutations with all diamonds. If this route does not exist, it continues with the permutations with the diamond number decreased by one. For each route of a permutation, A* is used for each node pair and the complete route is merged together. In Subgoal A* an extra property is added to the A* node to store the number and order of collected diamonds. This technique uses A* to search for the complete route with all diamonds or decreases the number of diamonds to find a route with fewer diamonds. MCTS uses Monte-Carlo Tree Search to find the best route. MCTS & A* uses Monte-Carlo Tree Search to have the diamond order and calculates the route for each node pair with A* and merges the route together. A rule-based driver has been created to follow the calculated route of the search technique node by node. It returns the actions to perform. The different search techniques with different properties have been compared to each other in the levels for the 2013 and 2014 competitions. The techniques have been also compared to the best agent of the competitions. Also 10 new difficult levels have been created to achieve a larger and more challenging test set. It has been discussed why and which search technique is either more or less effective than the others. The search technique Subgoal A* has been concluded as best search technique for Geometry Friends with a 81% and 32% higher score for the 2013 and 2014 competitions, respectively, than the best agent for the 2013 and 2014 competitions.

Now the research questions can be answered:

- How can the game world be abstracted by taking into account physics?

  A domain dependent abstraction for Geometry Friends has been built with a directed graph. The graph consists of nodes, which are created out of obstacles and diamonds. With, for instance, the fall-down node the physics is already taken into account. Between all node pairs it is checked whether an edge is possible. This checking takes into account the size of the rectangle shapes and

also, for instance, that the rectangle cannot reach high obstacles. The abstraction performs well in all rectangle levels and with an average calculation time of 123.6 ms it is quite fast.

- How can A* be incorporated in Geometry Friends?

  With the information of the abstraction, A* can search for a route. Because there is not only one start and one goal node, A* has been adapted with different approaches. Four approaches have been developed: Greedy Goal A*, Y-Heuristic A*, Permutation A* and Subgoal A*. Y-Heuristic A* and Permutation A* show a strong performance in the competition levels but Y-Heuristic A* is not as strong as Subgoal A* in the new levels and Permutation A* has a long search time in levels with many diamonds. Greedy Goal A* is not as strong as the other A* versions because the shortest route is not the best route due to physical conditions. Subgoal A* has been concluded as best search technique for Geometry Friends with a 81% and 32% higher score in the 2013 and 2014 competitions, respectively, than the best agent for the 2013 and 2014 competitions.

- How can MCTS be integrated in Geometry Friends?

  MCTS also uses the information of the abstraction. Each play-out it is checked whether a new best route is found. If this is true, the new best route is set. MCTS has the property to selected a route with many nodes. It is often visited the same node several times although this is not necessary. To prevent this, parts of the route are deleted if there is no diamond in between the same nodes. MCTS has not shown a strong performance in Geometry Friends. A second approach is a combination of MCTS and A*. MCTS determines the order of the diamonds and A* calculates the route between the diamonds. MCTS & A* has been shown a stronger performance than the best agent of the 2013 and 2014 competitions but performs not as strong as the A* versions.

- How can the driver take into account the physics and missing information due to abstraction?

  The rule-based driver follows a route of a search technique. There are 13 rules, which ensure that the driver reaches the next node of the route. The driver accelerates, decelerates or does not move depending on the applicable rule. The physics is also included in the route because of the abstraction. Missing information is not taken into account. If the rectangle player gets stuck, the route is recalculated. The driver is able to follow a strong route and to solve a level. It gets stuck sometimes if the route is less strong.

Finally, the problem statement can be answered:

- How can we develop an agent for Geometry Friends?

  To develop an agent for Geometry Friends a graph-based domain dependent abstraction has been created, which takes into account physics. The abstraction is fast and performs well in all rectangle levels. MCTS and A* search techniques have been proposed, which use the abstraction to search a route where diamonds of a level can be collected. A rule-based driver has been created, which is able to follow a route and to solve a level. Subgoal A* is concluded as best search technique for Geometry Friends with a 81% and 32% higher score in the 2013 and 2014 competitions, respectively, than the best agent for the 2013 and 2014 competitions.

## 7.2 Future Work

In the section future work ideas for improvements of existing techniques or new approaches are described.

The abstraction can be improved by taking into account the directions upper right and upper left. At the moment it is not necessary because these two directions are not needed but maybe in the future there are levels where it is necessary. The edges of the abstraction are checked for three rectangle shapes. With more shapes, for instance, a shape between the square and the horizontal rectangle, some edges will be set as not possible. This would solve the problem of the competition level 9 where the driver is not able to fall down through a gap because there is another platform above the gab at the right side so that to morph up at the middle of the gap is not possible.

Subgoal A* can be improved by the evaluation function. At the moment the estimated costs to the goal node are set to 0 because there are several goals. The evaluation function has to take into account all goal (diamond) nodes but should not use the shortest distance in any way. If the shortest distance is used, the search will run into a similar problem like Greedy Goal A*. MCTS can be improved by parameter tuning, for instance, the UCT formula, the play-out node number limit, the calculation of the backpropagated value or the efficiency can be increased for more iterations in the same amount of time. To test the different values the automated testing has to work correctly because the manual testing would take a lot of time. If a forward model for the physics would be available, a MCTS search technique can be implemented, which replaces the driver and returns each action of the rectangle player.

The driver can have more rules. For instance, new rules are needed if the directions upper right and upper left are taken into account in the abstraction. Then also some old rules have to be changed. It is also possible to tune driver parameters like the maximum acceleration to improve the speed so that the level can be solved faster.

Bugs of Geometry Friend have been found, which were exchanged via e-mails with Rui Prada of the Organization Committee of Geometry Friends. Two out of three bugs are known and they are trying to correct them.

There is a 2015 Geometry Friends Competition at the end of August. The result of this thesis, the Subgoal A* agent, will participate at the 2015 competition to compete for the first place.

# References


Barto, A. G. (1998). *Reinforcement learning: An introduction*. MIT press, Cambridge. [1]

Bellemare, M., Veness, J., and Bowling, M. (2013). Bayesian learning of recursively factored environments. *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 1211–1219. [3]

Bresenham, J. E. (1965). Algorithm for computer control of a digital plotter. *IBM Systems journal*, Vol. 4, No. 1, pp. 25–30. [12]

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of Monte-Carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, Vol. 4, No. 1, pp. 1 –43. [2]

Chaslot, G. M. J. B., Winands, M. H. M., Herik, H. J. van den, Uiterwijk, J. W. H. M., and Bouzy, B. (2008). Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357. [2]

Coulom, R. (2007). Efficient selectivity and backup operators in Monte-Carlo tree search. *Computers and Games* (eds. H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers), Vol. 4630 of *LNCS*, pp. 72–83. Springer. [2, 22]

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, Vol. 4, No. 2, pp. 100–107. [2]

Jiang, N., Singh, S., and Lewis, R. (2014). Improving UCT planning via approximate homomorphisms. *Proceedings of the 2014 International Conference on Autonomous Agents and Multiagent Systems*, pp. 1289–1296, International Foundation for Autonomous Agents and Multiagent Systems. [3]

Kim, H.-T., Yoon, D.-M., and Kim, K.-J. (2014). Solving Geometry Friends using Monte-Carlo tree search with directed graph representation. *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pp. 462–463, IEEE. [3, 9, 10, 23]

Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo planning. *Machine Learning: ECML 2006* (eds. J. Fürnkranz, T. Scheffer, and M. Spiliopoulou), Vol. 4212 of *LNCS*, pp. 282–293. Springer. [2, 22]

Pepels, T. and Winands, M. H. M. (2012). Enhancements for Monte-Carlo tree search in Ms. Pac-Man. *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pp. 265–272, IEEE. [3]

Perez, D., Powley, E., Whitehouse, D., Rohlfshagen, P., Samothrakis, S., I., Cowling P., and Lucas, S. (2013). Solving the Physical Traveling Salesman Problem: Tree search and macro-actions. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6, No. 1, pp. 31–45. [2, 3, 9]

Rocha, J. B. G. (2009). Geometry Friends. M.Sc. thesis, University of Lisbon, Portugal. [3, 5]

Samuel, A. L. (1959). Some studies in machine learning using the game of Checkers. *IBM Journal of Research and Development*, Vol. 3, No. 3, pp. 210–229. [1]

Shannon, C. E. (1950). Programming a computer for playing chess. *Philosophical Magazine*, Vol. 41, No. 314, pp. 256–275. [1]

Turing, A. M. (1953). Chess. *Faster Than Thought*, pp. 288–295. Pitman Publishing. [1]

Weinstein, A., Mansley, C., and Littman, M. (2010). Sample-based planning for continuous action Markov Decision Processes. *ICML 2010 Workshop on Reinforcement Learning and Search in Very Large Spaces*. [3]

# Appendix A

# New Levels

In this appendix the new levels for Geometry Friends are shown. They can be downloaded as an XML file from the following GitHub link:
`https://github.com/Tidusmaster/GeometryFriendsMasterThesis/blob/master/DFischerMasterThesis/CompetitionCode_v32/Competition%20Code/GeometryFriendsAgents/GeometryFriendsFiles/Levels/Fischer.xml`

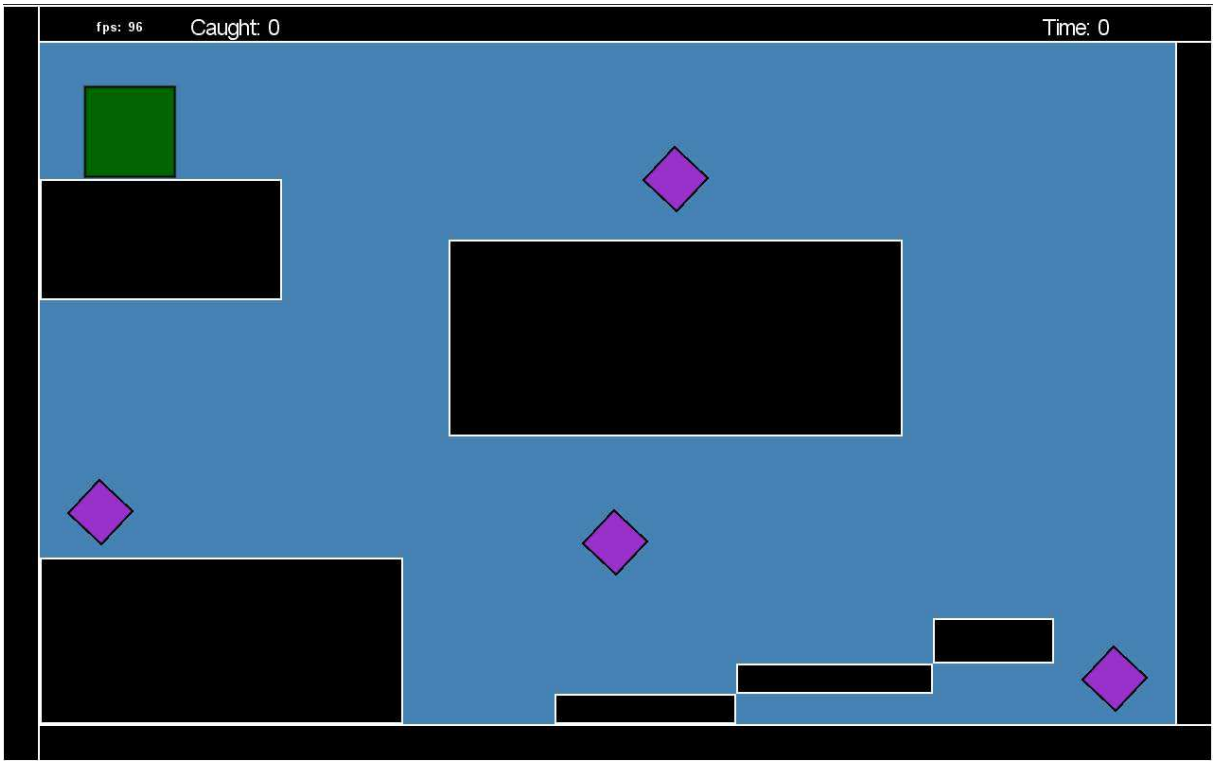


Figure A.1: Level 1 of the new levels

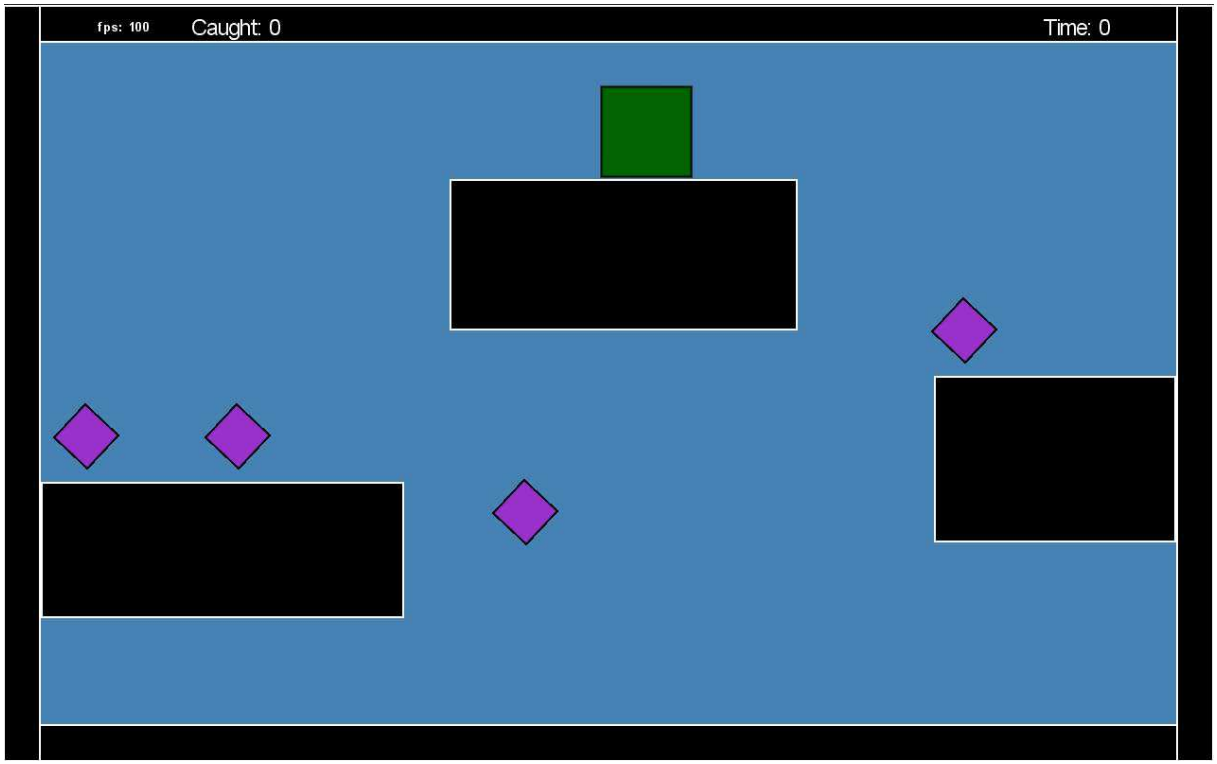


Figure A.2: Level 2 of the new levels

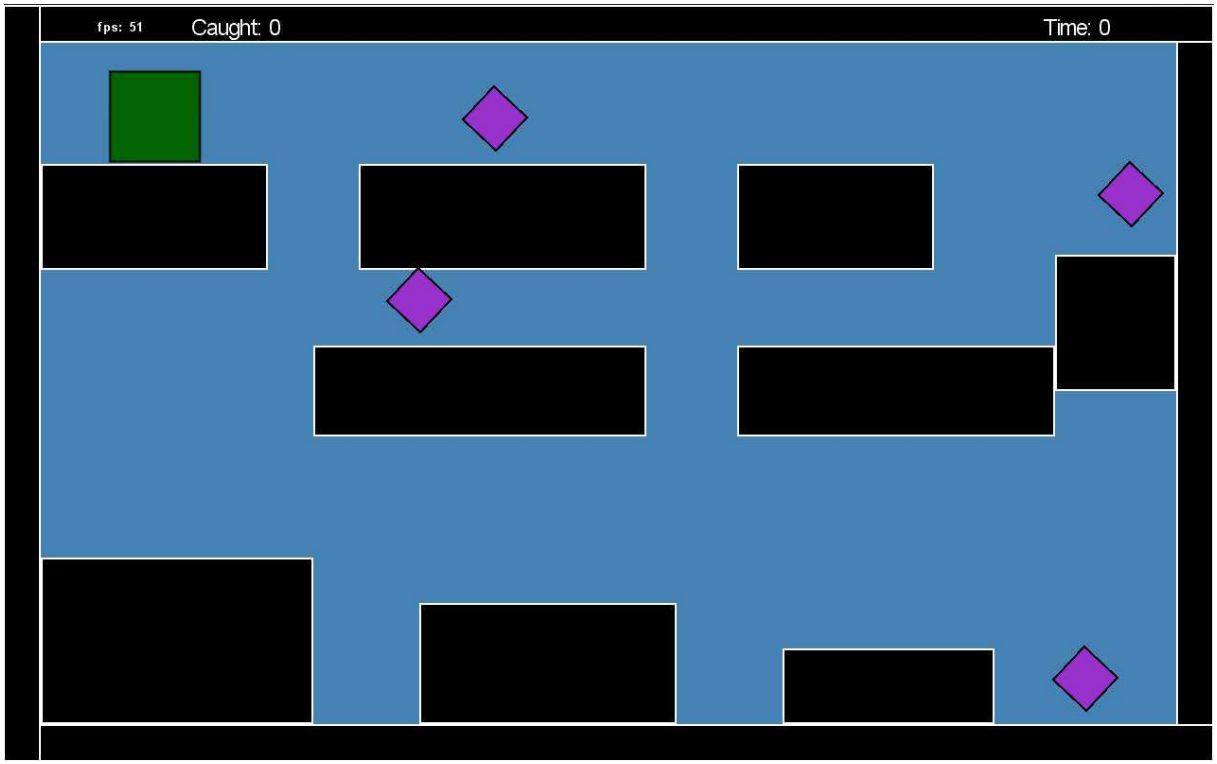


Figure A.3: Level 3 of the new levels

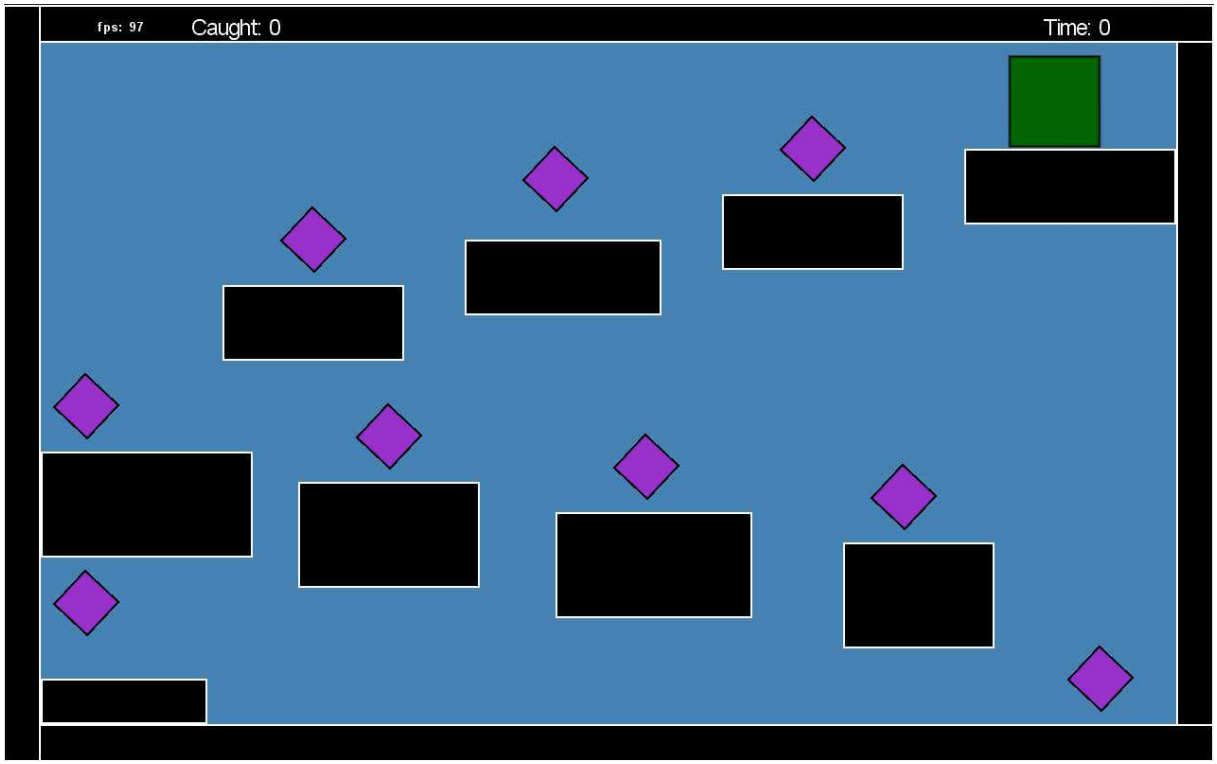


Figure A.4: Level 4 of the new levels

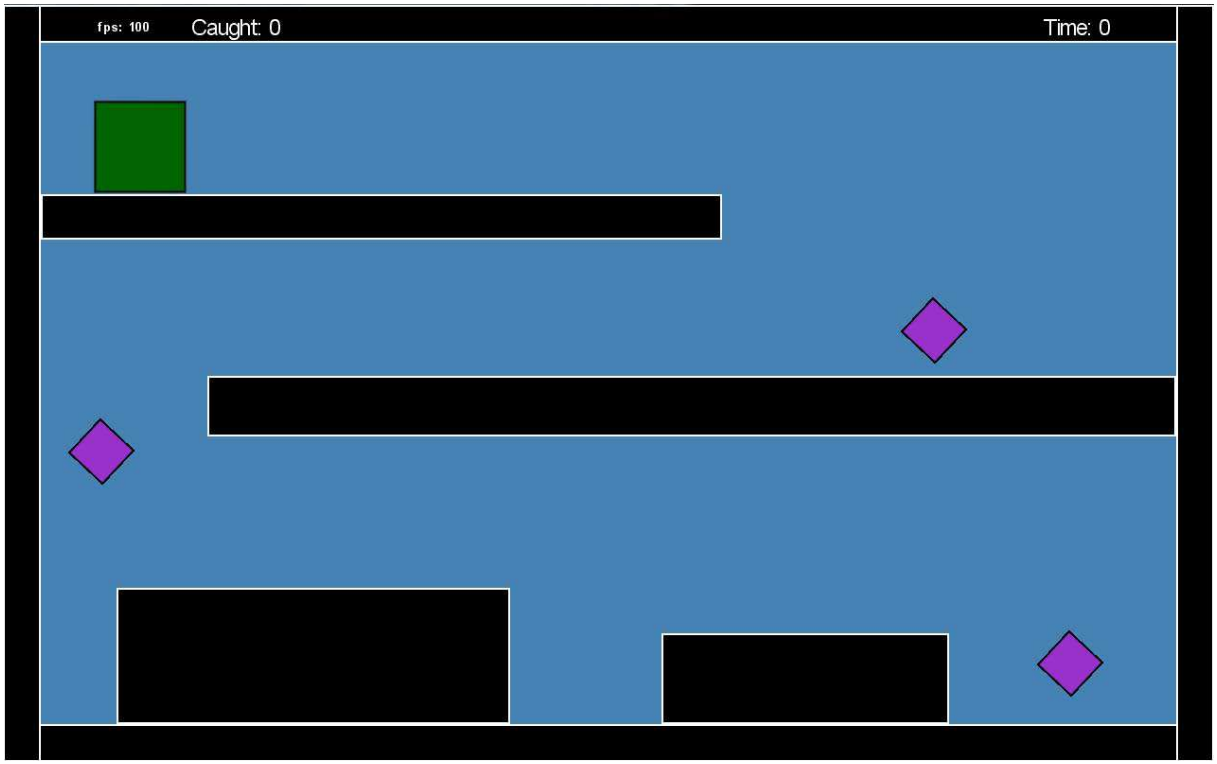


Figure A.5: Level 5 of the new levels

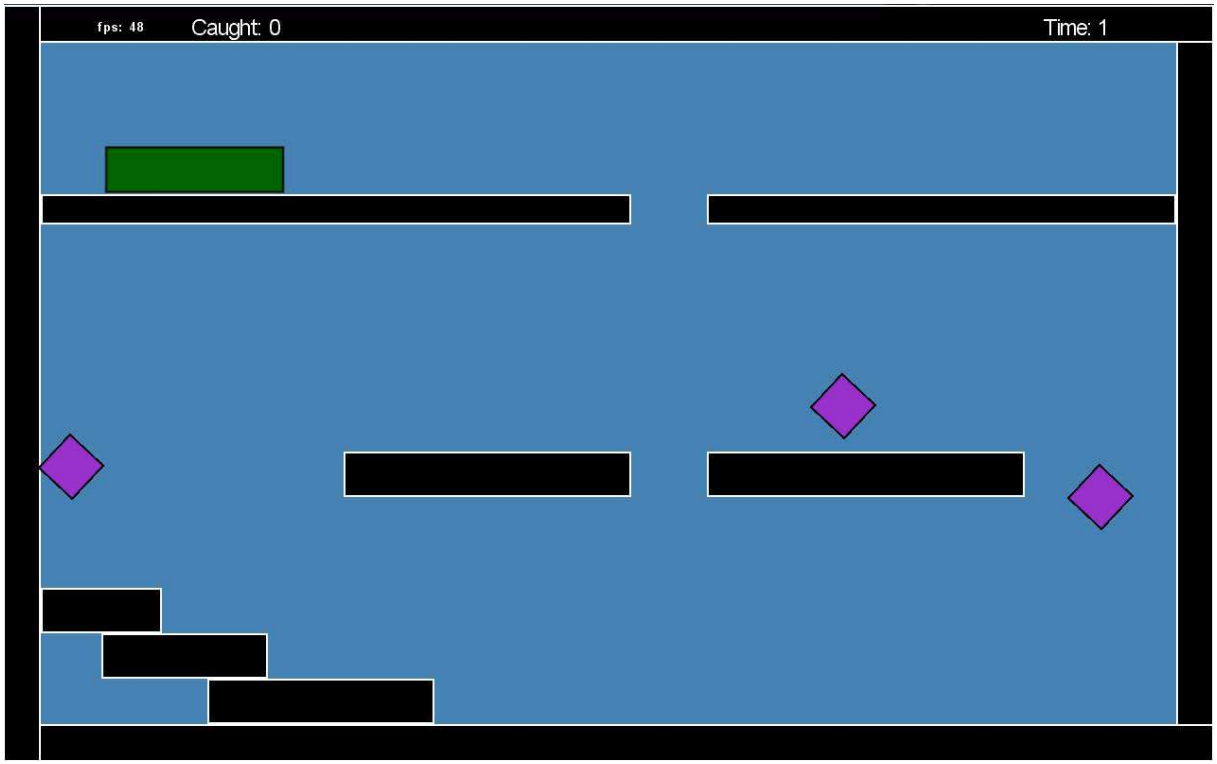


Figure A.6: Level 6 of the new levels

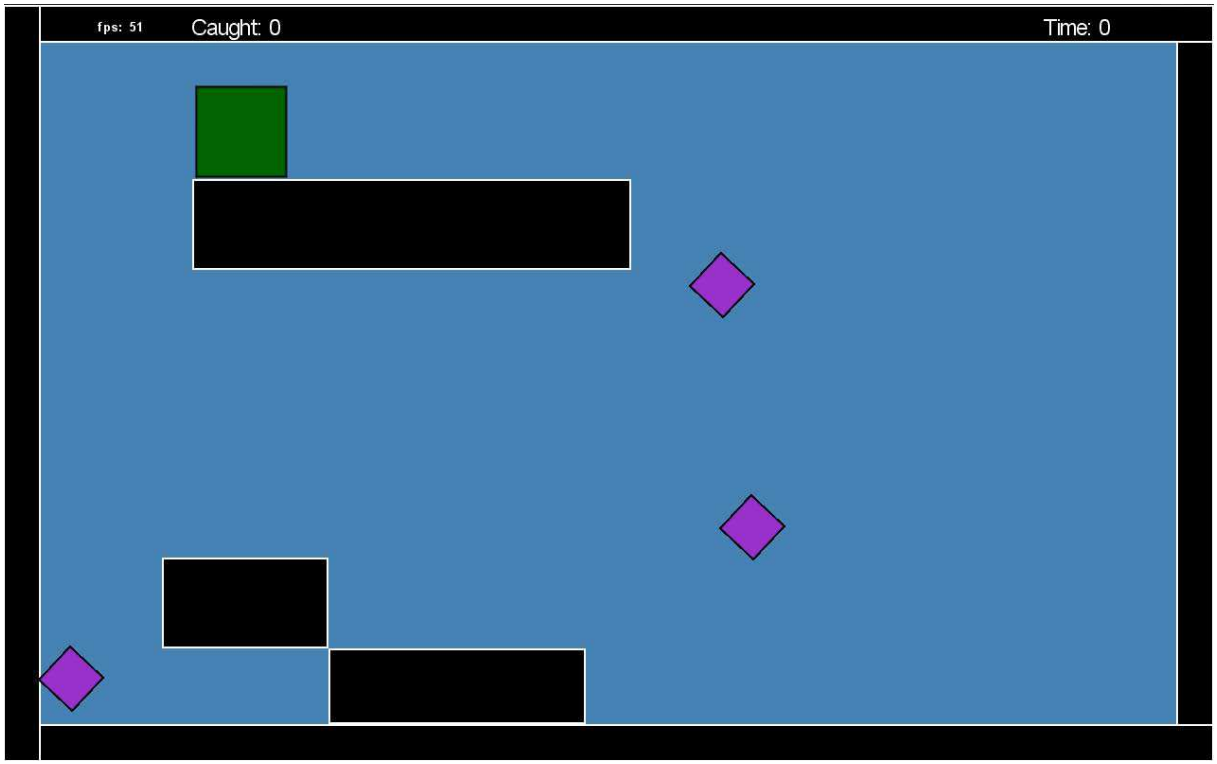


Figure A.7: Level 7 of the new levels

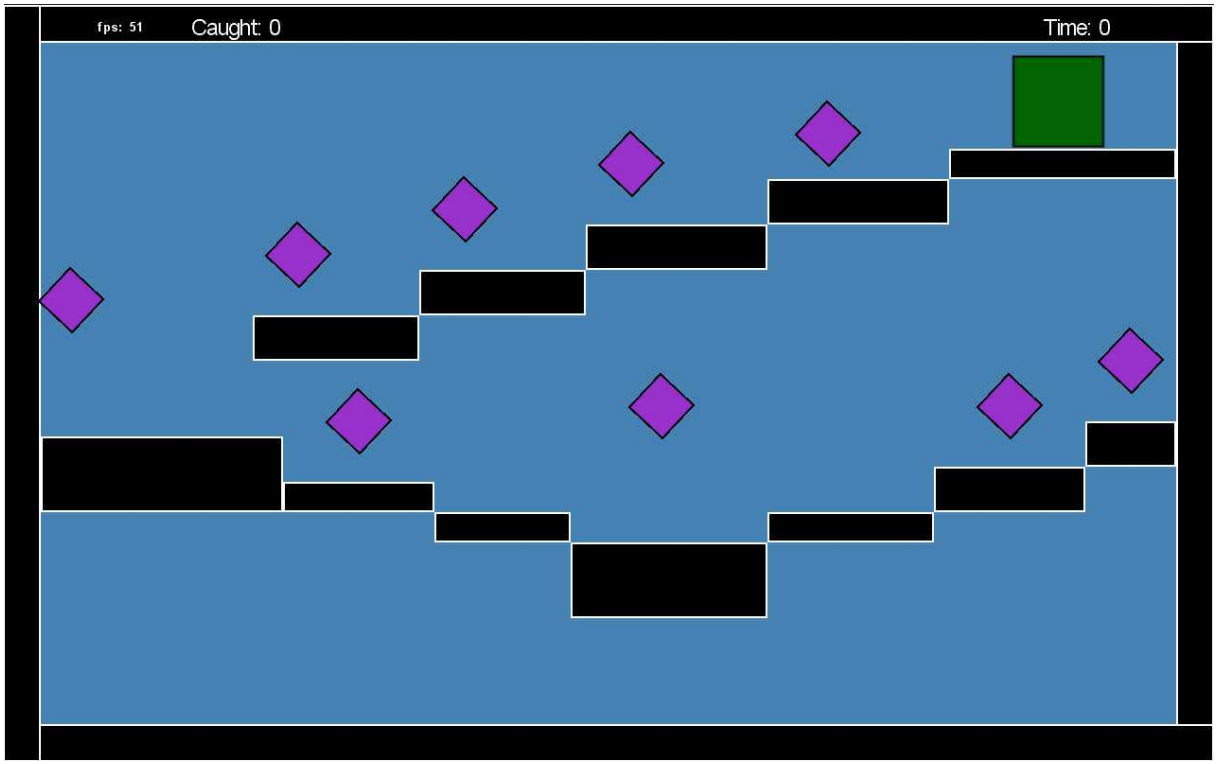


Figure A.8: Level 8 of the new levels

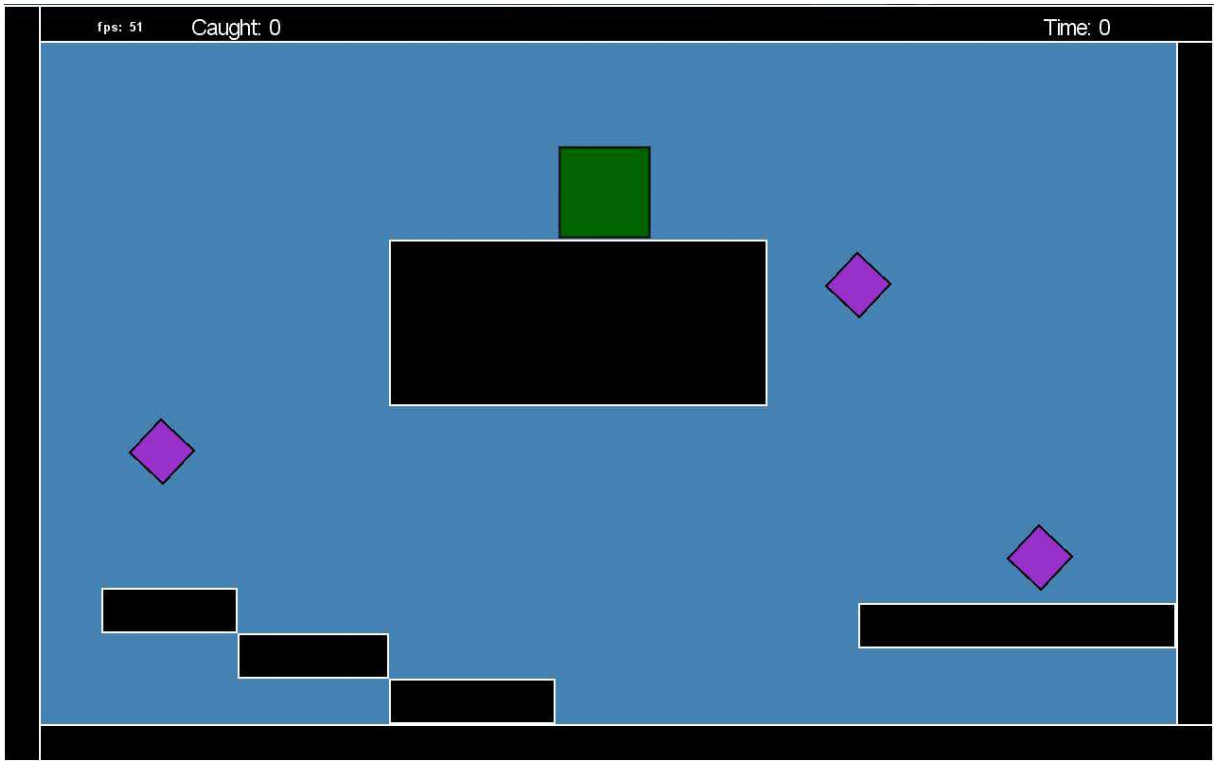


Figure A.9: Level 9 of the new levels

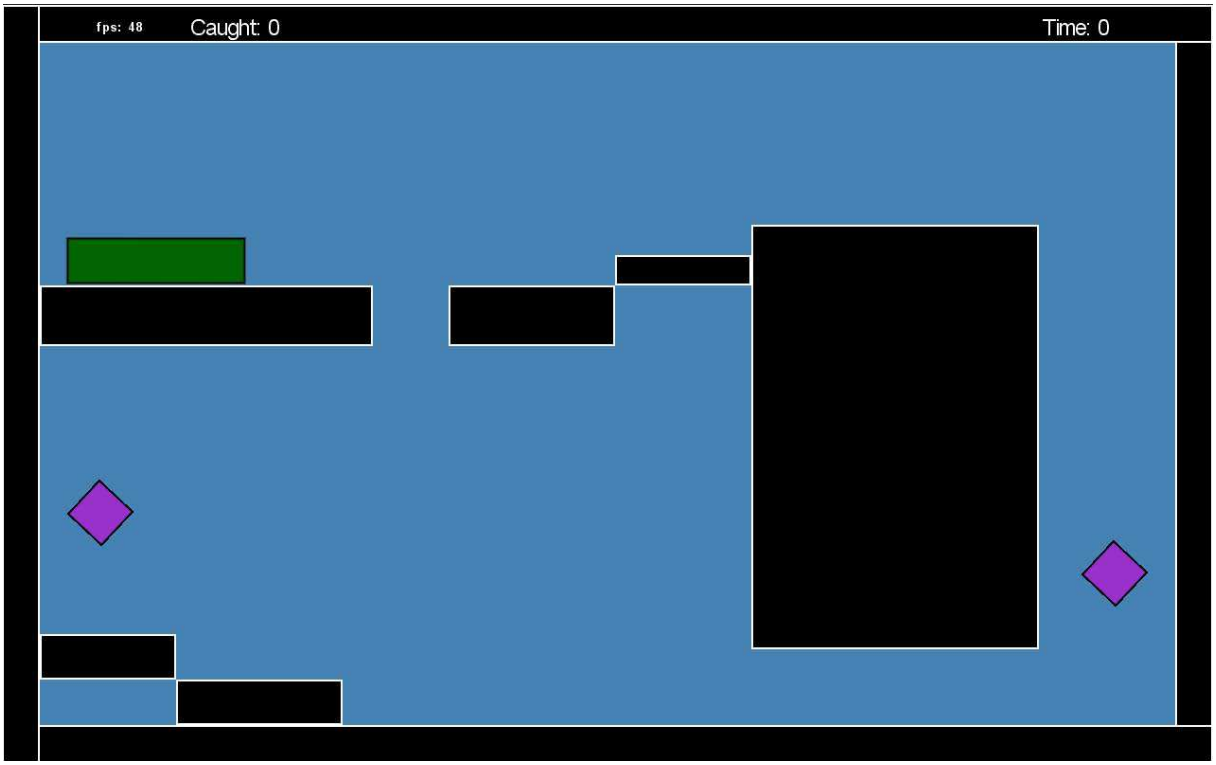


Figure A.10: Level 10 of the new levels