UCMAgent Rectangle Report

Alberto Almagro Sánchez[†]
Complutense University of Madrid
Madrid, Spain
alberalm@ucm.es
contact@alberalm.com

Juan Carlos Llamas Núñez[†]
Complutense University of Madrid
Madrid, Spain
jullamas@ucm.es
jcllamas2000@gmail.com

***Abstract*—This report presents our rectangle agent solution for the Rectangle Track of the Geometry Friends Competition at International Joint Conferences on Artificial Intelligence (IJCAI) and IEEE International Conference on Games (CoG) 2023.**

***Index Terms*—Artificial Intelligence, planning, physical modeling, expert system, physical environment, reinforcement learning.**

I. Introduction

This document aims to give an overview of our solution implemented for the rectangle agent of the Geometry Friends game. The strategy the UCMAgent follows to solve each level can be divided into three phases: representation of the level, action planning and execution of the plan. We will illustrate this strategy using the level shown in Fig. 1 as an example.

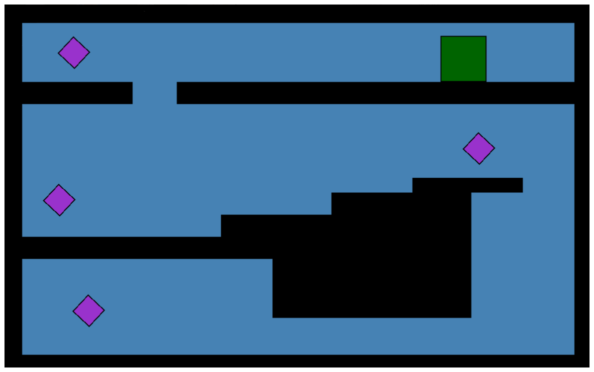

Fig. 1. Rectangle level example.

II. Level representation

To make an adequate representation of each level, the steps our agent follows can be summarized as the identification of the platforms of the level, the generation of movements that indicate connections between them and a filtering that allows to keep a minimal set composed of the best movements. In the following, we will discuss each of these stages.

A. Platform identification

To approach each level, we first make a discretization of the information that represents the level and that is provided by the game's interface. We perform this simplification by dividing the lengths of the objects by 8 and keeping the integer part. Although it may seem this compromises our agents precision, we concluded after several tests this first step had almost no harmful consequences and thus allowed us to work more comfortably.

Next, we establish the type of element represented in each discretized pixel (either obstacle, diamond or empty) and, from this distinction, we identify the platforms we would use throughout the rest of the process. Unlike in the case of the circle, the shape of the rectangle must also be considered when determining the platforms. For simplicity, we only consider the three basic shapes shown in Fig. 2, which we name horizontal, square and vertical. Therefore, in each "small" platform, we consider the set of shapes with which the rectangle can rest on it, as shown in Fig. 3 (this figure only shows the main small platforms, since in reality there are many more).

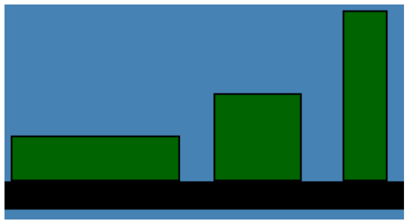

Fig. 2. Basic rectangle shapes considered.

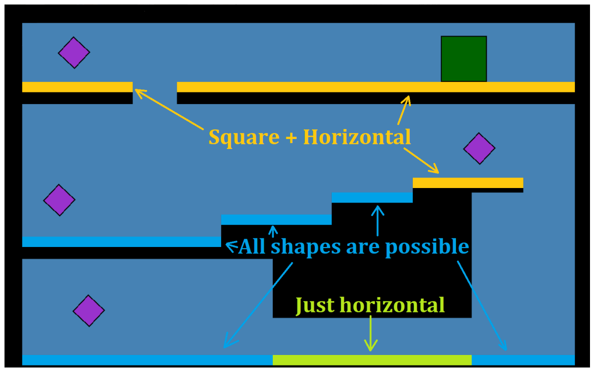

Fig. 3. Platform identification: small platforms.

After this small platforms are obtained, we juxtapose horizontally contiguous small platforms to create "simplified" platforms, which can be seen in Fig. 4. By construction, it can

[†] These authors contributed equally to this work and share first authorship.

be assured that, after maybe modifying its shape accordingly, the rectangle is able to move within the same simplified platform by just taking the MOVE_RIGHT or MOVE_LEFT actions. This is easily appreciated in the inferior platform of the example level.

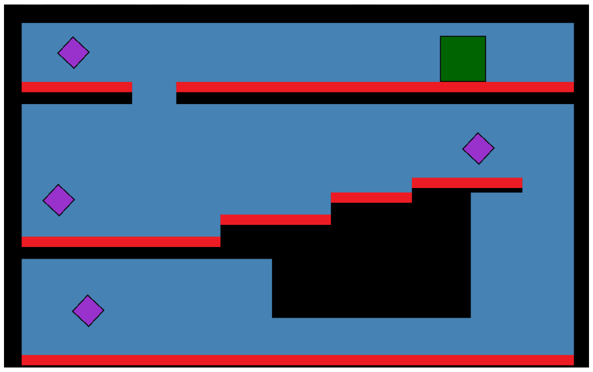

Fig. 4. Platform identification: simplified platforms.

B. Move generation

Once both small and simplified platforms have been identified, and in order to reflect the rectangle's ability to change platforms or reach diamonds, we introduce the concept of "moves", which represent sequences of atomic actions that achieve one or both of the above high-level objectives. For the rectangle agent, we consider multiple types of moves: NO_MOVE, ADJACENT, FALL, DROP, TILT, MONOSIDEDROP, BIGHOLEADJ, BIGHOLEDROP, HIGHTILT and WIDEADJ. We shall now describe each one of them.

1) NO_MOVE: This move represents the possibility of capturing a diamond that is on a platform with one of the three basic shapes and without the need of executing any action.

2) DROP: This move captures the rectangle's ability to fall vertically between two close platforms by performing a MORPH_UP action while resting with each of the vertices of its base in one of the platforms, as represented in Fig. 5.

3) MONOSIDEDROP: Essentially the same as a DROP move, except that one of the platforms is now a vertical wall, as in Fig. 5.

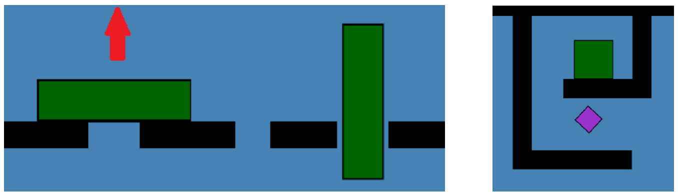

Fig. 5. Examples of DROP move (left) and MONOSIDEDROP move (right).

4) BIGHOLEDROP: This move is similar to a DROP but the gap's width between the horizontal platform is too high for the rectangle to rest with each of the vertices of its base in one of the platforms and too small to perform a FALL succesfully (we call this structure "big hole", hence the move's name). A new strategy is needed and will be explained later. Examples of levels that include this big hole structure are level 9 from the GF-CoG 2022 - Rectangle Track competition and level 8 from the GF-IJCAI-ECAI 2022 - Rectangle Track competition.

5) TILT: This move captures the rectangle's ability to climb stair steps by tilting through one of its corners (see Fig. 6).

6) HIGHTILT: Essentially the same as a TILT move, except that the stair steps considered are very high (as shown in Fig. 6).

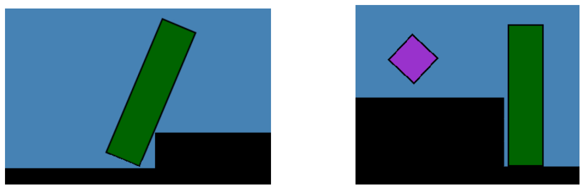

Fig. 6. Examples of TILT move (left) and HIGHTILT move (right).

7) FALL: This type of moves represent the possibility of the rectangle falling off the ends of the platforms. Its simulation is very similar to the FALL of our circle agent. In the case of the rectangle, we have also considered the three different basic shapes and have also modeled the rotation of the rectangle during falls around its center of mass, which improves the accuracy of the simulation. Since (contrary to the circle) the rectangle can change its trajectory during the flight, we simulate a total of three falls: the one in which, during the flight, the NO_ACTION action is always taken, the MOVE_LEFT action is always taken or the MOVE_RIGHT action is always taken.

8) ADJACENT: This move lets the rectangle slide to another platform at the same height provided that the gap between them is "small" (see Fig. 7).

9) BIGHOLEADJ: A BIGHOLEADJ is a FALL where strange collisions are allowed for the particular case in which the landing platform is at the same height as the departure platform (see Fig. 7). The gap between these platforms is greater than the one that would lead to an ADJACENT move, i.e. the rectangle cannot just cross with its horizontal form.

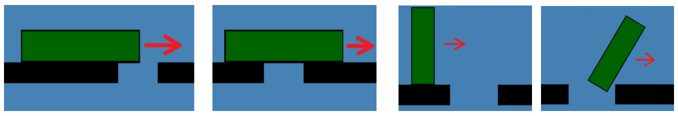

Fig. 7. Examples of ADJACENT (left) and BIGHOLEADJ (right) moves.

10) WIDEADJ: Initially, we just considered using BIGHOLEADJ moves for big hole structures. However, it can be generalized to wider gaps by just increasing the initial speed the rectangle had. We denote these generalized BIGHOLEADJ moves as WIDEADJ.

It is important to note some of these moves, while generated, are impossible to be completed. For example, if a WIDEADJ is generated but there is not enough room for the rectangle to obtain the speed needed to cross the gap, this move cannot be executed successfully. To avoid that these moves are included in the final plan our agent will follow, we set them as *risky*, which means that our search algorithm will try to avoid them whenever there is other path that does not have any *risky* move, even if it is longer. Our HIGHTILT and WIDEADJ moves require that the rectangle possesses some initial speed that is not taken into account when generating them, so they are marked as *risky* moves. This is the reason we differentiate them from TILT and BIGHOLEADJ moves, respectively, which can be executed with no initial speed, and are therefore safe moves that are always successful. We have also marked as *risky* the BIGHOLEDROP moves, since, while always possible, our strategy does not guarantee executing them correctly, so we want to avoid them as much as possible.

C. *Move filtering*

As in the circle agent, a large number of movements have been generated and it is necessary to perform a filtering in order to keep a reduced subset, which guarantees the maximum connectivity between platforms. We also use this stage to favour some moves over others e.g. ADJACENT and DROP over difficult FALL.

At the end of this stage, we would have a similar representation as the one shown in Fig. 8. We can see the different platforms and the connections between them with the most suitable moves left after the filtering. Each of these moves have characteristic information that identifies it and that is deduced from the simulation. This information includes the move type, the starting point, the landing point, the initial speed, the origin and destination platforms and the diamonds captured, among others.

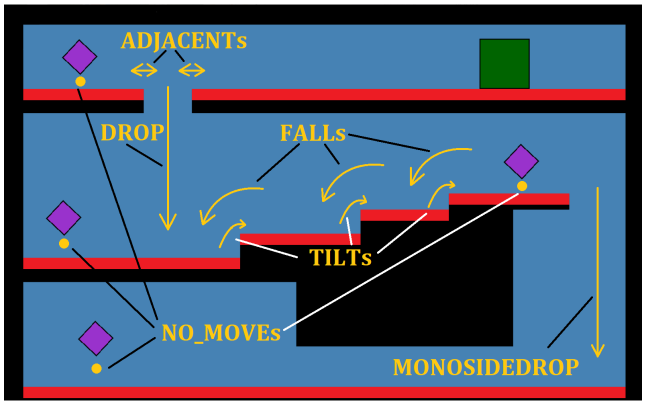

Fig. 8. Move generation and filtering.

III. Action planning

Once the level representation phase has been completed, we can collect the resulting information in a graph whose nodes are the platforms and whose edges are the moves connecting them. These moves are associated with the diamonds they capture according to the simulation previously performed. The objective of this phase is to generate a high-level plan, i.e., a sequence of moves to change platforms so that all diamonds are reached. During execution, our strategy involves that, once a platform is reached, all the collectibles of that platform are captured before moving to the next one, and that is why movements within the same platform are omitted.

The plan is generated by performing a time-limited breadth-first search with repetition control over the graph. If the search time expires, which is unusual because the graph is usually small in size, the best plan so far, i.e. the one that captures the highest number of diamonds, is returned. Our planning algorithm also takes into account real-time replanning and alternative paths to avoid *risky* moves.

IV. Plan execution

Once our agent has a plan to follow, the last step to solve a level is to perform the atomic actions necessary to complete it. For this purpose, Algorithm 1 is followed.

Algorithm 1 Rectangle's execution algorithm

```
1:  while Level not finished do
2:      if current platf. != depart. platf. of plan's 1st step then
3:          Replanning
4:      end if
5:      if There are diamonds left on the current platf. then
6:          $m \leftarrow$ Move that reaches the nearest diamond
7:      else
8:          if Plan is not empty then
9:              $m \leftarrow$ Plan's first move
10:         else
11:             Execute a random action
12:         end if
13:     end if
14:     $S \leftarrow$ best rectangle shape to complete $m$
15:     if Rectangle shape != $S$ then
16:         Morph rectangle to get closer to the shape $S$
17:     else
18:         Execute best action that leads to completion of $m$
19:     end if
20: end while
```

It is necessary to explain some details of the algorithm. First of all, we need to know which is the best shape to complete a certain move. We divide this task in two steps. Firstly, we need to update the shape of the rectangle during its approach to the initial point of the move, as it may not be straightforward like in the example of Fig. 9. This step can be solved in a relatively simple way by evaluating the valid shapes of the small intermediate platforms between the rectangle platform and the platform of the starting point of the movement, both on the same simplified platform. Secondly, once the rectangle is in the same small platform as the starting point of the move, it has to match its shape to the one required to begin that move, and that will depend on the move type: if it is ADJACENT, then the shape is horizontal; if it is TILT, then the shape is vertical; if it is FALL, then the target shape is the one used in the simulation; and so on. When the target shape has been determined, if the rectangle does not yet have such a shape, the first thing to do is to adopt it. To do this, the rectangle may use the MORPH_UP or MORPH_DOWN actions to grow or shrink as appropriate.

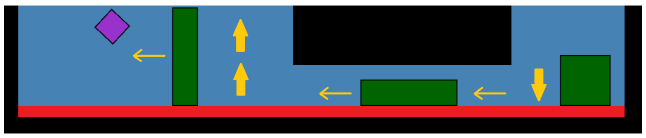
Fig. 9. Shape changes during approach to target point.

Once we have covered everything related with the shape of the rectangle, let us discuss how to execute the best action that leads to the completion of a certain move. The policy that our agent follows to get to the initial position of a move with the intended initial horizontal speed, omitting the shape changes considered above, is essentially the same as the one used for our circle agent. This system is explained in detail in our circle's report, but in the case of the rectangle we can relax some of the restrictions on the exact speed with which the rectangle must reach the target point, since it is not so decisive in some of the move types such as TILT or BIGHOLEADJ.

Assuming that the rectangle has managed to reach the starting point of the movement by fulfilling the preconditions of speed and shape, it may have to execute certain actions during the move, which depend on the type of move, but will mostly be the one that lets the rectangle gain momentum and helps it to cross to the other platform.

Exceptions to this are DROP or a MONOSIDEDROP move, during which we first need to perform a MORPH_UP action, followed by a MORPH_DOWN once the initial platforms have been avoided. This prevents the rectangle from falling through consecutive holes when an intermediate stop is desired.

Finally, the BIGHOLEDROP move is by far the most complex movement type that we have implemented, and, as we anticipated, we need to follow a new strategy to solve it. Due to the lack of space between the initial platforms, it is usual that the rectangle gets stuck between them, even when humans are controlling the rectangle. As humans are not able to find a consistent strategy to solve this situation and expert knowledge is not useful in this particular context, we tried to face this situation with reinforcement learning. In particular, we used Q-Learning so that the rectangle learned autonomously to fall between those small holes. Even though the environment does not offer us enough information (the lean of the rectangle is unknown) and the update rate is changeful, the given information was proved to be sufficient to successfully solve this situation an average of 62% of the times. The representation uses discretized values of the position of the rectangle with respect to the hole, the height of the rectangle, its velocity, and the gap's width.

During training, the reward is negative for each state in which the fall is not completed, so that the agent learns to perform the fall as quickly as possible. In addition, it is positive and very high when it is detected that the rectangle has a negative vertical position with respect to the gap between platforms, which means that the rectangle has succeeded in falling. We also set the characteristic parameters of reinforcement learning by Q-Learning to different values. For instance, the discount rate was set to $\gamma = 0.95$, while the learning rate was established as $\alpha = 0.1$. Finally, we followed an ε-greedy policy, starting in $\varepsilon = 0.3$ and exponentially decreasing its value.

This concludes our basic explanation of our agent's behaviour. In the following section, we will point out some minor improvements we have implemented.

V. Minor Improvements

To complement and refine the above algorithm, we present some of the systems we use to improve the performance of the rectangle. First, the recovery system is used to escape from situations where the rectangle is stuck. To do this, our rectangle performs random actions for a short period of time and, despite its simplicity, it is a very effective system without which it would be difficult to complete many of the levels. As in the case of our circle, but less critical, there is also a system to avoid unwanted falls from the ends of the platforms.

Our rectangule agent also includes an orientation recovery system. When the rectangle is flipped, sometimes the base of the rectangle is interchanged by one of its sides, and the game does not process this change until either a MORPH_UP or a MORPH_DOWN action is executed. Consequently, it is necessary to re-identify which are the bases and sides by detecting when the rectangle's height is not correctly managed.

It has also been implemented a shape change verification system. Before the rectangle takes the MORPH_UP and MORPH_DOWN actions, it is necessary to check that it has enough space to grow horizontally or vertically, as the case may be. Failure to perform this check may result in unwanted bugs where the rectangle trembles or is trapped inside an obstacle.

Finally, the environment possesses a bug that sometimes takes place whenever the rectangle agent falls from a very high platform and lands with a high speed. If this happens, the rectangle may be left unable to change its shape, which often leads to the level being unsolvable. We discovered this behaviour can be avoided if the MORPH_UP and MORPH_DOWN actions are executed when landing, and we added this function to our agent whenever it performed a high FALL move.

VI. Conclusion

To conclude, and in a similar way to our circle agent, we have based our solution for the rectangle agent in expert knowledge, by identifying and studying many different situations the rectangle may face. Our search algorithm, combined with our efficient movement generation and level representation, makes our rectangle solve almost every level from previous competitions.