A Reinforcement Learning Approach for the Circle Agent of Geometry Friends

João Quitério[*], Rui Prada[†], Francisco S. Melo[‡]
INESC-ID and Instituto Superior Técnico, University of Lisbon
Av. Prof. Dr. Cavaco Silva, 2744-016 Porto Salvo, Portugal
[*]joaollquiterio@tecnico.ulisboa.pt
[†]rui.prada@gaips.inesc-id.pt
[‡]fmelo@inesc-id.pt

Abstract—**Geometry Friends (GF) is a physics-based platform game, used in one of the AI competitions of the IEEE CIG Conference in 2013 and 2014. The game engages two characters, a circle and a rectangle, in a cooperative challenge involving collecting a set of objects in a 2D platform world. In this work, we propose a novel learning approach to the control of the circle character that circumvents the excessive specialization to the public levels in the competition observed in the other existing solutions for GF. Our approach proposes a method that partitions solving a level of GF into three sub-tasks: solving one platform (SP1), deciding the next platform to solve (SP2) and moving from one platform to another (SP3). We use reinforcement learning to solve SP1 and SP3 and a depth-first search to solve SP2. The quality of the agent implemented was measured against the performance of the winner of the Circle Track of the 2014 GF Game AI Competition, CIBot. Our results show that our agent is able to successfully overcome the over-specialization to the public levels, showing comparatively better performance on the private levels.**

I. Introduction

Geometry Friends (GF) is a physics-based platform game involving two "characters": the Circle and the Rectangle. To complete the game, the two characters must overcome several levels, in which they must collect all the diamond-shaped objects in the environment in the minimum amount of time. The levels of the game can be designed to be played by a single character or by both characters simultaneously, in a cooperative fashion. From a single character's point-of-view, GF imposes challenges in the navigation within the game space, namely in terms of fine control and adequate timing of the agent's actions. At the same time, it requires the agent to plan ahead and decide what path to follow in order to solve the level. From a cooperation point-of-view, the coordination of movements of both characters is also a challenge to be tackled.

Due to the difficulty and variety of the challenges it imposes, GF is an adequate platform to develop new AI algorithms. For this reason, it has been featured in the game AI competitions of the IEEE CIG Conference in both 2013 and 2014 editions.[1] The competition includes two single agent tracks, each containing levels to be solved by one of the two characters, and a cooperative track, where both agents must play cooperatively to solve a number of levels. Competitors are asked to produce an AI system that is able to control the corresponding character(s) towards the solution of 10 levels, 5 of which are publicly available while the submissions are open.

Existing AI systems for GF (presented in past editions of the competition) were able to successfully tackle the public levels. However, their performance in the unknown levels was significantly worse, suggesting that such systems were over-specialized in the levels that were made available. Hence, an AI system that is able to successfully tackle previously unknown GF levels should be able to break down each new level in its simplest components, and then robustly control the agent in each of these components.

In this work, we propose a novel solution that is supported on both search and reinforcement learning. We focus on the Circle agent, although our proposed approach is not circle-specific and can, therefore, be also applied to the Rectangle character. The proposed solution was developed to overcome the over-specialization to the public levels observed in past solutions. It also aims at becoming a springboard on the development of AI agents that are able to solve any possible level configuration without previous playing on it. To evaluate the quality of our solution, we compare our results with those of the winner of the IEEE CIG 2014 GF AI Competition, CIBot.

II. Geometry Friends

Geometry Friends is a physics-based platform game that is set in a two-dimensional environment. There are two characters in the game that the player can control: a yellow circle and a green rectangle. The environment (depicted on Fig. 1) is populated by diamond-shaped objects that can be collected by any of the characters and with obstacles that restrict the character's movement . There are two types of obstacles: black obstacles that restrict the movement of both characters; coloured obstacles that only restrict the movement of the character of the opposite colour. The agents must collect every diamond available on a particular level. The game has different levels, each one with a distinct layout for the obstacles, the collectibles and initial position of the characters.

[1] For more information about the competition, we refer to the website http://gaips.inesc-id.pt/geometryfriends/.

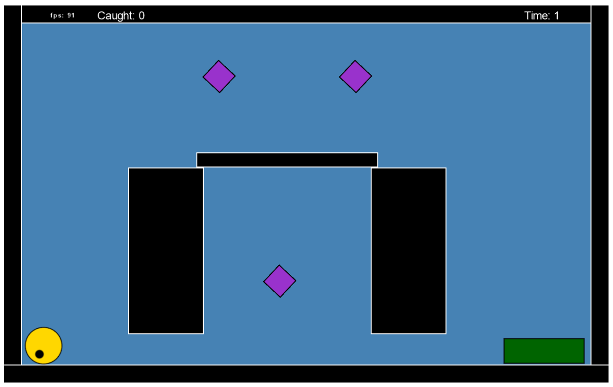

Fig. 1: Geometry Friends level

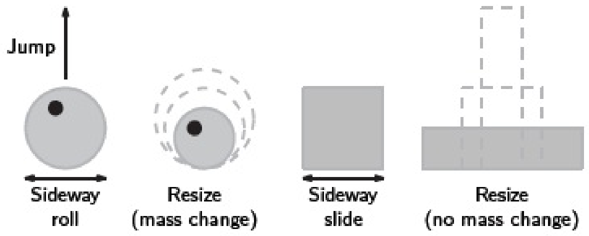

Fig. 2: Possible movements of both characters. Taken from Geometry Friends AI Competition Website

Each character has a specific set of actions that it can perform. The circle can *roll* both to the left and to the right, *jump* and *change its size*. The rectangle, on the other hand, can *slide* both to the left and to the right, and *morph* to become wider or slimmer, while maintaining a constant area (see Fig. 2). Both agents are affected by gravity, attrition and collisions with obstacles and one another. Since each character has different motor skills, there are levels that be solved by only one character; levels that can be solved by any of the two; and finally levels that can only be solved by both agents acting cooperatively.

III. Related Work

The use of AI to solve games is a long standing tradition, with such outstanding showcases as DeepBlue in chess [1], Chinook in checkers [2], and Watson in jeopardy [3]. And while solutions to complex games such as chess or checkers rely heavily on search algorithms, more recent successes arise from the combination of powerful search algorithms with an equally powerful *learning component* [3], [4]. For example, recent results on computer Go rely heavily on Monte-Carlo tree search algorithms rooted in reinforcement learning, such as the UCT algorithm [5]. In a closely related line of work, the Deep-Q system combines reinforcement learning with a deep neural network to achieve human-level play of several Atari games [6]. Ross and Bagnell [7] use structured learning to develop a player for the Infinite Mario game. Tsay *et at* apply reinforcement leaning to an agent that plays Super Mario, and outperforms other learning approaches but still under-performs when compared a search-based A* approach [8].

Specifically with respect to GF, the game was first mentioned on [9] as an example of a cooperative game. The position of the diamonds was calculated so as to force cooperation between both characters and assumed that the game was to be played by human-controlled agents. Carlos Fraga [10] presented an AI solution to GF using a navigational graph. This graph has different types of edges depending on whether the edge is traversable by the circle alone, by the rectangle alone or by both characters. The nodes of the graph are positioned on both agents' starting positions and on the diamonds'positions. Other nodes are generated by expanding the initial nodes. Once the graph is built, the agents run the A* algorithm to determine the path to follow. One of the limitations of this approach is the processing overhead caused by running the A* algorithm every time an action has to be made. Furthermore, the complexity of the analysis of the level needed to generate the graph, failed to solve many different situations in the game, suggesting a lighter approach could be used.

Yoon and Kim [11], developers of the winning agent of the 2014 GF AI Competition circle track and Benoît et al [12], runner up in the 2014 rectangle track of the same GF competition, use similar approaches based on path planning. Both agents create a graph from the level layout and use Dijkstra's algorithm to find the shortest path through the graph. Yoon and Kim's agent uses the edge points of the platforms as nodes. Whenever is possible for the circle to go from one of those edge points to another a graph edge is created. Every time the agent has to play, it runs Dijkstra algorithm to find the shortest path to the closest diamond. To avoid constantly running the Dijkstra algorithm, edge points along the path to a diamond are stored in a queue in the order they should be visited. With this optimization, the algorithm is only run when the queue is empty. Benoît's agent creates a meta-graph with special points called objective points. With this meta-graph the agent calculates the order in which it must visit those points using Dijkstra. Finally, the agent plans the set of actions it has to perform to be able to follow the path found. The main difference between those two solutions, is the fact that Benoît's agents plan everything on an initial setup phase whereas Yoon's make plans while playing the level. Both controllers use a simple rule based-system, although differing on the type of control encoded, that move the circle, by rolling and jumping, in a greedy way to move it closer to the target position. Jumping strategies (i.e. the distance and velocity to jump to a platform) are pre-computed in these rules.

Yen-Wen Lin et al. [13], developed the KUAS-IS agent that also competed on the 2014 GF Game AI Circle Track. Their agent uses A* and Q-Learning to solve GF. A* is used to find the shortest among the paths that go through all the diamonds on the level. However, A* can compute a path that leads the agent to a pitfall, such as tight hole that is very hard to get out. To avoid these pitfalls, their agent uses Q-Learning to bias the A* heuristics.

In our work, we combine reinforcement learning with search as a means to learn policies that can be generalized across levels. Reinforcement learning (RL) algorithms enable an agent to learn a task by trial and error, driven by a reward signal that "encodes" the task to be learned [14]. RL methods are designed for situations where the environment is quite dynamic and non-deterministic, such as the GF domain.

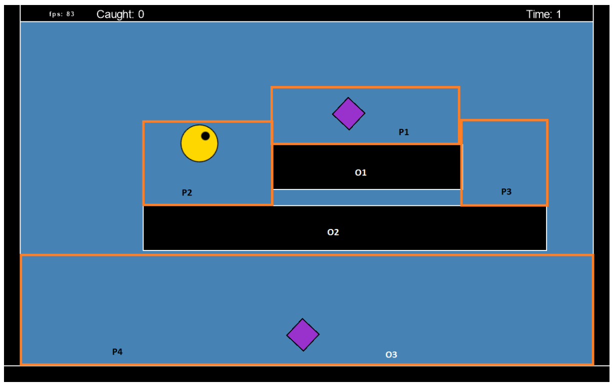

Fig. 3: Platform division example. The obstacle $O2$ is divided into platforms $P2$ and $P3$ as $O1$ restricts the movement of the circle in the ground platform.

IV. Solution

As discussed in Section I, we seek to overcome the over-specialization to the public levels of GF observed in previous AI systems. In order to achieve the necessary generalization ability, it is crucial that the agent can identify and solve patterns that are repeated throughout the game, instead of focusing on level-specific situations.

In order to increase the probability of finding those repeatable patterns, we use a divide-and-conquer approach to the problem of solving a GF level. In our approach, we divided that problem in three sub-problems (SP):

SP1 Catching all the diamonds that are on a platform;

SP2 Deciding the next platform to go;

SP3 Moving to the other platform.

With this division, we are reducing the number of possible game configurations the agent is looking for, as the agent can only be solving one of those problems at a given time. Therefore, the problem of GF is now solved by repeatedly solving the series of ($SP1 \rightarrow SP2 \rightarrow SP3$) existent in the level, starting from solving the platform where the character is initially placed. We are looking for repeatable patterns on SP1 and SP3 as solving SP2 always needs to take into account the layout of each level to decide where the character can move next.

For the definition of those three sub-problems, we need to divide the level into platforms to be solved. A platform is a region of an obstacle where the character can move without having the need to jump. In Fig. 3 we see that obstacle $o2$ is is divided into two platforms ($p2$ and $p3$) as obstacle $o1$ restricts the area of that obstacle that the circle character can reach. By splitting the game obstacles into different platforms in the agent's world representation, we reduce even more the number of possible configurations the agent faces without losing valid positions for the circle character.

The diamonds available on the level are assigned to one of those platforms. The attribution of a diamond to a platform is made in a left to right and top to bottom order. The diamond is always assigned to the platform that is exactly bellow him. This diamond assignment strategy has some flows, as it can assign a diamond to a platform from where it is impossible to catch it. This limitation is discussed later in Section VI.

The agent has a world-model that is queried whenever the agent needs to perform an action. In this model, the agent stores the list of platforms of the level it is solving together with the collectibles assigned to them. Moreover, it stores a navigation graph that is used as a map when the agent needs to decide which platform it should go next. The world-model also stores the current position, speed and radius of the character. The agent also has a knowledge-base that stores information that it uses to solve the sub-problems. Whenever the agents performs an action, it queries the knowledge-base for the best action to perform on the situation the agent is facing. This knowledge-base is updated in the end of each level with the information the agent gathered on it. This topic is discussed in more detail in Section IV-E,

We formulate the problem of finding the next platform to go (SP2) as path planning problem similarly to what other approaches to GF have done. We want to find a path, starting on the character's current position that goes through all platforms with diamonds to be caught. However, in our solution, we run a Depth First Search (DFS) instead of an A* algorithm. This decision has to do with the fact that our search space is very small, that we are not trying to find the optimal path and that a DFS is efficient to be ran several times while the agent is playing the level. More details on solving this sub-problem follow on Section IV-B.

To solve problems SP1 and SP3 we opted to use reinforcement learning. We are looking to learn two policies (one for each sub-problem) that the agent can follow to solve any possible configuration of the GF problem.

To use reinforcement learning, we need to define feature vectors that are used to calculate the reward of each game-state. SP1 and SP3 need different feature vectors as the problems differ very much from one another. To solve SP1, the agent only needs to capture features of the platform the character is in, whilst to solve SP3 it must take account the destination platform and the distances between the two platforms. Sections IV-A and IV-C discuss in detail the features used in SP1 and SP3 respectively.

A. *Solving one platform*

When solving one platform we assume that the character can catch all the diamonds assigned to it in a greedy way. This assumption, even though doesn't take into account that the character can fall off the platform without solving it completely, makes this sub-problem's feature-vector easier to define. With this simplification, the feature-vector only needs to capture the position of the diamond , within that platform, that is closer to the character. Although this may lead to sub-optimal solutions, it speeds up learning as the same value for this feature is repeated, regardless on where the other diamonds are positioned on the platform. To capture the relation of positions between the character and the diamond, the feature-vector also stores the distance vector that goes from the character's current position to the position of that diamond. The coordinates of the distance vector are measured as the number of circle radii that are needed to fill the distance between the character and the diamond.

Using only this vector to the closest diamond has the drawback of making it harder to track the number of available diamonds within the platform at a given time, which is an important feature to use in the calculation of the reward for the state. Another important feature to take into account is the presence of any obstacle that avoids the character from falling off the platform when it reaches the edge. If this feature is not used, depending on the previous training, the agent can either become careless and try to go at full speed towards a diamond that is on the edge of the platform or become too careful and go slowly and lose precious time. To capture such cases we take into account this obstacles' presence when the distance of the closest diamond to the platform edge is less than 5 times the circle radius (more than this distance the circle character can still reverse its movement without falling).

Another feature used is the distance in the xx axis from the character's current position to the leftmost and rightmost coordinates of the platform. The usage of this feature instead of using the relative position of the character to the platform as to do with the latter failing to capture the size of the platform. For instance, on a very tiny platform, the character being at 50% of it is not same as being on the middle of a large platform. On the former, a single move can make the character fall from the platform, whilst on the latter it would not fall. Nevertheless, both situations would be perceived as the same by the agent. This feature is also measured as the number of circle radii needed to fill the distance.

The horizontal speed of the character is also used as a feature. Since the speed value in the game varies from -200 to 200, we use an empiric constant of $1/20$ to weight this value. The speed value is truncated to an integer value to improve generalization of situations.

The reward for a state on this sub-problem must take into account the following features:

- The number of diamonds on that platform that were collected on the end of the level ($\#DiamondsFinal$). This is an indication of how good was the outcome of that platform was. The more diamonds we collected in the end of the level, the better our platform performance was and consequently the better the state is;
- The number of diamonds on the platform that were already collected by the time the agent was on that state ($\#DiamondsState$). The greater the number, the better the state is; Together with the $\#DiamondsFinal$ it gives the number of remaining diamonds;
- Distance to the closest diamond ($Distance$). The closer the agent. the better the state is.
- Percentage of time available ($TimeAvailable$). The more time the agent still has to solve the level the better the state is.

The reward function for this sub-problem, for a certain state s, is given by Equation 1. The distance factor has a greater weight as it is the feature that has most importance on this

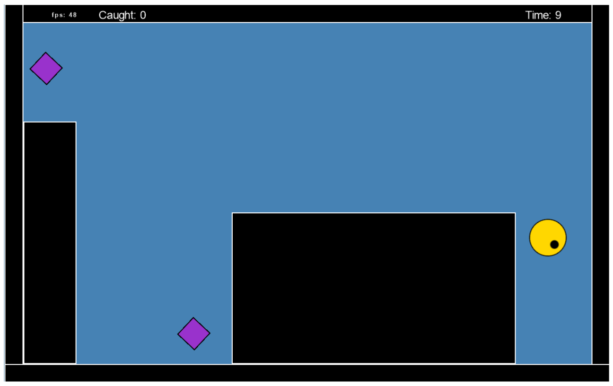

Fig. 4: Level from GF IEEE CIG 2014 AI Circle Single Track Competition. The circle will not be able to catch any diamonds due to the bad decision of falling to its right.

sub-problem.

$$\begin{aligned} reward(s) = &\#DiamondsFinal + \#DiamondsState \\ &+ 10 \times \frac{1}{Distance} + TimeAvailable \end{aligned} \quad (1)$$

This game-state definition is only applied when the character is on a platform which still has diamonds to be caught. When this is not the case, the agent needs to choose which platform the character should move to next.

B. *Planing next platform*

Making a good decision regarding the next platform to move to is crucial, since in certain levels a wrong decision can jeopardize the successful conclusion of that level, as can be seen on Fig. 4.

To make these decisions, the agent runs a modified version of a DFS in the navigation graph that is stored in its world-model. In our implementation, we removed the validation if the node was already visited from the classical DFS implementation as we may have to return a platform we already visited to go to others. In the graph, each node represents a platform of the level. Whenever it is possible for the character controlled by the agent to go from one platform to the other, there is a directional edge connecting the nodes representing those platforms. The creation of this graph is made before the first action on the level. In this graph, the edge stores not only the source and destination nodes, but also the x coordinate of the point where the edge has its start. To those specific points we give the name of jump-points. We use the jump-points to distinguish the edges whenever there are more than one edge connecting a pair of nodes. Another information stored in the edge structure is the difference in both xx and yy axis between the jump-point and the destination platform. This distance is calculated using the point on the destination platform that is closer to the jump-point. An example of such graph can be seen in Fig. 5.

The DFS returns the path (in nodes) that maximizes the number of diamonds that can still be caught. This algorithm assumes that for every platform, the agent can collect all the diamonds on it. Since it is often possible to go back and forth to another platform, the DFS can go into an infinite cycle. To

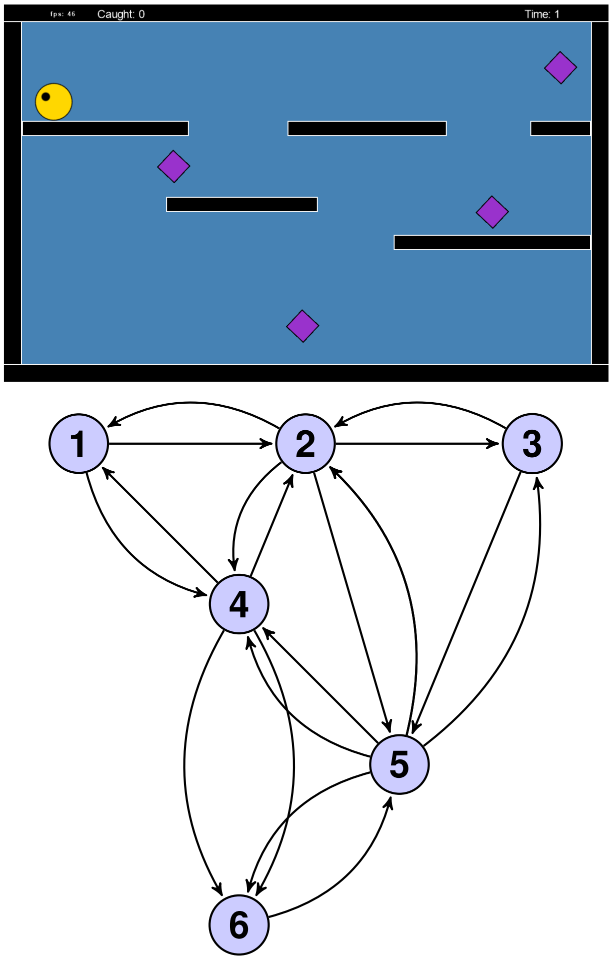

Fig. 5: The navigation graph models the level shown above. The nodes represent the obstacles on the level and the ground. There is one edge whenever it is possible to go from one platform to the other. Notice the two edges from node 4 to node 6.

avoid being caught in such a cycle, the search depth is limited to $2 \times (numberOfPlatforms - 1)$.

After computing the path, the agent checks what is the closest jump-point leading the character to the first platform of that path. The agent commits itself in moving to next platform through that jump-point until it leaves the platform it is in. This can happen when the agent jumps or fell off the platform. Once the agent lands on a platform again, and that platform doesn't have any diamonds to catch, it re-runs the DFS and chooses a new jump-point to move to. In certain cases, this new jump-point is the same as the one the agent was targeting before. Such situation happens when the agent fails to get to the intended platform and remains on the same platform it was before. The agent repeats the process of choosing a new jump-point until it reaches a platform where there are diamonds do catch.

The usage of this DFS is efficient as the depth of the search is always limited. This enables the agent to run the DFS every time the agent has to make a move. Moreover, it allows a quick recalculation of the path when a deviation from the original one is made.

C. *Moving to another platform*

After deciding to where the character has to go, the agent must produce a set of actions that makes the character it is controlling reach that target platform. As was stated before, this sub-problem is also modelled as a learning problem.

In this particular problem, the feature-vector has to capture the distance between the character and the jump-point the agent is targeting, the characteristics of the destination platform and the distance between the current platform and the destination platform. Moreover, these features must also capture the character's speed. The speed can be critical to fulfil the agent's goal. One of the examples on why speed is important, is the situation where the character needs to jump to a small platform. If the agent doesn't correctly control the speed, the character can jump over the intended platform instead of landing on it.

When moving to another platforms, the agent looks at the following features of the environment:

- Agent speed – integer value that ranges from -10 to 10 (uses the same 1/20 constant that was used when solving SP1);
- Distance to jump-point – integer value that indicates the distance in the xx axis between the character's current position and the jump-point position. It is measured as the number of circle radii;
- Distance vector – two-dimensional vector that stores the difference in the xx and yy axis between both platforms. It uses the same metric as the distance to jump-point feature.
- Landing platform size – the portion of the destination platform that can be used for the character to land on it. It is also measured as the number of circle radii.
- The edge the agent is committed to – the edge of the graph the agent is committed to solve.

Whenever the character gets to the intended platform, the time-stamp of that moment is stored in the edge of the graph the character traversed. This time-stamp is used to calculate how much time the agent took to move from one platform to the other.

The reward function for a given state on this sub-problem takes into account:

- The fraction of the time the agent spent moving to the next platform. This time is measured from the first time the agent commits to an edge until it reaches the destination platform of that edge.
- The fraction of the time from the first time the agent committed to the edge ($initialEdgeTimeStamp$), to the time taken when the agent played was on that state ($stateTimeStamp$). The closer that fraction is to 1, the higher the reward the state gets, as it is an indication that the agent was closer to solving the sub-problem.

The reward function for a given state s is calculated using Equation 2, where $totalEdgeTime$ and $levelTimeLimit$

represent the total time the agent took to get to the destination platform and the level time limit, respectively.

$$reward(s) = 100 \times \left(1 - \frac{totalEdgeTime}{levelTimeLimit}\right) \times \frac{stateTimeStamp - initialEdgeTimeStamp}{totalEdgeTime} \quad (2)$$

After solving all sub-problems, we need to define how the agent chooses its next action when it is called to do so.

D. Agent decision flow

When the agent is prompted for its next action, it updates its world-model according to the latest update from its sensors. The agent has sensors that can capture the following information:

- The character's current position and speed (two 2D vectors);
- The character's radius;
- Position of all the remaining diamonds ;
- Position and size of all the obstacles.

After finishing the update, the agent determines if there are still diamonds to be collected in the platform the character is in. If the platform is not solved yet, the agent will continue trying to catch all the diamonds on it. If the platform is solved, the agent decides which platform the character should go next and starts solving the sub-problem of moving towards that platform.

Regardless of the sub-problem the agent is in, it always queries its knowledge-base for the best move for its current state. If such state is not found, the agent will always play a random action from the set of possible actions. If the state is found, the agent chooses the best known action for that state with a certain probability x. Otherwise, it will also play a random action. In all situations, the set of possible actions are: roll left, roll right and jump. We didn't use the morph action of the circle as the outcome of that action can be achieved by a combination of jump and roll actions. In our work, the probability x was set to two distinct values, one for training purposes $x = 40\%$ and another for competition purposes $x = 80\%$. This percentage is the exploitation ratio of the agent and in the future can be set dynamically according to the number of states the agent knows at a given moment.

There are some particularities on the random choice of actions as the 3 actions do not have the same probability of being chosen. We detected that, if the jump action is done frequently, the learning process gets slower. This happens because the jumping action takes much more time to terminate than any other. When the agent jumps, it can only make another action when the character lands again. When the agent plays any other action it can choose another one as soon as it is again prompted to do so. To avoid frequent jumps, this agent only has a 5% probability of choosing the jump action when choosing a random action. Table I shows the probability of the agent choosing each action when playing randomly.

TABLE I: Actions probability when playing randomly

Action	Probability
Left	0.475
Right	0.475
Jump	0.05

E. Update and store learned values

When a level finishes, the agent must update its knowledge-base to take into account what happened during that run. The agent stores its knowledge-base in two separate files, one for each of the two learning sub-problems. Both files have the same structure. For each state identifier (s) there are values for each one of the possible actions (a). These values are real values and start at 0. The value 0 is used to represent lack of knowledge for the pair $< s, a >$.

For each action the agent performs, it stores the game-state and the action performed.

The update of the agent's knowledge for a given game-state s and an action a is made with the algorithm that follows. In the algorithm, we use $!a$ to represent all actions different from the action played and n to represent the current level. $n - 1$ is the level that was played immediately before.

```
if $s$ visited for the first time then
    reward($s$,$a$) =reward($s$,$a$,$n$)
    reward ($s$,!$a$) = 0
else
    reward($s$,$a$) = 0.95*reward($s$,$a$,$n-1$) +
                      0.05*reward($s$,$a$,$n$)
    reward($s$,!$a$) = reward($s$,$a$,$n-1$)
end
```

When the agent starts a new run, it loads all the information contained on both learning files. All the pairs $< s, a >$ with value 0 are ignored.

F. Training

The agent was trained in two distinct situations: simple levels to train specific scenarios, such as solving a small platform with two diamonds or jumping from one platform to another; and full levels similar to those featured in the GF AI Competition. As examples of the training levels designed we have the following situations:

- Solving a platform with a diamond in each edge (tested with and without the risk of falling from the platform);
- Solving the above situation but having the need to jump to catch the diamonds;
- Solving a 2-platform level with a diamond on the platform that was not the starting platform of the agent;
- Solving a "step" level where the circle has to consecutively jump from one platform to another until it reaches the one with the diamond;

TABLE II: Summary of competition results

	CIBot	Agent α
Runs Completed	50	22
#LevelsWon	5	5
Score	4337	2411

0

TABLE III: Percentage of diamonds collected (public levels)

#Level	CIBot	Agent α
1	100%	73%
2	100%	53%
3	100%	33%
4	30%	50%
5	50%	60%
Average	76%	54%
Std. Dev	30.07%	13.08%

The agent ran a total of 19871 games, on a set of 28 training levels with 5 of them being the public levels of the 2014 AI Competition, with a exploitation ratio was of 40%. The agent started without any prior knowledge so it always played randomly.

V. Evaluation and Discussion

To evaluate our agent, we put it playing the ten levels that were featured in the 2014's edition of the GF AI Competition. The results were taken in an Intel i7 processor at 2.4 Ghz and 16 GB of RAM. Windows 8.1 64-bit edition was used.

We compared our results with the winner of the competition, CIBot. Table II presents the summary of the performance of CIBot and our agent (Agent α) on the 2014 GF Competition. Our agent was ran with the same rules and time constrains as if it was also in competition. In the competition, the agents run 10 times on the same level in order to mitigate the chance factor. In total, our agent ran 100 runs. The update of knowledge values was disabled to avoid learning between runs on the same level. The last 5 levels of the competition were completely new to our agent. The row Runs Completed indicates the total number of runs in which the agent solved the level. Its value ranges from 0 to 100. The row #LevelsWon represents the number of levels where the agent had the best score of the two. The value ranges from 0 to 10. Finally, the Score row represents the final score of the agent. This value is calculated by averaging the score of each run. The score of a run is obtained using Equation 3 which is the same that was used on the GF AI Competition, where $V_{Completed}$ and $V_{Collect}$ are the bonuses for completing the level and catching a diamond , respectively. In our tests, to mimic what was done in the competition, those values were set to 1000 for $V_{Completed}$ and 100 for $V_{Collect}$. $agentTime$ is the time the agent took to solve the level, $maxTime$ is the time limit for the level being played and $N_{Collect}$ is the number of diamonds caught.

$$ScoreRun = V_{Completed} \times \frac{maxTime - agentTime}{maxTime} + (V_{Collect} \times N_{Collect}) \tag{3}$$

As can be seen on Table II, CIBot performs better overall and would still have been the winner even if our agent had taken part in the 2014 GF Competition. The huge difference in the final score between both agents is mainly due to the fact that the formula used to calculate the results favours having more runs completed than collecting more diamonds. To measure the performance of each agent on the public and private levels, we also looked at the percentage of diamonds each agent collected in each one of those level sets. Tables III and IV show the percentage of diamonds collected on the public and private level sets respectively. Through both tables, it is possible to notice that CIBot is the agent that collects the greater percentage of collectibles on the public levels, whereas Agent α is the one with a better performance in the private levels. Moreover, by the values of the standard deviation, we can see that Agent α has a more consistent behaviour on the levels. However, even more important than collecting more collectibles on the private levels or having a more stable behaviour than CIBot, is the fact that the difference between the percentages of the public and the private levels is very slim in our agent (only 1% difference). This difference shows that Agent α didn't become specialized on any of the level sets.

TABLE IV: Percentage of diamonds collected (private levels)

#Level	CIBot	Agent α
6	50%	70%
7	100%	27%
8	0%	76%
9	100%	73%
10	0%	20%
Average	50%	53%
Std. Dev	44.72%	24.42%

VI. Limitations

There are some limitations that we found on our approach. One of them is due to the specificities of the game while the others are typical limitations of the reinforcement learning problems.

A. Jumping problems

When the agent faces levels that have small platforms to which it needs to jump to, it often jumps over the platform instead of landing on it. However, when the circle manages to land, it has the correct speed and rarely slips out of the platform. An explanation can be the lack of training on these situations, either because there was not enough training runs for the agent to learn how to tackle such scenario or because levels didn't capture well such situation. Another minor problem found on our agent is the fact that it is unable to correctly map the diamonds that are between two platforms as the agent, while building the navigation graph, assigns that diamond to the platform immediately below it. However, if the diamond is located in a situation similar to the one depicted in Fig. 6, the diamond is assigned to a platform that is very far from it. For these types of diamonds', the catching must be done while jumping from one platform to the other, so the diamonds must also be able to be assigned to the edges of the graph.

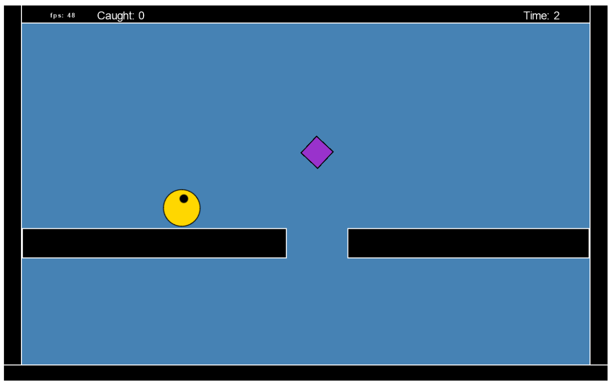

Fig. 6: Limitation of the agent. The diamond is assigned to the ground platform but is impossible to catch it from there.

B. Finding the next state

Another limitation in our solution occurs when the agent performs an action but its outcome generates the same game-state. This situation creates confusion on the reinforcement learning algorithm as the for the same pair $< state, action >$ two different outcomes occur. This goes against the deterministic world assumption that is made in [14]. This limitation only happens when is moving in a very slow way that the new state is perceived to be the same as the previous one. Currently, the agent only calculates the rewards when the level ends, so the problem is mitigated by only taking into account the reward and the outcome of the last of the repeated states. A way of avoiding this workaround is to increase the number of features captured by the game-state to get more detailed information. However, by doing so, we will have many states that are very similar. If we have a huge number of possible states, then the learning process will be much slower.

C. Finding known states

Currently the storage of the known states is done by using two different structures, one to store the knowledge on how to solve a platform and another to store the knowledge on how to go from one platform to another. These structures are indexed by the id of the state. The id is currently a string describing the features that are captured so as to easy the debugging task. However, if the knowledge-base becomes too big, both the loading and the searching in the knowledge-base can become the bottleneck of this approach. In the future, more intelligent ways of storing and searching through the knowledge have to be used in order to speed-up the decision process.

VII. Conclusion and Future Work

The presented solution applies a reinforcement learning algorithm together with path planning to solve the GF problem. Despite getting a final score much lower than CIBot, we are pleased to know that it got a better score in half of the levels played. This gives us confidence that a learning approach can be used to solve GF as it adapts better to new situations. Finally, our agent didn't become over-specialized to any level set, as it caught approximately the same percentage of diamonds on both level sets.

The same approach is going to be tested with the rectangle agent. This agent has to have a different set of features to match the character's specific motor capabilities. After that test, it is important to try to play in the cooperative track. An approach to the cooperation problem can rely on the emergence of cooperation when one agent sees the other as another platform on the game space.

We believe that this approach may be able to deal with the Human and AI Players Cooperation extension of the game discussed in the GF website. If the agents learn how to cooperate with another agent without knowing what algorithm the other is using, than those agents have a good chance of being able to cooperate with humans.

Acknowledgements

This work was supported by FCT – Fundação para a Ciência e a Tecnologia (INESC-ID multi annual funding) under project UID/CEC/50021/2013.

References

[1] F. Hsu, *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, 2002.

[2] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, "Checkers is solved," *Science*, vol. 317, no. 5844, pp. 1518–1522, 2007.

[3] D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. Kalyanpur, A. Lally, J. Murdock, E. Nyberg, J. Prager, N. Schlaefer, and C. Welty, "Building Watson: An overview of the DeepQA project," *AI Magazine*, vol. 31, no. 3, pp. 59–79, 2010.

[4] M. Bowling, N. Burch, M. Johanson, and O. Tammelin, "Heads-up limit hold'em poker is solved," *Science*, vol. 347, no. 6218, pp. 145–149, 2015.

[5] S. Gelly and D. Silver, "Achieving master level play in 9×9 computer Go," in *Proc. 23rd AAAI Conf. Artificial Intelligence*, 2008, pp. 1537–1540.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," arXiv:1312.5602, 2013.

[7] S. Ross and J. Bagnell., "Efficient reductions for imitation learning," in *Proc. 13th Int. Conf. Artificial Intelligence and Statistics*, 2010.

[8] J.-J. Tsay, C.-C. Chen, and J.-J. Hsu, "Evolving intelligent mario controller by reinforcement learning," in *Technologies and Applications of Artificial Intelligence (TAAI), 2011 International Conference on*, Nov 2011, pp. 266–272.

[9] J. B. Rocha, S. Mascarenhas, and R. Prada, "Game mechanics for cooperative games," in *Zon Digital Games 2008*. Porto, Portugal: Centro de Estudos de Comunicação e Sociedade, Universidade do Minho, November 2008, pp. 72–80.

[10] C. Fraga, R. Prada, and F. Melo, "Motion control for an artificial character teaming up with a human player in a casual game," 2012.

[11] D.-M. Yoon and K.-J. Kim, "Cibot technical report," http://gaips.inesc-id.pt:8081/geometryfriends/?page_id=476.

[12] D. A. Vallade Benoît and T. Nakashima, "Opu-scom technical report," http://gaips.inesc-id.pt:8081/geometryfriends/?page_id=476.

[13] L.-J. W. Yen-Wen Lin and T.-H. Chang, "Kuas-is lab technical report," http://gaips.inesc-id.pt:8081/geometryfriends/?page_id=476.

[14] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, pp. 237–285, 1996.