Mario AI - Gameplay Track
Developing a Mario Agent

Rik Claessens

June 19, 2012

Abstract

This article discusses the development of an agent for the Mario AI Competition Gameplay Track, using the $A*$ pathfinding algorithm. A dynamic goal generator is proposed to select different goals throughout a level. From the experiments, it is clear that the $A*$ is a well suited algorithm for pathfinding in the Gameplay Track of the Mario AI Competition. However, performance of the agent decreases rapidly on the higher difficulty settings of the game. The dynamic goal generator makes the agent play more humanlike.

1 Introduction

The Mario AI competition is based on the video game *Infinite Mario Bros.* [8], made by Markus Alexej Persson, a Swedish game developer, best known by his nickname *Notch*. The game is based on the platform game series *Super Mario Bros.*, made by Nintendo. It is made in java and available as an open source game.

Super Mario Bros. is a real-time 2D platform game. In this type of game the player controls a character, in an environment made out of different surfaces that have different sizes and heights (platforms) interspersed by gaps. Within the level, there are different kinds of enemies that pose a danger to the character. The goal is to survive and reach the end of the level by moving up, down, left, right or jumping. Of course, there are many platform games with variations on the definition above, but it includes the general characteristics most platform games have in common.

This article explains two algorithms which could be applied to platform game AI, $A*$ and *TBA**. A Mario AI agent is developed using the $A*$ algorithm. Whereas the agent from Baumgarten [10], the winner from the Mario AI Competition Gameplay Track in 2009, uses the $A*$ algorithm to determine the fastest path to the end of the level, a dynamic goal generator is proposed in order to obtain different goals throughout the level.

The following three research questions are formulated:

1. How does the $A*$ algorithm perform for a Mario AI Gameplay Track agent?
2. Is the *TBA** algorithm suited for a Mario AI Gameplay Track agent?
3. How does dynamic goal generation perform for a Mario AI Gameplay Track agent?

The structure of this article is as follows. First, Section 2 introduces the competition, its rules and a short history. In Section 3, the game environment (3.1) and the requirements for implementing an agent for the Mario AI Gameplay Track (3.2) are explained. Next, Section 4 introduces the $A*$ algorithm (4.1) and the *TBA** algorithm (4.2), which could be used by the Mario AI agent and their (dis)advantages are compared. Subsequently, Section 5 explains the implementation of the $A*$ algorithm and the dynamic goal generator, as well as the motivation for implementing $A*$ in favor of *TBA**, in detail. After this, Section 6 explains the experiments performed and discusses the results. Finally, in Section 7 a number of conclusions are drawn based on the performance of the Mario AI agent and a number of future research possibilities and areas of improvement are discussed.

2 The Mario AI Competition

The Mario AI Competition has been organized by Sergey Karakovskiy and Julian Togelius [10]. Together they organized the Mario AI Competition for the first time in 2009, at the CIG conference. In this edition there was only one track, *the Gameplay Track*, where the goal is to complete as many randomly generated levels as possible. Because there were two agents that completed all levels, the average computation time was the tie-break rule.

On the CIG conference of 2010, two new tracks were introduced to the competition: *the Learning Track* and *the Level Generation Track*. The learning track, similar to the gameplay track, instead focuses on online learning, where each agent is tested a large number of times for a single level where only the last attempts' score will

count for the competition. The level generation track does not test the agent AI, but level generators that create the levels for the agents to play. These generated levels are evaluated by human game testers live at the competition. For the competition in 2011, organized on multiple conferences, the organizers introduced a new track: *the Turing Test Track*. As the name suggests, this track requires the developers of the agents to let Mario play as a human would. The audience attending the competition has to vote whether they think a human is playing or a computer is controlling Mario.

In 2012 the Mario AI Competition is organized during the CIG conference in Granada, running in September. At the time of submission of this article as Bachelor Thesis for the Department of Knowledge Engineering, Maastricht University, it was not decided yet if the agent would participate in the Mario AI competition of 2012.

3 The Game Environment

This section explains the game environment. First, in Subsection 3.1, the similarities and differences with the original *Super Mario Bros.* games are discussed. Second, Subsection 3.2 explains the implementation details of the Mario AI Competition.

3.1 The Game

The game environment of the Mario AI Competition is similar to the original *Super Mario Bros.* series, with a few exceptions. The player controls Mario, the main game character. The goal of the game is to reach the end of the level, which is always located at the rightmost position of the level. Mario is able to perform the following actions: moving left or right, in the form of running or walking, jumping, ducking, shooting fireballs when Mario is in the *fire* state and picking up and throwing shells. An action which was not possible in the original *Super Mario Bros.* series is wall jumping. Mario is able to jump away from a wall he is sliding down, which makes it possible to prevent him from dying when he is falling into a gap.

Mario can have four different states in this game, as illustrated in Figure 1, from left to right: *fire*, *big*, *small* and *raccoon*. In this version of the game, Mario always starts in the *fire* state. When he is hit by an enemy he will subsequently go from *fire* to *big* to *small*. When Mario is hit in the *small* state, he dies. Mario will temporarily transform into a *raccoon* form when he picks up a shell, but this does not change his actual state, after releasing the shell Mario will be back in the state he was in before.

Super Mario Bros. games are also famous for the question bricks and the breakable bricks, in which power-ups or coins are hidden (Figure 2 from left to right: a red

Figure 1: The states of Mario

mushroom, a fire flower, a green mushroom and a coin on top of a number of question bricks and breakable bricks). Where the question bricks always contain a reward, the breakable bricks may contain nothing. Eating a mushroom will transform Mario in the *big* state when he is in the *small* state. Similarly, eating a fire flower will result in Mario being in the *fire* state, regardless of his current state. The coins Mario collects only count towards the agent's evaluation score. In the original *Super Mario Bros.* games, a green mushroom or collecting 100 coins would give the player an extra life, but this is not relevant for the Mario AI competition.

Figure 2: The power-ups and collectibles of Mario

There are also different kinds of enemies that form a danger for Mario in this platform game. They are represented in Figure 3. From left to right there is the *goomba*, *winged goomba*, the *red koopa*, the *winged red koopa*, the *green koopa*, the *winged green koopa*, the *spiky*, the *winged spiky*, the *piranha flower* and the *bullet bill*. The spiky cannot be killed by Mario. All other enemies can be killed by a fireball or shell, and all enemies, except the piranha flower, can be killed by jumping on them. When jumping on a koopa, the shell will be dropped for Mario to pick up and throw, or to jump on, which will make the shell move in the direction it was hit.

Figure 3: The enemies of Mario

The game has a variable difficulty. On the easiest difficulty there are no gaps in the level. The only enemies are the *goomba* and the *piranha flower*. On higher difficulty settings, Mario faces a higher number of enemies, now including the *winged goomba*, the *red* and *green koopa*, which can also be winged, and the *bullet bill*. Additionally, there are gaps that Mario has to jump over. Even higher difficulty settings include all types of enemies presented in Figure 3 and have increasing number of enemies and gaps.

3.2 The Competition Environment

In order to run a competition based on the game *Infinite Mario Bros.*, the organizers had to modify the structure of the game. First, they changed the continuous-time element of the game to a discrete form. This allows the world state to be advanced in time steps. The interval of the time steps is 40 milliseconds, corresponding to a rate of 25 frames per second.

Second, they provided an agent interface to allow the implementation of an agent for the competition. At each time step, the game provides the agent with the information about the environment. This includes the structure of the level, the location of Mario, the location and type of the enemies and a number of different statistics like number and type of kills and the health status of Mario (*fire*, *big* or *small*). Only information about the level and enemies that is on-screen at the current time step is received by the agent. The agents' job is then to determine an action based on this information in the form of a combination of basic actions, *right*, *left*, *down*, *jump* and *speed*. The speed button also will make Mario shoot a fireball when pressed while Mario is in the *fire* state.

The information the agent receives at each time step comes in different information levels. The structure of the map is represented by a 19×19 matrix containing values, which can be used by the agent to determine the type of world element present at that location. The most basic level of information about the environment tells an agent if there is a passable obstacle or not, represented by a *0* or *1* at the corresponding position in the matrix. The second information level makes a distinction between the platforms, the bricks and the enemies that are part of the environment (the flower pot and the cannon). The highest level of information tells exactly which type of world element is present except for the locations of the hidden rewards. Only on this point, the level information differs from the information the game engine uses.

The information about the enemies comes in two different forms. The most accurate form is a list containing the type of enemy and his $x-$ and $y-$ coordinate on the map. The second form is similar to the structure of the level information, a 19×19 matrix containing constants, which in turn represent the different types of enemies. The first form is more accurate since for the matrix, the coordinates of an enemy get rounded to the nearest cell. The agent is able to receive three possible information levels. The first one is again the roughest of the three, it only provides the information if there is an enemy or not at a certain location on the map. The second level of information distinguishes between enemies Mario is able to kill by jumping on them and the enemies that will hurt him if he jumps on them. As all enemies, except spiky (which cannot be killed by Mario), will die from an impact with a shell or getting hit by a fireball, it is of no importance for further separating enemy types into groups. The most accurate level of information provides again the exact same information as used by the game engine, only neglecting some graphical elements which are of no importance to the agent, such as the blinking animation of coins.

In order to implement an agent for the Gameplay Track of the Mario AI Competition, each agent always has to perform two operations: handling the information received from the game and determining an action for Mario to perform the next time step. For the competition, an agent may on average not exceed 40 ms of computation time. Otherwise the agent is disqualified from the competition.

The winner of the Gameplay Track is determined by letting the agents play 40 prior unknown levels, increasing in difficulty. The agent with the highest total distance covered wins the competition. If there are agents that complete all levels, an evaluation function is used to calculate how well the agent performed. During the competition in 2009, this evaluation function was not used. Instead, as there were agents that completed all levels, the agent with the least amount of computation time was declared the winner. Because the total distance covered was the only goal for the competition in 2009, all submitted agents only tried to survive in order to reach the end of the level. The evaluation score is based on the weighted sum of different features of the game and is given by Formula 1. The weight (w_i) and range (r_i) of all features (f_i) is given in Table 1.

$$\sum_i w_i \times f_i \tag{1}$$

These evaluation weights pose interesting heuristic possibilities, because the evaluation features are interrelated. This means that some actions directly influence a certain feature while indirectly affecting other features. For example, when Mario is in the *big* state, the only benefit from eating a mushroom is an increase of the evaluation score of 58. However, when eating a mushroom while Mario is in the *small* state, the agent benefits from an additional increase of the evaluation score of 32 for transforming into the *big* state, and indirectly has an increased chance of surviving the level. The same principle holds for eating a flower, there is a cumulative benefit from an increase in different statistics.

In the exceptional case, where there are agents which complete all 40 levels and have the same evaluation score there are four tie-break rules for the 2012 competition:

1. Total number of enemies killed (more is better)
2. Total number of coins collected (more is better)
3. Total in-game time left (more is better)
4. Total computation time (less is better)

f_i	w_i	r_i
Distance covered	1	0 - 4096
# of flowers devoured	64	0 - # of flowers
Level completion bonus	1024	0 or 1
Mario mode	32	0, 1 or 2 for *small*, *big* or *fire*
# of mushrooms devoured	58	0 - # of mushrooms
# of coins collected	16	0 - # of coins
# of hidden blocks found	24	0 - # of hidden blocks
# of kills	42	0 - # of enemies [1]
# of kills by stomp	12	0 - # of "stompable" enemies [2]
# of kills by fire	4	0 - # of enemies [1]
# of kills by shell	17	0 - # of enemies [1]
Time left	8	0 - 200

[1] The maximum number of kills is equal to the total number of enemies substracted with the number of spikies.
[2] Mario cannot kill the *fire flower* by jumping on it.

Table 1: Evaluation function features, including their weight and range

4 Pathfinding Algorithms

An agent playing a 2D platform game would need to find an optimal path through the level while avoiding different kinds of dangers. In the game of *Super Mario Bros.*, Mario has to get to the end of the level while avoiding gaps (static dangers) and enemies (dynamic dangers). In order for the agent to achieve its goal, a suitable pathfinding algorithm is required.

This section introduces the $A*$ algorithm and an enhanced version of this algorithm, *Time-Bounded* $A*$ ($TBA*$).

4.1 A*

A fundamental algorithm for pathfinding is the $A*$ algorithm. Although the core of the A* algorithm is as old as 1968 [4], it is still one of the most used algorithms for pathfinding. The algorithm applies best-first search to find a path with the lowest possible cost from a given start node to a goal node. The algorithm uses a heuristic function given by Formula 2. The heuristic function $f(n)$ is the sum of two elements. The first element, $g(start, n)$, is the cost of the path from the start node *start* to the current node n. The second element, $h(n, goal)$, is the estimated cost of the path from the current node n to a goal node *goal*.

$$f(n) = g(start, n) + h(n, goal) \tag{2}$$

The A* algorithm keeps track of the search progress using two lists, the *open* and the *closed* list. All nodes which have been generated, but not yet expanded are stored in the *open* list, while expanded nodes are stored in the *closed* list. The algorithm uses iterations to reach a solution. At each iteration, the node from the *open* list with the lowest f-value, is expanded. When there is a tie-break for the lowest value, a tie-break rule has to select which node the algorithm will expand.

Algorithm 1 A*, *Procedure A**

Input: State space with Start node *start*, Distance function d, Heuristic h, Successor generation function *Expand*, and Goal node *goal*
Output: Cost-optimal path from *start* to $t \in T$, or $\emptyset$ if no such path exists

```
closed ← ∅
open ← start
f(start) ← h(start)
while open ≠ ∅ do
  Remove u from open with minimum f(u)
  Insert u into closed
  if u = goal then
    return Path(u)
  else
    Succ(u) ← Expand(u)
    for all v in Succ(u) do
      Improve(u,v)
    end for
  end if
end while
return ∅
```

Algorithm 2 A*, *Procedure Improve*

Input: Nodes u and v, v successor of u
Side effects: Update parent of v, $f(v)$, *open* and *closed*

```
if v in open then
  if g(u) + d(u,v) < g(v) then
    parent(v) ← u
    f(v) ← g(u) + d(u,v) + h(v)
  end if
else if v in closed then
  parent(v) ← u
  f(v) ← g(u) + d(u,v) + h(v)
  Remove v from closed
  Insert v into open with f(v)
else
  parent(v) ← u
  Initialize f(v) ← g(u) + d(u,v) + h(v)
  Insert v into open with f(v)
end if
```

When expanding a node, all successor nodes from the current node are generated and inserted in the open list when the successor node is neither in the *closed* list nor in the *open* list with a lower g-value. The node which is expanded gets inserted into the closed list. This is to

avoid that nodes are re-expanded. Also the *closed* list is used to retrieve the path from the start node to a goal when a solution is found. The process described above is repeated until a goal node is selected from the *open* list, which means a solution has been found. The path from the start node to the goal is then backtracked using the closed list.

The A* algorithm is proven to be complete [9], guaranteeing that a solution will be found if one exists. The optimality of the solution is guaranteed when the heuristic function is admissible. Admissibility means that the heuristic function never overestimates the cost of path, i.e. is optimistic. This condition is sufficient to ensure optimality when the state space is a tree. However, the state space for this game is a graph. To guarantee optimality in this case, the algorithm requires an additional constraint in the form of a consistent (or monotonic) heuristic function. The requirement of consistency (Formula 3) holds when the estimated cost from node n_1 to a goal node is smaller than or equal to the true distance (the d-value) from node n_1 to node n_2 plus the estimated cost from node n_2 to a goal node. A consequence of a consistent function is that the f-values along a path are non-decreasing.

$$h(n_1, goal) \leq d(n_1, n_2) + h(n_2, goal) \tag{3}$$

The pseudo code for the standard $A*$ algorithm is given by Algorithms 1 and 2.

A disadvantage of the $A*$ algorithm is that all expanded nodes are kept in memory for each iteration. When the state space is too large, the $A*$ algorithm is not able to find a solution with limited memory. A depth-first variant of $A*$, called *Iterative Deepening* $A*$ (*IDA**) [5], reduces memory usage by only storing the nodes of a single path at a time in memory. However, the state space of the Mario AI Gameplay Track is of such size that the $A*$ does not store so many nodes that it runs out of memory.

4.2 TBA*

The real-time aspect of video games makes them a challenging subject for different types of agent algorithms. An agent for the Mario AI Gameplay Track has to calculate its next action within 40 ms. The standard $A*$ algorithm is not designed to handle this extra constraint, since it requires a complete solution before an action is selected. When the state space increases in size, the required processing times also grows, increasing the chance the algorithm is terminated prematurely. To cope with this time constraint, *Time-Bounded* $A*$ was proposed [1].

The *TBA** algorithm falls into the category of real-time heuristic search algorithms [1]. The main difference with the $A*$ algorithm is that *TBA** interleaves the planning and execution phase. Nodes are expanded, just like with $A*$, in the direction of the goal. However, to ensure real-time behavior, the search is periodically halted and the agent performs the most promising action found so far. Unlike $A*$, *TBA** does not start with an empty *open* and *closed* list at each time step. Instead, it continues the search using the *open* and *closed* lists from the last time step. While each time step both lists grow in size, real-time behavior is still guaranteed when the times it takes to expand or backtrack each node is constant-bounded. This is achieved by a careful choice of the data structure used for the two lists, providing amortized constant-time operations for the algorithm. For the open list, two different data structures are used. A hash table is used to check if a node is already in the open list. Additionally, a priority queue, implemented using a heap, is utilized for selecting the best node. The

Algorithm 3 TBA*

Input: Start node *start*, Goal node *goal*, State space P, Node expansion limit N_E, Traceback limit N_T

```
loc ← start
solutionFound ← false
solutionFoundAndTraced ← false
doneTrace ← true
while loc ≠ goal do
  PLANNING PHASE
  if not solutionFound then
    solutionFound ← A*(lists, start, goal, P, N_E)
  end if
  if not solutionFoundAndTraced then
    if doneTrace then
      pathNew ← lists.mostPromisingState()
    end if
    doneTrace ← traceBack(pathNew, loc, N_T)
    if doneTrace then
      pathFollow ← pathNew
      if pathFollow.back() = goal then
        solutionFoundAndTraced ← true
      end if
    end if
  end if
  EXECUTION PHASE
  if pathFollow.contains(loc) then
    loc ← pathFollow.popFront()
  else
    if loc ≠ start then
      loc ← lists.stepBack(loc)
    else
      loc ← loc_last
    end if
  end if
  loc_last ← loc
  move agent to loc
end while
```

closed list is stored in a hash table.

The *TBA** algorithm is complete [1]. Additionally, the agent will backtrack towards the start node when there is no path to the goal from the current node. The pseudo code for the algorithm is given by Algorithm 3. The *TBA** algorithm utilizes a standard *A** search. A difference with the given Algorithm 1 is that the extra parameter N_E is used to cut of the while loop when N_E nodes are expanded. Also, not all parameters (d, h and *Expand*) are explicitly given as input for the *A** algorithm.

5 Implementation

This section discusses the implementation details of the developed agent for the Gameplay Track of the Mario AI Competition. As explained in Section 4, two possible algorithms were considered for the Mario agent, the standard *A** algorithm and the time-bounded version of *A**, called *TBA**.

5.1 A* Agent

The developed agent is based on the standard *A** algorithm. However, before the pathfinding algorithm is started, the agent performs a number of steps. First, the agent receives the latest information from the game environment. Only the structure of the level which is on-screen at the current time step is received by the agent. This part of the level is mapped to a simulated world state of the complete level. Second, the type and location of each enemy is obtained. The agent tries to match each enemy to a corresponding enemy already present in the simulated world state and updates its location and speed accordingly. When a match cannot be found, a new enemy is added to the set of enemies in the simulated world state. The third step in updating the simulated game environment is updating the position and speed of Mario.

For the information of the level as well as the information of the enemies the agent uses the highest information level. They give the agent the most accurate simulated world state. The available information levels are explained in Subsection 3.2.

When the simulated world state has been updated with the latest information from the game, the agent selects its goal coordinates for this time step. The pseudo code for selecting the goal is given by Algorithm 4. First the agent checks if a power-up has appeared on screen. If this is the case, this is the goal location for the agent. If there is no power-up on-screen, the agent tries to stomp the closest "stompable" enemy. When the increase in the evaluation score for killing the enemy is nullified by the time it takes for the agent to kill it, this enemy will not be considered as a goal. If there is neither a power-up nor a "stompable" enemy close to Mario, the agent selects the cell above the highest and rightmost platform, which is known. The agent selects the cell above a platform to avoid selecting a space over a gap as a goal. Planning to the end of the level is not useful as only the part of the level which is on-screen is known to the agent.

Algorithm 4 Goal Generator

```
Input: current world state

if Power-up on screen then
    goal ← location(power–up)
else if Enemy close then
    goal ←cell above closest enemy which can be killed by
    jumping on it
else
    goal ←cell above highest and rightmost platform
end if
return goal
```

After selecting the goal, the *A** pathfinding algorithm searches for the optimal path from the current game state to this goal. The pathfinding algorithm is different from the standard *A** algorithm in four ways.

First, an extra constraint is added to the while loop from the open list. Now the loop is executed when there are still unexpanded nodes in the *open* list and the elapsed time since the start of the search is smaller than 40 ms.

The second difference is a direct consequence of the added time constraint. It is possible that the algorithm has not yet found a goal node when it runs out of time and is terminated prematurely. Therefore, it is necessary to keep track of the best node so far. So when the algorithm is terminated because of violating the time constraint, the path to this best node is selected instead of the path to a goal node.

The third difference concerns the selection of the node to expand. The standard *A** picks the node with the lowest f-value. In order to avoid Mario getting hurt, the agent selects the node with the lowest f-value from all nodes where Mario did not get hurt. This increases both the evaluation score for the agent and the chance of completing the level. The tie-break rule for equal f-values applied to this agent is to pick the node with the highest g-value. As the g-value is the real cost along the path so far, a higher g-value means a bigger part of the f-value is certain.

The fourth and final difference from the standard *A** algorithm is the fact that not all possible nodes are expanded. To decrease the chance of violating the time constraint, nodes where it is not useful to jump, are not expanded. The *jump* action only affects the movement of Mario in two situations. In the first situation Mario can initiate a jump when he is walking on a platform or

sliding down a platform. The second situation is when Mario has not achieved the maximum jump height yet. Mario is able to control the height of his jump by performing the *jump* action a number of consecutive time steps. Only nodes where Mario is in one of the two situations mentioned above, are expanded. This decreases the number of expanded nodes and therefore the total computation time.

As heuristic function, the algorithm calculates the minimum number of time steps required for the agent to reach the goal, assuming maximum acceleration from the current position and neglecting any obstacles.

The real distance between nodes after simulating Mario's action, is also calculated using the minimum number of time steps still required to reach the goal. Additionally, the node gets a penalty if Mario gets hurt or dies. By doing this, paths where Mario does not get hurt or dies have a higher cost than paths where Mario obtains no damage. This is a big difference with the agents which were submitted to the Mario AI Competition in 2009. They only tried to reach the end of the level as fast as possible.

After implementing the $A*$ agent, the dynamic nature of the game posed a bigger problem than the time constraint. The algorithm is able to determine a path that avoids enemies but has difficulties reacting when the game world suddenly changes. This may be caused by the fact that each time step the agent needs to match the information on the enemies it receives from the API to the enemies that are present in the simulated world state. When there are multiple enemies close to each other, this is error-prone.

5.2 TBA* Agent

As explained in Subsection 4.2, the standard $A*$ algorithm is not designed to handle a time constraint as it will only terminate if a solution is found. From the Mario AI Competition in 2009 [10], it is clear that the state space of the game is of such a size that it allows agents based on the $A*$ algorithm, to find a goal node without violating the time constraint. The $A*$-based agent which needed on average the most time per time step, required 15.19 ms/time step. Additionally, experiments (Section 6) showed that the agent described in Subsection 5.1 never ran out of computation time before calculating a path to a goal node. Because of these reasons, improving the $A*$ based agent was preferred to implementing a $TBA*$ agent.

6 Experiments and Results

In this section the performance of the developed Mario AI agent is discussed. First, in Subsection 6.1, the experimental setup is explained. Second, the results are given and discussed in Subsection 6.2. Conclusions about the performance of the agent are drawn in Section 7.

6.1 Experimental Setup

In order to test the performance of the implemented agent, a series of experiments were performed. These experiments were run on the KECS cluster of Maastricht University. Each node on the cluster has a quad core AMD Opteron F2216 2.4 GHz processor, 8 GB of memory and is running Linux kernel version 2.6.18.

Because the experimental setup of the competition in 2009 is not available, custom experiments were designed. All experiments are run on 100 levels, with level seeds 1 to 100. Each level is run five times. Additionally, the experiments are performed on three different difficulty settings. The first difficulty level consists of levels without gaps and the only enemy types are the *goomba* and the *piranha flower*. On the second difficulty setting Mario faces levels with gaps and includes the *koopa* enemy type, as well as the *bullet bill*. The third and highest difficulty setting consists of levels with more gaps and includes all different enemies in the game.

The performance indicators of the Mario AI agent, in order of importance for the competition, are the following:

1. Average distance covered (each level has a length of 256 cells)
2. Average evaluation score
3. Kill ratio
4. Coin collection ratio
5. Average in-game time left
6. Average computation time

Not every statistic gives a lot of information about the performance of the Mario AI agent. This is because every statistic is interrelated with others. For example, the fact that the average in-game time left is greater on the two higher difficulty settings can have two causes. Either Mario reaches the end of the level faster, or he dies faster. From the average distance covered on the higher difficulty settings it is clear that the latter is true. The same argument holds for the number of enemies killed. A conclusion cannot be made if the agent is better at killing enemies when selecting them as a goal, because the agent also dies faster, i.e. Mario comes across fewer enemies.

In order to compare the influence of the dynamic goal generator, the considered alternative goals were enabled individually. First, the performance of the agent is tested when always choosing the cell above the highest and rightmost platform as a goal (Platform). Second, experiments test the performance of the Mario agent with

Difficulty	Avg. Distance	Avg. Evaluation Score	Kill ratio	Coins collection ratio	Avg. in-game time left	Levels completed
Platform						
0	230.2 ± 4.1	6682.9 ± 149.7	29.8 ± 0.7%	31.1 ± 0.9%	108.1 ± 7.1	42.8%
1	61.3 ± 2.3	2586.1 ± 70.2	5.8 ± 0.4%	5.8 ± 0.3%	153.6 ± 6.2	0.0%
2	46.2 ± 1.5	2379.1 ± 49.9	3.7 ± 0.3%	3.9 ± 0.2%	167.8 ± 5.1	0.0%
Power-ups						
0	225.8 ± 4.6	6516.3 ± 154.1	29.9 ± 0.8%	30.6 ± 0.9%	98.5 ± 7.4	40.2%
1	62.8 ± 2.3	2559.1 ± 71.9	6.1 ± 0.4%	6.1 ± 0.3%	144.6 ± 6.8	0.0%
2	48.8 ± 1.7	2366.2 ± 59.1	3.6 ± 0.3%	4.1 ± 0.2%	159.8 ± 5.7	0.0%
Enemies						
0	144.4 ± 6.3	4625.3 ± 170.8	20.7 ± 1.1%	33.5% ± 1.4%	95.0 ± 6.6	9.4%
1	33.0 ± 1.5	1955.6 ± 63.1	3.4 ± 0.3%	5.9 ± 0.3%	137.6 ± 7.0	0.0%
2	26.4 ± 1.3	1878.6 ± 57.3	2.0 ± 0.3%	4.2 ± 0.2%	146.1 ± 6.7	0.0%
Power-ups and Enemies						
0	151.9 ± 6.1	4743.4 ± 172.1	21.9 ± 1.0%	35.7 ± 1.5%	82.3 ± 6.5	10.0%
1	33.4 ± 1.5	2002.5 ± 65.1	3.5 ± 0.3%	5.9 ± 0.3%	142.1 ± 6.8	0.0%
2	27.4 ± 1.3	1893.9 ± 59.7	2.1 ± 0.3%	4.3 ± 0.2%	145.5 ± 6.7	0.0%

Table 2: Mario AI Agent Results, 100 levels, seeds 1 - 100, 95% confidence intervals

dynamic goal generation. In one experiment only power-ups are considered as an alternative goal (Setup *Power-ups*), then only enemies (Setup *Enemies*). Finally, both enemies and power-ups are considered to be goals (Setup *Power-ups and Enemies*). The results are presented in Table 2.

6.2 Results

One important thing to note is that some levels are impossible to complete. Because of the randomness of the level generator, it may sometimes happen that the end of the level is unreachable. Therefore the total distance is a better indicator than the number of completed levels. Additionally, completing a level increases the evaluation score by 1024. Therefore, the average evaluation score shown in Table 2 is slightly lower than it would be if all levels were possible to complete. For the sake of being complete, the percentage of the levels completed by the Mario AI agent is also presented in the last column of Table 2.

The experiments revealed that selecting a dynamic goal does not have any noticeable effect on the average computation time of the agent. The agent has an average computation time for selecting its move of 31 ms/time step.

The implemented Mario agent covers about 90% of the total distance for the lowest difficulty setting when selecting the highest rightmost platform as a goal. When increasing the difficulty setting of the game, the total distance Mario is able to cover decreases drastically. Selecting power-ups as goals does not have a big impact on the agent's performance, although the performance is slightly worse. Also it ensures the agent to play more humanlike, because human players also tend to collect power-ups before advancing. Selecting "stompable" enemies as a goal, increases the number of kills noticeably. However, while trying to stomp enemies, Mario is in danger more of the time than when trying to reach the end of the level as fast as possible. Because of this increased danger, Mario is also killed by enemies more often, which results in a lower kill ratio.

7 Conclusion and Future Research

Because the experimental setup of the available results from the Mario AI competition held in 2009 was not available, a definite comparison with the proposed approach could not be made. However, from the performance of the agent using different heuristics for selecting the goal at each time step, a number of conclusions can be drawn.

As expected, the A^* algorithm is a well suited algorithm for the Mario AI Gameplay Track. The agent is able to cover on average 230.2 ± 4.1 of the 256 cells for the lowest difficulty setting by selecting the fastest path to the highest rightmost platform. By selecting this goal, Mario often avoids enemies by moving over them on higher platforms.

The dynamic goal generation performs worse than ex-

pected. The reason for it is that Mario has to follow more dangerous routes. As the *A** algorithm performs worse when dealing with enemies, the overall performance of the agent also decreases.

While researching the possible algorithms that could be applied to an agent for the Mario AI Gameplay Track, the time constraint for selecting an action seemed to be important to take into account. However, the Mario AI game world is of such a size that the standard *A** algorithm is able to find a goal node within the time constraint, with an average of 31 ms of computation time. Because the time constraint proved not to be the limiting factor for the *A** algorithm, the *TBA** algorithm is expected to not improve the performance of the agent.

As discussed in Section 5.1, the *A** algorithm has difficulties with the dynamic game environment. Especially when there are many enemies close to each other, the implemented algorithm cannot respond sufficiently. Also, the *A** algorithm is not designed for the possibility that the agent may have to wait to guarantee a safe passage.

For future research, enhancements have to be made to the implemented algorithm in order to handle the dynamic game environment better. When the performance of the algorithm dealing with enemies improves, the dynamic goal generation is expected to perform better too. This could be achieved by performing *A** multiple times. First, the general route the agent takes can be selected. Then a more thorough local search can be performed to determine the action sequence the agent will perform on the selected route.

An improvement of the dynamic goal generation can be achieved by making a more careful consideration when selecting a goal. Whereas the developed agent will select an enemy when killing it will result in an increase of his evaluation score, the agent should also consider the level of danger selecting this enemy as a goal.

References

[1] Bulitko, V., Björnsson, Y., and Sturtevant, N. (2009). TBA*: Time-Bounded A*. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 431–436. AAAI Press, Pasadena, California.

[2] Bulitko, V., Björnsson, Y., Sturtevant, N., and Lawrence, R. (2011). Real-time Heuristic Search for Pathfinding in Video Games. *Artificial Intelligence for Computer Games* (eds. P. González-Calero and M. Gómez-Martín), pp. 1–30. Springer New York.

[3] Edelkamp, S. and Schrödl, S. (2012). *Heuristic Search Theory And Applications*. Elsevier.

[4] Hart, P., Nilsson, N., and Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, Vol. 4, No. 2, pp. 100–107.

[5] Korf, Richard E. (1985). Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, Vol. 27, No. 1, pp. 97–109.

[6] Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., and Thrun, S. (2005). Anytime Dynamic A*: An Anytime, Replanning Algorithm. *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)* (eds. S. Biundo, K. Myers, and K. Rajan), AAAI Press.

[7] Millington, I. and Funge, J. (2009). *Artificial Intelligence for Games*. Morgan Kaufmann, 2nd edition.

[8] Persson, M. (2012). Infinite Mario Bros. `http://www.mojang.com/notch/mario/`.

[9] Russel, S. and Norvig, P. (2003). *Artificial Intelligence, A Modern Approach*. Pearson Education, 2nd edition.

[10] Togelius, J., Karakovskiy, S., and Baumgarten, R. (2010). The 2009 Mario AI Competition. *IEEE Congress on Evolutionary Computation 2010 (CEC)*, pp. 1–8.