A Rapidly-exploring Random Trees based approach for General Video Game Playing

Extended Abstract of MSc Dissertation

Rui Silva
Instituto Superior Técnico
Lisboa, Portugal
Advisor: Professor João Dias

ABSTRACT

General Video Game Playing (GVGP) is a field of Artificial Intelligence (AI) where the goal is to design an agent that can play a variety of video games successfully. This field approximates the important problem of general AI. Rapidly-exploring Random Trees (RRT) is a search algorithm known to perform well in a variety of domains, however it has not been used so far in a GVGP agent.

In this work, we investigate how RRT can be applied as the exploration algorithm of a GVGP agent. We identify the problems that prevent RRT from being directly applied in a GVGP agent and propose three variants of it, that circumvent those problems. We also investigate two possible enhancements for the algorithms: Species First (SF), a novel technique designed by us to avoid the repeated exploration of identical states and Tree Reuse (TR), a common technique in GVGP agents.

The proposed solution is evaluated using the deterministic single-player games of General Video Game AI (GVG-AI), a GVGP competition. Results show that the variants of RRT we propose make for viable GVGP agents, however not competitive with the best vanilla algorithm, Monte Carlo Tree Search (MCTS) – average win ratio of 12.9% on our best variant vs 17.0% for MCTS. The SF enhancement proved beneficial for our agents, being part of our best agent, which has a win ratio of 14.2%.

KEYWORDS

Rapidly-exploring Random Trees, General Video Game Playing, General Video Game AI, Artificial Intelligence for Games, video games

1 INTRODUCTION

GVGP is the field of AI regarding the problem of creating AI players (agents) that are capable of playing any video game. Considering that video games have theoretically unlimited constraints, GVGP is an approximation of general artificial intelligence, which is one of the most important goals of AI [11].

Traditionally, game playing agents are developed specifically for a game, using game-specific heuristics. However, agents in GVGP cannot take advantage of such heuristics given that the games available to train the agents are different to the evaluation games. The main challenge of GVGP is that the agent should be sufficiently general to learn the rules of any game, guide searches to relevant states, and be able to adapt to a large spectrum of situations.

Research on GVGP focuses on problem-independent algorithms, which are designed to be effective with little to no domain knowledge. Some previously developed agents for GVGP using these algorithms, guide searches exclusively by the game score value, which is one of the few pieces of information available to the agents.

RRT is a problem-independent search algorithm that has been used with success in a variety of domains [2, 3, 17], including game-specific playing agents [13], and has yet to be applied to GVGP. Its characteristics and similarities with other algorithms used with success in GVGP suggest that an RRT approach may be appropriate for a GVGP agent.

In this work we will determine if RRT is a viable exploration algorithm for a GVGP agent.

The rest of the report is organized as follows. Section 2 describes GVGP and GVG-AI in more detail. In section 3 we survey the related work. In section 4 we describe the proposed solution and how we have implemented it. Section 5 describes the setup and results of the experiments performed to evaluate our solution. Finally, in section 6 we present the conclusions of our work and some ideas for future research.

2 GENERAL VIDEO GAME PLAYING

GVGP is a field of AI concerned with the design of agents that are able to play every game. It can be considered an extension of General Game Playing (GGP) into the realm of video games [6].

As it is not possible to evaluate an agent in all games, the performance of an agent is measured by its ability to play a varied set of games (called evaluation games). For the evaluation results to be reliable it is essential that:

1. The set has a large number of elements, because the more evaluation games are used the closest we are from evaluating the ability of the agent to play all games, i.e. we are closer to the goal of GVGP.

2. The set be as heterogeneous as possible – i.e. it is essential that games of different types are included, like puzzles, mazes and games that require fast reaction times, because the more varied the set is the more skills will be required to play the games and the more skills are required the closest we will be from a general agent. Otherwise, if all the evaluation games are of the same type for example mazes we would be testing only the ability of the agent to solve mazes.

3. The evaluation games are not the same that were used to train the agent, otherwise we would be evaluating only the ability of the agent to play the training games.
For the creators of the agents to be able to create an agent capable of playing so many types of games, it is essential that there is a single interface to interact with them. The interface should be flexible enough to abstract the differences between the games and provide a reduced set of primitives that work on every game. This interface is important because without it the creators of the agents would need to foresee all the types of games that their agents would interact with, which would impossibilitate the agent from playing with a novel game type – i.e. one type of game that the creator of the agent had not foreseen.

To design an agent that can play every game, agent designers typically benchmark their agents on a set of publicly available games.

### 2.1 General Video Game AI

GVG-AI is a concretization of GVGP, it provides all the required infrastructure to develop and evaluate a GVGP agents. GVG-AI is simultaneously the implementation of the interface that allows agents to interact with the games, which coupled with the benchmarking system and a set of compatible games forms the GVG-AI framework, and a competition, the GVG-AI competition, with a specific set of rules and a ranking system.

Every edition of the competition uses new evaluation games, created specifically for that edition. These games are only revealed to the participants after the competition is over, i.e. the participants develop and submit their agents without knowing the games that will be used to evaluate their agents. However, other games are available, which are typically used as a benchmark and as a training set.

Currently, the GVG-AI framework contains more than 80 publicly available games. This set of games include the evaluation games used in previous editions of the competition and other games that are available since the creation of GVG-AI (which were used to provide a set of training games for the first edition of the competition). This set of games is diverse in mechanics, rules, and winning conditions subject to the limitations of the language they are defined in – Video Game Description Language (VGDL) – which is described in detail on section 2.3. This includes versions of classic games such as Pac-Man, Boulder Dash, and Space Invaders.

The GVG-AI framework, written in Java, provides an object-oriented interface for creating agents that can play in any game defined in VGDL. An agent can query the game status (winner, time step, score), the state of the avatar (position, orientation, resources), history of events or collisions during the game, and position of the different sprites in the level.

The GVG-AI framework can be executed in multiple ways, including as a human-playable game and with the game played by an agent in headless mode (without visuals).

The rules of the game, and the victory requirements are never provided to the agent. It is the responsibility of the agent to discover the game mechanics while playing. The main resources the agents have to reason about the environment is the forward model and game score provided by the framework.

The forward model allows agents to explore different sequences of actions in a simulated environment. The simulations must begin on an actual game state and have no width or depth limit. Given the nondeterministic nature of the games, there is no guaruntery that applying an action on the simulated environment and on the actual game leads to the same successor state.

The GVG-AI competition features single–player and, more recently, two–player tracks. The single–player track has the largest amount of games. The two–player track was added in 2016 and is composed of cooperative and competitive games.

Games in GVG-AI are nondeterministic – applying the same action twice in two copies of the same state does not necessarily result in the same successor state –, and fully observable – the locations of all objects in the game are known to the agent.

Before a game starts, the framework provides agents an initialization time with the duration of 1 second. Agents are given a StateObservation object and are free to use this time for anything, i.e. the framework does not demand anything from the agents in this stage.

Agents have a limited time to select an action per game step. The GVG-AI framework allows for the configuration of this value. The default value in the framework, which is the one used in the competitions, is 40 milliseconds. During this time the game is stopped.

Agents that take longer to decide than they are allowed to are penalized. The framework controls the execution of the agents, however it is technically possible that agents exceed the limit imposed by the framework. As such, and to ensure fairness between agents, the framework employs a time controlling system on the execution of the agents. Penalties go from ignoring the action the agent selected – on the game step the time was exceed – up to disqualification.

Each tick has a duration of 40 milliseconds. A match has a maximum of 2000 ticks, i.e. if the agent has not won or lost the game at the 2001th tick, the game is considered as lost by the framework.

### 2.2 Ranking system

The ranking system can be conceptually divided in two stages. The first ranks the agent with respect to its performance in a single game. The second ranks the agent with respect to the whole set of games of the competition.

On the first stage of the ranking system the agents are ordered according to the following three criteria, listed by decreasing order of significance.

- **Victories** The total number of victories achieved by the agent.
- **Score** The average of the score the agent got in all the matches of the game.
- **Time** The total duration of the matches of the game, measured in game ticks. The smaller the duration the better the rank.

On the second ranking stage, points are attributed to the agents according to their ranks in the games. Each agent gets 25, 18, 15, 12, 10, 8, 6, 4, 2, or 1 points for each game they have placed in first, second, third, etc., respectively. After, the points are summed up per agent and the agents are ordered by points, with more points meaning a better rank. This stage of the ranking system is based on the one used by the Formula 1.
2.3 Video Game Description Language

VGDL [12] is a simple and high level language for 2-dimensional (2D) video games. The language was designed to facilitate the implementation and generation of games that would be used in an AI research environment.

The language was designed for 2D arcade-style games, nonetheless it still allows for the implementation of many different game mechanics. Through the combination of primitives it is possible to define games with different types of physics (continuous or grid based, defining the friction, gravity, etc.), movement dynamics of objects (straight or random motion, player-controlled, etc.) and interaction effects upon object collisions (bouncing, destruction, spawning, transformation, etc.) [12].

A game is defined by two separate components, the level description and the game description. Each of the components is stored in its own text file. A game can have multiple levels.

VGDL defines the entities and the interactions that can take place in the game. Interactions between entities can occur when they collide with each other and the consequences are defined in the game rules.

**Game description** defines which objects exist in a game, the effects of the interactions between objects and between the objects and the player, and the winning and losing conditions.

**Level description** defines the initial position of the game-relevant objects, i.e. the initial state.

Originally, the language was designed alongside an interpreter, PyVGDL. This interpreter is written in Python, in which the syntax of VGDL is based on. However this interpreter is not used in GVG-AI, instead GVG-AI developed its own interpreter written in Java, which is integrated in the GVG-AI framework.

3 RELATED WORK

3.1 Iterated Width

Even though Iterated Width (IW) originates from classic planning, it has also been used for playing real-time games. Two examples of such games are the Arcade Learning Environment (ALE) and the GVG-AI competition. For such, some adaptations are required. Whereas in classic planning IW is used to find a goal state, in real-time games it is used to find a promising game state (either a winning state, or a state with high score) in a short amount of time [15].

Given the real-time constraints, both [7] and [5] concluded that it was infeasible to run the complete IW algorithm. As such, both studies considered only the fastest iterations, such as IW(1) and IW(2), and variations of them. IW(1) was shown to have a state-of-the-art performance in the Atari 2600 games of ALE, outperforming Upper Confidence Bound 1 applied to trees (UCT) in the majority of games [7], and to outperform MCTS on average over three sets of different GVG-AI games [5]. Adaptations considered using novelty measure for ordering states in a priority queue [7] and tuning the pruning strategy to include domain knowledge, an adaptation named IW(3) [5]. IW(3) was developed specifically for GVG-AI and considers only pairs of atoms where at least one of the atoms describes information about the avatar. Both adaptations performed worse than IW(1) on average.

GVG-AI presents several challenges to a straightforward implementation of IW, such as determining the state representation (what atoms to consider) and the nondeterministic state transactions. [5] consider only two types atoms to describe a game state: avatar(cell, orientation, atype) for avatars and at(cell, stype) for sprites. Many features of the game are purposely left out of this representation given that the number of boolean atoms in GVG-AI tends to be too large, even for IW(1) [5].

Although nothing prevents the use of a plain IW in GVG-AI, ignoring the stochasticity of the games might be too risky. For example, if the avatar has a monster to its right moving randomly, it would be wise to move away from the monster. However, IW may find the action move right safe, if the simulator returns a state where the monster also happened to move right. As such, [5] developed a technique, called safety pruning, in which before selecting an action, M samples are taken of the successor state of each action and it is counted the number of times the avatar dies as result of that action. Actions with the lowest count of deaths (and only those) are deemed safe and considered for the remainder of the algorithm.

3.2 Knowledge-Based Fast Evolutionary Monte Carlo Tree Search

The Knowledge-Based Fast Evolutionary Monte Carlo Tree Search (KBFEMCTS) algorithm was designed to improve the performance of a vanilla MCTS algorithm on GVG-AI. The two key modifications by the algorithm are the ability to reuse past information from events that provided score gain (exploitation), and to encourage interactions with unknown entities (exploration).

To guide the MCTS simulations, the default policy is calculated as weighted sum of feature values that are extracted from each game state. The features used in this implementation are the Euclidean distances to the closest types of non-playable character (NPC), resource, non-static object and portal. As there can be more than one type of these entities, the feature vector may have a variable length from a game step to another.

The need for a variable length feature vector comes from the fact that sprites exist also in a variable number, throughout a game. For instance, there might be more than one type of NPC, and they might spawn and disappear at any game step.

Each piece of knowledge corresponds to one event that represents the collision of the avatar – or sprite produced by the avatar – with another sprite. Specifically, the knowledge base contemplates only the types of sprites used to extract the features (NPC, resource, non-static object and portal).

The experimental work done by D. Perez et al, showed a significant improvement in performance, both in percentage of victories and scores achieved [9]. These experiments consisted on evaluating the algorithm on 10 games of the single-player track of GVG-AI. For comparison other tree algorithms were used, a vanilla MCTS, a Fast Evolutionary MCTS using the adaptation for the dynamic number of features, and a Knowledge-based MCTS using a random default policy in the simulation step of MCTS (i.e. no Fast Evolution used). KBFEMCTS approximately doubled the percentage of games won of all the other three algorithms used for comparison.
The authors concluded that adding both the knowledge base and evolution to bias the MCTS simulations provides a strong improvement to MCTS, but adding one of these features separately does not impact the vanilla MCTS algorithm significantly [9].

### 3.3 Rolling Horizon Evolutionary Algorithms

A version of Rolling Horizon Evolutionary Algorithms (RHEA) that handles macro-actions was applied to the Physical Traveling Salesman Problem (PTSP) in [10], which showed that evolutionary algorithms can be a viable and competitive alternative to MCTS. Then, a vanilla version of RHEA was applied on the GVG-AI framework which showed that a vanilla version of the algorithm is not able to explore the search space quickly enough, given the limited budget [4].

RHEA are suited for real-time problems, however the GVG-AI presents two issues to a vanilla version of the algorithm. One of the issues is that the limited time budget for choosing an action may be too short for evolution to produce its results. A recent study, where RHEA was applied on the GVG-AI framework, showed RHEA is unable to find better solutions than a random search [4]. The same study also showed improvements on the solution quality of RHEA when the time budget was increased. However, RHEA was the only algorithm tested on this extended time budget so it’s not possible to understand how the performance of RHEA scales with the time budget in relation with other algorithms. The other potential problem is the stochasticity of the games available on the GVG-AI framework. RHEA, by default, simulate an action only once so it is possible that the fitness of an individual is affected by an unlikely state transition. For example, consider a case where the avatar is supposed to run away from monster and has an adjacent monster to its right. Also consider that monsters chase the avatar 95% percent of time and on the other times the move elsewhere. While determining the fitness of an individual, a simulation is requested for the “move right” action. It is possible that on that particular simulation the monster moves up, by chance, and in that case the avatar survives. The fitness attributed to that individual will not be representative of its average performance as only one simulation was made. To solve this problem one possible solution is to simulate the same action (or individual) multiple times and average the results.

### 3.4 MaastCTS2

MaastCTS2 is an agent developed by D. Soemers et al for the GVG-AI competition. It placed first on the single-player track of the 2016 edition of the competition. The algorithm used is an enhanced version of MCTS. Some of the enhancements were known from existing research while other were introduced by D. Soemers et al [15].

The algorithm combines into MCTS several IW features, which had been the algorithm used by the previous winner of the GVG-AI competition. The following enhancements for MCTS inspired by IW were used [14]:

- **Breadth-First Tree Initialization**: before starting MCTS, the direct successors of the root node are generated, as a Breadth-First Search of depth 1.
- **Safety Preprunning**: before starting MCTS, each action available on the current game state is simulated $M$ times and it is counted how many of those simulations led to an immediate loss. Actions that had a number of losses strictly superior to any other are pruned.
- **Loss Avoidance**: whenever the simulation step of MCTS ends in a losing game state, that result is not backpropagated as would commonly be done in MCTS. Instead, one state is generated for every sibling of the last node, and only the evaluation of the node with highest evaluation is backpropagated.
- **Novelty-Based Pruning**: a pruning technique, with the goal of pruning redundant lines of play, based on novelty texts. As in the novelty tests of IW, this also requires states to be defined in terms of a set of boolean atoms. The atoms used by MaastCTS2 were the same atoms used by [5] for IW. The novelty test threshold used is 1, as in IW(1). In the selection step of MCTS, when one of the successors of $n$ should be selected, non-novel successors are ignored, unless $n$ has a small score. In such cases, all successors are considered.

The other enhancements used by MaastCTS2 are:

- **Progressive History** and **N-Gram Selection Technique**: these techniques introduce a bias in the selection and simulation step, respectively, towards playing actions that performed well in earlier simulations.
- **Tree Reuse**: this technique consists in initializing MCTS with a part of the tree built in the previous game step. When transitioning from a game step $t - 1$ to $t$, the subtree corresponding to the action played at $t - 1$ becomes the initial tree of MCTS at game step $t$. Given that games in GVG-AI can be nondeterministic, all scores and visit counts in the tree are decayed.
- **Knowledge-Based Evaluations**: to distinguish between states with identical score, MaastCTS2 uses an evaluating function that considers the knowledge collected during simulations, and distances to objects that could potentially be interesting.
- **Deterministic Game Detection**: MaastCTS2 acts differently whether it detects the game is deterministic or not. For instance, in deterministic games it does not apply the decay on the Tree Reuse technique. To detect if a game is deterministic, multiple sequences of actions are applied several times, testing for divergent states between repetitions. If a difference is found the game is considered nondeterministic. This technique is applied only once, at initialization time.

Besides the aforementioned enhancements, MaastCTS2 uses a particular configuration of the core algorithm, MCTS, to better suit the problem of GVG-AI. MaastCTS2 uses an open-loop implementation of MCTS. It uses the UCB1 [1] selection policy. Simulations are limited to a depth of 10 actions. In the expansion step of MCTS, the tree is expanded by adding the whole simulation play-out. Finally, for state evaluation purposes it uses the function given by eq. (1), where $score(s)$ is the game score value of a state $s$ in GVG-AI. Note that the sample MCTS agents included in the GVG-AI framework also use this evaluation function.
A RRT based approach for GVGP

\[
X(s) = \begin{cases} 
10^7 + \text{score}(s), & \text{if } s \text{ is a winning state} \\
-10^7 + \text{score}(s), & \text{if } s \text{ is a losing state} \\
\text{score}(s), & \text{if } s \text{ is a non-terminal state}
\end{cases}
\]  

(1)

The enhancements used by MaastCTS2 were evaluated individually and most of them showed to significantly increase the performance over a vanilla MCTS [15]. Several different versions of the algorithm were created with all but one enhancement implemented and tested against a baseline vanilla MCTS.

The complete version of the algorithm (with all enhancements) was evaluated and compared with the following agents: Open-Loop Monte Carlo Tree Search (OLMCTS), a basic MCTS, IW(1), and YBCriber (which won the GVG-AI competition at the IEEE CEEC 2015 conference). Results showed an average win increase, of 17.4\%, over the basic MCTS and an average win decrease, of \(-4\%\), when compared to YBCriber. Note that YBCriber was not submitted to the last couple of competitions, including the one that MaastCTS2 won.

### 3.5 Rapidly-exploring Random Trees approach for Geometry Friends

Geometry Friends (GF) is a physics-based puzzle-platform game. It is also a game AI competition where participants create agents to play the game. The competition features single and multiplayer tracks where an agent must control one and two characters, respectively. A small subset of the games used in each competition is publicly available before the competition while the rest is not revealed. The goal of the game is to collect a set of diamonds scattered though the level and the score depends on the time spent to do so.

An RRT based approach was used for the individual levels of Geometry Friends. This solution uses a RRT based approach to solve only the planning and applies another algorithm for controlling the motion of the character. The agent was submitted to the 2015 Geometry Friends Game AI Competition and it was able to plan both public and private levels and able to control the motion on most of them.

The goal of the planning phase of the agent is to produce a sequence of high level instructions for the motion controller. These high level instructions are composed of coordinates and an high level action, such as “turning”, where the character must turn direction, “fall”, where the character should fall, and “diamond above”, where the agent should reach for the diamond right above.

The planning algorithm is an adaptation of RRT, where states are added to a graph, iteratively and by random exploration. It starts with an initial state, \(x_{init}\), with all actions applicable on that state unexplored. One of those unexplored actions is selected, applied, leading to a new state, \(x_{new}\), and marked as explored. \(x_{new}\) is validated for any constraint violation, such as having the character go through an obstacle. If any constraint is violated \(x_{new}\) is ignored, terminating this iteration of the algorithm. Otherwise, \(x_{new}\) is added to the graph. This process is repeated iteratively a number, \(N\), of times, where the state selected is one from the graph that still has actions unexplored. The algorithm ends when either the number of iterations reaches \(N\) or the goal is found.

While the algorithm is running it is always known what is the best state so that the algorithm is able to provide a solution when the maximum number of iterations is reached. After validating a state, that state is compared with the previously best state and a variable updated if a new best state was found. The value of a state is defined by the number of diamonds that have yet to be collected. As such, a goal state is state with value 0.

### 3.6 Discussion

Evolution, as the core algorithm of an agent, appears to not work on GVG-AI given the limited computational budget available to agents. [4] showed that the 40 milliseconds to decide on an action, per game step, are not enough for an evolutionary algorithm to produce better decisions than a random controller.

[4] also showed that by increasing the amount of time available to the algorithm the solution quality would increase, suggesting that the used technique requires a larger computational budget to be competitive with state-of-the-art solution. It should be noted, however, that other approaches, such as tree search algorithms, also produce higher quality decisions given more time per game step. So, it is not clear if a RHEA approach ever outscalers a tree search approach, and if it does from what computational budget per game step.

Nevertheless, it is possible to use evolutionary algorithms in GVG agents with success. Both KBFEMCTS and MaastCTS2 use the technique of Fast Evolutionary MCTS which uses an evolutionary algorithm as a source of control parameters to adapt the behaviour of each iteration of MCTS. Both studies reported improvements when using this technique.

Despite being designed for GVG, some agents do use limited domain knowledge. The GVG-AI framework provides agents with some information about the game world, such as the position and type of the sprites in the game – if it is a NPC, a static sprite, a resource, etc. For instance, KBFEMCTS exploits this information to guide the avatar towards other sprites, to interact or collide with them.

Tree search based algorithms are the most successful ones, in particular MCTS and IW. These two algorithms use very different search strategies. While IW is based on Breadth First Search (BFS), MCTS uses sampling with Depth First Search (DFS). [15] noticed the problem of how a basic implementation of MCTS sometimes does not explore all the actions of the root node, given the limited computational budget. This leads to a very unbalanced tree, where some local (in terms of time) actions are ignored in favour of some other remote actions. To compensate this issue, they applied the technique of Breadth-First Tree Initialization (BFTI). This technique solves the problem of finishing a game step with immediate actions of the root node unexplored. However, the overall problem of having an unbalanced tree is only reduced as the issue is only pushed to the next depth level of the tree. The search strategy of IW does not suffer from this problem but suffers from others, such as being restricted to a small lookahead.

RRT has a balanced search strategy. The search strategy of RRT leads to a uniform state space exploration, which might be a good compromise for GVGP, between the search strategies of MCTS and
4.1 Design of the agents

All of the proposed agents share the same design, which can be split in three stages: initialization, exploration, and action selection. The initialization stage occurs once per match, while the other two stages occur once per tick. The limited time frame of each tick will be used mostly for the exploration stage of the algorithm, as it is expected to be the stage that most influences the performance of the agent.

The primary data structure of the algorithm is the search tree where the exploration results are collected. In this tree vertices represent states and edges represent actions.

In the initialization stage the search tree is created with one vertex, the initial game state, $x_{init}$ – this game state is the only real game state available to the agent, all others are results of simulations.

In the exploration stage, the graph is expanded iteratively. Each iteration starts by randomly sampling the model of state space $q$ and selecting the unexplored state $x$ – a state with at least one unexplored action – from the tree that is closest to $q$. From the set of unexplored actions of $x$ a random one, $a$, is simulated on $x$, leading to a successor state, $y$. Both $y$ and $a$ are added to the graph (action $a$ is represented on the graph as an edge from $x$ to $y$). The new state $y$ is evaluated with an heuristic function and the value is stored as a property of the vertex, concluding the iteration. This process is repeated every game step and stops when the time left to decide on an action is inferior to the combined time of another iteration and the action selection stage.

The action selection stage takes the constructed tree and outputs the best action. From the information stored on the tree, the state with the highest value is selected and the path of states and actions is traced back until $x_{init}$. The last action of this path is the output action.

4.2 Agent framework

Each of the variants was implemented in an agent framework that we developed. The agent framework facilitates the experimentation and implementation of the variants as it contains the implementation of features that all the agents share, such as the infrastructure required to integrate with the GVG-AI framework and the task scheduling algorithm.

4.2.1 Scheduling algorithm. Given that GVG agents are subject to real-time constraints and that extra processing time improves their performance, they should be designed to use the available time efficiently. For that reason we have implemented a task scheduling algorithm, which controls the execution time of each stage of the agent.

The task scheduling algorithm was designed to maximize the time allocated to the exploration stage of the agent, as long as there is enough time left for the action selection algorithm. This is implemented by checking how much time is left in the tick after every iteration of the exploration stage. When there is not enough time to run another iteration and the action selection algorithm (plus a safety margin), the exploration stage is ended and the final stage – action selection – is executed.

The variants we propose have an identical task scheduling, which is an iterative approach with the period of a tick: the agent explores during most of the tick and at the end of it runs the action selection algorithm. Alternative task scheduling solutions are possible. An example would be an agent that keeps exploring – executing a random or constant action in each tick – until it finds a winning action sequence.

We did not find any research comparing the iterative design with the one mentioned in the example neither did we evaluate the designs ourselves. We chose the iterative design as it is the most common among GVG agents.

### Algorithm 1: The task scheduling algorithm

```
algorithm execute()

  % current time and average duration
  initTime ← now()
  numIterations ← 0
  timeBuffer ← 2ms
  averageDuration ← 0
  remainingTime ← 1

  repeat
    agent.explore()
    numIterations ← numIterations + 1
    averageDuration ← averageDuration + remainingTime
    action ← agent.actionSelection()

    remainingTime ← remainingTime - timeBuffer
    timeBuffer ← timeBuffer + 2ms
    averageDuration ← averageDuration - (initTime + timeBuffer)

  until remainingTime < averageDuration

return action
```

4.2.2 Sampling the state space. A fundamental component of the RRT algorithm is to draw random samples from the state space, this works well when the algorithm knows the definition of the state space or is able to determine if a random sample belongs in the state space. In GVG, an agent is never sure about the state space
and has no way of checking if a state belongs in the state space. As such, a workaround is required to maintain this fundamental component of RRT.

Our solution is to create an approximation of the state space and to draw random samples from it. The approximation is created, at the beginning of the match, by extrapolating the initial state.

The approximation of the state space contains the following information: dimensions of the game, and the type and the amount of the entities currently in the game. This model serves as the upper bounds of the approximated state space, i.e. the agent assumes that the game will not grow bigger – the size of game is constant throughout a match – and that there won’t spawn any new entities or more of the same entities throughout the match – entities can and do spawn in games, as such, this is not an ideal approximation.

To ‘randomly sample’ the state space, we generate a partial state with an avatar in a random position of the game and a random amount of each type of entity, both using an uniform distribution.

4.2.3 Distance function. In RRT, after randomly sampling a state it is necessary to identify which of the states already present in the tree is the closest to the randomly sampled one. To implement this functionality, everyday we randomly sample a state we compute its distance with every state of the tree.

To compute the distance between two states, s1 and s2, we designed eq. (2) which considers the avatar position and the cardinality of each type of sprite.

\[
d f(s_1, s_2) = \text{avatarDistance}(s_1, s_2) + \text{spriteDifference}(s_1, s_2)
\] (2)

In eq. (2), the avatarDistance function takes two states as input and computes the distance between the position of the avatar of each state. As the environments in GVG-AI are 2D and considering \( s_1 = (x_1, y_1) \) and \( s_2 = (x_2, y_2) \) we can calculate the squared of Euclidean distance as shown in eq. (3).

\[
\text{avatarDistance}(s_1, s_2) = (x_1 - x_2)^2 + (y_1 - y_2)^2
\] (3)

In eq. (2), the spriteDifference function takes two states as input and computes the sum of the difference of the cardinality of each type of sprite, present in either state. For example, consider a state \( s_1 \) with 4 sprites of type A and 0 sprites of type B and a state \( s_2 \) with 3 sprites of type A and 2 sprites of type B; in this example the \( \text{spriteDifference} \) is 3, because state \( s_1 \) has one more sprite of type A and state \( s_2 \) has two more sprites of type B (\( 1 + 2 = 3 \)). If you consider \( T \) to be the union of the types of sprites of \( s_1 \) and \( s_2 \) and \( t_i \) to be the number of sprites of type \( t \) that state \( i \) has, then \( \text{spriteDifference} \) can be computed as shown in eq. (4).

\[
\text{spriteDifference}(s_1, s_2) = \sum_{t \in T} |t_1 - t_2|
\] (4)

4.3 Variants of Rapidly-exploring Random Trees

This section contains the description of the specifics of each one of the four variants of RRT that we propose. For a description of the agent framework on which these variants are integrated see section 4.2.

4.3.1 Vanilla Rapidly-exploring Random Trees. The Vanilla Rapidly-exploring Random Trees is the simplest variant we implemented.

- **Initialization** The exploration tree is created with the current game state only.
- **State selection** The state selected for expansion is the closest unexplored state to a state \( q \), which is randomly sampled from the approximation of the state space maintained by the agent framework.
- **Expansion** All of the actions of the selected state are explored and all of the successors are added to the exploration tree.
- **Action selection** The selected action is the first action in the action-path that leads the agent towards the state with highest score.

4.3.2 Rapidly-exploring Random Trees Essential. The motivation for this variant is that the vanilla variant of RRT (presented in section 4.3.1) behaves differently from the RRT algorithm. The vanilla variant, in the expansion stage, adds all the successor nodes of the selected node to the tree, regardless of the distance between them and the randomly sampled state, \( q \). In RRT at most one node is added per expansion step and for a node to be added, it needs to be closer to \( q \) than its parent. This variant introduces the concept of essential nodes to approximate the behavior of the RRT algorithm.

The concept of essential nodes affects most stages of the agent:

- **Initialization** The initial node (the root of the tree) is marked as essential.
- **Node selection** Essential nodes have a higher priority of being selected. First the algorithm searches for the unexplored essential node that is closer to \( q \). In case there is no unexplored essential node, all unexplored nodes are considered.
- **Expansion** All of the actions of the selected node are simulated and the resulting successor states are added to the tree (as the vanilla variant). From the set of successor states, the one that is closer to the randomly sampled state is marked as essential.
- **Action selection** All states are considered equal during action selection, ignoring the essential attribute.

As an alternative, it would also be possible to only add essential nodes to the tree, discarding any node that was not considered essential. This has the advantage of a smaller memory footprint but has the disadvantage of possibly requiring a state to be expanded twice. For example, consider such an agent and the beginning of a very simple game, where the initial state, \( a \), has only two possible actions, \( \beta \) and \( \gamma \), which the agent expands, leading to state \( b \) and \( c \), respectively. The agent marks \( c \) as essential and discards \( b \). However, later the agent discovers that \( c \) is a terminal state and now has no essential states. Now, the agent needs to expand \( a \) again (with \( \beta \)) since it is the only state that the agent knows and that is partially unexpanded.

4.3.3 Rapidly-exploring Random Trees Essential Strict. The RRT Essential Strict variant is based on the previous variant, RRT Essential, it has the same motivation and a slightly different implementation. The difference is in the expansion step: while the non-strict version always marks a successor state as essential, the strict version does not. In this version, there is an additional condition that a
successor must meet for it to be marked as essential, it needs to be closer to q (the randomly sampled state) than its parent.

This solution tries to address the problem that in GVGP it is not knowable how to get closer to a particular state. Consider a scenario where the agent wants the avatar to go right. One could assume that performing the move right action would get the avatar closer to the goal, however now assume that there is a portal to the right of the avatar that when stepped on takes the avatar to the left. In this scenario performing the move right action would make the avatar farther away from the goal. If you also assume that performing any other action would not get the avatar any closer to the goal, you will be thinking of a scenario where the Strict version of RRT Essential would not mark a single successor as essential, but the non-strict version would.

4.4 Enhancements

In this section we describe the techniques we implemented in our agent to improve its performance.

4.4.1 Tree Reuse. Given that we are considering deterministic games only, we implement a simple form of TR. After applying an action, the agent preserves the subtree which corresponds to the chosen action.

Consider the transition from tick $t - 1$ to tick $t$, where the agent has applied the action $a$ and let $r_t$ denote the root of the tree at tick $t$. The new root of the tree becomes the node that is obtained when $a$ is applied to $r_{t-1}$, while its sibling nodes and parent are discarded.

4.4.2 Species First. SF is a novel technique for GVGP agents that we propose as an enhancement for our agents. It arose from the will to fix an observed behavior of our agents, which is that we noticed our agents visiting identical states too often. As such, we wanted to implement a technique that would deter the agents from that.

The obvious solution would be to compare the newly found successor states with all the other states already present in the search tree. In case there were any two equal states we would disregard one of them from being considered for future state selection stages. However, this is not possible in GV-GAI. In GV-GAI the agents do not have access to the state itself, they only have access to a StateObservation object, which for most cases is the same as having access to the state but not on this problem. There could be any number of StateObservation objects that look identical from the perspective of the agent but that actually represent different game states. In practice this is the problem of partial observability. As such, we had to design a more complex solution.

The solution we ended up designing revolves around a new concept called species. Species group similar observations, i.e. a species is a set of similar observations. An observation can only belong to a species. To be considered "similar" observations must have an equal part of information about the state they represent. For example consider a scenario with a state which is defined by the tuple of values $<A, B, C, D>$, with agents that only have access to the first three values, through an observation, and with only the two first values being used for the definition of a species and consider the following three states: (1) $<1, 1, 1, 1>$, (2) $<1, 1, 1, 2>$, (3) $<1, 1, 2, 3>$, and (4) $<1, 2, 3, 4>$. In this scenario, states (1), (2), and (3) would be members of a single species as they all share the first two values, $<1, 1, ?, ?>$, and state (4) would be the only member of another species.

The concept of species influences the state selection and expansion stages of the agents.

State selection While there are unexplored species – species in which all members are unexplored –, the agent only considers the states of those species for state selection. When there are no unexplored species, the agent returns to normal, considering all unexplored states equally.

Expansion After simulating an action on a state $x$, the resulting successor is assigned to a species. If the species of the successor is new, it inherits the exploration status of the state, i.e. if the successor is already explored (happens if the state is terminal) then the species is marked as explored and marked unexplored otherwise. Also, if this expansion step completely explores $x$, the species of $x$ is marked as explored.

5 EVALUATION

This chapter describes the experiments used to evaluate the performance of the RRT search technique in GVGP and the results obtained. The setup of the experiments is described first and the results are described afterwards.

5.1 Setup

The primary measurement used to compare the performance of the agents is the percentage of victories. There are other potentially interesting criteria such as score, ticks required to win, etc., however we consider victories to be the most important criteria in a GVGP competition.

The games for all experiments were played using the revision a809f68 of the GVGAI framework, with the default configuration and on a personal computer. We chose this particular version of the framework because it was the most recent available version of the framework when we began the experiments. The personal computer specifications were the following: Intel i5-6500 (3.6GHz) with 16GB of RAM running at 2133MHz. The operating system was Arch Linux x64 with kernel version Linux 4.18.1-arch-1-1-ARCH.

5.1.1 Methodology. To produce and collect the experimental data, we matched every agent listed in section 5.1.3 with every level of every game listed in section 5.1.2, at least 20 times (each game has 5 levels which means that the agents played each game 100 times). To facilitate this process we used GNU Parallel [16]. This tool was useful because it can generate all the combinations among its input sources – i.e. a list of items – and execute a program for each combination. In our case, fed GNU Parallel with three lists, the agents, the games, and the iteration of the benchmark, and a program that matched an agent with every level of a game.

At the end of each match the following statistics are collected: if the agent won, the duration of the match (measured in ticks), the final score, the number of warnings of time overspent, and if

---

1. If you consider the tuple of values to be the DNA (biology concept) of a state, you may find this definition of "species" to be similar to the biological definition of the same term, which is the reason why we chose this name.
2. Found online at https://github.com/EssexUniversityMCTS/gvgai/commit/a809f68
the agent was disqualified. From this set of data we compute the average win percentage of a pair agent-game by computing the arithmetic mean of the set of matches of that pair, with victories being represented with the value 1 and losses with 0.

5.1.2 Games. All experiments, described in this section, were run on the deterministic games of the GVG-AI framework. To determine the set of deterministic games, we analyzed the source code of all games and excluded the ones that contain stochastic primitives.

5.1.3 Agents. Given that each research has its own setup and that we want to determine how viable are RRT-based approaches in GVGP, we also compare our agents with other agents, namely: Random, Random Search, and MCTS.

This set of agents contains simple and state of the art agents. The former ones serve as a baseline for comparison and the latter ones are considered to determine competitiveness of our approaches.

5.2 Experiments and Results

In this section we describe the experiments performed and their results. Firstly we will compare the performance of the proposed RRT implementations. Secondly, we will evaluate the proposed enhancements. Finally, we will apply those same enhancements on a Random Search (RS) algorithm to determine if the results of the RRT with enhancements are merit of the algorithm or of the enhancements themselves.

5.2.1 Variants of Rapidly-exploring Random Trees. In section 4.3 we described four variants of the RRT algorithm for GVGP. With the goal of understanding which variant of RRT performs best in GVGP we matched each variant without enhancements with every game. We also consider the RS and MCTS agents, also without enhancements, for comparison. We chose these two agents as they define our expectations for the results of our agents, with RS representing the minimum acceptable performance and MCTS representing great performance.

Overall, the performance of the variants of RRT we propose fits in the range we expected it to, between the Random and MCTS agents. The Random agent is the least performant of the agents with an average win ratio of 4.8 ± 0.6% while MCTS is the most performant of the agents with a win ratio of 17.0 ± 1.1%. Sorting our proposed agents from least performant to most performant we get an order consistent with order of the description in section 4.3, which were described in an order of increasing complexity. The average win ratio of our agents is 7.5 ± 1.0% for the RRT Vanilla variant, 11.0 ± 1.3% for the RRT Essential variant and 12.9 ± 1.4% for the RRT Essential Strict.

In section 4.3 we presented two versions of the RRT Essential variant, the strict and non-strict version. Results shows that the strict version has on average a 2% higher win ratio versus the non-strict version with non-overlapping 95% confidence intervals.

5.2.2 Enhancements. In this section we describe the experiments we used to evaluate the proposed enhancements and show their results.

To evaluate the two proposed enhancements, Species First and Tree Reuse, we ran an experiment for each. In each of these experiments we compared two agents, one with the enhancement being tested and another without the enhancement. The base agent we chose for this is the RRT Essential Strict – our proposed variant with the highest win ratio.

Species First The total average win ratio of the non-enhanced agent is 12.9 ± 1.4% while the enhanced version has 14.2 ± 1.1%, i.e. the presence of the Specie First enhancement has potentially caused an increase on the performance of the agent.

Tree Reuse The total average win ratio of the non-enhanced agent is 12.9 ± 1.4% while the enhanced version has 12.7 ± 1.2%, i.e. the presence of the Tree Reuse enhancement has potentially caused a decrease on the performance of the agent. This results went against our expectations and as such we decided to repeat the experiment on other variants to determine if this performance regression with the Tree Reuse technique is specific only to this agent.

We repeated the Tree Reuse evaluation experiment on two other agents, the Vanilla and Essential Strict with the Specie First enhancement agents. Once again, the total average win ratio of the non-enhanced agents is higher than of those without Tree Reuse, with the Vanilla agent averaging at 7.5 ± 1.0% without the enhancement and 8.4 ± 1.9% with it, and with the Essential Strict with the Specie First enhancement averaging at 14.2 ± 1.1% without the enhancement and 14.0 ± 0.7% with it. These results support the evidence obtained with the experiment on the Essential Strict agent and suggest that the Tree Reuse technique is not beneficial to these agents in this context.

Results consistently show that the agents with the Tree Reuse enhancement have a lower average win ratio than those without said enhancement. This result goes against our expectations as theoretically this enhancement provides more information to the agent, which should be beneficial. Our theory for this result is that the extra information causes the agent to take more time to run. This may happen by either (1) longer or more frequent garbage collection pauses or (2) algorithms of the agent such as state selection or action selection taking longer to run.

5.2.3 Random Search with enhancements. In the previous sections we evaluated the variants of RRT we propose, without and with enhancements. In this section, we compare our best agent with a RS agent with the same enhancements. This experiment should help us determine if the results we have obtained are result of our agent framework and enhancements or merit of the algorithm itself. The RRT agent we use in this experiment is the RRT Essential Strict with the Specie First enhancement and is compared with an RS agent also with the Specie First enhancement.

The RS agent we are using in this experiment is much simpler than the RRT agent. The RS agent has identical Initialization, Expansion and Action selection stages with the Vanilla RRT. In the State selection it selects a random unexplored state.

The total average win ratio of our best agent, RRT Essential Strict with the Specie First enhancement, is 14.2 ± 1.1% while the the average of the RS agent is 13.2 ± 2.1%. As these results have overlapping 95% confidence intervals it is not possible to determine which of
these agents is more performant without a statistical hypothesis test.

6 CONCLUSIONS

In this dissertation we surveyed the existing work on GVGP agents and identified a lacunae on the used algorithms. Most of these agents use problem agnostic search algorithms such as MCTS and IW, with enhancements specifically developed for the GVGP domain. RRT is a problem agnostic search algorithm, that has been used successfully in various domains, but it has never been used in the context of GVGP.

We decided to develop an RRT-based agent for the deterministic single-player games of GVGP, a GVGP competition. Our analysis of this domain and the RRT algorithm made us identify some issues for a GVGP RRT-based agent. We summarize these issues in the following list:

- RRT requires the ability to draw random samples from the state space. This is typically not possible in video games given that usually players do not know the definition of the state space, which also happens in the GVGP-AI competition.
- RRT requires the ability to determine relative distances between states (usually in the form of a state-distance function). This is also typically not possible in games, and also not possible in the GVGP-AI competition.
- In GVGP, it is not possible to check if a given state (for instance one obtained from randomly sampling the "state space") is reachable, i.e. not an "obstacle", or even valid.

These issues prevent the direct implementation of RRT in the GVGP domain. However, in this work we have adapted the RRT algorithm to overcome these issues. The three variants of RRT we propose and the Specie First enhancement were designed by us and are our proposed solution to this problem.

We analyzed how the agents interact with the GVGP-AI framework and conceptually divided the agents into several stages. We adapted the RRT algorithm for the exploration stage of a GVGP agent and proposed three variants of it. We developed an agent framework to complement these algorithms with the remaining stages of an agent. The agent framework consisted of the commonalities of the agents which not only facilitated the development of the algorithms but also ensured a consistent platform for the evaluation of the algorithms.

We proposed two techniques as possible enhancements for our algorithms, the Specie First and Tree Reuse. The former is a novel technique designed by us for GVGP agents and the latter is a common technique used in GVGP agents.

We evaluated our proposed solution on the deterministic single-player games of the GVGP-AI competition. For the non-enhanced variants of RRT we propose, results show that they have an average win ratio that is higher than a random agent and lower than a MCTS agent. Which suggests that RRT is a viable exploration algorithm for GVGP. Regarding the two techniques we proposed as enhancements, results showed that the Specie First had a positive impact on the performance of the tested agent and that the Tree Reuse technique had a negative impact. Lastly, tried to determine if the performance of our agents was merit of the agent framework or of the algorithms themselves. To achieve that we developed a simple agent on our agent framework and applied the Specie First enhancement on it. We compared the performance of this agent with our best agent, RRT Essential Strict with the Specie First enhancement and results showed that the RRT-based agent had a 1.3% higher average win ratio with overlapping 95% confidence intervals.

This work can be extended in multiple directions. One obvious direction would be to integrate our work in an agent that is capable of playing stochastic games. An interesting way to achieve this is to develop an agent that combines two different algorithms, one for stochastic and another for deterministic games. As a way to improve our agents we would have liked to have tested a system that dynamically updates the model of the state space throughout a match. Lastly, we suggest that other researchers try to tackle this problem, of integrating RRT in GVGP as we believe there is space for innovations in this area.

REFERENCES