Abstract

In this thesis a framework for generating games- and levels in the video game description language VGDL is presented. The generator is able to automatically generate games, test the playability and quality using AI controllers, and score each game according to a series of evaluation criteria.

The research and framework is separated into two parts, divided by different game genres: First an analysis of 2-dimensional action-arcade games, with the goal of producing a generator focusing on evolving a set of game-rules, using the performance of general game-playing algorithms in generated games. And second, an examination of 2-dimensional turn-based puzzle games-, and especially the levels of puzzle games, with the ambition of being able to not only generate simple sets of game-rules, but also to generate non-trivial puzzles (levels) for the games.

The results of this thesis show that automatically generating game design is indeed very promising, with several interesting game designs generated – but also many bad ones. It seems a human designer is still needed to assess which generated game designs actually work.

Contents

Chapter 1

Introduction

Below I will introduce the idea and problem of generating both the content for games-, and the concept of generating complete games. The research goals of the thesis is then shown, and an overview of the project is presented.

1.1 Introduction to content generation

Procedural generation of game content (levels, textures, items, quests, audio, characters etc.) has become a widely used tool for video-game developers, and has been applied to an array of different game- genres and types. Automatically generating content allow game developers to provide more content, with more differentiation, from a limited supply of assets, most often by helping to produce levels, maps or dungeons for their games. PCG is often used to maintain a challenge for the player by presenting an unpredictable and renewed game experience on re-playing the games in which it is used.

Rogue [?] is one of the earliest examples of a game using procedurally generated levels (dungeons) as a main part of the game design. The space-trading game Elite [?] used PCG to both save disk space, and to generate a large array of planets and galaxies, each with a different set of unique properties generated for every play through.

Since then several highly successful games, from genres including platform-, first person shooter- (FPS) and strategy games, have used PCG to generate levels-, maps and worlds, as either used in their main game mode (e.g. Spelunky [?], Minecraft [?]), or as an extra feature, making the games have additional re-playability (e.g. Civilazation II(-V) [?], Heroes of Might and Magic III [?]). PCG has additionally been used to create a wider array of weapons and items, especially in role-playing games (RPGS) where "looting" (finding new gear) is a main part of the game design (e.g. Borderlands-series [?], Diablo-series [?], Dark Age of Camelot [?]).

Several game-developers, companies and research groups has additionally used PCG in the generation of other types of game-content, or at different stages, including complete worlds, environments and stories (e.g. Dwarf Fortress [?]), and graphics and audio (e.g. RoboBlitz [?], .kkrieger [?]).

1.2 Introduction to game generation

An interesting next step to take is to not only generate content for a video game, but to generate completely new games and game design – i.e. to generate the set of rules and mechanics of a game, and necessarily to generate the content or scenarios to actually play the game[1].

A possible approach to automatically generating complete games might be to search through a space of programs represented in a programming language like C or Java. However, the proportion of programs designed in such languages that can even be considered a game is rather small – and much less, the amount games that a human player would find enjoyable. To reduce this problem a game description language (GDL), designed to encode games – and only games – can be used, severely increasing the density density of well-defined games.

[1] Even simple games like Tetris or Pong is dependant on a playing field (or level) and a definition of each object

However even searching through a fairly well defined space of possible games, one still have a need for some way of actually telling good games from bad ones (or at least, mediocre games from really bad games) – that is, a fitness function is needed. A fitness function could partly consist of inspecting the generated rules as expressed in the GDL, e.g. to make sure that the player can interact with the game in some way, and that there are winning- and losing conditions, which could in principle be fulfilled. However there are many bad games that fulfil such criteria, so intuitively it seems one need to actually play the games that are generated.

A fitness function for generating complete games therefore needs to incorporate a capacity to automatically play the games it is evaluating, giving each game a score dependant on certain values from the results of playthroughs.

1.3 Research Objectives

The main objectives of this study was to construct a generation process able to create simple games and levels using the game description language VGDL, with the goal of making games as enjoyable for human players as possible. The focus of the study was to create fitness functions able to differentiate between games (and levels) of different quality, to be able to search through, and importantly, evolve, a set of generated games, and thereby create interesting content.

The main approach used to create the fitness function was to use the results of a series of general game-playing (knowledge free) algorithms, and using the collection of their performance profiles to construct individual fitness features. A series of human-designed games, assumed to be human-enjoyable, was used as a baseline for "good games" and was intensely used in choosing and weighing features of the algorithms' performance profiles. This approach was focused on a specific genre and type of games (*arcade-action games*), which is the main focus of the framework used (the GVG-AI framework). The general game-playing algorithms used are well-fitted to play the game genre, since they are excellent at quickly finding a decent action to take in the environment of the games – where quick movements to avoid enemies, and increasing the player-score is in the focus.

The second part of this thesis is focused on games of the *puzzle*-genre described in VGDL. This approach is mainly focused on generating new game levels for either existing human designed-, or for generated VGDL games, by using the results of a breadth-first search algorithm (which is able to find the shortest solution), with the goal of being able to judge the difficulty, and thereby the quality of a game-level pair. The VGDL puzzle games examined consist in general of a much smaller game-state space (i.e. the amount of possible arrangements of sprites and values), and so potentially are more well-suited for a more in-depth analysis.

Assumptions

As mentioned above, I assume that a subset of the human-designed VGDL games described (Section 2.3, 3.1) are of high quality (i.e. enjoyable for human players). This is not always clear when actually playing the games in the framework used, since they mostly need crucial element like, audio, proper graphic assets- and effects and an input mapping scheme fitting to the game.

I additionally assume that completely randomly generated games, using randomly generated levels in the VGDL language (without sorting/searching using a fitness function) produce games with an extremely (almost negligible) low chance of being human-enjoyable. This assumption is mostly made from initial tests into game generation using the VGDL language.

When analysing- and generating *puzzle games* (see below) I make the assumption that the quality of a game- and level of the genre is directly related to the difficulty of the game – that, at least, having a possibility of challenging levels or situation is deeply important for puzzle game.

1.3.1 Definitions

This project is focused a games of two restricted genres and types – these restrictions mostly stems from the framework used in the work, but allows a more focused analysis:

1) Single-player, 2-dimensional, top-down, action games, in which the player controls a single avatar which can (maximally) be moved in the four cardinal directions, possibly with an "action"-(shoot etc.) button. The games are all focused on increasing the players score, and often winning

requires killing/destroying all of enemies on the screen. These games will be simply be referred to as *action-arcade games*.

And: 2) Single-player, 2-dimensional, top-down, turn-based puzzle games, in which the player controls a single avatar which can (maximally) be moved in the four cardinal directions, possibly with an "action"-(shoot etc.) button., and all events occur as a result of player actions (i.e. no NPC's or falling boulders, and no random events). In the VGDL framework these type of games are simply a subset of the action-arcade genre, and only a subset of the VGDL classes- and functions defined in the GVG-AI framework (see Section 2.3) are "allowed". This type of games will be referred to as *puzzle games* in the project.

1.4 How to read this thesis

In the following chapters I will define the problem in more detail, explain the existing research and framework used, present the results of fulfilling the research goal and finally the ultimate products of the analysis of this thesis: Generated games.

The figure below shows the structure of the thesis, explaining the content and reasoning behind each chapter and how it fits in to the larger picture.

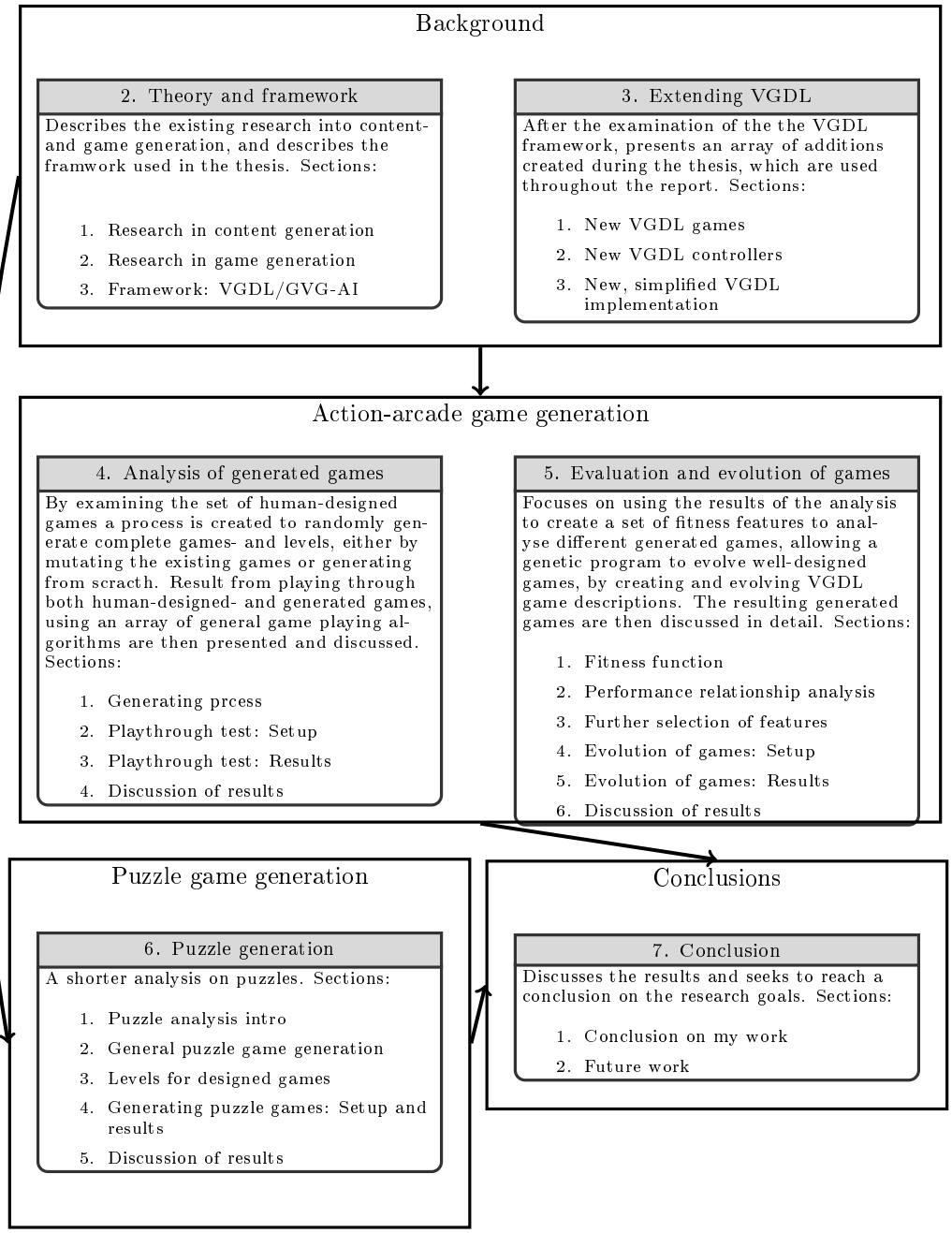

Figure 1.1: How to read this thesis

Chapter 2

Existing research and framework

In this chapter I describe the existing research on the topic of procedural content generation (PCG) for games, and specifically game- and level-generation, which is of relevance to the project. Additionally the framework used for game- and level-generation, and testing is described thoroughly.

2.1 Content generation

The task of PCG is to create elements for specific parts of a game, for instance levels, weapons, environments, textures and sounds.

Several different approaches are used in PCG (both in commercial games and in research), however the "search-based" approach (explored by ?, ?) dominates the problem. In search-based generation a *fitness function* is used to score generated content by their quality, making it possible to ensure a high quality of the generator, by either simply generating a large amount of content and choosing the best, or using an evolutionary strategy to evolve better content.

2.2 Game generation

Generating complete games through algorithms is a problem that is being increasingly researched, but progress on the topic for last decade. Because the problem is in general quite large, a subset of the problem is usually handled: Only generating certain types of games-, or using different (restricted) frameworks. Video games may consist of a large number of tangible and intangible components, including rules, graphical assets, genre conventions, cultural context, controllers, character design, story and dialog, screen-based information displays, and so on ???.

In this project I look specifically at generating the game-play and -setting of games, i.e. defining a set of game-rules and -objects, and specifying the levels in which the game-play takes place. The two main approaches that have been explored in generating game rules are reasoning through constraint solving ? or search through evolutionary computation or similar forms of stochastic optimisation ???. In either case, rule generation can be seen as a particular kind of procedural content generation ?.

It is clear that generating a set of rules that makes for an interesting and fun game is a hard task. The arguably most successful attempt so far, Browne's Ludi system, managed to produce a new board game of sufficient quality to be sold as a boxed product ?. However, it succeeded partly due to restricting its generation domain to only the rules of a rather tightly constrained space of board games. A key stumbling block for search-based approaches to game generation is the fitness/evaluation function. This function takes a complete game as input and outputs an estimate of its quality. Ludi uses a mixture of several measures based on automatic playthrough of games, including balance, drawishness and outcome uncertainy. These measures are well-chosen for two-player board games, but might not transfer that well to video games or single-player games, which have in a separate analysis been deemed to be good targets for game generation ?. Other researchers have attempted evaluation functions based on the learnability of the game by an algorithm ? or an earlier and more primitive version of the characteristic that is explored in this paper, performance profile of a set of algorithms ?.

A typical approach for generating complete games is searching in a space of possible games. This basically requires two things: That the ratio of enjoyable games in the set is not too low - otherwise those games might never be found. To increase this ratio a game description language (GDL) is often used (searching through all possible Java or C programs would lead to an enormous amount invalid games). Also, to search through a set of games it is necessary to be able to calculate a fitness value for each game, valuing how enjoyable the game is.

2.3 Framework: VGDL and the GVG-AI framework

Regardless of which approach to game generation is chosen, one needs a way to represent the games that are being created. For a sufficiently general description of games, it stands to reason that the games are represented in a reasonably generic language, where every syntactically valid game description can be loaded into a specialised game engine and executed. There have been several attempts to design such GDLs. One of the more well-known is the Stanford GDL, which is used for the General Game Playing Competition [?]. The language is tailored to describing board games and similar discrete, turn-based games. However, it is arguably too verbose and low-level to support search-based game generation. Another attempt at an VGDL is called PuzzleScript [?], created by game designer Stephen Lavelle. The language – as its name suggest – is focused on (turn-based) puzzle games, but the engine allows for simple forms of animation and movement, for a more action-oriented game design. The language is relatively high level compared to Stanford GDL, but at as result the complete space of games is rather restricted.

2.3.1 VGDL game descriptions

The various game generation attempts discussed above feature their own GDLs of different levels of sophistication; however, there has not until recently been a GDL for suitably for a larger space of video game types- and genres. The Video Game Description Language (VGDL) is a GDL designed to express 2D arcade-style video games of the type common on hardware such as the Atari 2600 and Commodore 64. It can express a large variety of games in which the player controls a single moving avatar (player character) and where the rules primarily define what happens when objects interact with each other in a two-dimensional space. VGDL was designed by a set of researchers **??** (and implemented by Schaul **?**) in order to support both general video game playing and video game generation. In contrast to other GDLs, the language has an internal set of classes, properties and types that each object can defined by, which the authors suggest the user to extend.

Objects have physical properties (i.e. position, direction) which can be altered either by the properties defined, or by interactions defined between specific objects. Playing the games also required a specified level which defines a set of game tiles, indicating the initial locations of sprites. Each sprite can move a distance specified by its `speed`-parameter in the four cardinal direction, which supports decimal values causing sprites to be able to move in-between game tiles.

A VGDL description has four parts:

SpriteSet

Defines which sprites can appear in the game. Each sprite must be designated by a class, in which a set of predefined actions exists. Also a set of parameters can (and sometimes, must) be fed to each sprite, configuring for instance the `speed` or how often the sprite takes an action – using the `cooldown`-parameter – and a set of parameters unique to the different sprite-classes. For instance, for the class `Flicker`[1] the lifetime can be adjusted, and the `Chaser` sprite class[2] it is necessary to define which object to actually chase. Another interesting sprite types is the `Resource`-class, which, in combination with interaction-rules, can be "collected" by other sprites and several effects can occur (again defined in the interaction-rules), like "if avatar has collected more than 10 coins, he/she is killed when touching wall sprites". Sprites can additionally be designed in a tree structure, where multiple sprites have the same parent sprite, making descriptions for interaction- and termination rules (described below) less verbose.

[1] `Flickers` is a simple extension to the base sprite; the sprite is destroyed after a speecifed amount of time.
[2] `Chaser` sprites move towards a specified other sprite, every time it is allowed to move.

InteractionSet

Each line of the `InteractionSet` specifies a pair of sprites and an effect, defining the resulting effect of the two sprites interacting with each other (i.e. colliding with each other). Possible interaction effects include `stepBack`, causing the first sprite specified to be pushed back to its last position, `bounceForward`, causing the sprite to be pushed one tile forward, and `killSprite`, which simply kills the first sprite defined on collision. Many interaction effects additionally have parameters which must be set for the rules to have a function, for instance for the rule `transformTo` (which converts the first sprite defined from one type to another), it must be defined which sprite to transform to. All interaction effects also have a possibility of changing the players score on collisions.

TerminationSet

The termination set defines how the game can end. Each line in this set has a win parameter, which is set to true or false; determining if the effect causes a winning or losing the game. Each termination-function is represented by a class defining under which conditions the game should end, for instance the `SpriteCounter` class can cause the game to be won or lost, when there exists 0 of a certain sprite (when all of the sprites are killed). Most of the VGDL games examined- and generated in this thesis simply contain two termination rules: A win- and a lose- termination.

LevelMapping

The job of the `LevelMapping` (see figure 2.2) is to translate from a character (`char`) in a level-file (explained below), to sprites from the `SpriteSet`. A single mapping can be shared by several sprites, causing a series of sprites to be created on the same game-tile.

Level description

Besides the four sets used to describe a game in VGDL, to actually play a game a level file is additionally needed, defining the initial positions of a set of sprites using the `SpriteSet` combined with the `LevelMapping`, in which each textual character defines which sprites to appear in which tile-, while spaces defines empty tiles of the game.

2.3.2 The GVG-AI framework: AI controllers and testing

The GVG-AI framework is a testbed for testing general game-playing controllers on games specified using VGDL. Controllers are called once at the beginning of each game for setup, and then once per clock tick to select an action. Controllers do not have access to the VGDL descriptions of the games. They receive only the game's current state; the position of each sprite, and their "types" (e.g. portal, immovable, resource), passed to a controller when it is asked to select an action. However these states can importantly be forward-simulated to future states. Thus the game rules are not directly available, but a simulatable model of the game can be used.

GVG-AI sample controllers

A series of general game playing controllers is additionally available in the framework, using a series of different approaches to select an action on each clock tick. One of these controllers were chosen to be used in this project to examine designed- and generated games:

SampleMCTS Simulates the game space of possible actions within a small area of the games, using a open-loop search and a "Vanilla" MCTS approach using UCT.

GVG-AI example games

The framework additionally contains 20 human-designed games, which partly consist of interpretations of classic video games (e.g. *Boulderdash*, *Frogger*, *Missile Command* and *Pacman*), while some are original creations by the General Video-Game AI Competition's organisers. The interpretations of existing games all have different stages of simplifications to them, causing several of the games' features to be missing – for instance, the player cannot push boulders in the VGDL version of Boulderdash and they do not "roll" as in the original, and the ghosts in Pacman behave

```
BasicGame
    SpriteSet
        forestDense > SpawnPoint stype=log prob=0.4  cooldown=10 img=forest
        forestSparse > SpawnPoint stype=log prob=0.1  cooldown=5 img=forest
        structure > Immovable
            water > color=BLUE img=water
            goal  > Door color=GREEN img=goal
        log     > Missile   orientation=LEFT  speed=0.1 color=BROWN img=log
        safety > Resource  limit=2 color=BROWN img=mana
        truck  > img=truck
            rightTruck  > Missile   orientation=RIGHT
                fastRtruck  > speed=0.2  color=ORANGE
                slowRtruck  > speed=0.1  color=RED
            leftTruck  > Missile   orientation=LEFT
                fastLtruck  > speed=0.2  color=ORANGE
                slowLtruck  > speed=0.1  color=RED
        # defining 'wall' last, makes the walls show on top of all other
            sprites
        wall > Immovable color=BLACK img=wall

    InteractionSet
        goal avatar  > killSprite scoreChange=1
        avatar log   > changeResource resource=safety value=2
        avatar log   > pullWithIt   # note how one collision can have multiple
             effects
        avatar wall  > stepBack
        avatar water > killIfHasLess  resource=safety limit=0
        avatar water > changeResource resource=safety value=-1
        avatar truck > killSprite scoreChange=-2
        log     EOS  > killSprite
        truck   EOS  > wrapAround

    TerminationSet
        SpriteCounter stype=goal   limit=0 win=True
        SpriteCounter stype=avatar limit=0 win=False

    LevelMapping
        G > goal
        0 > water
        1 > forestDense water       # note how a single character can spawn
            multiple sprites
        2 > forestDense wall log
        3 > forestSparse water       # note how a single character can spawn
            multiple sprites
        4 > forestSparse wall log
        - > slowRtruck
        x > fastRtruck
        _ > slowLtruck
        X > fastLtruck
        = > log water
        B > avatar log
```

Figure 2.1: Example of a VGDL game description: An interpretation of the classic arcade game *Frogger*

in a much more simplified fashion than in the original. A short summary of each of the example games can be seen in Appendix A

The games are – except for the game *Soko-Ban* – of the *action-arcade* genre (defined in the introduction), in that the player controls a single avatar which must be moved quickly around in a 2D-setting to win, and to get a high score. Figure 2.1 shows the VGDL description of the game *Frogger*, and 2.2 a level description from the same game. Figure 2.3 show how some of the games are visually represented when playing in the GVG-AI framework.

Game genres of VGDL and the GVG-AI framework

Using the VGDL implementation of the GVG-AI framework the games that can be described are rather restricted to certain genres- and types, without greatly extending the program. For instance,

```
wwwwwwwwwwwwwwwwwwwwwwwwwwwww
w                wGw                w
w00==000000===0000=====000=2
w0000====00000000000====00012
w00===000===000====0000===02
www   ww   www     www   wwwww
w   ———   ———   —   ———   w
w—      xxx         xxx      xx  w
w  —   ———      —    ———  ——  w
w          A                    w
wwwwwwwwwwwwwwwwwwwwwwwwwwwww
```

Figure 2.2: Example of a VGDL level description: A level for the interpretation of *Frogger*

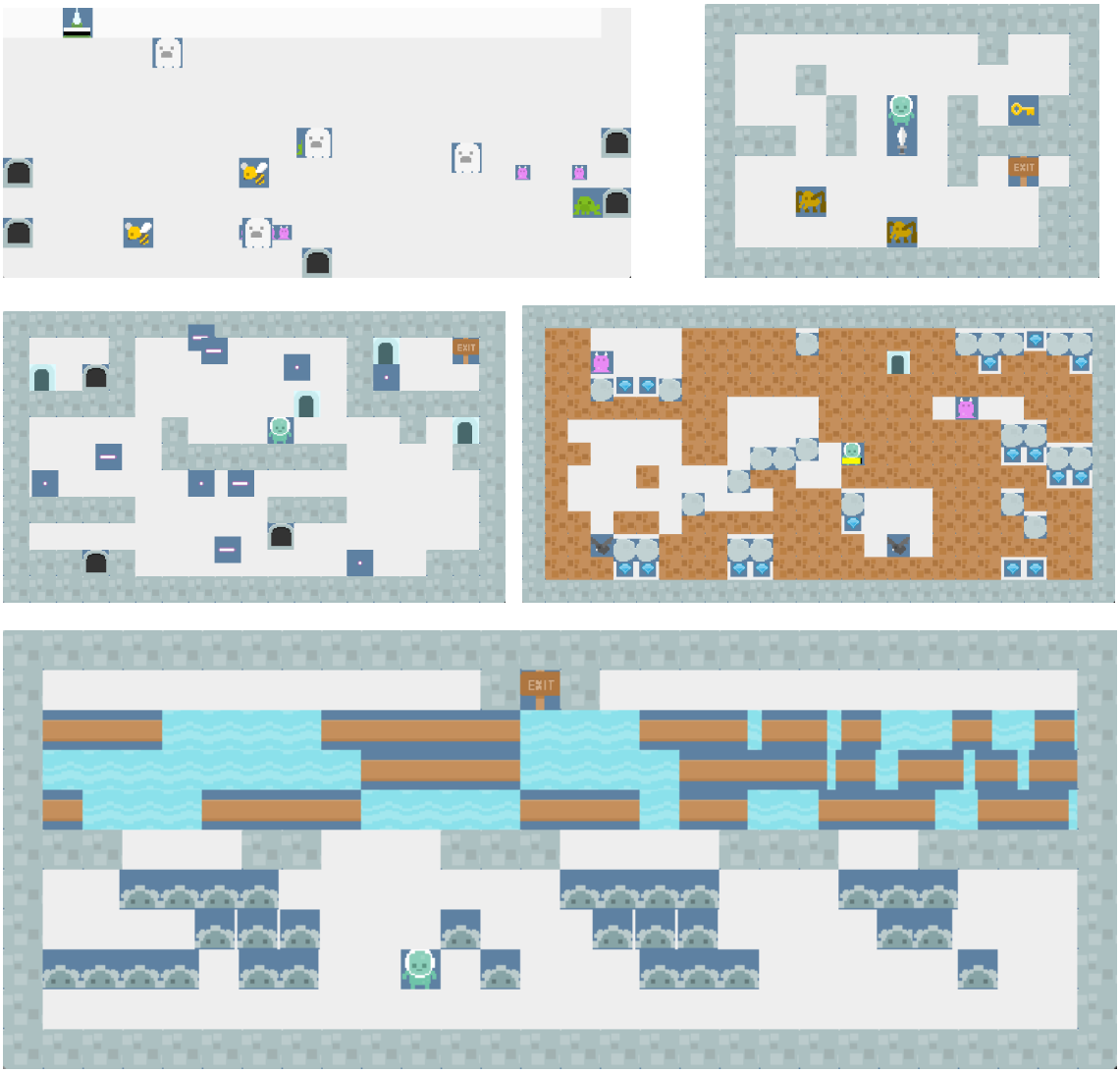

Figure 2.3: The visual representation of a few of the VGDL example games which are interpretations of commercial games. From top-left: *Seaquest*, *Zelda*, *Portals*, *Boulderdash* and *Frogs* (*Frogger*).

the lack of possibility to traverse different levels in the same play through make *adventure* games unattainable, and the restriction of only moving a single character with the keyboard (the avatar) makes great restriction in creating *strategy* games.

The games examined- and generated in this thesis are of a type that makes them suitable for

the framework: Action-arcade- and puzzle games.

As mentioned in the introduction, I make the simple distinction between the two types of games in the VGDL framework, that arcade games can contain elements (sprites) that move by themselves, whereas interactions only occur as a result of the avatar moving in puzzle games. As a results, certain sprites- and interaction classes are restricted to the action-arcade games (e.g. NPC- and Missile sprites), making the space of possible VGDL *puzzle games* slightly smaller.

Chapter 3

Extending VGDL

This chapter describes a series of extensions I constructed for the GVG-AI framework, making a more in-depth analysis possible. I created a series of VGDL games, AI controllers and a implemented a new, simplified framework to play through *puzzle games* with less time and memory use.

3.1 Writing new VGDL games

To increase the size of the set of designed games, to allow for a more precise analysis of what makes a good game, I created fourteen new game descriptions in VGDL.

Another important reason for creating new descriptions, was to introduce a series of *puzzle games* to be analysed, since, as mentioned in Section 2.3.1 the example games from the GVG-AI competetion are almost all of the *action-arcade* genre.

Describing existing games in VGDL

The main goal when developing a game - which also applies in this case - is to ensure that it is enjoyable to human-players. Therefore the games implemented are all interpretations of published existing games (and not original creations). As described in Section 2.3.2 VGDL can only describe relatively simple games of certain types/genres, and so only a limited number of games can be translated without severely changing the gameplay.

The games I created (described below) are in general "true" copies of the originals - containing all the gameplay features and interactions but lacking elements like audio, graphics and controls. However many lack certain (non-essential) game-play features like bonus points (like fruits in Pac-Man), infrequently appearing enemies (UFOs in Space Invaders) or features which only appear in later levels of the games.

Interpreted games

A series of fourteen commercial games was re-created in VGDL to be used in future tests. Figure 3.2 shows one of the games translated to VGDL. Minor changes to the GVG-AI framework was made for a small number of the games, to be able to correctly copy the games' core gameplay features. The original game levels were additionally translated into VGDL level description, with five levels being created for each game. For some of the games the levels could not be completely copied, for instance because the levels were too large in size (*Bolo Adventures*), or because a lack of game features implemented in the clones (*Chip's Challenge*), and the resulting levels are therefore simplifications of the originals..

Action-arcade games

A set of four Atari -arcade and -2600 games were found to be suitable for interpretation; *Solar Fox*, *Crackpots*, *Centipede* and *Astrosmash*, and a VGDL game description was written for each. A description of each of the games and their interpretations can be seen in Appendix B.

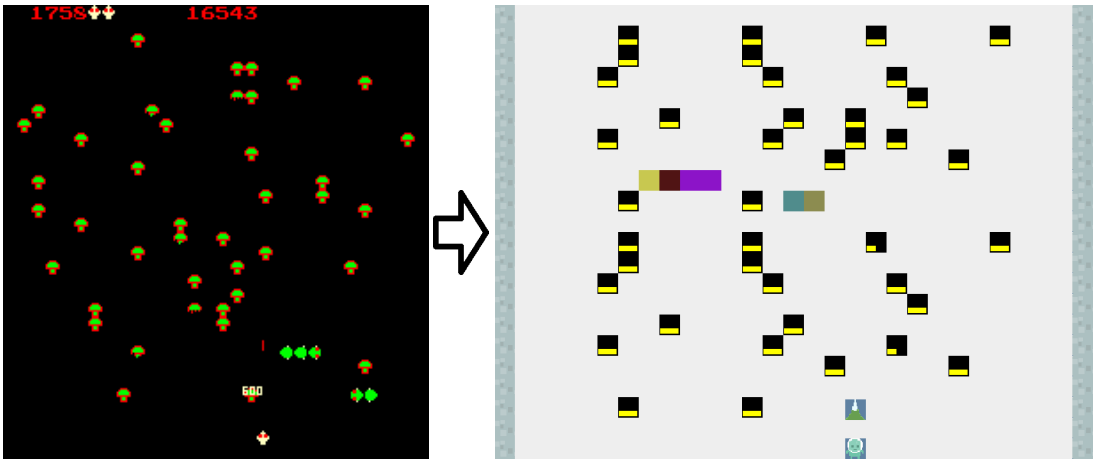

Figure 3.1: Interpretation of the classic arcade game *Centipede* [?]

Puzzle games

A set of puzzle games from several different platform, all featuring a single avatar, was found to be suitable to be described in VGDL. The games can mostly be described as *Soko-Ban*-clones with different spins on the gameplay. Some of the games have action- gameplay aspects, with enemies hindering the progression of the player, but as per the defintion of *puzzle games* used in the thesis these parts were removed only leaving the puzzles of games.

However, even with the removal of enemies, a few of the interpreted games contain elements which are not strictly in line with the definition of VGDL *puzzle games*, for instance some of games feature gliding boxes or rolling boulders (implemented using the `Missile` sprite class), which can cause effects to happen many frames after the avatar has acted.

The games interpreted were *Soko-Ban*, *Bait*, *Bombuzal*, *Bolo Adventures*, *Zen Puzzle*, *The Citadel*, *Brainman*, *Chip's Challenge* and *Modality*. A summary of the games can be seen in Appendix B.

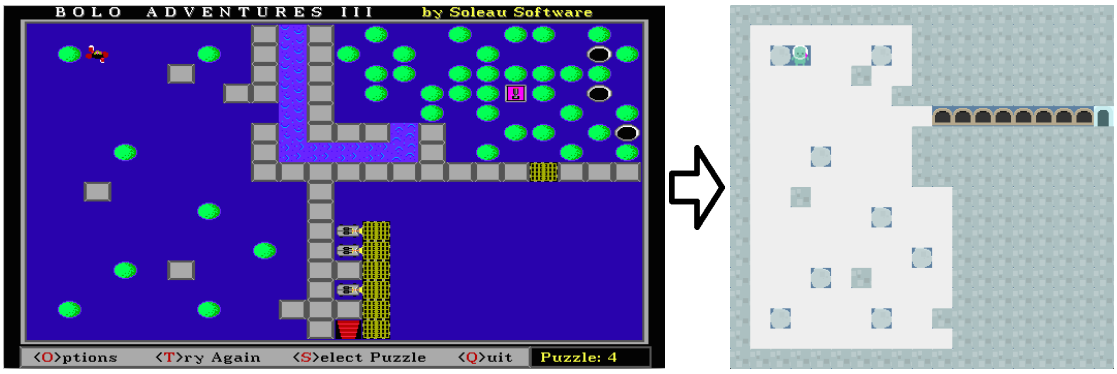

Figure 3.2: Interpretation of both the game and part of a level, from the DOS game Bolo Adventures III

3.2 Creating new VGDL controllers

To be able to probe VGDL games in more detail, I created a series of new AI controllers with various approaches to finding the action to take. In total four new controllers was generated: Three to play the action-arcade style games, and one only focused on solving puzzle games.

Action-arcade controllers

Three controllers of an increasing degree of cleverness was introduced: *One-step* (simplest), *Deep-search* and *Explorer* (most complex), with different strategies used in simulating the forward model

(accessible when running the GVG-AI framework).

OneStep The One-step algorithm only attempts to advance the forward model once for each allowed action in the game. The score and win/lose state of the resulting game states are then considered, and the action with the resulting best state is chosen, with a random action chosen when no state is better than others. The controllers approach can be seen below:

Algorithm 1: One-step algorithm

```
for action : PossibleActions do
    newGameState = gameState.copy().advance(action)
    value[action] = value(newGameState)
return action leading to highest value, or random action if values are equal
```

DeepSearch This controller starts out in a similar fashion as the One-step, by expanding using all possible actions by copying the initial game-state. These game-states are then simulated further upon by advancing each game state a single time, with a partly[1] random action, without copying the state (which is a rather costly procedure). This expansion is continued until the controller runs out of time, and a value for each action is calculated by considering the score and win/lose-value for each state that can be achieved from each of the initial states.

Algorithm 2: Deep-search algorithm

```
queue ← initial gameState
while has time left do
    gameState = queue.poll()
    if gameState == initial gameState then
        for action : PossibleActions do
            queue ← gameState.copy().advance(action)
    else
        gameState.advance(random action)
        d ← depth of search //amount actions performed
        value[action] += value(newGameState) *PossibleActions.length^d
return action leading to highest value, or random action if values are equal
```

Explorer The explorer controller was designed specifically to play the arcade-style games of the GVG-AI framework, and utilizes a series of methods to strengthen its decisions. Unlike the other controllers which utilise open-loop searches, it stores information about visited tiles and prefers visiting unvisited locations. In addition the controller uses three different arrays to choose an action on each clock tick: 1) One considering how probable death is from each action, 2) one examining the score that can be achieved from the actions, and 3) one only looking at the boringness – which changes dependant on which tiles has been visited. If the *death*-array have too high values, that array is used to take the decision (the action leading to the lowest value is chosen), otherwise the best action from the score-array is used. If the values of the *score*-array happen to be too similar, the boringness-array is used as last resort.

The controller also addresses a common element of the VGDL example games, randomness. The controller gains an advantage in many of the games by simulating the results of actions repeatedly, before deciding the best action.

The Explorer was proven to be of a decent quality by getting a 1st / 5th place in the GVG-AI's competition between controllers [2].

[1] A key part of success of the algorithm is that it, after the initial expansion, never tries to advance in a opposite movement direction to the initial.

[2] The controller (nicknamed *thorbjrn_other*) has at the time of writing (28/02-2015) a 1st place in the training set of the competition (first 10 games of the GVG-AI framework), and 5th in the validation set (the second 10 games): http://gvgai.net/gvg_rankings.php?rg=1

Puzzle controller

The general game playing algorithms from the GVG-AI competition- and the algorithms described above are not well-suited for playing *puzzle games*, because the goal can often only be achieved by applying a specific series of actions in the genre, which the controllers are not very good at. To analyse an arbitrary puzzle game and be able to find the fastest solution, a breadth-first search algorithm was implemented, using a few methods to decrease the state-space by assumptions of the games.

PuzzleSolver The PuzzleSolver algorithm use the fact the puzzle games (using the definition of the introduction) only contain elements which can be moved or interacted with by the player-avatar, and contains no random elements, and so the initial state of a game in GVG-AI can be used to simulate all possible actions that eventually will lead to finding the solution of the games. Additionally the PuzzleSolver uses the face that most of the puzzle games contain wall-sprites which always push back the player, and so the controllers never try to move into a wall – this is achieved by storing the positions of every wall-sprite at the start of the game, and never simulating actions that lead to the avatar in a walls position.

Also, for each path the algorithm attempts, a signature of the game state is calculated and stored, constructed from the position and types of each (non-wall) sprite appearing in the given state. A path is cut off if the calculated game state is the same as has appeared before.

A problem of the approach is that controller adds several more signatures than needed if the game contains for instance `Missile-` or `Flicker` sprites, which are used in a few of the designed *buzzle games.*. For the latter type of sprite a solution was implemented by "skipping" game states in which a `Flicker`-sprite exists (which have a limited lifetime), simply returning a NIL action until all `Flickers` has disappeared.

3.3 SimpleVGDL

Since the GVG-AI framework has some functions and properties which are not important for, especially, the *puzzle games* analysed in this work, I created an implementation of a lighter version of VGDL, solely focused on analysing puzzle games in more detail. The implementation is basically a clone of the GVG-AI framework, but with several time- and memory consuming features removed, which in part was possible due to the puzzle games being relatively more simple (for instance, only movement from one tile to another is possible in SimpleVGDL, whereas sprites can move and collide in-between tiles in the GVG-AI framework).

In the SimpleVGDL framework the PuzzleSolver algorithm was additionally geared up with a two different approaches to memory handling: 1) Storing the actual game state for nodes in the search queue, and 2) only storing the game state signature and path, making it necessary to re-create the game state from the initial state of the game, whenever an element is polled from the queue.

Below I present the results of a small test of running a increasingly difficult game/level-pairs using the puzzle solver, in both the GVG-AI framework and using SimpleVGDL, both with- and without the "low-memory" option.

framework	*time*	*mem.*
GVG-AI	962 ms	32 MB
SimpleVGDL	535 ms	6 MB
SimpleVGDL_LowMem	3642 ms	5 MB

Figure 3.3: Results from playing through the game *(Real) Soko-Ban*, level 4 (of 5) using the puzzle solver controller

The results show a clear difference in time and space usage between the GVG-AI framework and SimpleVGDL, with SimpleVGDL being about twice as fast. At the same time there is a significant time difference between using the low-memory approach or not, but this does not carry over when

framework	*time*	*mem.*
GVG-AI	9420 ms	270 MB
SimpleVGDL	4354 ms	59 MB
SimpleVGDL_LowMem	41691 ms	58 MB

Figure 3.4: Results from playing through the game *(Real) Soko-Ban*, level 2 (of 5) using the puzzle solver controller

framework	*time*	*mem.*
GVG-AI	314210 ms	1892 MB
SimpleVGDL	142547 ms	638 MB
SimpleVGDL_LowMem	163385 ms	386 MB

Figure 3.5: Results from playing through the game *The Citade*, level 5 (of 5) using the puzzle solver controller

framework	*time*	*mem.*
GVG-AI	OutOfMemoryError	OutOfMemoryError
SimpleVGDL	OutOfMemoryError	OutOfMemoryError
SimpleVGDL_LowMem	327441 ms	347 MB

Figure 3.6: Results from playing through the game *Brainman*, level 2 (of 5) using the puzzle solver controller

the levels become increasingly difficult (see Figure 3.5). The space usage is even more varying when the program is running, because the memory-heavy solutions store a large amount of data in the queue, but the memory used to store each game state signature are the same, often causing almost the same amount of memory to be used up at the end of the process (when the queue is exhausted when the solution is found). As can be see from the last test (Figure 3.6) the main advantage of the approach is that it can actually finish certain game/level-pairs, where the others reach a `OutOfMemoryError Exception`.

Chapter 4

Analysis of randomly generated games

This chapter describes the setup- and methodology used in generating new games (i.e. game descriptions). Additionally a play test of a set of both human-designed and generated games, using an array of general game playing algorithms, will be presented.

This chapter is only focused on *action-arcade* games - analysis and generation of *puzzle games* will be presented and discussed in Chapter 6.

Selection of designed arcade-action games

A subset of the human-designed games mentioned in Section 2.3 and 3.1 were selected for the following analyses games, and for creating new games by mutation.

After my addition of VGDL game descriptions to GVG-AI framework there is a total 33 human designed games which could potentially be used as a base-line for quality in games. However, using the definitions from the introduction, only 23 of the games are of the *arcade-action* genre explored in this section, which the general game playing algorithms play decently. Additionally, as mentioned in Section 2.3.1 several of the example games from the GVG-AI competition are original creations by the completion organisers, which cannot equally be assumed to be human-enjoyable.

With these limitation, thirteen interpretations of existing games are left and used as a baseline for testing and game generation: Aliens, Boulderdash, Frogs, Missile Command, Zelda, DigDug, Pacman, Seaquest, Eggomania, Solar Fox, Crackpots, Astrosmash and Centipede.

Besides the additions and changes to the GVG-AI framework to make new games possible, a slight change was done to remove the possibility of players scoring lower than 0, which is the norm in the original games mentioned – and also makes comparing scores across controllers more straightforward.

4.1 Generating processes

Using the GVG-AI framework I constructed processes for generating large sets of new game description. I used two different approaches in generating new game description in VGDL: Mutating existing (designed) game descriptions-, and randomly generating new ones (from scratch). In both approaches several constraints were set upon the generation process, either to prevent crashes or limit parameters to values known to be good in the designed games, causing a slight decrease in the space of potential generated games.

4.1.1 Mutation of example games

A process was built for parsing existing VGDL game descriptions, mutating certain sprites- or rules, and returning new, valid game descriptions,. The set of human-designed games described at the beginning of this chapter, were chosen to be used as a basis for mutating new game descriptions.

Generating approach

It is not immediately apparent which approach to use in mutating the different elements of each VGDL game description. Since the sprite definitions- and rules of a VGDL game are constructed from a combination of a class and a series of parameters specific to the class, it is necessary to change the parameters if the class is changed, but the parameters of an existing class can freely be mutated.

A process was built for mutating each of the different parts of game descriptions, i.e. changing the sprites available of the `SpriteSet`, the interactions rules (`InteractionSet`) and the termination rules (`TerminationSet`). The process additionally allowed for changing the amount of rules, i.e. generating new rules or removing existing ones.

Since the sprites and rules of each game description is described by both a class and a series of parameters, several partly subjective decisions was made on defining probabilities of different parameters' values. For instance it was decided that a sprite being mutated or generated only has a 25% chance of using the `cooldown` parameter (making sprites have pauses between acting), and that the parameter have a random value between 1 and 10. The values and probabilities used were mostly decided by examining the set of designed games descriptions. Several constraints were also used to avoid game with non-valid descriptions (which can cause crashes in the GVG-AI framework), for instance by ensuring that avatar-sprites cannot be spawned and sprites cannot transform to an avatar, but an avatar sprite can still transform to another type of avatar sprite.

In this work I decide to only mutate interaction-rules from the `InteractionSet` of designed VGDL games, and simply change every part (classes, parameters and references) of a random amount of rules, with each rule having a 25% change of being changed for each game to mutate. In addition to simplifying the mutation process, and having games more with a higher chance of being playable this allows us to use the original designed games' levels for testing.

Figure 4.1 shows a typical outcome of mutating the `InteractionSet` of a VGDL game: Some of the new generated rules might never be triggered or have a neglible effect, while overwriting some of existing rules.

```
1  %zelda
2  InteractionSet
3    movable wall  > stepBack
4    nokey goal    > stepBack
5    goal withkey  > killSprite scoreChange=1
6    enemy sword > killSprite scoreChange=2
7    avatar enemy > killSprite scoreChange=-1
8    key  avatar   > killSprite scoreChange=1
9    nokey key     > transformTo stype=withkey
10
11 %zelda_mutation_3
12 InteractionSet
13   movable wall > stepBack
14   monsterQuick monsterQuick > attractGaze
15   sword sword > killSprite
16   enemy sword > killSprite scoreChange=2
17   avatar enemy > killSprite scoreChange=-1
18   key avatar > killSprite scoreChange=1
19   nokey key > transformTo stype=withkey
```

Figure 4.1: Mutation of the `InteractionSet` of the VGDL interpretation of the game *Zelda*

4.1.2 Random game generation

A similar process as mentioned above was used to randomly put game descriptions together, creating new games completely from scratch and constructing the textual lines for the four parts of a VGDL description: Generating an array of sprites (for the SpriteSet), interaction-rules (InteractionSet), termination-rules (TerminationSet) and level mappings (LevelMapping).

Several partly subjective choices were again made for the possible amount of different sprites, and interaction- and termination rules. Even though some of the original games contain a larger set of elements, I constricted the ranges to sprites: $3-8$, interactions: $3-10$ and terminations: 2 (a "win-" and a "lose"-termination), to reduce the space of generated games.

When generating descriptions, I used similar constraints to those mentioned in the previous section, partly to avoid generating descriptions with invalid elements, and partly to increase the proportion of interesting outcomes. To simplify the sprite creation process, no parent-child structure was used in the `SpriteSet`[1].

All sprites were given a random sprite image, while the avatar-sprite was given the same sprite for each game, to make the game more easy to understand when visualising the games (and actually playing through them). Figure 4.2 shows an example of a game generated using the process.

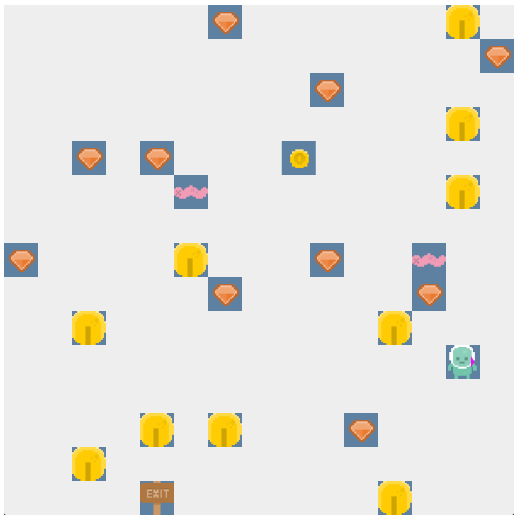

Figure 4.2: Visual representation of a randomly generated VGDL game

Level generation

The problem of generating a game is more often than not intimately linked to a generation of levels. I.e. one could end up with generating game descriptions of high quality games, but if the level descriptions does not fit to the game-play (by being too trivial, or too hard for instance) the games might not be found if searching through a set of generated games.

A simple level generator were constructed with the simple goal of making the randomly generated games playable. Using each sprites mapping of a game description (defined in the `LevelMapping`), the generator designates a level using a given size, and inserts a random amount of each sprite, at random positions.

Figure 4.3 shows an example of generated game description-, and Figure 4.4 the level generated for the game, by using process described above.

4.2 Playthrough of generated games: Experimental setup

Using the generation presented in the previous section, I performed a playthrough test of both the designed- and generated games using different general game playing algorithms (controllers), and discuss different ways to analyse the resulting data.

Controllers

Six different controllers were used to play through the games. The controllers use an array of varying approaches, which can be described as a different degree of intelligence. One of the more "intelligent" controllers used were included in the GVG-AI framework (SampleMCTS), while the remaining were implemented for this work (Explorer, DeepSearch and OneStep). In addition two extremely simple controllers were created to be used as a basis for "bad" play: 1) A *Random* controller, which always returns a random action, and 2) a controller which never acts at all and always return an "NIL"-action, with the appropriate name *DoNothing*.

[1] However, the goal of the parent-child structure is only to make rule definitions less verbose, and does not make actual new game features possible.

```
BasicGame
  SpriteSet
    avatar > HorizontalAvatar img=avatar cooldown=2
    gen1 > Resource limit=10 value=5 img=fire
    gen2 > RandomNPC img=wall
    gen3 > Resource limit=4 singleton=TRUE value=1 img=explosion
    gen4 > OrientedFlicker limit=23 orientation=RIGHT img=water
    gen5 > Spreader limit=14 stype=gen1 img=spaceship
  InteractionSet
    avatar gen1 > killIfFromAbove
    avatar gen2 > killIfOtherHasMore limit=4 resource=gen1
    gen5 gen1 > cloneSprite
    gen4 gen5 > pullWithIt
    gen3 avatar > changeResource value=1 resource=gen1
    avatar gen1 > undoAll
    avatar gen4 > stepBack
    avatar gen5 > stepBack
  LevelMapping
    $ > gen1
    % > gen2
    & > gen3
    ' > gen4
    ( > gen5
  TerminationSet
    SpriteCounter limit=0 stype=avatar win=TRUE
    MultiSpriteCounter limit=0 stype1=avatar stype2=gen3 win=FALSE
```

Figure 4.3: A randomly generated VGDL game description

```
                %
     %%
       %'
                %
           %
  %
              %

  $             &

          % &
  %

     (   %    A%
         %    %
```

Figure 4.4: A random generated level description, generated for the VGDL game above

Generated games

A set of 130 VGDL games were generated by mutating each of the 13 designed games ten times, using the approach described in the previous section. 400 randomly generated games (generated from scratch), each accompanied with a single randomly generated level of size 15x15, were additionally generated and used in testing.

Testing and result analysis

The six controllers were used to play through the set of human-designed-, mutated and randomly generated games. Because of CPU budget limitations each game was played through ten times, with a maximum amount clock ticks of 2000 and 50 ms allowed for each tick. In addition, a function was implemented to stop playing through games if a clock tick ever takes more than 50 ms, which would happen often when generated games had rules generating too many sprites, which can end up taking a severe amount of time to play through if not discarded quickly. For each playthrough of games, only the most essential data from the GVG-AI framework was retrieved: The score, the win-lose value, the amount of clock ticks used and a list of actions performed.

A process to analyse the data was built, to calculate averages, standard deviations, max-, minimum and other statistics of each of the values, for each controller, for both each individual game and over a whole set of games. To more accurately compare the score for the different controllers when playing across a range of different games I additionally calculate a normalised score using a max-min normalisation ($\frac{score-min}{max-min}$), using the maximum/minimum score for each play through, across the controllers. The entropy of actions performed for each playthrough, and averages across playthroughs- or set of games, was additionally calculated

4.3 Playthrough of generated games: Results

Using the setup described above averages of all play-throughs for each controller, is presented and the results compared with each other.

In the results below, a series of data from the generated set of games were removed due to being either too difficult to analyse, or for having certain values which I assess could not possibly result in a quality game. Namely I remove games by three different considerations: 1) When a controller was disqualified (because of too many sprites- and/or interactions happening, making a frame using more than the allowed 50 ms), 2) when all controllers get the same score – and win-rate – for all playthroughs, and 3) where the game always (for all controllers, for all playthroughs) end in less than 50 clock ticks[2].

It should be noted that the uncertainties for average score, clock-ticks and action entropy in the data below assume a normal distribution of the data, which in general is not true, and so these values should be taken with a grain of salt.

Designed games

The six controllers were first of all set to play through each of the human-designed VGDL games. Below the averages across all the games from the designed set can be seen (Figure 4.5). Averages across each single game can be seen in Appendix C,

The distributions show that there is a large difference between the results of the more intelligent algorithms and that of the intended ´´intended-to-be-bad" controllers, with both a higher win-rate and a significantly higher average score. Surprisingly, the DeepSearch algorithm is has a significant lead over the other algorithms, in spite of its simple approach to selecting actions.

The *clock-ticks* and *action entropy* does not show a similar clear relationship between the more- and less intelligent algorithms, for the collection of all games. Examining the results from each game it is noticeable that the profiles for these values (*clock-ticks* and *entropy*) are very different from game to game.

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	138.67 (20.59)	0.69 (0.03)	0.65 (0.04)	556.35 (46.71)	0.75 (0.04)
DeepSearch	370.82 (61.86)	0.82 (0.03)	0.72 (0.04)	789.95 (59.73)	0.75 (0.04)
MCTS	231.17 (48.47)	0.57 (0.03)	0.52 (0.04)	985.31 (59.20)	0.77 (0.04)
Onestep	33.16 (4.56)	0.24 (0.03)	0.16 (0.03)	442.77 (47.54)	0.76 (0.04)
Random	22.22 (3.25)	0.13 (0.02)	0.08 (0.02)	261.98 (35.30)	0.78 (0.04)
Do Nothing	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	734.72 (70.09)	0.00 (0.00)

Figure 4.5: Averaged results across all the 13 designed games

Mutated games

After removing games by the requirements mentioned above I was left with a total of 90 game descriptions with some form of playability. The results of playing through these games with the

[2]This is also the maximum lifetime of generated `Flicker`-sprites – initial trials into VGDL game generation showed that many games was won when a certain type of sprites was all destroyed, but the sprite in question were `Flicker`s and so even the simplest controller would always win.

controllers can be seen in Figure 4.6. The scores have higher means and standard deviations, indicating outliers in the data. The ordering of the *normalised score mean* and *win-rate*, however, shows a similar pattern as for the example games, but with the DoNothing controller scoring points and even winning games. In general the extreme values of norm. score and win-rate is smaller than for the designed set of games.

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	8533.35 (1576.40)	0.70 (0.01)	0.44 (0.02)	1084.34 (26.61)	0.69 (0.02)
DeepSearch	4224.94 (756.10)	0.68 (0.01)	0.43 (0.02)	1206.04 (26.17)	0.74 (0.01)
MCTS	2002.26 (377.34)	0.47 (0.01)	0.29 (0.02)	1381.99 (24.68)	0.78 (0.01)
Onestep	2256.70 (535.78)	0.32 (0.01)	0.19 (0.01)	1036.34 (27.64)	0.72 (0.02)
Random	873.78 (277.09)	0.21 (0.01)	0.15 (0.01)	745.30 (26.63)	0.79 (0.01)
Do Nothing	985.85 (275.77)	0.08 (0.01)	0.14 (0.01)	1208.18 (27.73)	0.00 (0.00)

Figure 4.6: Averaged results across the 90 (out of 130) mutated games, which fulfil the criteria described at the beginning of this section

Randomly generated games

Figure 4.7 shows results for the remaining 66 randomly generated games (of 400), with problematic games removed according to the same criteria as above.

The *normalised score means* and *win-rates* both have values that are more closely clustered togther, than in the previous game sets. For this set of games the *action entropy* of the intelligent controllers has also fallen quite a bit (relative to the Random-controller), indicating that more of the games have simple solutions to win or increase the score

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	1649.97 (208.06)	0.52 (0.02)	0.11 (0.01)	1800.27 (22.87)	0.68 (0.02)
DeepSearch	1276.81 (132.87)	0.53 (0.02)	0.11 (0.01)	1795.23 (23.13)	0.72 (0.02)
MCTS	546.75 (64.27)	0.40 (0.02)	0.10 (0.01)	1811.42 (22.27)	0.73 (0.02)
Onestep	702.92 (139.02)	0.34 (0.01)	0.08 (0.01)	1857.14 (19.24)	0.74 (0.02)
Random	278.14 (34.81)	0.29 (0.01)	0.08 (0.01)	1860.77 (19.06)	0.75 (0.02)
Do Nothing	1191.83 (253.25)	0.24 (0.01)	0.02 (0.01)	1964.15 (9.94)	0.00 (0.00)

Figure 4.7: Averaged results across the 66 (out of 400) randomly generated games which fulfil the criteria of a "not-completely-terrible" game

Outcome and graphs

By examining the graphs below (especially Figure 4.8 and Figure 4.9) presenting the data discussed, it is clear that there is some relationship between the performance profiles of playing with controllers of different types, on different set of games. However it should be noted here that some games from both the mutated- and generated game-set, contains games, which when examined by themselves have similar distributions as the average designed game (these will be discussed below).

4.4 Discussion of initial test results

The results display some interesting patterns. Especially the win-rates suggest a relationship between intelligent controllers' success and better game design; for better designed games, the

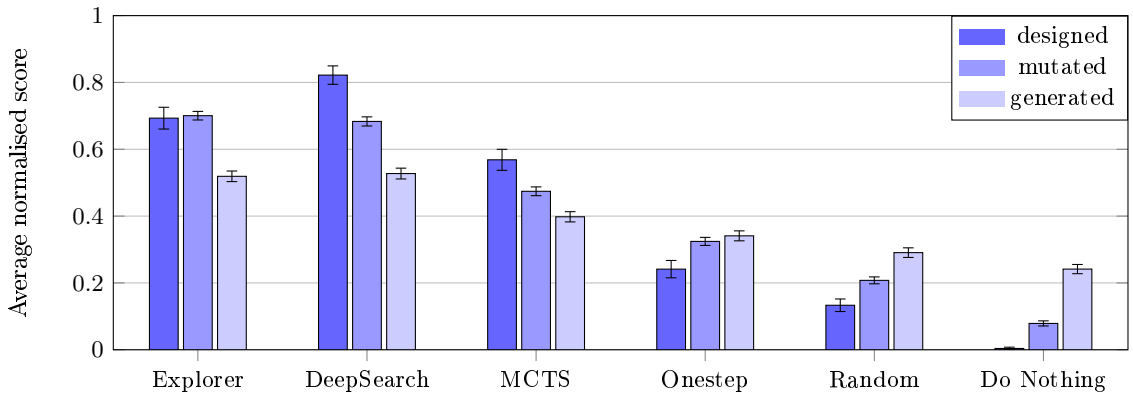

Figure 4.8: Averaged normalised score across all games of the three different set of games: Designed-, mutated- and completely generated games

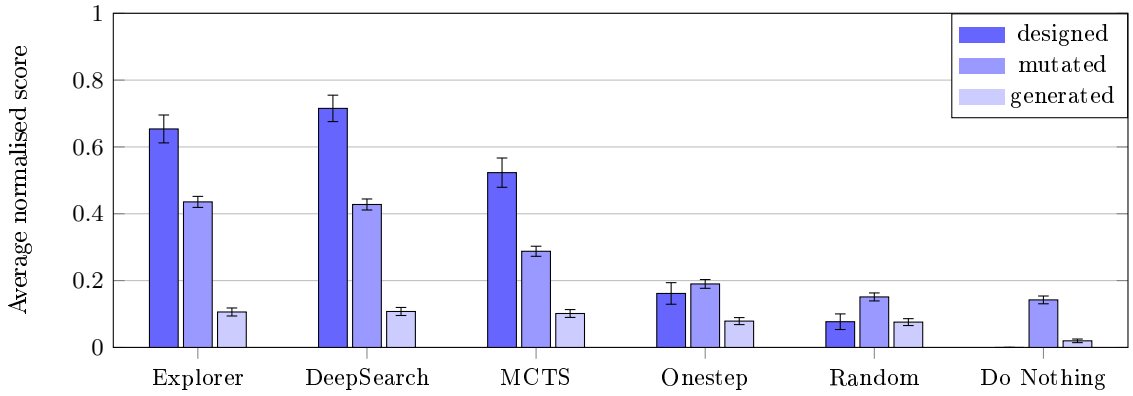

Figure 4.9: Averaged win-rate for the three sets

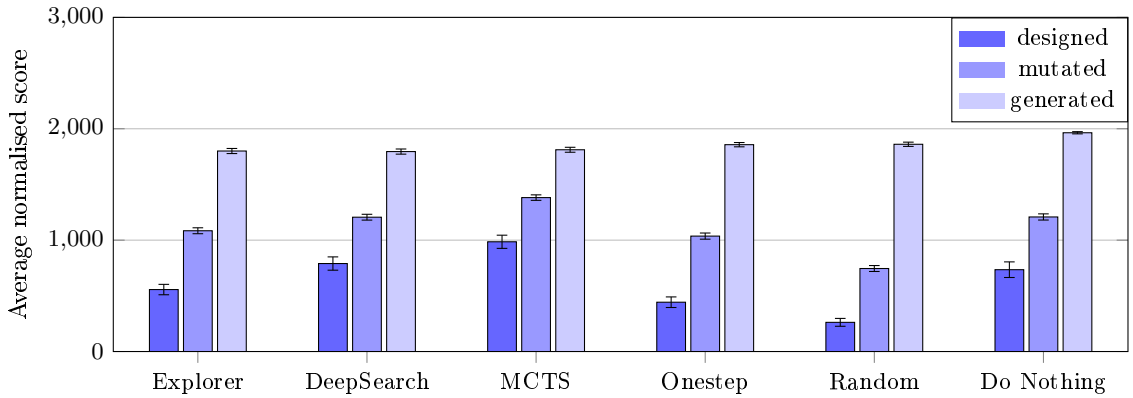

Figure 4.10: Averaged clock tick for the three sets

relative performance of different types of algorithms differ more. This supports the hypothesis that relative algorithm performance profiles can be used to differentiate between games of different quality. In randomly generated games, which arguably tend to be less interesting than the others, smarter controllers (e.g. Explorer and MCTS) do only slightly better than the worse ones (i.e. Random and DoNothing). This is due to a general a lack of consistency between rules generated in this manner. Mutated games, however, derive from a designed game. Therefore, they maintain some characteristics of the original idea, which can improve the VGDL description's gameplay and playability.

While it is possible that random actions can result in good outcomes, this chance is very low, especially when compared to the chance of making informed decisions. In spite of that both Ran-

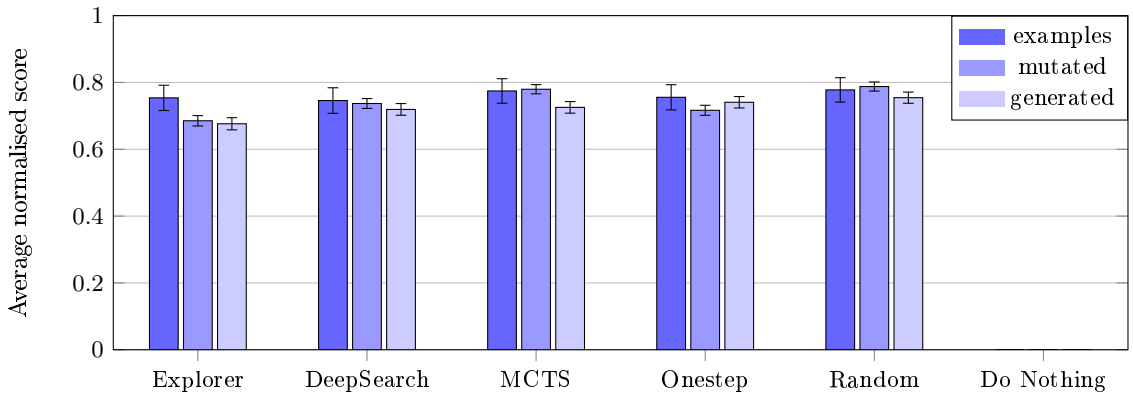

Figure 4.11: Averaged action entropy for the three sets

dom and DoNothing do fairly well in randomly generated games. The performance of DoNothing emerges as a secondary indicator of (good) design: In human-designed games, DoNothing very rarely wins or even scores.

Human play of generated games

As mentioned in Section 4.3 there was a series of generated games with performance profile basically indistinguishable from the designed games. I decided on examining many of these games further by studying their game descriptions, and playing the games with visuals enabled, both with AI controllers and human players. Besides a few mutated games which kept most of the original rules, and therefore were successful games, the tested generated games were in general too trivial and aimless to be entertaining. However, by also comparing these games with other without the "well-formed" performance profiles, it became clear that performance profiles of algorithms could at least be used to separate "not-completely-bad" games from terrible games.

This however gives an indication of in important assumption: That I can assume that the generated games, at least from the randomly generated set, are of a overall low quality. I will carefully use this assumption to construct an evaluation function for games, in the next chapter,

Besides the above, a few re-occurring problems was found in the generated games.

- Sprites often completely leave the level playing field. For instance because they are of the `Missile-` or `Fleeing` (which tries to move away from a specified sprite each frame) sprite classes.
- The game can only be won in the first few (<50) frames. Because sprites disappear before that (for instance because the sprite is
- There is too much purposeless and arbitrary going on: Often some sprites can not be interacted with at all, and just wander around on the screen, or they can only be interacted by certain non-avatar sprites.
- Several of the sprites- and/or rules are never used.

Chapter 5

Evaluation and evolution of games

The initial tests of the previous chapter shows that there is reason to explore the relationship of AI's perfomance profiles to analyze a given VGDL game, and give some indications of interesting statistical values to analyse.

This chapter focuses on, using the results of the previous chapter, creating a fitness function to analyse different generated games, allowing a evolutionary algorithm to evolve well-designed games, by creating and evolving VGDL game descriptions. The resulting generated games are then discussed in detail.

5.1 Fitness function: Introduction

The tests of the previous chapter indicates that the relationship between general game playing controllers performance could potentially be used to determine the quality of games. To make it possible to use the performance relationships, while not overfitting the parameters to the small set of "good" data points (13 human-designed games), it is necessary to find the most important and selective statistical values from playthrough results for use in a fitness function, while trying to exclude statistics that do not signify game quality.

As mentioned in Section 4.4 the score and win-rate seems to have the most distinct distributions for the different type of games, with human-designed games all having a significant difference between e.g. the Explorer- and Random controller, making them interesting to use for an evaluation function. The *action entropy* might also be appropriate to use, but the difference in the statistical distributions are much smaller for these values.

Limiting fitness features to score and win-rate, there is still an enormous amount of possible ways to calculate features-values for the fitness function. For instance, I could have a feature for the difference in average score between every single controller but it would result in $\binom{6}{2} = 15$ features for a single statistic. Also, I could use a series of different values simply to compare the score: *minimum*- and *maximum* score, *standard deviation* of score, *median*, *quartiles* or any form of normalised score, and of course I could use combined values for the features, e.g. the score increase per clock tick.

Additionally it is not straightforward to calculate a feature even after deciding the controllers and value to compare (e.g. score or clock ticks between Explorer and Random). I could use a simple relative difference formula (e.g. $f = |\frac{a-b}{max(a,b)}|$), and use the result as a feature, but it might make more sense to hand-craft each feature to a specific goal, for instance for the action entropy it seems (from the data of the last chapter) designed games cause the intelligent controllers to have a similar value as the Random controller, while the same is not true for the randomly generated games – in the generated games the intelligent controllers often find a simple solution requiring similar actions. A feature could therefore be made to have a value of 1 if the entropies are the close (and going toward 0/-1 when random has a higher entropy), while not rewarding the game for having the intelligent controllers action entropy be higher than Random (which the data do not show as a relationship in designed games).

Also, how each fitness feature adds to the complete fitness function is not straightforward. One could use a simple sum of each chosen feature, maybe with a weight constant attached signifying how much each feature adds the overall value ($f = w_1a_1 + ... + w_na_n$), but the approach can have the outcome of some features overruling others.

In the following I will try to shed light on which features are the most interesting, by both analysing the data in more detail, and by evolving games using the features.

5.2 Performance relationship analysis

Since the relationships between controllers acting intelligent, and acting random or doing nothing, seems to be a possible indication of game quality, an analysis of the relationship between different controllers using varying values was performed. As indicated in the previous section, choosing how to evaluate game for the fitness function almost certainly have an element of subjectivity. To make an as informed decision as possible I ran a series of test described below, using different features, evolving weights for the features, and examine the results.

In the following I repeatedly to use a *relative difference*-function ($f = |\frac{a-b}{max(a,b)}|$) to compare statistical values between controllers performance results, retrieved from playthroughs. Additionally I chose to simply construct the fitness function as a sum of the individual features.

Also, the assumption that the games from the generated- and mutated set simply are of a low quality (discussed in Section 4.4), is used throughout the following analysis.

Analysis of possible relationship features

As mentioned in the previous sections, the relationship of score and win-rate between controllers seems to be obvious choices to adopt as fitness features. Also, the there seems to be a consistent difference in the values between the intelligent controllers (especially Explorer and DeepSearch) and the less intelligent ones: The OneStep, Random and Do Nothing-controllers.

To measure the strength of different possible choices I performed a small test, counting how often a specific intelligent controller has a higher value than a simple one, and calculating the average relative difference between the pair. The results can be seen in Figure 5.1.

Some of the features in the table immediately seems interesting and significant, for instance the score average- and win-rate difference between the intelligent- and the DoNothing controller is consistently higher for the intelligent controllers, while the same is not true for several of the games from the mutated- and randomly generated set. The average clock ticks between, for instance Explorer and OneStep additionally could be interesting to use as an indication of a bad game, since the value is never higher for OneStep in the generated game set. However the actual relative difference between the controllers is tiny, and so the relationship might not be that significant. Also the designed games do not have a clear relationship profile for the clock ticks (as discusses in Section 4.4).

Analysis of feature sets

From the analysis of features, a selection of interesting values are available, and of these I continuously tested the results of selecting different sets of features (between 2-20) of relative difference values between pairs of controllers.

From the varying sets of feature selections, I first of all calculated and compared the resulting fitness (by summing the features) of games from both the designed- and generated set, noticing how each game was scored – while examining the data from high-fitness generated-, and low-fitness designed games further.

I continuously tried evolving a set of weights for individual features, to balance the contribution from each type of values using a CMA-ES algorithm[1]. For this task another problem arise however: Creating a fitness function to score a chosen set of weights. Several different approaches were tried for this task, but all relying on the assumption that all the games from the randomly generated set of Chapter 5 were of a low quality. Using the assumption, the fitness was increased if the set of weights caused the overall fitness of the designed games to be larger than for the randomly generated set (from Chapter 4).

Because the amount of games assumed to be of a good quality were rather small (13 human-designed games) this approach was discarded, in fear of overfitting the fitness function to the data points.

[1] https://www.lri.fr/ hansen/cmaes_inmatlab.html

data type	Designed		Mutated		Rnd. gen.	
	count	*diff.*	*count*	*diff.*	*count*	*diff.*
Mean score:Explorer>Onestep-S	13 (of 13)	0.717	78 (of 90)	0.494	43 (of 66)	0.240
Mean score:Explorer>Random	13 (of 13)	0.848	81 (of 90)	0.651	43 (of 66)	0.334
Mean score:Explorer>Do Nothing	13 (of 13)	1.000	82 (of 90)	0.823	45 (of 66)	0.404
Mean score:DeepSearch>Onestep-S	13 (of 13)	0.736	73 (of 90)	0.474	35 (of 66)	0.214
Mean score:DeepSearch>Random	13 (of 13)	0.864	81 (of 90)	0.659	41 (of 66)	0.307
Mean score:DeepSearch>Do Nothing	13 (of 13)	1.000	83 (of 90)	0.821	42 (of 66)	0.375
Winrate:Explorer>Onestep-S	11 (of 13)	0.675	36 (of 90)	0.282	4 (of 66)	0.035
Winrate:Explorer>Random	11 (of 13)	0.756	39 (of 90)	0.327	5 (of 66)	0.033
Winrate:Explorer>Do Nothing	11 (of 13)	0.846	36 (of 90)	0.349	6 (of 66)	0.086
Winrate:DeepSearch>Onestep-S	12 (of 13)	0.762	35 (of 90)	0.299	3 (of 66)	0.029
Winrate:DeepSearch>Random	12 (of 13)	0.846	39 (of 90)	0.350	5 (of 66)	0.026
Winrate:DeepSearch>Do Nothing	12 (of 13)	0.923	35 (of 90)	0.345	6 (of 66)	0.081
Mean tick:Explorer>Onestep-S	8 (of 13)	0.197	39 (of 90)	0.026	0 (of 66)	-0.076
Mean tick:Explorer>Random	9 (of 13)	0.439	56 (of 90)	0.280	1 (of 66)	-0.084
Mean tick:Explorer>Do Nothing	7 (of 13)	0.082	34 (of 90)	0.015	2 (of 66)	-0.072
Mean tick:DeepSearch>Onestep-S	8 (of 13)	0.251	42 (of 90)	0.098	1 (of 66)	-0.083
Mean tick:DeepSearch>Random	10 (of 13)	0.504	59 (of 90)	0.328	2 (of 66)	-0.086
Mean tick:DeepSearch>Do Nothing	6 (of 13)	0.160	35 (of 90)	0.082	2 (of 66)	-0.075

Figure 5.1: Simple analysis of possible fitness values

Outcome

From the above analyses I decided to only use two different relative difference values from performance profiles of the controllers, to use in the fitness function, both having a unique form in the designed set of games, while not appearing similarly in the generated set of games.

The chosen features were:

A function comparing score means between DeepSearch and DoNothing. Returning the relative difference between the two values.

A features comparing win-rates between DeepSearch and DoNothing. Returning the relative difference between the two values.

Where the total fitness of a game was given by the sum of the two values. Using these features, 12 of the designed games has a "perfect score" of 1, while the game DigDug only score 0.5 (since the DeepSearch controller was never able to win). 30 of the mutated games has a score of 0.95 or higher (23 has perfect score), while only 4 of the generated games has a fitness above 0.5 (however these are all perfect scores of 1).

I tested many of the games from the mutated set of games, but they can in general be described as a twist on the original games, but with some game-breaking element (for instance the avatar suddenly becomes able to walk through wall, dirt or water), and because of the sheer amount of games with high fitness they become difficult to analyse thoroughly (too see if any of them actually contains interesting game design).

The four randomly generated games additionally were not much fun to test, and in general they contained similar problems as those discussed at the end of the last chapter (Section 4.4): They were too trivial, and the winning solutions caused the game to end too quickly. Of the games,

`gen_game44` probably contains the most interesting game design (see Figure 5.2). The player avatar can only move left and right, but also place ground sprites. There is a "door" sprite falling down slowly from above, and the goal is simply to place a ground sprite below the door. The crate- and gem sprites has no effect in the game.

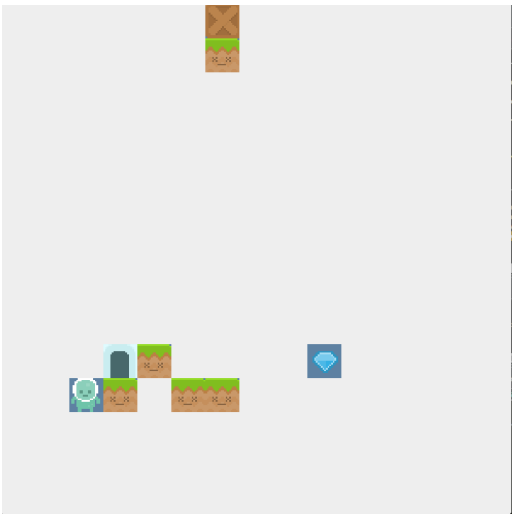

Figure 5.2: Screenshot of a generated VGDL game (`gen_game44`), where the DeepSearch algorithm plays consistently better than DoNothing

5.3 Further selection of features

After examining the resulting fitness values from generated VGDL games, and increasingly starting to evolve games using the fitness function (described in the next section), it became clear that I would need to introduce more fitness features, especially using non-relative values – i.e. instead of comparing controllers' results, using for instance averaged results across the controllers or only use results from a single controller.

The first insight was (also discussed in Chapter 4) that games where the intelligent controllers can win too quick – I decided on 50 clock ticks – should have a reduced fitness. A correlated problem is that the intelligent controllers often has a low *action entropy* in these games, but this is seemingly directly related to the fact that they can win with only a few precise actions.

The second insight was that the the intelligent controllers is often able to both win AND lose in many of the designed games (controllers standard deviation of win-rate above zero), while this happens rarely in the generated games. Although this is not true for all of the designed games, it is able to give an indication of the difficulty of a game which seems to otherwise be missing.

As a result, the fitness function used to evolve games had the following form:

$$fitness = \frac{relDiff(\text{average score}) + relDiff(\text{average wins}) + [\text{can win in 50 ticks}] + [\text{can win and lose}]}{4}$$

Where [can win in 50 ticks] returns -1 if a controller is able to win in less than 50 clock ticks, and otherwise gives 1, and [can win and lose] similarly returns either a 1 or -1 if the standard deviation of win-rate is above 0 (1 if true). The resulting fitness is a numerical value between -1 and 1.

In addition a game is automatically given a fitness of -1, if it does not fulfil the criteria used in accepting games in Chapter 4: That the controllers are not disqualified, that the score and wins is not always the same, and that the controllers do not all finish in less than 50 clock ticks.

5.4 Evolution of games: Setup

A process for generating and evolving new VGDL games were built using the fitness function found in the previous section. Two different approaches were again used to create new game description:

1) Evolving from an existing, human-designed game description, using human-designed levels, and 2) evolving from randomly generated games, using randomly generated levels.

In both approaches a simple evolution strategy (ES) approach, using mutation- and crossover operations across several generations, in order to increase the fitness of a population (of games). For each generation a population of game descriptions are tested using the two controllers DeepSearch and DoNothing, and a fitness is calculated for each game, with the lowest fitness games being removed from the population. As in the previous chapter, only the interaction rules (of the `InteractionSet`) were evolved over each generation.

Because of CPU scheduling issues[2] the populations size was limited to 50, with 25 members always surviving each generation, with a limit of 15 generations. Additionally the maximum amount of clock ticks for playthroughs was limited to 800.

The evolution process was stopped when the highest fitness stopped increasing over 10 generation, when the best games fitness were above 0.98 (an almost perfect score), or when the 20 generations allowed were over.

Evolving randomly generated games

An additional process was used to evolve games from a randomly generated game (using the generation process of Section 4.1), which also requires a level generated for the description, to allow playing it. Since the evolution process itself is only focused on changing the interaction rules, it should be certain that the `SpriteSet`, `TerminationSet` and the level description (and `LevelMapping` are well-formed, so the evolution strategy is actually able to find games of high fitness.

Therefore a process was built to first continuously generate a new game- and level descriptions, playing through them with the controllers, and then checking for 1) that the fitness is above -1 (which simply means that the minimum criteria for a game has been fulfilled), and 2) that intelligent controller was actually able to win. From the results of Section 4.3 I know that DeepSearch has a average win chance of 11% in the set of randomly generated games, however, where 334 games has already been removed for not fulfilling the minimum criteria, and so we can expect that at least $\frac{66}{400} \cdot 11\% = 1.8\%$, or every 55th game description, will be fulfil the criteria and be selected to evolve from.

As mentioned in the discussion of the results of Chapter 4 the previous randomly generated games had a problem of sprites leaving the playing field. To counter this an interaction rule was made for each sprite in the generated descriptions, making sprites take a step back if trying to move out of the level.

5.5 Evolution of games: Results

Because of the ability of the evolution strategy to return early if the fitness increased over 0.98, a large amount of game description were generated, especially when generating game by mutating the game Boulderdash, were a "perfect" fitness games (fitness of 1) were often found in the first generation – which however caused them to only have a small number of differences from the original game description of Balderdash. A series of 54 games were generated and evolved over 4 days, 45 by evolving from the human-designed game Boulderdash, and 9 by evolving games from a randomly generated game- and level description. Of these games 40 mutated-, and 6 randomly generated games had a perfect, or near-perfect fitness of >0.98, making them the most interesting to examine further.

Playing through an array of the Boulderdash mutations I noticed that the games still had many of the problems first discussed in Chapter 4 when examining mutated games: The mutation often cause certain important game rules of the originals to be broken – suddenly the player can walk through walls- or boulders, gems does not disappear after they have increased the players score, or enemies can no longer harm the player. With that being said, some of the games certainly contain some interesting aspects. In `boulderdash_mut09` (see Figure 5.3 for changes in the games' rules),

[2]The problem is mostly that it just takes a lot of time for intelligent controllers to play through each game. Each controller is allowed up to 50 ms per tick, and so even with a maximum of 800 ticks, playing through a game 6 times can take up to $50ms \cdot 800 \cdot 6 = 240s$. With a population size of 50 over 15 generation, a single game can in the worst case take up to 50 hours to evolve.

which is also one of the better games[3], the avatar has gained an ability to push boulders making them glide across the level with the ability to kill certain enemies. However, the enemies that can be killed by the boulders no longer harm the avatar, and so the new ability does not cause much change to the actual game.

```
1  InteractionSet
2    dirt avatar > killSprite
3    dirt sword  > killSprite
4    diamond avatar > collectResource
5    diamond avatar > killSprite scoreChange=2
6    moving wall > stepBack
7    moving boulder > stepBack
8    avatar boulder > killIfFromAbove scoreChange=-1
9    avatar butterfly > killSprite scoreChange=-1
10   avatar crab > killSprite scoreChange=-1
11   boulder dirt > stepBack
12   boulder wall > stepBack
13   boulder diamond > stepBack
14   boulder boulder > stepBack
15   enemy dirt > stepBack
16   enemy diamond > stepBack
17   crab butterfly > killSprite
18   butterfly crab > transformTo stype=diamond scoreChange=1
19   exitdoor avatar > killIfOtherHasMore resource=diamond limit=9
20
21   -->
22
23 InteractionSet
24   dirt avatar > killSprite
25   dirt sword > killSprite
26   boulder avatar > attractGaze
27   dirt diamond > killIfFromAbove
28   moving wall > stepBack
29   moving boulder > stepBack
30   avatar boulder > killIfFromAbove scoreChange=-1
31   butterfly dirt > killIfHasMore limit=15 resource=diamond
32   avatar crab > killSprite scoreChange=-1
33   boulder dirt > stepBack
34   butterfly avatar > killIfFromAbove
35   sword crab > cloneSprite
36   boulder boulder > stepBack
37   avatar EOS > killIfHasLess limit=13 resource=diamond
38   diamond dirt > transformTo stype=diamond
39   crab butterfly > killSprite
40   diamond avatar > collectResource
41   exitdoor avatar > killIfOtherHasMore limit=9 resource=diamond
42   butterfly boulder > killSprite
```

Figure 5.3: The original- and evolved set of interaction-rules of a VGDL description, generated by mutating the description of Boulderdash.

I played through all of the 6 games of "perfect" fitness evolved from randomly generated descriptions, which in general has a more unique design that the Boulderdash-evolved descriptions. The outcome of several of the games are partly- or completely random, no matter what the player does the results are the same: This is likely a product of using a rather small statistical sample when playing through each game (6 playthroughs for each controller). The games are overall quite simple, but several of them contain some interesting game design for human players – some which seemingly has appeared by accident:

In the evolved game `evol_game01` (see Figure 5.4 (description) and Figure 5.5 (screenshot)) the goal is to kill a block of dirt that whizzes around the level, trying to flee from the avatar. However, true to the action-arcade genre of the game, the player can additionally increase his/her score by moving the avatar to a laser-sprite (`gen2` in the description). Only one laser-sprite can exist in the level – because of its parameter `singleton=TRUE`, and the player can only increase his/her score

[3]Gems do no disappear when collected, making the player able to win very quickly (one just has to stand still on gem for a second, and then head for the exit), but at the same time enemies can walk through ground making the game has some sense of challenge and difficulty – compared to the other generated games.

by repeatedly going to each laser that is spawned. The game `evol_game02` is rather simple, in that the goal is simply to kill the avatar, which can be achieved by walking to the "cog" sprite (at the top of the screen in Figure 5.6. However the game is lost if the bat sprite (a `RandomNPC`) ever collides with the same object, making the outcome rather random.

```
BasicGame
  SpriteSet
    avatar > MovingAvatar img=avatar
    gen1 > Passive img=honey
    gen2 > Resource limit=4 singleton=TRUE img=missile value=2
    gen3 > Immovable img=pellet
    gen4 > Bomber orientation=DOWN stype=gen5 img=frog prob
        =0.22090000000000004
    gen5 > Spreader limit=33 stype=gen2 img=fire
    gen6 > Fleeing stype=avatar img=dirt
  InteractionSet
    gen6 gen4 > attractGaze
    avatar EOS > killSprite
    gen5 EOS > wrapAround
    gen1 gen3 > killIfHasLess limit=3 resource=gen2
    gen2 avatar > transformTo stype=gen1 scoreChange=2
    gen2 avatar > wallStop
    gen2 avatar > changeResource resource=gen2 value=0 scoreChange=7
    gen6 gen2 > killIfOtherHasMore limit=0 resource=gen2
    avatar EOS > stepBack
    gen1 EOS > stepBack
    gen3 gen3 > stepBack
    gen3 EOS > stepBack
    gen1 gen1 > spawnIfHasMore limit=11 stype=gen6 resource=gen2 scoreChange
        =-2
    gen5 EOS > stepBack
    gen6 EOS > stepBack
    gen2 gen5 > collectResource
  LevelMapping
    $ > gen1
    % > gen2
    & > gen3
    ' > gen4
    ( > gen5
    ) > gen6
  TerminationSet
    SpriteCounter limit=0 stype=gen6 win=TRUE
    SpriteCounter limit=0 stype=avatar win=FALSE
```

Figure 5.4: The VGDL game description for the evolved game `evol_game01`

5.6 Discussion of results

Many of the games generated by in this chapter definitely have some interesting properties and features, but overall the VGDL game generation process was not able to create any complete games of a high enough quality to compare with other games of the genre. Additionally the generator was only able to find games with interesting qualities, and did not guarantee that each generated games were any good. The generator was simply able to find sets of games in the space of VGDL game descriptions with a significantly higher chance of quality, than for completely randomly generated games.

To be able to increase the quality of the game generator one would almost definitely need to refine- and extend the fitness function, which could identify many more aspects of games- and playthroughs of games. For instance, one could examine how each controller increased its score across a playthrough, th proportion of sprites- and rules that have been in use in the game, the amount of sprites that has left the level field, or the amount of objects that have been killed or created. I have implemented functions for retrieving several more value (like these) in the GVG-AI framework, but initial examination of the values did not show any clear form or profile, for the

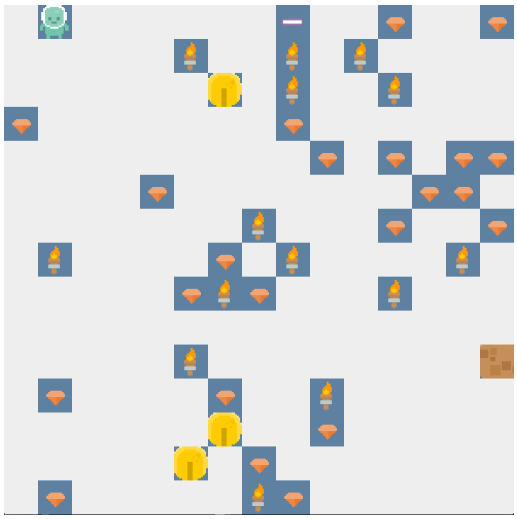

Figure 5.5: Screenshot of a evolved-from-random-generated VGDL game (`evol_game01`) with a fitness score of 1

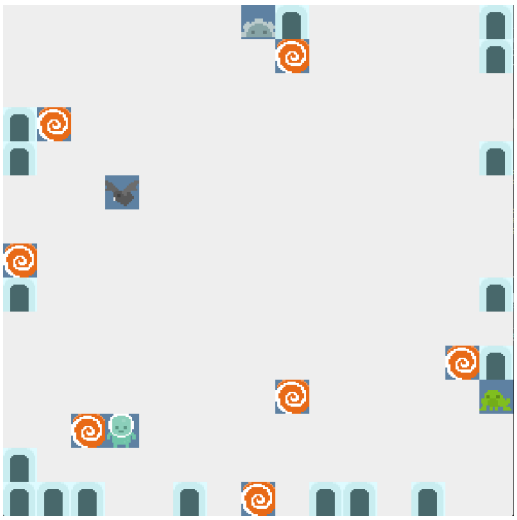

Figure 5.6: Screenshot of a evolved-from-random-generated VGDL game (`evol_game02`) with a fitness score of ≈0.98

human-designed set of games[4].

[4] https://docs.google.com/spreadsheets/d/1HAuIdXBcv4MXTlerkSqz8tK-zXMypzzQAU8sRlt95TA/edit?usp=sharing

Chapter 6

Puzzle generation

In this chapter I will explore the prospect of automatically generating VGDL puzzle games- and levels. First an array of human-designed puzzle games will be examined using a general puzzle solving algorithm. Then, using the results of the algorithm, a general level generator for puzzle games will be introduced, and generated levels for designed games presented. At the end, a process for both generating a (puzzle) game description and evolving a level, will be shown and the results of the generator discussed.

6.1 Puzzle analysis introduction

To analyse and generate puzzle games more quickly, my implementation of VGDL: SimpleVGDL (Section 3.3), was used to test- and generate levels, instead of the GVG-AI framework.

Games

From the GVG-AI framework (Section 2.3) and my own additions (Section 3.1), there is a total of 10 puzzle games to use in the following analysis.

However, several of the games were deemed unfit for a complete examination. Some, because the levels were too difficult for the a breadth first search algorithm to have a chance (Bolo Adventures and Chip's Challenge). Three other games were removed from a more technical reason: The games Bombuzal and ZenPuzzle all has a continuous creation of `Flicker`-sprites, which (as mentioned in Section 3.2) halts the PuzzleSolver in its progress. Because of this, only the following five games were used as a base-line for "good" puzzle games: Bait, Brainman, The Citadel, Soko-Ban and Modality.

Finding the shortest solution

The general puzzle solving controller introduced in Section 3.2 was used in analysing all VGDL games described in this chapter, which, as mentioned, is able to find the shortest solution in a given VGDL *puzzle game* However as indicated from Section 3.3 the controller can easily meet games which game-spaces are too large to fit in memory, to be able to find the solution using the algorithm.

Solving designed games

To analyse the six designed puzzle games mentioned above, the puzzle solving controller was set to play through all of the levels of the games, finding both the fastest possible solution, and all other possible routes. In the following tests, the PuzzleSolver was allowed an hour of computation time to solve each game level.

Figure 6.1 show the results of playing through the games.

As can be seen, many of the levels were never completed, because the solutions were too complex and the PuzzleSolver ran out of time.

game	*lvl*	*found sol.*	*sol.length*	*time*
Bait	0	true	9	500ms
Bait	1	true	38	2440ms
Bait	2	true	71	612040ms
Bait	3	false	-	-
Bait	4	true	35	753400ms
Brainman	0	true	30	20800ms
Brainman	1	true	32	221400ms
Brainman	2	false	-	-
Brainman	3	false	-	-
Brainman	4	true	32	9600ms
Modality	0	true	6	240ms
Modality	1	true	33	1440ms
Modality	2	true	66	2920ms
Modality	3	true	158	16640ms
Modality	4	true	27	1080ms
Sokoban	0	true	25	106920ms
Sokoban	1	true	107	28960ms
Sokoban	2	false	-	-
Sokoban	3	true	97	6080ms
Sokoban	4	true	30	1280ms
The Citadel	0	true	15	3920ms
The Citadel	1	false	-	-
The Citadel	2	false	-	-
The Citadel	3	false	-	-
The Citadel	4	false	-	-

Figure 6.1: Results from the five human-designed puzzle games examined

6.2 General puzzle level generation

Since the PuzzleSolver controller is able to complete an arbitrary level of a puzzle game, as long as the solution(s) are not too complex, it becomes interesting to use it to solve generated levels for games. By using an approach similar to that used to generate general level descriptions, for randomly generated games in Section 4.1.2, it became clear that the PuzzleSolver could be used to not only find out if randomly generated levels are any good, but use its results to evolve new and better levels.

A setup for generating levels for a given game were constructed using an evolutionary strategy, utilising both mutation- and crossover operations on a population of levels. The overall goal for the generator was to create levels having a non-trivial and difficult solution.

The input to the generator was the preferred size of the level, but also a more subjective element of choosing the textual character identifying "wall" and "ground"-sprites which some of the puzzle-games utilises[1], to make generated levels for those games behave as intended.

[1] In (Real) Soko-Ban every "empty" tile contains a ground sprite (signified by a dot "."), while most other sprites

Importantly, the fitness function for the evolution process was implemented by a simple comparisons of a series of results of PuzzleSolver playing the levels, by first sorting with the first value, the by the second, and so on. The values used to compare results were, rather subjectively, chosen to be:

1. The number of actions required for the shortest solution. This element is focused to fulfil the goal of generating as difficult levels as possible.
2. The amount of times a non-player sprite has been moved throughout the solution.
3. The number of times sprites has interacted with another throughout the solution.
4. The time used in finding the solution.

For the following tests of the generator, the evolutions strategy was run with a population of 100 level, with 50 subjects surviving every generation, evolved over a maximum of 100 iterations.

6.3 Generating levels for designed games

The general puzzle level generator was set to create levels for the five designed puzzle games analysed above. Because space of game states quickly become too large for the PuzzleSolver, all levels were generated using a level size of 8x8, with walls on all edges – making the actual levels consist of only 6x6 tiles.

Even with the tight restriction, the generator was able to create several levels of high difficulty for the games. Figure 6.2 show how a generated level for the game Bait increases in difficulty over generations.

```
1  1.              30.             100.
2  wwwwwwww        wwwwwwww        wwwwwwww
3  w  w w w        wG  wK w        w   wK w
4  www 1  w        w   w  w        w G ww0w
5  w w G  w >>     w 1  w w >>     w 1  wMw
6  w w wK w >>     ww   w  >>     ww0  wMw
7  w   0ww w        w A  0 w        w A    w
8  wwwwA Mw        w1   w  w        w   0  w
9  wwwwwwww        wwwwwwww        wwwwwwww
```

Figure 6.2: The evolution of a randomly generated level for the game Bait. The three levels are the best in the population at generation 1 (requires 7 actions), generation 30 (requires 35 actions and generation 100 (requires 57 actions), respectively.

Figure 6.3 shows a few generated levels for four of the games, retrieved from the best (i.e. longest solution) level in each evolution process.

6.4 Puzzle game generation

After showing that the general level generator is able to create levels of a certain difficulty, it becomes interesting to see if it can be used in combination with a game description generator (as introduced in Chapter 4), to create both a set of game rules- and the levels, for a game.

As mentioned in Section 2.3.2 puzzle games allow a smaller set of sprite classes, and interaction-rules defined in their VGDL description[2], making the overall space of possible games smaller.

However, even with the smaller space of games, it is not at all certain that a randomly generated game description would contain a interesting combination of rules and sprites, making it an viable subject for generating a difficult level.

(e.g. boxes, the avatar) are spawned with ground sprite below them to make it possible to detect when a box is pushed of its target – for other puzzle games the ground sprite was simply set to " " (empty space)

[2]Sprites like `RandomNPC`, and interaction-rules like `reverseDirection` (changing direction is only interesting for constant moving sprites like `Missiles`) can not be used in the definition of puzzle games used.

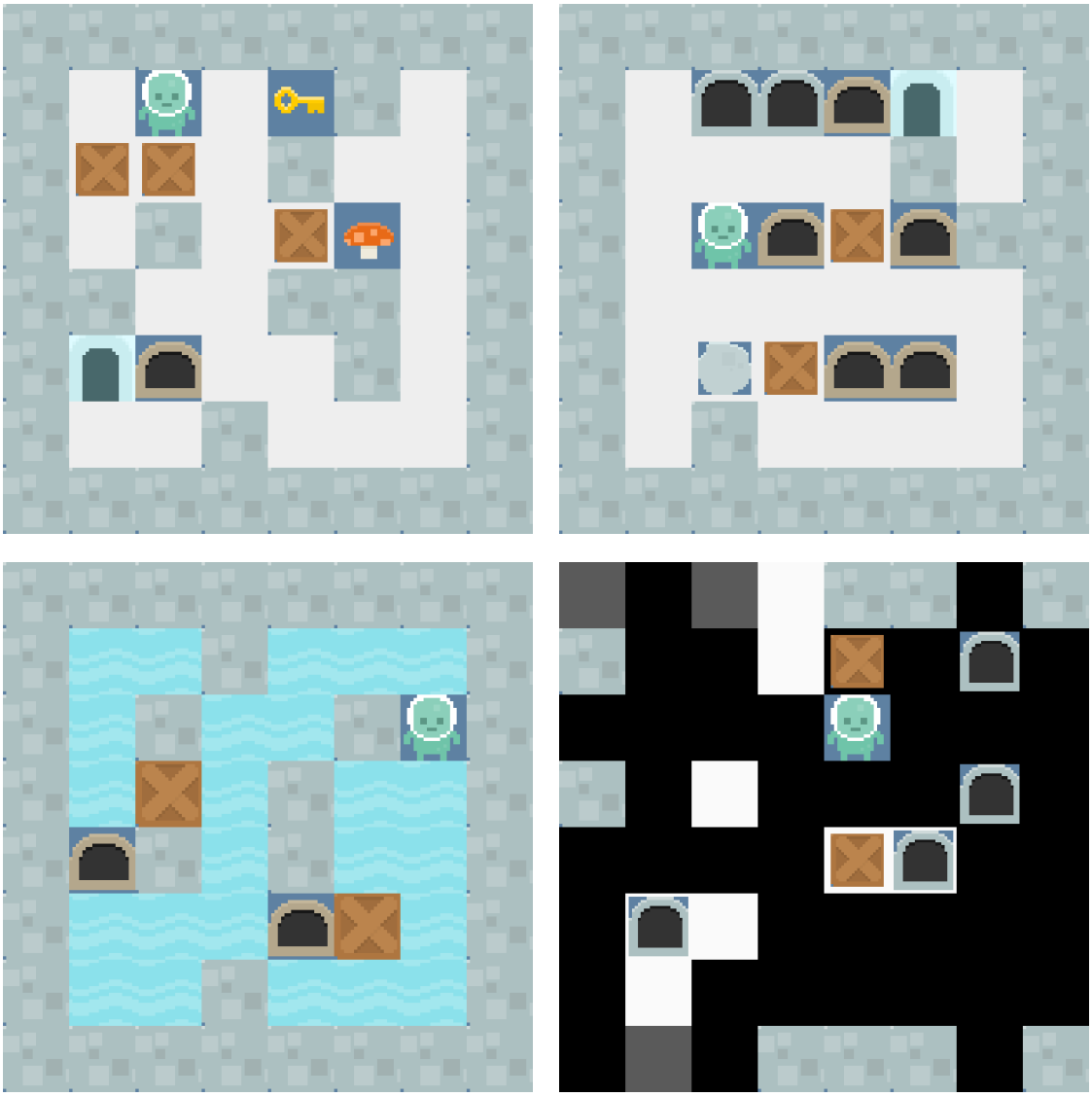

Figure 6.3: Generated levels for some of the designed puzzle games. From top left: *Bait* (shortest solution is in 65 actions), *The Citadel* (77 actions), *(Real) Soko-Ban* (60 actions) and *Modality* (88 actions).

6.4.1 Experimental setup

To increase the chance of retrieving well-formed game description I decided to implement a process to first check the if a game description at all could be interesting, and only after starting to evolve a level. The process worked by:

1. Generate a simple game description, using the small set of allowed classes, with a small number of sprites (2-3 + the avatar)
2. Generate 50 randomly generated levels for the game, and let the PuzzleSolver play through them all.
3. If the most difficult level from step 2 requires at least 2 actions AND 1 non-avatar sprite has been moved in the solution, go to next step, otherwise generate a new game description (step 1).
4. Start evolving a level for the game

6.4.2 Results

Over a period of about 10 days, the generator was able to generate a long series of VGDL puzzle game description, each with a 8x8 level to play through the game with. However of these, most had the solution to the levels be rather short, with many games requiring less than 20 actions to complete. Of the more difficult game-level pairs, two requiring 50 or more actions for the shortest solution, and eight pairs requiring more than 35 moves. The Figure 6.4 shows the levels of the two of the more interesting game-level pairs (`evol_puzzle015` and *evol_puzzle009*), and Figure 6.5 the game description of the most difficult.

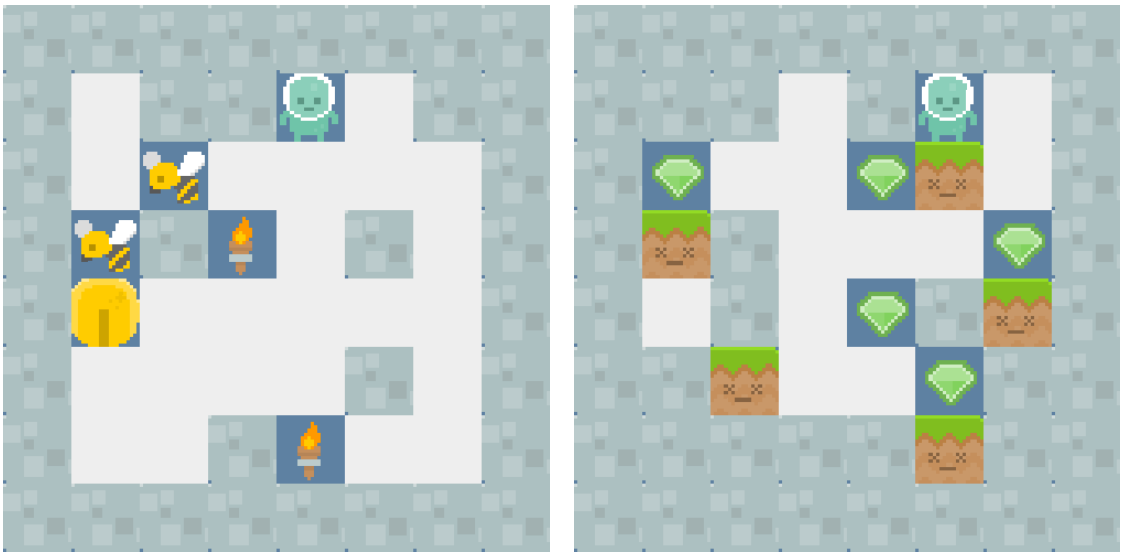

Figure 6.4: Visual representation of the two generated puzzle game-level pair, `evol_puzzle015` requiring 85 actions (left) and `evol_puzzle009` requiring 24 (right) actions for the shortest solution.

```
BasicGame
  SpriteSet
    avatar > OrientedAvatar img=avatar
    gen1 > Passive img=bee
    gen2 > Portal stype=gen1 img=fire
    gen3 > Immovable img=honey
  InteractionSet
    gen3 gen3 > stepBack
    gen1 gen2 > killSprite
    gen3 gen2 > undoAll scoreChange=6
    gen2 EOS > stepBack
    gen1 avatar > bounceForward
    gen2 gen1 > bounceForward scoreChange=-1
    avatar wall > stepBack
    gen1 wall > stepBack
    gen2 wall > stepBack
    gen3 wall > stepBack
  LevelMapping
    $ > gen1
    % > gen2
    & > gen3
  TerminationSet
    MultiSpriteCounter limit=0 stype1=gen1 stype2=null win=TRUE
```

Figure 6.5: The game description of the generated puzzle game `evol_puzzle015`

From the set of the more difficult pairs of game-levels, several non-trivial game-designs has emerged. For instance, in the most difficult game (`evol_puzzle015`) the goal is to destroy all the bees, which can by achieved by pushing them into the fire sprites. However the fire sprites are

pushed back when bees are pushed into them, and so bees can only be destroyed if the fire is up against a wall. Additionally the player has to think of when to kill the bees, since they are what allows the player to interact with the level and push fire sprites to fitting locations. Several of the games are otherwise quite similar to *Soko-Ban*, but with different twists. For instance, the generator has features like: Boxes (i.e. the object(s) that can be pushed in the game) can be destroyed by the player if pushing them into a wall, boxes that can be destroyed by pushing them into each other, or a game where new boxes are created when pushed to their goal (`evol_puzzle009`).

The generated games all have a strong similarity to Soko-Ban however, which is likely because of the requirement of the solution to require moving non-avatar sprites, and the games could be in general be described as Soko-Ban- clones or imitations.

6.5 Discussion of results

The fitness function used to evolve levels were chosen rather subjectively, and could almost definitely be improved, to be able to generate levels of a higher quality. It might be that better levels could be generated if the amount of clock ticks were the most focused, but because of time schedule issues I did not have time to explore the function further.

The generator is able to generate levels for most designed VGDL games, and certain generated games, restricted to a subset of classes, that – at least – have non-trivial solutions which are not immediately obvious, even with the tight restrictions of the level size (6x6 playing field). The drawback of the generator is that it is rather slow at generating even small levels, and has severe problems with larger level- and game space sizes. Also, it is not guaranteed to find the most interesting levels for the games – human level designers would almost definitely be able to create harder and more enjoyable levels.

Probably the most interesting aspect of the level generator is however the process of generating both a complete game- and level description. As mentioned, the tight restrictions of my definition of VGDL puzzle games, and the process the evolution strategy use (requiring that a sprite has been moved), causes the actual games- and levels generated to have strong similarities. Several of puzzle games generated definitely contain interesting features, but if any of the games can be considered of a high quality enough to be enjoyable for human players is a question left unanswered.

Chapter 7

Conclusion

7.1 My work

The goal of this project was to create a process for creating games and levels in the game description language VGDL of a "as-high-possible" quality. To reach this goal, two different approaches was attempted : 1) Generating general game description – action-arcade games, using all the possible classes and functions of VGDL. And 2) Generating game descriptions restricted to a subset of VGDL classes – puzzle games.

The main focus was on creating the descriptions of action-arcade games using the performance of general game playing algorithms to assess the games' quality. Using an evolutionary strategy a set of playable (action-arcade) VGDL games was generated, some using an existing human-designed game (Boulderdash) as basis, and some using randomly generated games- and levels.

The process was able to generate several set of interesting game-rules, but at the same time several of the games created by the generator hardly contained any game-play, and in many games the outcome was largely based on luck. In general the generator is not able to instantly find the most interesting games in the space of VGDL games, but it is able to find a subset of more interesting game description, which human game designers can examine further to decide which games are actually "good". A problem in the process of evolving the action-arcade games used in this project, is that intelligent AI controllers are used which spend a lot of time in playing the through each game – just playing through a single game 6 times takes the DeepSearch controller up to 2000 ticks * 40 ms * 10 = 800 seconds. At the same time using these, rather limited, parameters constraint the results the controllers can potentially retrieve. A more clever approach might be to first probe the games with some more simple controllers, or decreasing the allowed time for intelligent controllers, allowing the games to be played through more quickly, and decide if the game is worth spending time on for a more precise analysis (using intelligent controllers).

7.2 Future work

Maybe one could make an algorithm to play a game in a more similar fashion as humans: First reading, and understanding the rules (in pac-man: eat all the cheese, in boulderdash: get to the exit after getting X gems). Then, using a learning-approach, playing through the game/level, not know much about possible interactions, but maybe assuming that colliding with moving sprites is dangerous (they could be enemies!). And then playing the game repeatedly to learn to play the game better.

It might be interesting to analyse games using a wider array of controllers, or analysing by letting the same controller (or controllers) play with a different amount of time allowed per clock-tick, or even using controllers designed to "play bad" (trying to die, decrease score) could be interesting

Also, it could be interesting to retrieve more extrinsic data from playing through a game (e.g. the proportion of interaction-rules that was triggered).

Appendices

Appendix A

GVG-AI example games

Below is a short summary of each of VGDL games from the GVG-AI framework, discussed in 2.3.2.

Aliens VGDL interpretation of the classic arcade game *Space Invaders*. A large amount of aliens are spawned from the top of the screen. The player wins by shooting all the approaching aliens.

Boulderdash VGDL interpretation of *Boulder Dash*. The avatar has to dig through a cave to collect diamonds while avoiding being smashed by falling rocks or killed by enemies.

Butterflies The avatar has to capture all butterflies before all the cocoons are opened. Cocoons open when a butterfly touches them.

Chase About chasing and killing fleeing goats. However, if a fleeing goat encounters the corpse of another, it get angry and start chasing the player instead.

Digdug VGDL interpretation of *Dig Dug*. Avatar collects gold coins and gems, digs his way through a cave and avoid or shoot boulders at enemies.

Eggomania VGDL interpretation of *Eggomania*. Avatar moves from left to right collecting eggs that fall from a chicken at the top of the screen, in order to use these eggs to shoot at the chicken, killing it.

Firecaster Goal is to reach the exit by burning wood that is on the way. Ammunition is required to set things on fire.

Firestorms Player must avoid flames from hell gates until he finds the exit of a maze.

Frogs VGDL interpretation of *Frogger*. Player is a frog that has to cross a road and a river, without getting killed.

Infection Objective is to infect all healthy animals. The player gets infected by touching a bug. Medics can cure infected animals.

Missile Command VGDL interpretation of the classic arcade game *Missile Command*. Player has to destroy falling missiles, before they reach their destinations. If the player can save at least one city, he wins.

Overload Player must get to the level after collecting coins, but cannot collect too many coins, as he will be too heavy to traverse the exit.

Pacman A VGDL interpretation of *Pac-Man*. Goal is to clear a maze full with power pills and pellets, and avoid or destroy ghosts.

Portals Objective is to get to a certain point using portals to go from one place to another, while at the same time avoiding lasers.

Seaquest VGDL interpretation of *Seaquest*. Avatar is a submarine that rescue divers and avoids sea animals that can kill it. The goal is simply to a high score.

Survive Zombies Player has to flee zombies until time runs out, and can collect honey to kill the zombies.

Whackamole VGDL implementation of the classic arcade game *Whac-a-Mole*. Must collect moles that appear from holes, and avoid a cat that mimics the moles.

Zelda VGDL interpretation of *Legend of Zelda*. Objective is to find a key in a maze and leave the level. Player also has a sword to defend himself against enemies.

Camelrace Player needs to get to endpoint to win.

Soko-Ban The objective is to move the boxes to holes, until all boxes disappear or the time runs out.

Appendix B

Games designed during thesis

This chapter described each of the VGDL games introduced in Section 3.1. The games are all interpretations of existing games designed on different platforms.

B.1 Action arcade games

Astrosmash [?] The player controls a laser cannon at the bottom of the screen, with the goal of shooting down as many incoming meteors, bombs and other objects. Points are earned by destroying objects, but lost if the objects reach the ground.

Centipede [?] A centipede is approaching from the top of the screen. The player controls an cannon at the bottom of the screen able to shoot the enemy, but hitting the centipede in everywhere but its ends causes it to split into two parts.

Crackpots [?] Spiders are crawling up from the bottom of the screen, and the player must push flowerpots, which appear again a short while after being pushed, to stop and kill them.

Solar Fox [?] The player controls a spaceship which is constantly moving, and only the direction can be controlled. The goal is to collect all of the coins in each level, while evading the missiles of the enemies on the top- and bottom of the screen.

B.2 Puzzle games

Bait [?] *Bait* is a game with similar mechanics to *Soko-Ban* but otherwise unique game play. In each level the player must push boxes to get across the playing field, to reach a button, and afterwards head for the exit,.

Bolo Adventures [?] Another game closely related to *Soko-Ban*, but with several unique features like lasers and rolling boulders. The levels of the game are in general extremely difficult.

Bombuzal [?] The goal is to blow up all the bombs, which are spread wide across each level. When a bomb is exploded it erodes nearby tiles, or even other bombs, forcing the player to find a path that allows him/her to reach all the bombs.

Brainman [?] Has similar characteristics as the other *Soko-Ban*-like games, but also has a focus on collecting gems to increase the players score.

Chip's Challenge [?] *Chip's Challenge* is again a *Soko-Ban*-like puzzle game, but with an large amount of features compared to the other games, like water where either boxes can be used to make a bride across, or the player can swim across if he has a specific item (a swimsuit).

The Citadel [?] An additional game closely related to *Soko-Ban*. This game features boxes which can be moved in a line, i.e. if the player pushed a box of this type, all boxes behind it will additionally be pushed.

(Real) Soko-Ban [?] The original push-boxes games.

Modality [?] *Soko-Ban*-like game with a twist: Each level is divided into white- and black tiles, and the avatar can only move between the two colours by going through the grey tiles on the playing field.

Zen Puzzle [?] The player must "paint" every tile in an area by moving the avatar across them. After a tile has been painted it becomes solid and impossible for the avatar to move through, making the player have to find a path through the area to reach all tiles.

Appendix C

Designed games results

This chapter shows the individual data for each game, for the test discussed in Section 4.3. Each game was played through ten times by each controller.

Aliens

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	82.50 (0.60)	0.97 (0.01)	1.00 (0.00)	511.90 (9.34)	0.68 (0.15)
DeepSearch	84.60 (0.82)	1.00 (0.00)	1.00 (0.00)	413.10 (7.19)	0.68 (0.15)
MCTS	69.60 (1.02)	0.82 (0.01)	1.00 (0.00)	609.70 (33.35)	0.68 (0.15)
Onestep-S	59.80 (1.56)	0.70 (0.02)	0.30 (0.14)	832.90 (34.24)	0.68 (0.15)
Random	35.80 (6.57)	0.42 (0.08)	0.10 (0.09)	441.30 (99.43)	0.68 (0.15)
Do Nothing	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	581.90 (56.51)	0.00 (0.00)

Astrosmash

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
DeepSearch	1306.50 (46.75)	1.00 (0.00)	1.00 (0.00)	1000.00 (0.00)	0.68 (0.15)
MCTS	487.50 (37.47)	0.37 (0.03)	1.00 (0.00)	1000.00 (0.00)	0.68 (0.15)
Onestep-S	31.60 (10.43)	0.03 (0.01)	0.80 (0.13)	932.60 (46.49)	0.68 (0.15)
Random	47.50 (13.02)	0.04 (0.01)	0.30 (0.14)	709.70 (99.00)	0.68 (0.15)
Do Nothing	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	380.90 (50.84)	0.00 (0.00)

Boulderdash

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	25.10 (2.74)	0.92 (0.06)	0.70 (0.14)	879.00 (160.96)	0.96 (0.06)
DeepSearch	12.30 (3.74)	0.45 (0.13)	0.20 (0.13)	531.70 (154.27)	0.99 (0.03)
MCTS	16.30 (2.22)	0.64 (0.10)	0.40 (0.15)	1200.30 (218.37)	1.00 (0.00)
Onestep-S	4.10 (0.97)	0.16 (0.04)	0.00 (0.00)	186.00 (55.39)	1.00 (0.01)
Random	1.20 (0.42)	0.05 (0.02)	0.00 (0.00)	63.50 (9.19)	1.00 (0.02)
Do Nothing	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	2000.00 (0.00)	0.00 (0.00)

Centipede

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	84.30 (3.59)	0.88 (0.05)	0.90 (0.09)	167.40 (14.82)	0.65 (0.15)
DeepSearch	92.70 (4.39)	0.95 (0.02)	1.00 (0.00)	173.60 (13.87)	0.66 (0.15)
MCTS	79.20 (3.30)	0.82 (0.04)	0.70 (0.14)	176.40 (15.07)	0.68 (0.15)
Onestep-S	59.90 (4.79)	0.63 (0.06)	0.30 (0.14)	200.20 (5.45)	0.68 (0.15)
Random	58.40 (4.82)	0.60 (0.04)	0.40 (0.15)	182.00 (9.99)	0.68 (0.15)
Do Nothing	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	188.00 (0.00)	0.00 (0.00)

Crackpots

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	476.00 (71.36)	0.20 (0.03)	0.00 (0.00)	242.40 (27.64)	0.68 (0.15)
DeepSearch	2426.00 (71.00)	0.99 (0.01)	1.00 (0.00)	1000.00 (0.00)	0.68 (0.15)
MCTS	2046.00 (121.95)	0.84 (0.05)	0.50 (0.16)	877.10 (56.70)	0.67 (0.15)
Onestep-S	104.00 (20.61)	0.04 (0.01)	0.00 (0.00)	80.80 (11.07)	0.68 (0.15)
Random	76.00 (19.73)	0.03 (0.01)	0.00 (0.00)	69.00 (8.02)	0.68 (0.15)
Do Nothing	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	45.50 (1.15)	0.00 (0.00)

Dig Dug

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	21.40 (2.06)	0.80 (0.08)	0.00 (0.00)	1752.90 (176.37)	0.98 (0.05)
DeepSearch	19.00 (1.11)	0.73 (0.07)	0.00 (0.00)	2000.00 (0.00)	0.94 (0.08)
MCTS	19.80 (3.49)	0.70 (0.10)	0.00 (0.00)	2000.00 (0.00)	1.00 (0.01)
Onestep-S	8.50 (2.96)	0.32 (0.09)	0.00 (0.00)	596.20 (174.14)	1.00 (0.02)
Random	2.60 (1.10)	0.09 (0.03)	0.00 (0.00)	272.30 (50.20)	1.00 (0.00)
Do Nothing	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	1104.00 (158.94)	0.00 (0.00)

Eggomania

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	23.00 (14.26)	0.21 (0.13)	0.20 (0.13)	263.80 (54.24)	0.57 (0.16)
DeepSearch	110.00 (2.08)	1.00 (0.00)	1.00 (0.00)	587.90 (118.36)	0.61 (0.15)
MCTS	3.50 (2.09)	0.03 (0.02)	0.00 (0.00)	343.10 (85.83)	0.68 (0.15)
Onestep-S	0.40 (0.16)	0.00 (0.00)	0.00 (0.00)	120.60 (16.39)	0.68 (0.15)
Random	0.20 (0.20)	0.00 (0.00)	0.00 (0.00)	107.00 (18.71)	0.68 (0.15)
Do Nothing	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	139.60 (26.71)	0.00 (0.00)

Frogs

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	0.90 (0.10)	0.95 (0.05)	0.90 (0.09)	193.30 (31.22)	0.85 (0.11)
DeepSearch	0.10 (0.10)	0.15 (0.11)	0.10 (0.09)	1821.50 (178.50)	0.73 (0.14)
MCTS	0.50 (0.17)	0.55 (0.16)	0.50 (0.16)	641.00 (136.13)	0.81 (0.12)
Onestep-S	0.00 (0.00)	0.05 (0.05)	0.00 (0.00)	95.90 (26.92)	0.61 (0.15)
Random	0.00 (0.00)	0.05 (0.05)	0.00 (0.00)	5.40 (1.85)	0.85 (0.11)
Do Nothing	0.00 (0.00)	0.05 (0.05)	0.00 (0.00)	2000.00 (0.00)	0.00 (0.00)

Missile Command

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	5.60 (0.40)	0.74 (0.06)	1.00 (0.00)	104.80 (5.33)	0.94 (0.08)
DeepSearch	7.40 (0.40)	0.96 (0.04)	1.00 (0.00)	68.00 (7.96)	0.99 (0.03)
MCTS	1.70 (0.47)	0.23 (0.06)	0.70 (0.14)	151.40 (13.72)	1.00 (0.00)
Onestep-S	1.60 (0.64)	0.20 (0.08)	0.50 (0.16)	160.30 (15.66)	1.00 (0.01)
Random	0.40 (0.27)	0.05 (0.03)	0.20 (0.13)	136.40 (18.27)	1.00 (0.01)
Do Nothing	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	110.00 (0.00)	0.00 (0.00)

Pac-Man

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	277.20 (8.57)	0.94 (0.03)	0.30 (0.14)	592.10 (95.18)	0.86 (0.11)
DeepSearch	290.80 (6.22)	0.98 (0.01)	0.40 (0.15)	1418.00 (196.47)	0.86 (0.11)
MCTS	222.40 (12.37)	0.75 (0.03)	0.00 (0.00)	2000.00 (0.00)	0.86 (0.11)
Onestep-S	143.10 (21.47)	0.48 (0.07)	0.00 (0.00)	1616.40 (223.47)	0.86 (0.11)
Random	64.50 (13.08)	0.22 (0.05)	0.00 (0.00)	1150.50 (246.99)	0.86 (0.11)
Do Nothing	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	2000.00 (0.00)	0.00 (0.00)

Seaquest

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	334.60 (153.63)	0.53 (0.16)	0.30 (0.14)	705.60 (103.02)	1.00 (0.02)
DeepSearch	433.60 (164.71)	0.65 (0.14)	0.60 (0.15)	745.90 (105.85)	0.99 (0.03)
MCTS	25.00 (2.05)	0.27 (0.11)	0.70 (0.14)	898.80 (68.89)	1.00 (0.01)
Onestep-S	10.90 (3.95)	0.05 (0.03)	0.00 (0.00)	340.70 (79.92)	0.94 (0.07)
Random	1.40 (0.75)	0.02 (0.01)	0.00 (0.00)	93.00 (31.79)	1.00 (0.01)
Do Nothing	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	103.30 (26.90)	0.00 (0.00)

Solar Fox

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	25.90 (2.51)	0.81 (0.08)	0.60 (0.15)	194.60 (14.56)	0.00 (0.00)
DeepSearch	32.00 (0.00)	1.00 (0.00)	1.00 (0.00)	176.20 (4.09)	0.00 (0.00)
MCTS	31.50 (0.34)	0.98 (0.01)	0.80 (0.13)	1416.00 (169.42)	0.00 (0.00)
Onestep-S	5.40 (3.04)	0.17 (0.10)	0.10 (0.09)	132.50 (27.73)	0.00 (0.00)
Random	0.10 (0.10)	0.00 (0.00)	0.00 (0.00)	59.60 (9.96)	0.00 (0.00)
Do Nothing	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	15.00 (0.00)	0.00 (0.00)

Zelda

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	6.60 (0.85)	0.83 (0.11)	0.90 (0.09)	90.80 (12.97)	0.98 (0.04)
DeepSearch	5.60 (0.83)	0.75 (0.11)	1.00 (0.00)	333.40 (131.67)	0.89 (0.10)
MCTS	2.20 (0.71)	0.28 (0.09)	0.50 (0.16)	1495.20 (176.43)	1.00 (0.01)
Onestep-S	1.80 (0.96)	0.23 (0.12)	0.10 (0.09)	460.90 (229.22)	1.00 (0.01)
Random	0.70 (0.40)	0.13 (0.08)	0.00 (0.00)	116.00 (47.80)	1.00 (0.01)
Do Nothing	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	883.20 (210.98)	0.00 (0.00)