Initial Results From Co-operative Co-evolution for Automated Platformer Design

Michael Cook, Simon Colton, and Jeremy Gow

Computational Creativity Group, Imperial College, London
`http://ccg.doc.ic.ac.uk`

Abstract. We present initial results from ACCME, A Co-operative Co-evolutionary Metroidvania Engine, which uses co-operative co-evolution to automatically evolve simple platform games. We describe the system in detail and justify the use of co-operative co-evolution. We then address two fundamental questions about the use of this method in automated game design, both in terms of its ability to maximise fitness functions, and whether our choice of fitness function produces scores which correlate with player preference in the resulting games.

Keywords: automated game design, procedural generation, co-operative co-evolution

1 Introduction

Procedural content generation (PCG) is a highly active area of research that offers effective methods for generating a wide variety of game content. PCG systems tend to work in isolation, often as a supplement to a human-designed system, designing aspects of the game's world [1–4]; generating items or abilities suited to the individual currently playing [5, 6]; or generating quests or tasks for the player to undertake [7–9]. However, automating design as a whole – that is, the design of a game solely by a system and without direct human judgement – remains largely uninvestigated.

The problem of automated game design is an attractive one to address, because it not only provides us with a basis to build stronger, more capable procedural content generation systems, but also allows for more intelligent design systems, representing a move away from merely creating content and towards co-operating with a human designer on a shared creative task. We have shown in [10] that automated game design systems can help complete partially-specified designs – further work building systems such as this will drive the development of assistive design tools.

The remainder of this paper is organised as follows: in section 2, we introduce co-operative co-evolution, prior work in this area, and related work in automated game design. In section 3, we present ACCME, a system which employs co-operative co-evolution to automatically design 2D platform games. We give details of experiments conducted using ACCME to investigate the effectiveness of computational co-evolution as an automated design technique. In section 4, we present conclusions and look at future work in the area.

2 Background

A co-operative co-evolutionary (CCE) system solves a problem by decomposing it into several subtasks called *species*. These are represented as independent evolutionary processes that are evaluated by recomposing members from each population into solutions to the original problem, and evaluating the quality of the complete solution. A CCE system decomposes a problem, P, into n subtasks, $P_1, \ldots, P_n$, where a *solution* for P is a set $\{p_1, \ldots, p_n\}$ with $p_i \in P_i$.

A fitness function for such a system evaluates a solution to the larger supertask P, rather than evaluating members of a subtask's population. Therefore, to evaluate a member, p_x, of the subtask P_x's population, we extract the most fit member of every other subtask's population, compose this set to form a solution to P, and apply the fitness function to this hybrid solution. The notion of 'co-operative' evolution refers to the way in which the fitness of the solution is directly related to how well p_x *co-operates* with the other components of the solution. Since these other $n - 1$ components do not change during the evaluation of the population P_x, the fitness represents how well a member of the population of P_x contributes to the overall solution.

Co-operative co-evolution was proposed in [11] by Potter and De Jong, in the context of function optimisation problems. They state that "in order to evolve more complex structures, explicit notions of modularity need to be introduced in order to provide reasonable opportunities for complex solutions to evolve". We hold that for creative tasks such as game design, a similar level of modularity is desirable.

The ANGELINA System

In [10], we presented ANGELINA, a system that designs arcade games using co-operative co-evolution. We decomposed the task of designing such games into several species, each of which is responsible for a certain aspect of the design. The three species generate *maps*, two-dimensional arrays that describe passable and impassable areas in the game's level; *layouts*, which specify the arrangement of red, green and blue entities in the game world as well as the play character; and *rulesets*, which describe the way in which the non-player entities moved, and also define one or more rules that describe the effects of certain types of entity, player or obstacle colliding with one another. A combination of a map, layout and ruleset defined a game. We performed several experiments to support the claim that CCE is able to rediscover existing games in the target domain, such as PacMan, as well as games that were novel. In [10] we describe a demonstration of the software running independently to design games, and an assistive task where ANGELINA is given hand-designed maps and produces suitable rulesets and layouts.

Related Work

Although we are not aware of any other work which addresses the problem of automating whole game design, there are other related projects. In [9], the authors evolve rulesets for arcade games. The system uses a neural network to learn rulesets. The fitness of games is based on how hard or easy they were for the network to learn. This work inspired the design of our domain in [10]. In [7], Nelson and Mateas evolve simple 'minigames' by interpreting terms that describe actions or subjects. The work is interesting in terms of higher-level design tasks relating to the interpretation of themes and their relation to game mechanics, although the work does not specifically tackle interrelated or co-operating design tasks.

In [8], Browne and Maire present a system for automatically designing board games. The underlying task of designing a set of rules that govern ludic interactions is common to both projects. The work is primarily involved in both identifying 'indicators of game quality' and subsequently applying these as heuristics in an evolutionary process for generating games. The work culminated in major successes in the area, including the publishing of some computer-generated game designs as commercial board games.

3 ACCME

Metroidvania is a subgenre of 2D platform games. The term, a portmanteau of two games that popularised the genre, was coined by Sharkey [12]. Metroidvania games are "based... on exploration with areas that [can] only be reached after attaining items in other areas" [13]. Contemporary examples vary from casual to more challenging games [14, 15]. The subgenre's core concepts lend themselves well to fitness functions.

ACCME is a system we have developed that designs such games using CCE, built on an evolutionary framework derived from [10]. ACCME is comprised of a Map species, a Layout species and a Powerset species. We first show how ACCME represents a game, then examine the species making up the CCE process and how playouts were implemented. We also detail some evaluative work which investigates the usefulness of CCE and the relationship between our definition of fitness and game quality.

3.1 Representation

A game is represented as a 3-tuple consisting of a Map, a Layout and a list of powerups called a *Powerset*. A map is a two-dimensional array of integers, where each integer in the array maps to an 8x8 pixel tile within the finished game. An zero value in the array describes an empty space, and any value greater than zero represents some tile texture (such as grass, or water). A *collision index*, i, is chosen such that any integer less than or equal to i is non-solid in the game world. This is used to define which integers describe solid platforms and walls, and which describe scenery. The collision index allows for tiles to change at runtime, allowing the representation of locked and unlockable doors.

A layout defines what we call *archetypes*, a description of a class of enemy. An archetype consists of one or more *actions* and a *movement behaviour*. Movement behaviours describe how the enemy moves through the game, selected from one of `STATIC`, where the enemy does not move, `PATROLS`, where the enemy moves horizontally until it meets an obstruction or there is no solid ground to walk on, and `FLIES`, which is similar to patrolling but does not require solid ground. Actions describe things that enemies can do during the game that provide a challenge to the player or somehow differentiate their behaviour from other enemies. An archetype has zero or more actions, selected from `TURRET`, which fires a projectile at the player whenever they are within a certain sight range, `POUNCE`, which causes the enemy to leap towards the player when they have an unbroken line of sight, and `MISSILE`, which fires a slower projectile that follows the player. A layout also contains a list of enemies, which are described by an (x, y) starting co-ordinate in the map, and an archetype number. The layout also describes the player's starting location and the location of the exit to the game.

A powerset is a list of powerups. A powerup is described by a co-ordinate representing its location in the map, as well as a *target variable* and a *target value*. When the player touches the powerup, the target variable is changed within the game code so

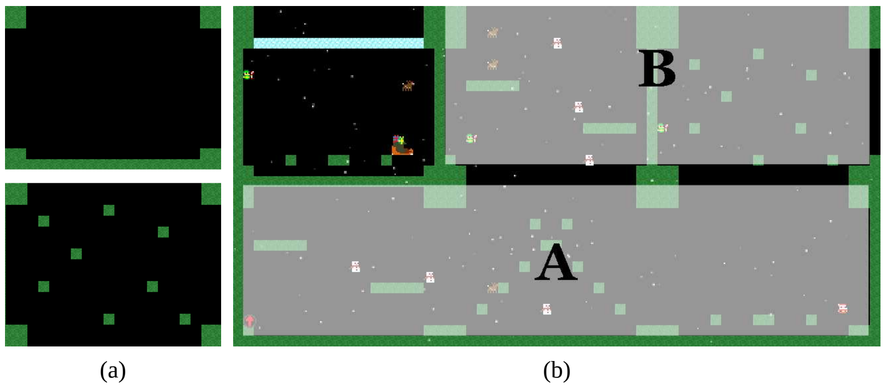

Fig. 1: Screenshots showing map templates (left) and regions (right).

that its value becomes that of the target value. There are three target variables available to ACCME - `jumpVelocity`, which describes the velocity applied to the player when the jump key is pressed, `globalAccelerationY`, which describes the effect of gravity applied to game objects, and `collisionIndex` which defines the integer value above which map tiles are considered solid. Target values are chosen from an integer range defined appropriately for each powerup.

3.2 CCE Species

Maps are constructed out of smaller two-dimensional arrays of fixed width and height called *Map Tiles*. The specific layout of a map tile is selected from one of 13 outer templates (defining the border around the tile). These outer templates define tile borders as blocked or unblocked - for instance, Figure 1(a) shows an outer template where the lower side of the tile is blocked. We provide all possible permutations, with the exception of the case where all sides are blocked. A map tile also selects one of 12 inner templates, which were hand-made. An example of such a hand-crafted template is provided in Figure 1a. Hand-designed templates were used to ensure some logical order to each tile, but with enough compositional variation that ACCME is responsible for the overall arrangement. A *Map*, therefore, is a two-dimensional array of map tiles, and the CCE process operates at no more detailed a level than map tiles when performing operations such as crossover.

The fitness function scores highly those maps which do not allow the player to leave the map bounds. It also heavily relies on playouts, assigning a higher fitness to those maps which have initially small reachable fractions, but whose maximal reachable fraction (having collected all relevant powerups) is high. We discount the fitness for contributions made to reachability early on in the game and towards the end. The intention here is to encourage steady progress throughout the game, where powerups make an increasing contribution to the player's abilities, and then after the game's midpoint, the player makes progressively smaller advances towards the exit. To describe the fitness function, first consider the game as a list of 'stages', beginning with the player start (p_s), culminating with the exit (p_e), and with intermediate stages representing the collection of a powerup. We represent the list of stages as: $[\ p_s, pup^1, \ldots, pup^n, p_e\ ]$.

Let $rch(x)$ be a function that returns the percentage of the map that is reachable at stage x of the game (but not reachable in the previous state). Then this list of stages contributes to the overall fitness proportionally, using fractional variables d_i as discounting factors to reduce the contributions made by each stage:

$$d_1 \times rch(p_s) + d_2 \times rch(pup^1) + \ldots d_x \times rch(pup^m) + \ldots d_2 \times rch(pup^n) + d_1 \times rch(p_e)$$

where $\forall x \forall y \; x < y \implies d_x < d_y$. The fitness function also assigns higher fitness to maps with longer paths between the start and the exit. In many games, it is considered poor practice to arbitrarily extend the player's path; however, due to the large state spaces inherent in ACCME, it is important to keep the maps small, while utilising as much of the space available as possible. Our intention was to generate games in which the optimal path through the map passes through as many map tiles as possible, to maximise the utilisation of the space available.

Crossover of two maps produces child amps that inherit either alternating rows or columns of the two parent maps. Mutation of a map replaces a number of randomly selected map tiles with newly generated ones, with a maximum of four replacements per map per mutation.

Powersets The fitness function for a powerset assigns fitness proportional to the amount of increase in reachability each powerup provides. Playout data is used to calculate this, and to calculate which powerups are reachable (and whether powerups can be collected in multiple orders, or whether there is a linear progression through the game). We employ the notion of a *trace object* which describes all possible routes through the game, recording the order in which powerups are collected, as well as if the exit is reachable. This is expressed as an ordering on the set of powerups, P. We are interested in traces where this ordering is partial, rather than total, as player choice is a desirable factor in the design of Metroidvania games. We define a trace T as a list of powerups, $\{p_1, \ldots, p_n\} \subseteq P$, where P is the set of all powerups in the game. We denote that the predicate $term(T)$ holds if, after executing the trace T, the player is able to reach the exit. We increase the fitness of a powerset relative to the number of *legitimate traces* it has in its trace object, where T is legitimate $\iff \forall T' \in (\mathcal{P}(T) \setminus \{T\}) \, . \, \neg \, term(T')$.

Note that $\mathcal{P}(T)$ is the power set of the set of powerups T. The above states that any sequence of powerups smaller than T would not permit the player to reach the exit. Preliminary experimentation showed this to be a useful balancing factor which encourages multiple traces through a game, but penalises designs in which the player is able to bypass a section of the game and ignore some powerups entirely. We also add value to a powerset's fitness relative to the average distance between each powerup. We calculate distance between objects by performing an A* search on the reachability map.

Powersets are crossed over by creating child powersets that randomly select powerups from the two parents, with a small chance to generate an entirely new powerup instead of inheriting from either. Mutation of a powerup randomises the magnitude of the change the powerup makes to its target variable.

Layouts The layout species is concerned with designing the enemy types present in the game and placing them within the map along with the player's starting location and the level's exit location. The task of enemy design is similar to the design of entities in the experiment described in [10]. We initially give very low fitnesses to any illegal or invalid placements. For ACCME, this involves penalising for enemies, player character or exit locations that are placed in walls. We penalise heavily for layouts that do not allow the player to reach the exit. To evaluate this, we use the same reachability calculations as present in the powerset evaluation described above.

Figure 1(b) shows a subsection of a game design. The player begins in section A, in which there is a powerup that allows access to section B. We identify these sections by calculating the player's reachability potential after picking up powerups in the game. We then reward layouts that introduce archetypes gradually, so that sections that are explored later in the game are more likely to have the full selection of archetypes, whereas sections explored early in the game may only have a subset.

Layouts are crossed over by exchanging locations of enemy archetypes, exit and player locations, and designs for archetypes themselves. Crossover can also switch the enemies of map tiles, in much the same way that map crossover exchanges map tiles, that is either by row, column or single tile. Mutation is applied to make small changes to the location of enemies, player start location and game exit. Mutation can also randomly change features in an enemy archetype, altering the movement type or adding and removing behaviours.

3.3 Playouts and Reachability

ACCME performs playouts in order to take a game state and establish which regions of its map are currently reachable. The system can apply powerups to change variables that affect reachability. Calculation of the reachable area is computationally expensive given the number of games assessed in a run of the system (although an individual reachability check merely tests each reachable tile for nearby reachable tiles, a sample run described in section 3.5 evaluates over 240,000 games). ACCME maintains an *open list* of map locations that are known to be reachable, initialised with the starting location. Upon removing a new location from the open list, it checks three possible scenarios:

Jumping If the player is standing on solid ground, they are capable of jumping. The formula used to calculate the potential height of the jump is ${V_{start}}^2/2g$, where g is gravity, expressed in pixels per second per second, and V_{start} is the upward force applied by the jump operation, also expressed in pixels per second.

The formula for jump height, combined with the knowledge that horizontal force can be applied regardless of the player's position, allows us to calculate the space in which the player has a positive vertical velocity (the *rising area*) by simply applying the maximum horizontal force in both directions for the duration of the jump. We then traverse the map locations in this area, and for each location we test to see if there are obstructions between the starting location and the target. If there is not, the area is reachable, and is added to the open list. We found that using line-of-sight as a check for accessibility is a cheap but effective method for deciding whether or not an area was reachable.

Walking If the player is on solid ground, we perform a Walking check. If a contiguous area of solid ground extends left or right of the current location, then the locations above this solid ground are also considered reachable. This helps cover some map areas that would otherwise take a longer time to detect using only jumping and falling.

Falling If the player is not on solid ground, then they are falling. This may be because they have walked off the edge of a platform, or are jumping. In this case, we calculate the horizontal extent of a jump to simulate the player's descent and label areas that are reachable during the fall. Because the player can apply horizontal velocity during a fall, this is different to a real-world physics simulation.

Quality and Accuracy of Reachability Estimations The above cases provide an estimate of reachability, allowing ACCME to assess levels and infer where the player can and cannot reach without having to simulate a full game playout. By avoiding such extensive simulation, we are able to greatly reduce the complexity of evaluating a game without much loss in reachability data; however, extensions to ACCME's domain that allow for other kinds of obstacles (such as enemies which cannot be destroyed) would, we think, require a full simulation in order to fully assess runtime reachability.

In deciding how best to estimate reachability, we opted for a system which, at worst, *underestimates* the amount of reachable map space. Overestimating in this case would produce games that were potentially unsolvable, but by underestimating we merely allow for the fact that through application of skill the player may be able to bypass certain sections of the game level (by reaching areas which ACCME had flagged as unreachable). Such situations are not uncommon for games, and give rise to *speed runs*, where players use such design flaws to complete a game in the fastest possible time[16].

A pilot study outlined in section 3.5 showed that, since reachability was not bidirectional, ACCME was unable to differentiate between areas that were reachable, and areas that could be reached and then returned from. This caused ACCME to design games with one-way jumps and inescapable pits. We modified the software to use a single additional check per reachable map tile to detect if it can be exited as well as entered. We proportionally reduce the fitness of maps that contain dead ends – this still allows for situations where a player is able to escape a dead end by obtaining a powerup.

3.4 Evolutionary Setup

A typical execution of the software is composed of 400 generations, undertaken with each species maintaining a population of 200 solutions. We utilise a steady-state, elitist selection method, with the fittest 10% comprising the parents of the next generation. The parents are also included for another generation of evolution; this is to allow trends to emerge more readily from the co-operating species, as existing progress towards co-operation is not lost between generations. We experimented with the application of some other selection techniques such as roulette-based approaches, but found them to be considerably less reliable. We plan to explore other such techniques in future work.

3.5 Evaluation

Effectiveness of CCE To compare the results of CCE with selecting from a comparable population of randomly-generated games, we generated 240,000 game designs at random and evaluated them using the same fitness functions that ACCME uses. The fitness of the highest-scoring game is shown as a dashed line in Figure 2. On the same graph, the line shows the fitness change over 400 generations for ACCME running as a co-operative co-evolutionary process with three species, each with a population of 200 members. Each species evaluates against only the fittest members of the other species,

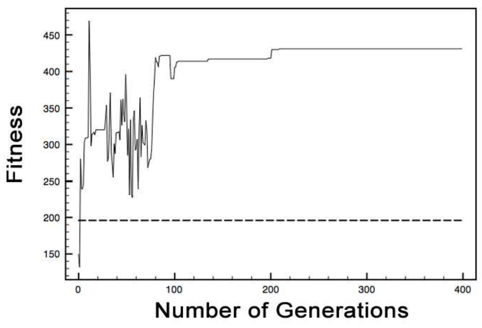

Fig. 2: A standard run of ACCME against a comparable random search.

hence ACCME evaluated 600 games per generation. The graph shows a clear improvement over random generation after only a handful of generations, and also highlights the fast convergence of ACCME. The strength of the convergence may point to a weakness in the CCE system, a matter which we discuss in section 4.

The early spike seen in figure 2 highlights a game with higher fitness than those of the final generations. We select our output from the final generation run, because we posit that these games exhibit the strongest co-operative traits, so a spike of this nature is, we believe, caused by one of the three species presenting a very high-fitness solution that counterbalances lower-fitness solutions in other species. CCE processes fluctuate wildly during the early stages of generation. At these stages, the species are far from co-operative, which means that the change between generations is often very large as they try to compensate for the lack of co-operation. We plan to examine such spikes in further work, as it may indicate that our fitness functions are not balanced in their evaluation of co-operative fitness, or that our method of synthesising fitness evaluations from the co-operating processes is not the best way of evaluating overall fitness.

Pilot Study We performed a pilot study to assess player response to the games produced, and to highlight issues in player evaluation of the games. 180 players played the same game and rated it between 1 and 5. Qualitative feedback was also sought to gain insight into player reactions. From the responses, we made several improvements to ACCME, including developing its understanding of reachability to include dead end detection (described in section 3.3). We noted that players' responses frequently highlighted areas of the game that were not ACCME's responsibility – such as control schemes or art direction. We discuss this in section 4 in the context of future work.

Comparative Study Following the pilot study, we performed a second smaller study in which 35 participants, responding to a call sent out to the 180 pilot study participants, were asked to play three games designed by ACCME. We chose three games with fitness valuations of 436, 310 and 183, labelled high, medium and low fitness respectively, to represent a range of game fitnesses. The unlabelled games were presented to each participant in a randomly-selected order, and the participants were asked to rank the games in terms of perceived quality after playing all three. Our hypothesis was that

	Best Rank	Middle Rank	Worst Rank
HighFitnessGame	19	9	11
MedFitnessGame	9	15	15
LowFitnessGame	11	15	13

Fig. 3: Data showing frequencies of ranks for the comparative study.

higher-fitness games should be preferable to players than low-fitness games. For our study data, we found a greater proportion of high fitness games were ranked highest: 49% compared to 25% for low/medium fitness games. However, the effect was not significant (chi-squared, p=0.15). We found a very weak but insignificant rank correlation between fitness and player preference, (Kendall's τ = 0.11, p = 0.17). In both tests, we were unable to reject the null hypothesis. Although these results are inconclusive, the data suggests to us that there may be some effect of fitness on preference, but further study is required to investigate the relationship.

One key piece of written feedback we received was that some players felt the games were too similar. This is partly down to commonality in features not designed by ACCME. However, the repeated nature of the goals and the restrictive set of powerups from which ACCME chooses from also contributed to this. In a creative task such as game design, the ability to create novelty is crucial, and ACCME does not appear capable of this in its current state. The brevity of the three games was also mentioned in some written feedback as a negative quality. The games in the study were restricted to 3x3 map tiles; given that a key element in Metroidvania games is gradual exploration, it is conceivable that the games did not last long enough for the players to experience this sense of gradual exploration. Hence, larger game sizes might improve future studies.

4 Conclusions and Further Work

We have introduced co-operative co-evolution in the context of automated game design, and presented ACCME, a system for designing simple Metroidvania platform games using CCE, reachability analysis and a flexible powerup system. We have shown CCE to be effective at developing high-fitness solutions, but a comparative study shows a gap between ACCME's concept of fitness and player preference.

The pilot study highlighted many difficulties in the evaluation of automatically-designed games by humans. The task of distinguishing decisions made by the automated designer, and decisions made by the authors in constructing the framework, is difficult. We received comments on aspects of the design that ACCME was responsible for, as well as things inherent in the template game supplied to ACCME. This shows a need for more forethought in presenting automatically designed games to players in future.

Our later comparative study highlighted the similarities in games produced by ACCME, even when the system considers there to be large differences in fitness. This leads us to two areas of further work. Firstly, we plan to reconsider the fitness valuations used, in order to strengthen some areas of evaluation, and add in new valuations to emphasise some areas that playtesters perceived as lacking, such as difficulty. One approach might be to simulate simplified combat, reducing the game to a turn-based simulation, thereby discretising and simplifying the evaluation. Secondly, it may be necessary to focus our surveys in future to avoid general concepts such as preference or fun. Designing experiments to evaluate specific parts of a design, such as level layout or powerup design, may provide a better way of estimating the impact of a system such as ACCME.

We noted earlier that ACCME converges on a solution very quickly. CCE has unique problems associated with it that affects its ability to locate global optima, which we are yet to investigate in ACCME. Such problems are discussed in [17] and some solutions proposed in [18]. We aim to apply these ideas to ACCME in the hope that it will improve the process. We also wish to investigate alternative selection methods for the evolution. All of the games listed in this paper can be played online at http://bit.ly/gbangelina.

5 Acknowledgements

The authors would like to thank Zack Johnson, Kevin Simmons and Riff Conner for their insight into Metroidvania design. We also thank the anonymous reviewers for their helpful comments and suggestions which have helped improve the paper.

References

1. J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, and G. Yannakakis, "Multiobjective exploration of the Starcraft map space," in *Proc. of 2010 IEEE Conf. on Computational Intelligence and Games*.
2. D. Ashlock, C. Lee, and C. McGuinness, "Search based procedural generation of maze-like levels," in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, issue 3, pp. 260-273, 2011.
3. G. Smith, M. Treanor, J. Whitehead, and M. Mateas, "Rhythm-based level generation for 2D platformers," in *Proc. of the 4th International Conf. on Foundations of Digital Games*, 2009.
4. L. Cardamone, G. Yannakakis, J. Togelius, and P. Lanzi, "Evolving interesting maps for a first person shooter," in *Applications of Evolutionary Computation*, vol. 6624, 2011.
5. E. J. Hastings, R. K. Guha, and K. O. Stanley, "Evolving content in the galactic arms race video game," in *Proc. of 2009 IEEE Conf. on Computational Intelligence and Games*.
6. A. Liapis, G. Yannakakis, and J. Togelius, "Neuroevolutionary constrained optimization for content creation," in *Proc. of 2011 IEEE Conf. on Computational Intelligence and Games*.
7. M. Nelson and M. Mateas, "Towards automated game design," in *Artificial Intelligence and Human-Oriented Computing*, pp. 626–637, 2007.
8. C. Browne and F. Maire, "Evolutionary game design," in *IEEE Transactions on Computational Intelligence in AI and Games*, vol. 2, issue 1, 2010.
9. J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *Proceedings of 2008 IEEE Conference on Computational Intelligence and Games*.
10. M. Cook and S. Colton, "Multi-faceted evolution of simple arcade games," in *Proc. of 2011 IEEE Conference on Computational Intelligence and Games*.
11. M. Potter and K. de Jong, "A cooperative coevolutionary approach to function optimization," in *Parallel Problem Solving from Nature PPSN III*, vol. 886, pp. 249–257, 1994.
12. S. Sharkey and J. Parish, *Debunking Metroidvania*. http://www.bit.ly/wiredmv/
13. *Metroidvania*, Gaming Wikia. http://gaming.wikia.com/wiki/Metroidvania
14. *Knytt Stories*, Nifflas Games, 2007. http://nifflas.ni2.se/
15. *Spelunky*, Mossmouth Games, 2009. http://www.spelunkyworld.com
16. *Portal Done Pro - Speedrun*, DemonStrate, 2010. http://www.j.mp/ygoJLh
17. R. Wiegand, "An analysis of cooperative coevolutionary algorithms," Ph.D. dissertation, George Mason University, USA, 2004.
18. A. Bucci and J. B. Pollack, "On identifying global optima in cooperative coevolution," in *Proc. of the 2005 Conf. on Genetic and Evolutionary Computation*.