

# An Argument for Large-Scale Breadth-First Search for Game Design and Content Generation via a Case Study of Fling!

**Nathan R. Sturtevant**

Department of Computer Science  
University of Denver  
Denver, CO, USA  
*sturtevant@cs.du.edu*

## Abstract

Search is a recognized technique for procedural content generation and game design, and it has been used successfully as part of commercial and academic games. In this context, search has almost always referred to selective search, as opposed to larger brute-force searches. The argument against brute-force search is that the state spaces of the games are almost always too large to be amenable for brute-force search. We believe, however, that brute-force search should not be too quickly dismissed. State spaces with trillions or tens of trillions states can now be exhaustively searched with relative ease, and growth in parallelism and computational power is expected to continue to scale this trend. We believe that this, combined with appropriate abstraction, will allow exhaustive search to be applied to many problems once thought to be prohibitively large. We explore this argument in the context of a game called ‘Fling!’, available for mobile devices, showing a system for interactively designing and analyzing puzzles.

## Introduction and Motivation

This paper explores the application of exhaustive search in content generation for games. Procedural content generation (PCG) is a growing field, with many diverse approaches. Many of these have been detailed in a recent survey paper (Togelius et al. 2011). This survey paper describes the majority of search approaches as being based in some sort of local search, often evolutionary mechanisms, although other search approaches are not excluded. Our goal in this work is to begin the exploration the applicability of large-scale brute-force search to the design process, both automated and with human interaction.

From our perspective, the work here addresses the following research question from this survey paper:

*Can we combine search-based PCG with top-down approaches?* Coupled with the right representation and evaluation function, a global optimisation algorithm can be a formidable tool for content generation. However, one should be careful not to see everything as a nail just because one has a hammer; not every problem calls for the same tool, and sometimes several tools need to be combined to solve a problem. In particular,

the hybridization of the form of bottom-up perspective taken by the search-based approach with the top-down perspective taken in AI planning (commonly used in narrative generation) could be very fruitful. It is currently not clear how these two perspectives would inform each other, but their respective merits make the case for attempting hybrid approaches quite powerful.

We say ‘from our perspective’ because the exact definition of ‘top-down approaches’ is not provided. But, we suggest a number of techniques for assisting with the solving of puzzles, including how we can use a brute-force analysis at the end of a puzzle along with selective forward search from a particular problem instance to interactively analyze puzzle instances.

Our ideas are fleshed out through a small case study of the game of Fling!, a popular puzzle game available for mobile devices. We use a combination of breadth-first search, retrograde analysis, and forward search. Our view is that this tool would be used by a designer to explore the design space possible within a domain that is being created. When design choices have been made about the types of puzzles desired for a given application, further automation could be applied to generate puzzles with desired properties, although we do not address the automatic content creation step here.

We then conclude with an overview of related work in large-scale search.

## Search Classifications

For completeness and clarity, we begin with a simple classification of search approaches. In particular, we distinguish between complete and incomplete search methods as well as between informed and uninformed methods. These approaches are most easily illustrated with how they are used for problem solving.

A *complete/informed* approach is usually used when solving a single instance of a problem. Heuristics or other guidance are used to prune away unpromising portions of the search, but the core of the search is exhaustive in nature. A\* (Hart, Nilsson, and Raphael 1968) and DPLL (Davis, Logemann, and Loveland 1962) (and more generally Depth-First Branch and Bound approaches) fall into this category, although A\* will run out of memory on large problems, while the DPLL algorithm will not.

A *complete/uninformed* approach is used when no guidance is available, or when the goal is to enumerate an entire state space. Example algorithms include breadth-first and depth-first search. When we use the term *brute-force search*, this is what we are referring to.

A wide variety of approaches can be described as *incomplete/informed*; these more broadly fall into the category of local search (Russell and Norvig 2005). In these approaches a solution (or genome which encodes a solution) is created and then iteratively altered to improve the solution. The algorithms converge to a local optima, with no guarantees of global optimality, but can run on problems that are not feasible for complete search. These algorithms can be informed both by the representation used, by selection heuristics, and by evaluation of the quality of the resulting solution.

*Incomplete/uninformed* approaches are less common; the simplest example is a pure Monte-Carlo search, where the best action is determined by random walks through the state space. But, to be meaningful, an evaluation (fitness) function is usually still required to evaluate the result of the random walks.

This paper focuses on the use of complete/uninformed search and its applicability to PCG.

### Example Domain: Fling!

Fling! is a game by CandyCane Software which has been available for iOS devices for a number of years, and more recently for other devices. A screenshot of the game is shown in Figure 1. The goal of the game is to fling the balls into each other and remove all the balls, except one, from the board. We demonstrate this in Figure 2. The left side of this figure contains a much simpler level, with only three balls. Moving the top ball down will cause it to collide with the ball below it and stop. The lower ball rolls off the board, and the upper ball remains. The board that results after the move is shown in the right half of the figure. Then, either ball can be rolled into the other ball to complete the level.

Fling! has several modes of play, but the primary mode has 35 successive levels. The number of puzzles needed to pass a level varies from three to eight. The number of balls gradually increases through the levels, increasing the difficulty of solving the levels. In the main mode there is no time limit on play.

If there are  $N$  balls on the board, every move will roll exactly one ball off the screen, so the solution is always of length  $N - 1$ . The number of balls on the board in the game varies between 3 and 14, but there is no fundamental reason why more balls cannot be used. There are 56 total locations on the board. Ignoring symmetry, there are  $\binom{56}{14} = \frac{56!}{14!42!} = 5,804,731,963,800$  ways to place 14 balls on the screen. Storing which combinations are solvable<sup>1</sup> would take 725,591,495,475 bytes, or about 675 GB, which is well within the range of current external storage, but might take a few weeks to solve. Table 1 shows the number of states and the memory required for storing all arrangements with a given number of balls on the board. If we were building this game and had finalized the mechanics of the

<sup>1</sup>Using 1-bit per state (without further compression).

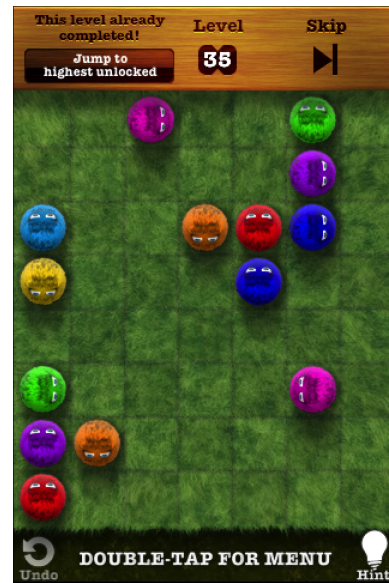


Figure 1: A level 35 (the highest level) Fling! board.

game, we would consider large searches to help us select interesting levels for the game, but significant time investments may not always pay off in early design stages. The full process of puzzle selection and design can easily take weeks or months, and so an exhaustive search could be used to validate the final solutions and to assist in later puzzle design.

### Building a tool to analyze Fling! puzzles

Given the previous analysis, we can now build a tool to analyze and explore Fling! puzzles. We begin by iteratively solving the Fling! boards of size  $1 \dots 10$  using a retrograde search approach (Schaeffer et al. 2003). Given that all boards of size  $n - 1$  have been solved, solving all boards of size  $n$  requires iterating through every possible board, and gen-

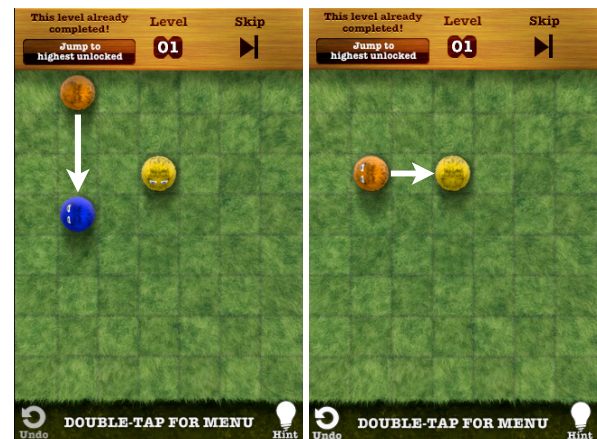


Figure 2: Sample moves on a fling board.

Table 1: Storage requirements for solving board with 1 . . . 15 pieces.

# Balls	# of Arrangements	Memory (GiB)
1	56	0.00
2	1,540	0.00
3	27,720	0.00
4	367,290	0.00
5	3,819,816	0.00
6	32,468,436	0.00
7	231,917,400	0.03
8	1,420,494,075	0.17
9	7,575,968,400	0.88
10	35,607,051,480	4.15
11	148,902,215,280	17.33
12	558,383,307,300	65.00
13	1,889,912,732,400	220.01
14	5,804,731,963,800	675.76

erating legal moves. If one of the moves leads to a level which is solvable, then the given board is considered to be solvable. There is nothing sequential about this process, and so it can be easily parallelized. On an 8-core 2.4GHz Intel Xeon machine with 12GB of RAM we were able to solve all boards with 10 balls in 129 minutes, or just over two hours using 16 threads. For perspective, this is less time that was spent developing the software to perform the search. Hyper-threading enables faster performance with more threads than processors; in this case we used 16 threads and 8 cores. We refer to this data as the *endgame* data, as it resembles the endgame databases built for solving the game of Checkers (Schaeffer 1997). Iterating through the boards can be done with a ranking and unranking function, also called a perfect hash function. If there are  $N$  possible boards, an unranking function can take the integers  $0 \dots N-1$  and convert them into unique boards. A ranking function reverses the process. We used well-known methods described in the literature (Edelkamp, Sulewski, and Yücel 2010) for this process<sup>2</sup>.

We then built a tool, shown in Figure 3 which, given a Fling! board, can perform searches to determine the following metrics for the given board.

1. **The number of states legally reachable.** This is determined by a forward breadth-first search from the current board state. The endgame data is not used, as it would prevent an accurate count of legally reachable states. This metric can be expensive for large boards. The problem in Figure 3 has 14,409 reachable states, which takes about 50ms to exhaustively search.
2. **The legal moves which lead to a goal state.** This is determined by a depth-first search with duplicate detection. The endgame data is used to speed this process. The white triangles in Figure 3 indicate which pieces can be moved

<sup>2</sup>Simple representations, such as a 1/0 bit-representation of each square on the board, can also be used, but are not necessarily space efficient when stored on disk

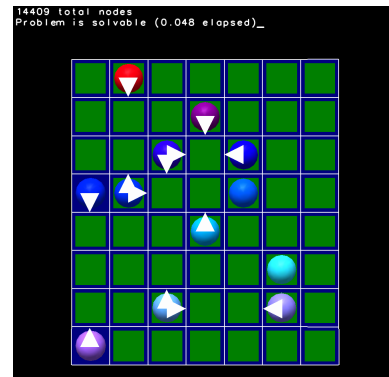


Figure 3: A screen shot of our tool for analyzing Fling! boards.

and in which directions to solve the board.

3. **How adding/removing pieces from the board change the solvability.** This process iterates through all 56 locations on the board. If a location has a piece, the piece is removed, and the resulting board is analyzed. If a location does not have a piece, a piece is temporarily added, and the resulting board is analyzed. Our tool highlights the squares on the board where adding or removing a piece would change the solvability of the board. Removing the piece in the lower-left hand corner of Figure 3, for instance, would render the puzzle unsolvable. The cost of this analysis varies greatly, but is most expensive when many positions are not solvable.

The tool also allows the user to add or remove balls from the screen by left-clicking on them, and balls can be ‘flung’ across the board by right-clicking and dragging.

After playing with the tool and its parameters, we opted to only load the 9-piece endgame data into memory, as it takes a significant time and memory to load the data for 10 pieces, and the puzzles that we are experimenting with do not require the larger endgame data for high performance. If the larger data was required, we would consider accessing the data directly from disk and caching results in memory. This actually suggests that the Fling! puzzle is relatively simple to analyze and does not require the full advantages of the endgame data for analysis; we present these results momentarily.

The puzzle in Figure 3 was generated randomly and has many possible solutions – almost any legal move will lead to a solution. There is only one legal move which does not lead to a solution. In collecting the solutions for the puzzles offered to the user in Fling!, we verified what a hint in the game suggested – that the puzzles in the game only have a single unique solution. We illustrate this in Figure 4, a puzzle from the game which has a single solution. But, as can be seen, there are many ways that we could add or remove pieces from the board and still have a solvable puzzle. Adding or removing pieces almost always significantly increases the number of possible solutions for the given board.

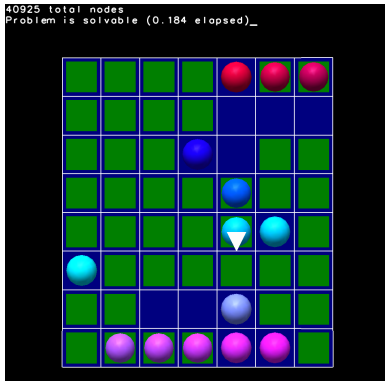


Figure 4: A problem from level 35 which only has a single solution.

The designer of this game<sup>3</sup> clearly chose this property as one that was important for creating interesting puzzles, although we could imagine allowing more than a single solution at earlier levels and decreasing the number of solutions at later levels. Generating all problems which only have a single solution can be done using the same retrograde analysis as we used to generate the endgames for the solvable states. But, when testing a state at level  $n$ , we would only set its related bit if there was only a single move that led to a solution. (Technically speaking there can be multiple moves, but when multiple moves are available, they all lead to the same successor state.) We could also modify our analysis to indicate which changes to the board lead to a board with a single solution path.

### Measuring Problem Difficulty

We believe that there are general metrics that can point to the difficulty of a puzzle, something that has been explored in other puzzles as well (Ritchie 2011). To us, the most obvious and generic metric is the number of states reachable from an initial puzzle instance. We stored all the boards given to us in a single play-through of the game and then solved them all with a breadth-first search, counting the number of legal states. The results are shown in Figure 5. The solid curve plots the average number of states in the BFS of each level, while the error bars indicate the easiest and hardest instances at a particular level. While we didn't find a monotonic relationship between difficulty and the number of states in the state space, there was a strong correlation through most of the levels. There is also a strong correlation between the number of balls on the board (shown in Figure 6) and the average reachable states in a level. The bulk of the time in this analysis was spent collecting levels; analyzing all levels can be done in about 15 seconds.

There are domain-specific metrics which could also be used. For instance, a fling-specific metric is how many times the user must switch from moving one ball to another during the solution, and how one move interacts with balls that were

<sup>3</sup>We attempted to contact the designer of the game, but did not receive a response.

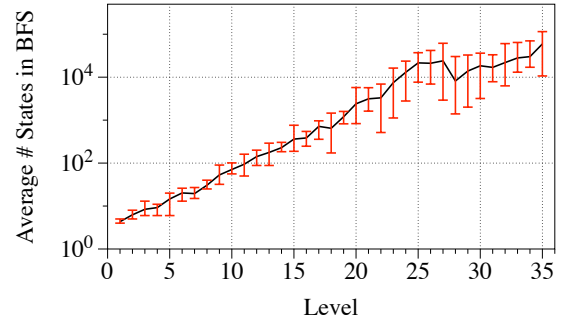


Figure 5: The reachable states of Fling! instances on each given level (a proxy for difficulty). The error bars represent the smallest and largest instances on a level. Note that the  $y$ -axis is logarithmic.

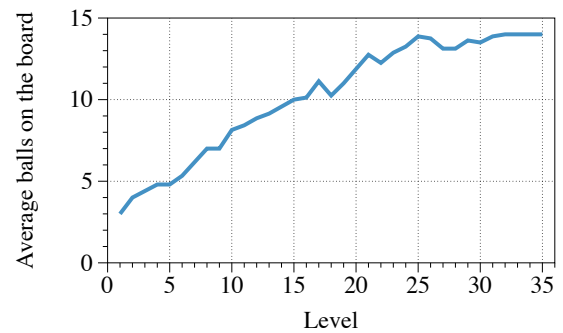


Figure 6: The average number of balls on each level of the game.

disturbed in the previous move. The puzzles that ship with the game all only have a single legal solution, which actually gives a logical flavor to the puzzles, as you can logically exclude moves to help solve puzzles. (If a pair of moves can be applied in any order and result in the same board configuration, then they won't be on the solution path.) Thus, you could also measure the difficulty of a problem based on the number of candidate moves after performing logical reasoning to eliminate incorrect moves. More work is needed to explore this, but it wouldn't be difficult to annotate states in our tool with different metrics.

### Savings from Endgame Analysis

Next we measured the savings from using the 9-piece endgame data during the analysis to determine which pieces could be added or removed from the game to preserve solvability. We generated 100 random problems with each of 14 through 16 pieces and solved them with and without the endgame data, measuring the time required to perform the analysis. The results are in Figure 7. Note that the  $y$ -axis is on a logarithmic scale to clarify the savings.

The overall savings from the endgame data is about a factor of 5 on the random problems with 14 pieces, but the sav-

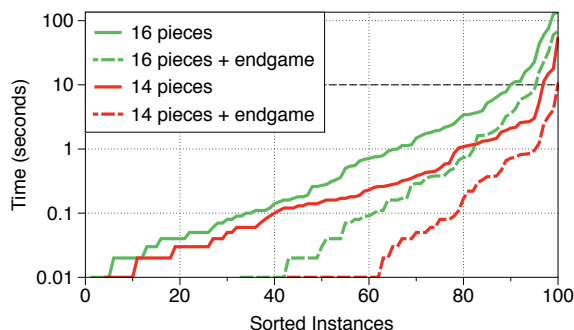


Figure 7: The savings from using the endgame analysis in Fling!

ings are larger on easier instances than harder instances. The boards that are most difficult are those that are unsolvable or have many locations which would cause the problems to be unsolvable. These boards require the full DFS to verify unsolvability, while solvable boards tend to terminate early and can be solved quite quickly. We also measured the savings over all the actual problems that we collected from the game and found the same factor of savings on actual instances and random instances.

These savings are less than we expected, but are still meaningful. They are related to the properties of the Fling! game; it is a point of future research to identify when retrograde analysis will save time and how much time it will save.

### Solvable Problems

One metric which influences the search for interesting instances is the number of problems that are solvable. We observed that many random instances were solvable without modification and so we looked into the percentage of solvable random instances. The results are in Figure 8. While only 10% of the problems with 4 pieces are solvable, around 65% of the problems with 10 pieces are solvable. This means that the problem is not in generating solvable instances, but generating interesting solvable instances. The difference in the number of solvable problems will significantly influence the features needed in a tool designed to help designers build and find interesting instances, something which is a topic of current research (Smith, Butler, and Popovic 2013).

### General Approach

Using our tool for Fling! as a model, we propose that there are a number of brute force search techniques which can be used to assist in the design of interesting puzzle instances. These include combining retrograde analysis and the creation of endgame databases, as well as forward search using this data. Other techniques include selective breadth-first search. Together, these techniques can be used to label and annotate puzzle instances to suggest how changes to a puzzle will influence solvability.

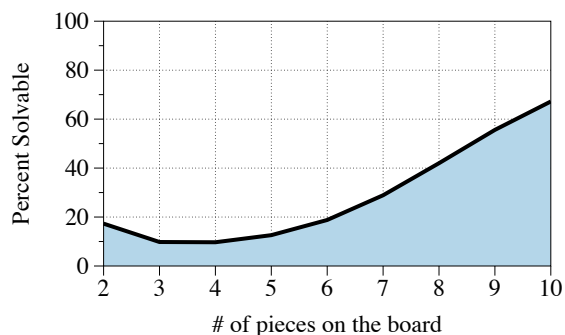


Figure 8: The percentage of problems which are solvable.

Overall, the state space of many games grows exponentially with the size of the puzzle, meaning that a slight growth in the size of the representation can result in a large multiplicative growth in the state space size. Although relatively large instances can now be solved exhaustively, the state space of many games is far beyond that which can ever be exhaustively searched. While we agree that these methods do not apply to all games, particularly those of a more continuous nature, many large state spaces can be decomposed or abstracted in ways that drastically reduce the number of possible states.

Consider, for instance, work on generating puzzles for a fraction-training game (Smith et al. 2012). In this game players must route lasers around a board, which is blocked in certain locations by rocks. While the full game is too large to exhaustively enumerate, it is possible to factor the board and generate all solutions for the lasers, ignoring the rocks. Then, the placement of rocks can be used to filter the legal solutions. This type of decomposition has the potential to significantly scale the size of the search space which can be handled via uninformed search.

### Related Work on Large-Scale Breadth-First Search

Large-scale searches, often with trillions or more states, have been more widely studied in the last few years. We highlight a few approaches here, although there is much more work on the topic than can be described here.

Notable in this work is Korf's complete breadth-first search of the 15-puzzle sliding-tile puzzle (Korf and Schultze 2005), which has over 10 trillion ( $10^{13}$ ) states. At the time this took approximately 30 days, and there was insufficient storage to store the results for later analysis, although this is possible today on consumer hardware. The approach took advantage of a large number of search techniques, including frontier search, which just stores a single layer of the BFS at one time.

Another line of research has been performed on Rubik's cube. This work successively reduced the bound on the maximum number of moves required to solve any state (Kunkle and Cooperman 2008). The primary method used to perform these computations was large-scale parallel breadth-

first search with hard disk drives for storage. The final bound of 20 moves was recently proven (Rokicki et al. 2010) using slightly different methods.

Both of these techniques are primarily one-shot, in that a computation is performed, but the results are not used for significant purposes afterwards. Work by Zhou and Hansen (2007) have used large-scale searches for finding optimal solutions to planning problems, and they have developed methods for using data stored in external memory efficiently (Zhou and Hansen 2005).

The largest search results which have been computed for later extensive usage has been the endgame databases used in the game of Checkers (Schaeffer et al. 2005). These databases were built after the Chinook program had finished competitive play, but could be used both for analyzing and playing the game of Checkers, and were later used for solving the game (Schaeffer et al. 2003).

More recently, we have computed and saved the results of a large-scale BFS of the edge cubes of Rubik’s Cube, and of the single-agent version of Chinese Checkers (Sturtevant and Rutherford 2013). Both of these state spaces have about 1 trillion states and are stored using 4-bits per state requiring approximately 500GB of storage. The Rubik’s cube search only required a week to complete, while Chinese Checkers took a month.

One other area where large-scale searches are also performed is in model-checking (Jabbar and Edelkamp 2006). In this area the searches are used to validate that a model of a system meets the given specification.

## Conclusions and Future Work

In this paper we have performed a case-study of the puzzle game Fling!. We have used a number of brute-force search techniques to enable designers to explore puzzles and how modifications of the puzzles influence solvability. This paper represents the preliminary stages of work on automated large-scale analysis of puzzles with the purposes of semi- or fully-automated design of new puzzle instances. Much more work is needed to improve and understand the limits of the approach, stretching the applicability of simple searches for puzzle design.

## Acknowledgements

Thanks to the anonymous reviewers which provided many excellent suggestions for improving this paper, including the taxonomy of search approaches.

## References

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *Commun. ACM* 5(7):394–397.

Edelkamp, S.; Sulewski, D.; and Yücel, C. 2010. Gpu exploration of two-player games with perfect hash functions. In *SOCS*.

Hart, P.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4:100–107.

Jabbar, S., and Edelkamp, S. 2006. Parallel external directed model checking with linear i/o. In Emerson, E. A., and Namjoshi, K. S., eds., *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, 237–251. Springer.

Korf, R. E., and Schultze, P. 2005. Large-scale parallel breadth-first search. In *National Conference on Artificial Intelligence (AAAI-05)*, 1380–1385.

Kunkle, D., and Cooperman, G. 2008. Solving rubik’s cube: disk is the new ram. *Commun. ACM* 51(4):31–33.

Ritchie, B. 2011. The inside story of how we created 2500 great rush hour challenges. <http://www.thinkfun.com/microsite/rushhour/creating2500challenges/>.

Rokicki, T.; Kociemba, H.; Davidson, M.; and Dethridge, J. 2010. God’s number is 20. <http://www.cube20.org/>.

Russell, S., and Norvig, P. 2005. *Artificial Intelligence, A Modern Approach, Second Edition*. Prentice Hall.

Schaeffer, J.; Björnsson, Y.; Burch, N.; Lake, R.; Lu, P.; and Sutphen, S. 2003. Building the checkers 10-piece endgame databases. In *Advances in Computer Games*, 193–210.

Schaeffer, J.; Björnsson, Y.; Burch, N.; Kishimoto, A.; Müller, M.; Lake, R.; Lu, P.; and Sutphen, S. 2005. Solving checkers. In Kaelbling, L. P., and Saffiotti, A., eds., *IJCAI*, 292–297. Professional Book Center.

Schaeffer, J. 1997. *One jump ahead - challenging human supremacy in checkers*. Springer.

Smith, A. M.; Andersen, E.; Mateas, M.; and Popovic, Z. 2012. A case study of expressively constrainable level design automation tools for a puzzle game. In El-Nasr, M. S.; Consalvo, M.; and Feiner, S. K., eds., *FDG*, 156–163. ACM.

Smith, A. M.; Butler, E.; and Popovic, Z. 2013. Quantifying over play: Constraining undesirable solutions in puzzle design. In *International Conference on the Foundations of Digital Games*.

Sturtevant, N. R., and Rutherford, M. J. 2013. Minimizing writes in parallel external memory search. *International Joint Conference on Artificial Intelligence (IJCAI)*.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Trans. Comput. Intellig. and AI in Games* 3(3):172–186.

Zhou, R., and Hansen, E. A. 2005. External-memory pattern databases using structured duplicate detection. In Veloso, M. M., and Kambhampati, S., eds., *AAAI*, 1398–1405. AAAI Press / The MIT Press.

Zhou, R., and Hansen, E. A. 2007. Parallel structured duplicate detection. In *AAAI*, 1217–1224. AAAI Press.