Exhaustive and Semi-Exhaustive Procedural Content Generation

Nathan R. Sturtevant
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
nathanst@ualberta.ca

Matheus Jun Ota
Institute of Computing
University of Campinas
São Paulo, Brazil
matheusota@gmail.com

Abstract

Within the area of procedural content generation (PCG) there are a wide range of techniques that have been used to generate content. Many of these techniques use traditional artificial intelligence approaches, such as genetic algorithms, planning, and answer-set programming. One area that has not been widely explored is straightforward combinatorial search – exhaustive enumeration of the entire design space or a significant subset thereof. This paper synthesizes literature from mathematics and other subfields of Artificial Intelligence to provide reference for the algorithms needed when approaching exhaustive procedural content generation. It builds on this with algorithms for exhaustive search and complete examples how they can be applied in practice.

Introduction and Related Work

There are many high-level approaches to procedural content generation (PCG); these approaches are often reflective of the broader techniques used in the Artificial and Computational Intelligence. Evolutionary algorithms (Eiben and Smith 2003; Mitchell 1998), for instance, were widely used for optimization long before they were applied to PCG for games (Frade, de Vega, and Cotta 2010; Togelius et al. 2010; Shaker et al. 2012). Similarly, languages like STRIPS (Fikes and Nilsson 1971) and PDDL (McDermott et al. 1998) were developed for other planning tasks before being adapted for procedurally generating narrative in games (Riedl and Young 2010). This trend is also seen in work on answer set programming (ASP) with early work in the field (Dimopoulos, Nebel, and Koehler 1997) occurring far before its use in games (Smith and Mateas 2011). Similarly, machine learning (ML) has also been widely studied prior to more formally defining the use of ML for PCG as PCGML (Summerville et al. 2017).

A recent book (Togelius, Shaker, and Nelson 2016) provides a taxonomy of PCG methods, including search-based approaches to PCG, also called SBPCG (Togelius et al. 2011). These are divided into two parts, evolutionary search algorithms (§2.1) and everything else (§2.2.1), which includes exhaustive search, random search, and solver-based approaches such as ASP. Exhaustive search methods apply when the state space is small enough to enumerate. There is relatively little work in this area and the book doesn't go into any deeper detail on the algorithms or representations needed for exhaustive search. Thus, the field currently lacks a clear starting point for those wanting to use this approach.

This paper remedies the issue by studying the problem of *exhaustive* procedural content generation (EPCG). EPCG approaches use a generator that models the problem and can systematically generate all content for the model. Then, there are many algorithms that can be placed on top of a generator to efficiently evaluate and select content. EPCG as a broad approach is particularly useful when the domain is not amenable to gradient methods or local search techniques. EPCG can also be more efficient than generic solvers (e.g. SAT or ASP) when constraints cannot be naturally encoded or when problem-specific optimizations are possible.

The primary contributions of this paper are (1) to synthesize work from mathematics and other subfields of AI regarding the exhaustive *generation* of content with respect to the state or content representation. Simple algorithms for generating all combinations and permutations are described, with suggestions of how the approaches can be further optimized. Additionally, the paper (2) describes several high-level algorithms that can be used for selecting content. Finally, the paper (3) provides examples of evaluation functions that are applied with these algorithms to determine the final content that is generated. Put together, this (4) provides a reference for EPCG that encourages future scholarship.

Sample Domains

This paper uses two domains as examples, Fling! and The Witnes. Our problems in these domains are small enough for exhaustive enumeration. The paper will discuss techniques for scaling to larger domains in the conclusions, but it should be clear that there are some problems representations where EPCG will have little benefit. Consider, for instance, terrain generation. Suppose that we need to generate a 1024x768 2D terrain. For each x-value in the terrain we would need to generate a single height. Thus, with no other restrictions, there are 768^{1024} possible terrains. Most of these terrains are meaninglessly random, but even if we restrict the height at each location to be within one of the neighboring cell, ensuring that the terrain is smooth, there are $768 * 3^{1023}$ possible terrains – still prohibitively large. But, more importantly, there are still large numbers of terrain with little

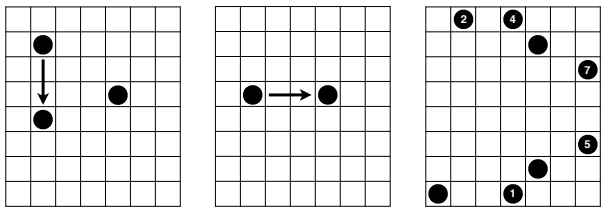

Figure 1: Movement in the game Fling!

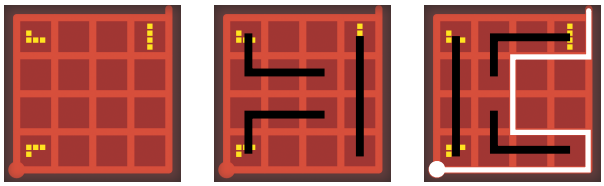

Figure 2: (a) A sample puzzle in The Witness; (b) a virtual layout for which no path can be drawn and (c) a virtual layout and corresponding path that solves the panel.

practical difference between them. Puzzle domains tend to be more focused and more amenable to exhaustive generation because they operate at a higher level of abstraction.

Example Domain: Fling!

Our first domain is the game Fling! (Sturtevant 2013). In this game the goal is to fling all the *pieces* off of a 7x8 game *board*. This is done by flinging one piece into another as illustrated in Figure 1. In part (a) there are two possible moves. Either the top piece can be flung down to hit the bottom piece, or the bottom piece can be flung up to hit the top piece. The solution requires the top piece to be flung down, as indicated by the arrow. Part (b) shows the result of moving the top piece down. After the collision the top piece stops just above the bottom piece, leaving two pieces side-by-side. The piece that was at the bottom continues off the bottom of the board. At this point either piece can be flung into the other piece resulting in a single piece left on the board. In order to initially fling a piece into another, there must be a gap of at least one space between the two pieces. If a piece is flung into a line of other pieces, there is a perfect transfer of momentum from one piece to the next until the last piece in the row/column is flung off the board.

Valid boards have only a single solution – that is, there is only one sequence of moves (excepting the last move) that leads to a solution. Of the 35.6 billion boards with 10 pieces, only 15 million (0.04%) have this property. Thus, the fitness landscape will likely be too sparse for evolutionary operations like crossover to produce valid boards. While solver-based methods could be used, generic solvers will not be able to match the efficiency of retrograde analysis discussed later in the paper.

Example Domain: The Witness

The Witness is a 2016 game by Jonathan Blow and Thekla, Inc. in which the player traverses a world solving a number of puzzle *panels* that appear in the world. These panels are solved by a self avoiding walk (Knuth 1976) from the start (typically the lower-left corner) to the goal (typically the upper right corner) within a grid according to the particular constraints of the puzzle. Each portion of the game introduces different constraints that are then combined and explored together in joint areas later in the game.

In this paper we select one type of panel and solution constraint to explore. This panel type is shown in Figure 2; it is necessary to explain the constraints for this panel here in order to motivate our approach, although these are not explained in the game. Figure 2(a) shows the basic puzzle.[1] We refer to the three tetris-like pieces as *piece constraints*. Without the piece constraints the solution is any path on the grid from the start (bottom-left corner) to the goal (top-right corner) such that the line does not intersect itself. What makes the panel interesting are the piece constraints.

In solving this puzzle, each of the pieces must be *virtually* placed on the board so that they do not overlap with each other and do not extend past the edges of the board. Figure 2(b) shows one possible placement of the virtual pieces as black lines. However, there are additional constraints. The line that is drawn between the start and the goal must divide the board into one or more *regions*, where each region only contains virtual pieces, or only contains empty space. No region can contain both a virtual piece and empty space. Thus, Figure 2(b) is not a valid layout of the pieces because there is no path from the start to the goal that does not cross itself and also separates the virtual pieces from the empty space. Two additional constraints are that (1) the path from the start to the goal may not divide a virtual piece into two parts and that (2) each virtual piece must be placed in the same region as the piece constraint from the game board (although the locations do not have to overlap). This is illustrated in Figure 2(c). There are two regions in the panel. The left region contains all three piece constraints on the board, and all three pieces can be virtually placed in this region without overlap or rotation. The right region is entirely blank, and the path between the start and the goal separates these regions.

These puzzle panels are an instance of a constraint satisfaction problem (CSP) (Russell and Norvig 2009), except that the spatial constraints in these puzzles are not found in typical CSP solvers. But, we are not interested in just solving the problems, but also in generating new content, which is more complex than standard CSP or ASP problems (Smith, Butler, and Popovic 2013). Taken together, this makes this problem well suited for exhaustive generation.

Exhaustive PCG

We begin with a definition of EPCG, which has not appeared previously in the literature.

Definition 1 Exhaustive Procedural Content Generation *(EPCG) describes approaches for generating procedural*

[1] http://the-witness.net/news/2016/02/printable-panels/

Algorithm 1: Basic puzzle generator

Input: Generator: G, Evaluator: E
Output: best puzzle b

```
1 b ←G.unrank(0)
2 for i ∈ {1, ..., G.maxRank − 1} do
3     t ← G.unrank(i)
4     if E(t) > E(b) then
5         b = t
6 return b
```

content where all possible content is methodically generated and evaluated.

The key to this definition is the *methodical* generation. That is, the content is generated in some well-understood order without repetition. This excludes randomized algorithms that, in the limit, would generate all content, but not without repetition. Note that in some problems, the fully exhaustive nature of the generation is not necessary. For instance, content can be pruned when it is shown to be sub-par, or when time or memory constraints limit the ability to enumerate all content. Algorithms that are capable of *methodically* generating all content, but that choose to to skip some content are *semi-exhaustive*. Algorithms underlying many generic solvers, such as the DPLL algorithm (Davis, Logemann, and Loveland 1962), are often semi-exhaustive in nature, but are not designed particularly for PCG.

A simple EPCG algorithm is shown in Algorithm 1. This procedure takes in an evaluator, which provides a numerical score to a state, and a generator, which we define below. The generator is the key to the procedure.

In order to methodically generate content, an algorithm must know how much content there is to generate and be able to generate this content. This usually requires some sort of combinatorial analysis of how content can be arranged, as well as functions to convert integers into content (a ranking function) and content back into integers (an unranking function) (Myrvold and Ruskey 2001). One could view the rank as an index into an implicit database of content from a given generator. In GAs, search occurs directly in the state representation, and the state representation is usually chosen to be amenable to GA operations like crossover. In EPCG the state representation is usually an array of piece locations which is amenable to ranking and unranking operations.

In EPCG we define a content generator, G, as a set of functions which must include *maxRank* and *unrank*, and optionally includes *rank* and *increment*. The *maxRank* function returns the total number of states that can be generated. The unrank function converts a index $\{0 \dots maxRank - 1\}$ into a state. The optional *rank* function covers a state back into an index, and the optimal *increment* function takes a state and directly computes the state with the next rank, without the full unranking process. We give examples of these in the next section.

Table 1: The first and last combinatorial ranks for $n = 20, k = 4$

Rank	State
0	0 1 2 3
1	0 1 2 4
2	0 1 2 5
...	

Rank	State
4841	15 16 17 19
4842	15 16 18 19
4843	15 17 18 19
4844	16 17 18 19

Algorithm 2: Unranking a combination into an array

Input: *rank, board[], numPieces, boardLeft, offset, boardSize*
Output: populated *board* array with positions of pieces for given rank

```
1 if numPieces > 0 then
2     b ← (boardLeft−1 choose numPieces−1)
3     if rank ≥ b then
4         unrank(rank−b, board, numPieces, boardLeft−1,
          offset)
5     else
6         board[offset] ← boardSize - boardLeft
7         unrank(rank, board, numPieces−1, boardLeft−1,
          offset+1)
```

Ranking and Unranking functions

In this section we demonstrate the functions for a content generator over the Fling! state space, which is described by combinations. The full explanation of the combinatorics is outside the scope of this paper, but can be found in mathematical references (Mazur 2010). Then we provide an overview of how other state spaces can be enumerated.

Consider placing pieces on a Fling! board with n board positions and k pieces. The first piece can be in any of n positions, and the second piece can be in $n - 1$, etc. But, because the pieces are not distinct, there are $k!$ symmetric orderings. Thus, the state space contains $maxRank(n, k) = \binom{n}{k} = \frac{n!}{(n-k)!k!}$ unique states. The initial and final states in the ranking for $n = 20$ and $k = 4$ are found in Table 1. This is a lexicographical ordering, as the digits are incremented in sorted order. The values are interpreted as the location of pieces, as opposed to representing the full board explicitly.

An *unrank* function is found in Algorithm 2. Initially *rank* is the value that is being unranked, *boardLeft* is the board size that hasn't yet been analyzed (initialized to boardSize). *Offset* is the offset in the *board* array for the next piece. The main loop of this function is testing to see if the next piece can go in the next location. The piece is placed in this position (line 6) if, after the piece is placed, the rank will be smaller than the number of combinations for the remaining pieces. This means that the sub-problem of placing the remaining pieces is well-formed. Otherwise the next location is skipped, and the rank is decreased by the corresponding number of combinations that were skipped. After returning from the function, the board array is populated with the locations of each of the pieces. This function runs in time linear in the size of the board; faster implementations can be written by pre-caching the computations.

Algorithm 3: Incrementing an array to the next combination. Piece i is in location i in the array. Assumes pieces are sorted with smallest piece in first location

```
Input: board[], numPieces, boardSize, currPiece
Output: populated board array with positions of pieces
        for next rank
1 currPieceLocation ← board[numPieces-currPiece-1];
2 if currPieceLocation == boardSize-1-currPiece then
3 |   newPieceLocation ← IncrementRank(board, numPieces,
  |   boardSize, currPiece+1)+1;
4 else
5 |   newPieceLocation ← currPieceLocation+1;
6 board[numPieces-currPiece-1] = newPieceLocation;
7 return newPieceLocation;
```

Algorithm 4: Ranking a board array into a combination

```
Input: board, numPieces, boardLeft, loc, offset
Output: rank of board
1 if numPieces == 0 then
2 |   return 0
3 if board[offset]-loc = 0 then
4 |   return rank(board, numPieces-1, boardLeft-1, loc+1,
  |   offset+1)
5 b ← C(boardLeft-1, numPieces-1)
6 return b + rank(board, numPieces, spaces-1, loc+1, offset)
```

It is often the case that algorithms iterate through the ranks in order. In such cases, one can compute the state with next higher rank directly at significantly lower cost. This procedure is shown in Algorithm 3. This procedure attempts to move the last piece to the next position. If this piece is already in the last position (line 2), the previous piece is incremented, and then the piece is placed just after the previous piece. When using the incremental procedure, Algorithm 1 has line 3 replaced with a call to the incremental procedure in Algorithm 3. Our tests suggest that this is 10-20 times faster than the full unrank procedure, although the savings depend on the cost of the evaluation function.

The final piece of a generator is the rank function, found in Algorithm 4, which takes a state and returns the rank. This isn't used in the simple pseudo-code of Algorithm 1, but we will demonstrate its use in the next section.

This section provides pseudo-code for generating states represented by combinations of pieces; C code for these procedures is available at `https://www.movingai.com/`. Other common states that are found in puzzle games are permutations, where a fixed set of elements are permuted in an array, $k-$permutations, where k elements from a permutation are permuted, and the remaining locations are blank. Bonet (2008) presents efficient algorithms for ranking and unranking permutations, including incremental approaches. Finally, multisets are used when there are a fixed number of different piece types that can appear in a puzzle. Multisets are written $\left(\!\binom{m}{k}\!\right)$ where there are m types of items, k total items will be chosen, and order does not matter. In practice multisets are related to combinations where $\left(\!\binom{m}{k}\!\right) = \binom{m+k-1}{k}$.

In The Witness puzzles, the locations of the pieces are determined by the same combinatorial approach as in Fling! Because the order of placement within the puzzle matters, the types of the pieces is determined by simpler counting rules. If there are p types of pieces and k locations, then there are p^k ways of arranging the difference piece types in the locations. The total number of locations is the product of the number of locations of the pieces and the arrangement of the piece types.

The basic approach naively generates all possible content, and does not take into account any constraints that might exist within a problem. For combinatorial puzzles, it will perform $O(\binom{n}{k})$ solving and evaluation operations, but only requires constant space. The approach can be easily parallelized by distributing the for loop across multiple threads.

To summarize this section, one key task in EPCG is breaking down the combinatorics of the problem being studied. Problems where identical pieces are placed across a board use the combinatorial algorithms in this section to generate content. Problems with different types of pieces, where order does not matter, are defined by multisets, and can also use these algorithms. Problems where unique pieces are ordered require permutations (Myrvold and Ruskey 2001). Other problems will require custom analysis.

Given this foundation, we now provide two alternate algorithms which build upon the ideas here to efficiently generate exhaustive content.

EPCG Algorithms

In this section we consider two algorithms, retrograde analysis and branch and bound search, that are more complex than the simple puzzle generator in Algorithm 1. We additionally give examples of the more complicated evaluators that can be used in EPCG along with these approaches.

Solving Large Game Boards

Suppose that, in the game Fling!, we want to find all boards that are uniquely solvable. There are 35.6 billion boards with 10 pieces on them (assuming a 7x8 grid), but only 15 million of these are uniquely solvable (0.04%).

We could directly use Algorithm 1 for these puzzles, but this approach would be inefficient due to the structure of the puzzles. Consider if we are solving boards with 6 pieces, of which there are 32.5 million possible boards. Assuming that on average there are 6 moves on each board (in practice this is higher), when solving all 32.5 million boards, there would be 195 million boards generated with 5 pieces on them during the solving process. However, there are only 3.8 million unique boards with 5 pieces, meaning that each board would be generated 51 times on average. By pre-computing information about all the boards with 5 pieces we can then efficiently re-use the information when solving the boards with 6 pieces. This retrograde or dynamic programming approach has been used to build endgames databases for two-player games (Schaeffer et al. 2004). In this case it can be used for

Algorithm 5: Retrograde analysis: Finding states with unique solutions

```
   Input: depth
   Output: bit arrays single and solvable
1  for i ∈ {0, ..., maxRank − 1} do
2      for each successor s of parent do
3          r ←rank(s)
4          sc, ss ← 0
5          if solvable[depth − 1][r] then
6              sc++
7          if single[depth − 1][r] then
8              ss++
9          if sc > 1 then
10             break;
11     if sc == 1 && ss == 1 then
12         single[depth][i] = true;
13     if sc > 0 then
14         solvable[depth][i] = true;
```

a single-agent puzzles, with special enhancements for finding the puzzles that have a single unique solution.

In Algorithm 5 we show the pseudo-code for this process. The input to the procedure is the current number of pieces on the board. After running, the *solvable* array has one bit set for every board that is solvable, and the *single* array has one bit set for every board that is uniquely solvable. When running with depth d it is assumed that the array has already been filled in for depth $d-1$. While this code is tailored to finding unique solutions, the main computation (lines 4–10 and 11–14) can easily be altered for other retrograde computations. This is particularly useful when there are very long solutions, such as when every action does not remove a piece from the board.

One interesting feature of this algorithm is how the uniquely solvable boards are computed. Doing this at depth d requires both the array of solvable boards at depth $d-1$ and the array of uniquely solvable boards at depth $d-1$. A board is uniquely solvable at depth d if exactly one child is solvable, and this child is also uniquely solvable. Thus, to identify a uniquely solvable state we need to keep both arrays in memory for the shallower depths.

A small optimization to this approach is found in line 9. If more than one solution to a board has been found, then we know that the board is not uniquely solvable, so we can stop analyzing the successors of the current board. The time to find boards with unique solutions among all 148.9 billion boards with 11 pieces required 8 hours and 29 minutes without this optimization and 4 hours and 13 minutes with it. These types of domain-specific optimizations can significantly scale the size of problem that is solvable, something that is lost when a generic solver-based approach is used. External-memory approaches (Zhou and Hansen 2004; Jabbar and Edelkamp 2006; Sturtevant and Rutherford 2013), where most data is stored on disk, have the potential to further scale this work.

This approach to retrograde analysis requires the ranking function (Algorithm 4), because we must look up the stored value of the successors of the current state being analyzed. The ranking function described in the previous section runs in time linear in the size of the board, but with simple precomputation it can be optimized to run in time linear in the number of pieces being ranked.

Evaluating selected boards The retrograde analysis procedure efficiently computes the puzzles that have exactly one solution, but we would still like to select content from among the puzzles that have a single solution. In particular, we would like to select the content that maximizes the ability for an expert player to use their expert knowledge when solving puzzles.

If there is a unique solution to a particular puzzle, each action will depend on the previous action in some way. If an action does not depend on the previous action, then that action could have been performed prior to the previous action, and there would be more than one unique solution to the board. For instance, in Figure 1(c), many of the pieces can be flung vertically into another piece. If two consecutive vertical actions were part of the final solution, the board could still be solved if the actions were performed in the opposite order, and thus there would be more than one solution. An expert with this knowledge can consider fewer possibilities.

Our evaluation function compares the size of the brute-force search tree with the size of the constrained tree that an expert would search, and selects the puzzles with the largest ratio. After all puzzles with unique solutions are found, the evaluator runs only on these puzzles and selects the puzzle with highest ratio between the brute force and constrained tree sizes. Figure 1(c) shows one puzzle that maximizes this metric. The pieces are labeled with their first move time in the final solution.

The approach uses the basic EPCG algorithm in Algorithm 1, except that we discard states that do not have a single unique solution. As a result, it is not efficient to use the incremental unranking function, because most states can be discarded without unranking.

Branch and Bound Search

With some content, the value of the content can be measured before it is complete. Thus, as content is incrementally generated, we can rule out content that is already shown to have a worse evaluation than our best content so far. We demonstrate this semi-exhaustive approach in The Witness where sets of joint puzzles are incrementally generated.

This work borrows a concept in the game which had been introduced, but not explored completely. The idea in these puzzles is to build a panel which contains p *sub-panels* which must be solved jointly. We illustrate one example for $p = 3$ in Figure 3. Although there are many solutions for each of the three individual sub-panels, there is only one solution that will jointly solve all three sub-panels (and thus the whole panel).

Our approach to these panels treats them like a secret sharing algorithm (Shamir 1979). The idea is that looking at any single sub-panel, or any subset of the sub-panels, should provide little or no information for solving the complete panel.

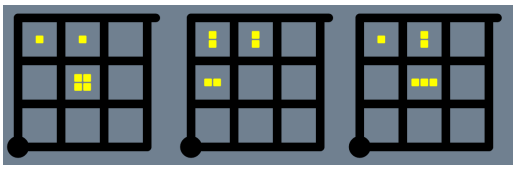

Figure 3: A panel with three sub-panels which must be jointly solved. All three sub-panels can be solved by the action sequence URDRUURU.

Only when all sub-panels are considered together can the user find solution to the panel.

Again, we could use an approach like Algorithm 1 to generate and evaluate all possible content. The evaluation of a set of panels is the minimum number of joint solutions for any subset of sub-panels. But, let us consider the combinatorics of this approach.

Our search begins with a library of 24 piece types. On the 3x3 sub-panels we restrict there to be only three pieces per sub-panel. Thus, there are $\binom{9}{3} = 84$ ways to place three pieces on a sub-panel and $24^3 = 13,824$ ways to choose the types of the three pieces, for a total of 1,161,216 possible sub-panel configurations. We are placing three sub-panels in a row, so there are 1.6×10^{18} possible panels to consider.

Exploring this many combinations exhaustively is prohibitively expensive, but finding the best combination is not, because many of them can be pruned away. The key feature of a branch and bound algorithm is the ability to evaluate a partial solution. In this case, the evaluation function can be applied to a set of sub-panels which is smaller than the full set. The result will be an upper bound on the evaluation of any full set that can be constructed by completing the partial set. Thus, if the partial evaluation is worse than the best solution, we don't need to further enumerate solutions.

This is illustrated in Algorithm 6, a recursive procedure for branch and bound search. When a complete set of boards is found, the best set is updated (line 1) if a better solution is found. If a partial set is worse than the current best solution (line 4), the search prunes the current branch as it will be suboptimal. Otherwise, the possible subpanels are recursively generated and added them to the current set. Although we have provided this code for the particular problem we are solving, the approach is more general and can be adapted to other combinatorial settings where incremental evaluation is an upper bound on the total evaluation.

Note that most of the sub-panels we can enumerate are not actually solvable. These evaluate to 0 and would be immediately rejected, but it is slightly more efficient to eliminate these *a priori*, which reduces the number of sub-panels to from 1,161,216 to 113,296. Taken together we can quickly find the optimal solution shown in Figure 3.

Further analysis demonstrated that our first puzzle was too easy to solve, because all of the virtual pieces are placed in the same region. We then repeated the process with the further restriction that the virtual pieces be in different regions in the final solution. This constraint gave us the panel in Figure 4(top), which is a significantly more complex puzzle. We

Algorithm 6: Branch and bound search

Input: $numBoards, S, E$
Output: best set of boards B

```
1 if numBoards = 0 && E(S) > E(B) then
2 |   B ← S
3 |   return
4 else if E(S) < E(B) then
5 |   return
6 else
7 |   for i ∈ {0, ..., maxRank − 1} do
8 |   |   s ←unrank(i)
9 |   |   BB(numBoards − 1, S ∪ s, E)
```

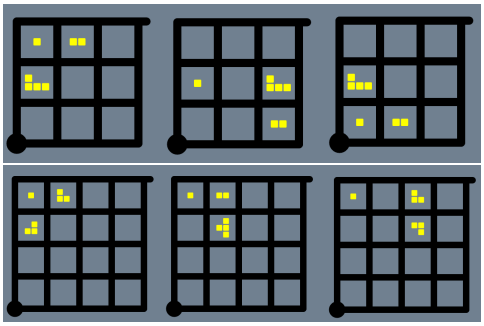

Figure 4: New panels with the evaluation constraint that the virtual pieces must be placed in different regions in the solution. The solution(s) are left as an exercise for the reader.

were also able to generate larger boards such as the puzzle in Figure 4(bottom) by more efficient parallelization of the solving process and by restricting to a smaller set of candidate boards. This is an example of semi-exhaustive search because we are using both the branch and bound constraints as well as artificially limiting the search space to find the best puzzles more quickly. Many people familiar with the game have had difficulty solving this puzzle because it explores the game in new ways.

Discussion and Future Work

This paper has provided a framework for exhaustive PCG, with examples of the ranking and unranking functions used in these frameworks. We demonstrate how these concepts can be used with retrograde analysis in Fling! and with branch and bound search in The Witness. Together, these examples of exhaustive and semi-exhaustive PCG form a foundation that will enable other researchers to build on these techniques for new applications.

As with all PCG, evaluating the quality of PCG content is extremely important (Shaker, Smith, and Yannakakis 2016). For the specific EPCG domains studied here, future work will perform user studies to measure the effectiveness of different evaluators in keeping players engaged and learning while they play the puzzles that are produced.

A broader question for future work is how to scale EPCG methods to different types of problems. For instance, it is not feasible to exhaustively generate all possible 200x14 Super

Mario levels. As with the previously discussed terrain example, most levels will be meaningless variations and will be mostly unsolvable. But, it would be feasible to define a library of 20x14 level segments that are semi-exhaustively chained together into full levels. Similarly, it also would be possible to semi-exhaustively generate 20x14 level segments for the library. As another example, we have been able to build a graph-based representation of some levels in the game *Braid* that could be exhaustively enumerated. But, significantly more work is needed to take these concepts and turn them into useful systems for generating content. Broadly speaking, in addition to more example applications of EPCG, there are also significant opportunities for research that abstracts game components so that EPCG approaches can be applied to the abstract space.

References

Bonet, B. 2008. Efficient algorithms to rank and unrank permutations in lexicographic order. In *AAAI-Workshop on Search in AI and Robotics*, 142–151.

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

Dimopoulos, Y.; Nebel, B.; and Koehler, J. 1997. Encoding planning problems in nonmonotonic logic programs. In Steel, S., and Alami, R., eds., *Recent Advances in AI Planning*, 169–181. Berlin, Heidelberg: Springer Berlin Heidelberg.

Eiben, A. E., and Smith, J. E. 2003. *Introduction to evolutionary computing*, volume 53. Springer.

Fikes, R. E., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2(3-4):189–208.

Frade, M.; de Vega, F.; and Cotta, C. 2010. Evolution of artificial terrains for video games based on accessibility. In *Applications of Evolutionary Computation*, 90–99. Springer Berlin/Heidelberg.

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.

Knuth, D. E. 1976. Mathematics and computer science: Coping with finiteness. *Science* 194(4271):1235–1242.

Mazur, D. R. 2010. *Combinatorics: a guided tour*. MAA.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. Pddl-the planning domain definition language.

Mitchell, M. 1998. *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press.

Myrvold, W., and Ruskey, F. 2001. Ranking and unranking permutations in linear time. *Information Processing Letters* 79:281–284.

Riedl, M. O., and Young, R. M. 2010. Narrative planning: Balancing plot and character. *Journal of Artificial Intelligence Research*.

Russell, S. J., and Norvig, P. 2009. Artificial intelligence: a modern approach (3rd edition).

Schaeffer, J.; Björnsson, Y.; Burch, N.; Lake, R.; Lu, P.; and Sutphen, S. 2004. Building the checkers 10-piece endgame databases. In *Advances in Computer Games*. Springer US. 193–210.

Shaker, N.; Nicolau, M.; Yannakakis, G. N.; Togelius, J.; and O'Neill, M. 2012. Evolving levels for super mario bros using grammatical evolution. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, 304–311. IEEE.

Shaker, N.; Smith, G.; and Yannakakis, G. N. 2016. Evaluating content generators. In Shaker, N.; Togelius, J.; and Nelson, M. J., eds., *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer. 215–224.

Shamir, A. 1979. How to share a secret. *Commun. ACM* 22(11):612–613.

Smith, A. M., and Mateas, M. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3):187–200.

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)*.

Sturtevant, N. 2013. An argument for large-scale breadth-first search for game design and content generation via a case study of fling! In *AI in the Game Design Process (AIIDE workshop)*, 28–33.

Summerville, A.; Snodgrass, S.; Guzdial, M.; Holmgård, C.; Hoover, A. K.; Isaksen, A.; Nealen, A.; and Togelius, J. 2017. Procedural content generation via machine learning (PCGML). *CoRR* abs/1702.00539.

Togelius, J.; Preuss, M.; Beume, N.; Wessing, S.; Hagelbäck, J.; and Yannakakis, G. N. 2010. Multiobjective exploration of the starcraft map space. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, 265–272. IEEE.

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.

Togelius, J.; Shaker, N.; and Nelson, M. J. 2016. Introduction. In Shaker, N.; Togelius, J.; and Nelson, M. J., eds., *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer. 1–15.

Zhou, R., and Hansen, E. 2004. Structured duplicate detection in external-memory graph search. In *National Conference on Artificial Intelligence (AAAI-04)*, 683–689.