State Representation and Polyomino Placement for the Game Patchwork

Mikael Zayenz Lagerkvist[1][0000−0003−2451−4834]

`research@zayenz.se`
`https://zayenz.se`

Abstract. Modern board games are a rich source of entertainment for many people, but also contain interesting and challenging structures for game playing research and implementing game playing agents.
This paper studies the game Patchwork, a two player strategy game using polyomino tile drafting and placement. The core polyomino placement mechanic is implemented in a constraint model using `regular` constraints, extending and improving the model in [8] with: explicit rotation handling; optional placements; and new constraints for resource usage.
Crucial for implementing good game playing agents is to have great heuristics for guiding the search when faced with large branching factors. This paper divides placing tiles into two parts: a *policy* used for placing parts and an *evaluation* used to select among different placements. Policies are designed based on classical packing literature as well as common standard constraint programming heuristics. For evaluation, *global propagation guided regret* is introduced, choosing placements based on not ruling out later placements.
Extensive evaluations are performed, showing the importance of using a good evaluation and that the proposed *global propagation guided regret* is a very effective guide.

Keywords: AI · Constraint programming · Games · Polyomino · Packing

1 Introduction

Game playing has long been a core area of traditional AI research. While classical board games such as Chess and Go are computationally very hard, they are also somewhat simple in their actual game play mechanics and representations. Modern board games, such as Settlers of Catan, Ticket to Ride, Agricola, etc. are a rich source of entertainment and creativity that often include complicated game states and complex rules. The more complicated game states, rules, and interactions represent a challenge for implementing game playing logic. In particular, the branching factor is often very large, and at the same time the complicated states make each step much more computationally expensive.

Patchwork [15] by Uwe Rosenberg is a two player strategy game that uses polyomino tile drafting and placement. A polyomino is a geometric form formed by joining one or more squares edge to edge, for example as in Tetris. It was released in 2014, and is in the top 100 games on Board Game Geek [3]. The game is simple to describe for a human, but the central tile placement mechanic is non-trivial to implement in an effective and correct manner. This paper shows how to implement the tile placement sub-problem using constraint programming.

Game playing AI typically uses a tree-based search such as Minimax, Alpha-Beta, or Monte-Carlo Tree Search. For reaching good performance, it is crucial to have heuristics for guiding the tree search into promising parts of the search tree quickly, AlphaGo [17] is a prime example of this. In Patchwork, this requires having good heuristics for packing polyominoes. This paper defines a *strategy* as a combination of a *policy* used for creating placements of the tiles and an *evaluation* that chooses among the generated placements.

Policies are designed based on classical packing literature, in particular the Bottom-Left strategy from [1]. In addition and as contrast, standard constraint programming heuristics are also used for placement policies, both simple heuristics such as first-fail as well as more advanced modern heuristics such as accumulated failure count/weighted degree [4]. For evaluation, *global propagation guided regret* is introduced, essentially choosing placements based on not ruling out later placements. This is contrasted with more obvious measures such as packing to the left or towards a corner.

Contributions This paper proposes using constraint programming for implementing parts of complex game states in modern board games. It extends and improves a previous polyomino placement model using `regular` constraints, including optional placements and explicit handling of transformations. A formulation for exact resource usage for tiles using `regular` constraints is also introduced. For guiding search, a new and straight-forward heuristic called *propagation guided global regret* is designed, that uses look-ahead and the results of propagation to guide search. Extensive evaluation of the model and proposed heuristics is done.

Plan of paper In the next section, some background on constraint programming and AI for game play is given. In Sect. 3 the game Patchwork is described in detail. Sect. 4 introduces the model for the core placement problem, and the following section introduces the placement heuristics developed. The heuristics are evaluated in Sect. 6, and finally some conclusions and directions for future work are given.

2 Background

This paper is concerned with implementing a representation of a part of a game state using constraint programming. To give the context, some general background on both game playing AI and constraint programming is needed.

2.1 Game playing

Game playing is a branch of AI where agents interact following a certain set of rules. Common board games are typically *discrete* and *sequential*. The number of potential actions in each step is the *branching factor* (b), and the number of actions taken by each agent during a game is the number of *plies* (d). The full tree defined by the potential actions is typically very large, in the order of $O(b^d)$.

Game playing is most often implemented using heuristic state space exploring game tree search. Typical classical examples are Minimax and Alpha-beta pruning. More recently, Monte-Carlo Tree Search (MCTS) has become very influential. For a survey of MCTS methods and results, see [5].

A core issue in implementing a game playing AI system is to represent the game state. Key requirements are *correctness*, *speed*, and *memory size*. Classical games such as Chess, Go, and Othello/Reversi have fairly simple game states, and much effort has been in creating very small and efficient representations. Modern board games in contrast have more complex state spaces and rules, which complicate game state implementation. Some recent examples of implementing game AI for modern board games include Settlers of Catan [19], Scotland Yard [11], 7 Wonders [14], and Kingdomino [7].

2.2 Constraint programming

Constraint programming is a method for modeling and solving combinatorial (optimization) problems. Modeling problems with constraint programming is done by defining the variables of the problem and the relations, called constraints, that must hold between these variables for them to represent a solution. A key feature is that variables have finite domains of possible values.

Constraints can be simple logical and arithmetic constraints as well as complicated global or structural constraints. Of particular interest for this paper is the the `regular` constraint introduced by Pesant [12], where a specification for a regular language (a regular expression or a finite automaton) is used to constraint a sequence of variables.

3 Patchwork

Patchwork [15] is the first game in a series of games by Uwe Rosenberg that uses placing polyominoes as a core game mechanic. Patchwork is the simplest of these games, with the polyomino placement being front and center to the game play. It is a top-ranking game on the Board Game Geek website, at place 64 out of more than a hundred thousand entries [3] in July 2019, around five years after the original release.

3.1 Rules

Patchwork is a two-player game with perfect information. The game consists of 33 polyominoes called *patches*, a marker, currency markers called *buttons*, two

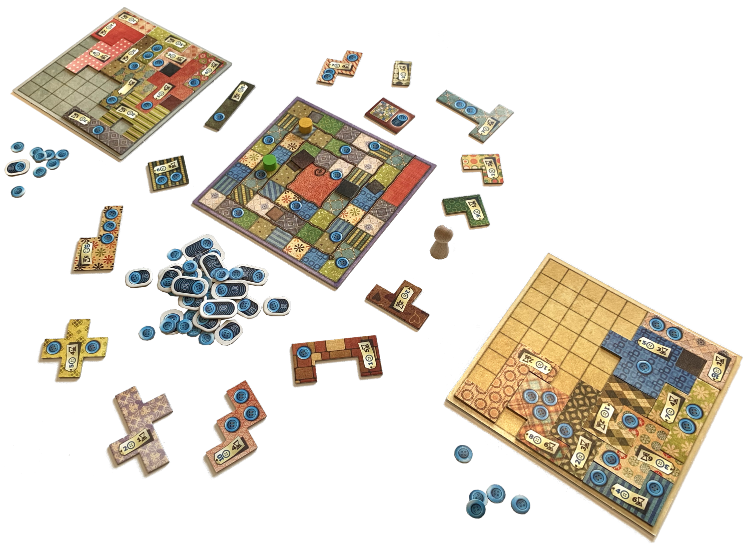

Fig. 1. Patchwork game in progress

player tokens, two 9 by 9 boards (one per player) where patches are to be placed, and a central time board. Each patch has a button cost, a time cost, and between 0 and 3 buttons for income. The time board has 53 steps in total. Five of the steps have a special 1 by 1 patch, and 9 steps are marked for income.

At the start of the game, the patches are organized in a circle, with a marker after the smallest patch. The player tokens are at the start of the time board, and each player has an empty board and 5 buttons. The game is scored based on the number of buttons gained and squares covered for each player at the end of the game.

In each turn, the player whose marker is the furthest back gets to move (if both players are at the same place, the last to arrive there gets to play). When making a move, a player may either advance their time marker to the step after the other player or buy and place a patch. Advancing the time marker to the step after the opponents time token is always possible, and gives the number of steps in button income.

To buy and place a patch, the player may choose one of the three next patches in the circle. The player must pay the indicated number of buttons on the patch (between 0 and 10) and place the patch on their board. The marker is moved to the place of the bought patch, and the player token is moved the number of steps indicated on the patch (between 1 and 6). To buy a patch, the player must have sufficient funds and the ability to place the patch on their board.

If the player when advancing passes a step marked for income on the time board, they collect new buttons based on the number of income buttons on the patches they have placed. If the player is the first to pass one of the special 1 by 1 patches on the time board, they get the patch and place it on their board immediately.

The game ends when both players have reached the center of the time board. The final score is the buttons they have acquired, minus two for each uncovered square on the board. The first player (if any) that filled a complete 7 by 7 area of their board gets an extra 7 points. For a full description of the rules, see [15].

3.2 Strategy

The game requires balancing income, filling squares, and placing patches to not disallow future patch placements. The time remaining for a player can be viewed as a resource that is spent when making moves.

Initially the player has $5 - 2 \cdot 9 \cdot 9 = -157$ points. If no patches are purchased (only advancing the time marker), the final score for the player would be $-157 + 53 = -104$. A normal good score for patchwork is positive, leading to a goal of earning at least 3 points per square advanced ($157/53 \approx 2.96$).

Each patch can be evaluated in isolation for the total change in score that patch would give. There are four main things to consider for a patch P:

- Size S_P, the number of squares the patch will cover.
- Button cost C_P, the number of buttons to pay for the patch.
- Time cost T_P, the number of steps to advance the time marker.
- Button income B_P, the number of buttons to collect at income spots.

The total income gained by the end of the game when buying patch P at time t is determined by the number of remaining button income spots on the board $I(t)$ (starting at 9 and mostly evenly spaced out).

Assuming that we are interested in maximizing our point gain per time used for patch P at time t, the following formula can be used

$$G(P,t) = \frac{2 \cdot S_P - C_P + I(t) \cdot B_P}{\min(T_P, 53 - t)} \tag{1}$$

Two additional issues relevant to the above evaluation are the 7-by-7 bonus and the size 1 squares on the board. The bonus is a property that needs to be planned for, so we leave it to the planning. The 5 size 1 squares are each on their own worth 2 points and can easily be added when evaluating G, but depend on the current game state. Their main use however is in either completing a 7-by-7 square or *preventing* the opponent from completing their own 7-by-7 square. Again, we leave this out of the basic evaluation since it is question of planning and not static evaluation.

3.3 Characteristics

The game is as mentioned a 2-player game with perfect information. The game set-up includes a shuffle of the polyominoes giving $32! \approx 2.6 \cdot 10^{35}$ different games.

The branching factor can be quite large. The first part is the choice of either advancing the time marker or buying one of the three next patches (if possible). Given that a patch is bought, it must be placed on the board. For example, in the initial step, a 2 by 3 L-part can be placed in 42 different positions for each of the 8 symmetries of the patch.

The average branching factor is experimentally found by running 100 random games. Both players use a simple heuristic, by choosing tiles based on Equation 1 and placing tiles using *BL-Every* and *Regret* (see Sections 5 and 6), In this setting, the game averages 23.2 plies for the first player and 23 plies for the second player, with an average branching factor of 83.2. There is also a slight first-player advantage in this setting, with the first player winning 56 games.

4 Placement model

Core for implementing a game state for Patchwork is the players individual boards where patches are placed. Implementing placement and packing from scratch is complicated and error-prone. We use constraint programming to quickly and reliably implement the packing part of the game state.

The model for placing parts on the board builds upon the placement model for polyominoes introduced in [8]. The model uses `regular` constraints to specify the required placement of a patch on a board. First the original model is explained briefly, and then the additions for Patchwork are given.

4.1 Original model

Consider the placement of the patch in Figure 2 in the 4-by-4 grid shown. Each square is represented by a 0/1-variable (1 meaning the patch covers the square), and the grid is encoded in row-major order. All placements of the patch are encoded by the regular expression $0^*110^310^310^*$. This encoding includes all valid placements, but it also includes some invalid ones (see right side in Figure 3). To forbid such placements, an extra column of dummy squares is added that are fixed to 0 as shown in Figure 4, and the expression is changed to $0^*110^410^410^*$. Rotations and flips are handled by combining the regular expressions for each transform using disjunction, relying on the DFA minimization for removing states representing equal rotations.

Unique sets B_p of 0/1 variables are used for each patch p. The variables for different patches are connected with integer variables B with domain values representing patches, empty squares, and the end column: $P \cup \text{empty} \cup \text{end}$. While the constraint can be defined directly on the B variables, [8] showed that using the auxiliary B_p variables performs much better.

4.2 Extensions

The original model is not enough for implementing a model for Patchwork. In particular, optional placements and control of transformations is needed, as is measuring usage.

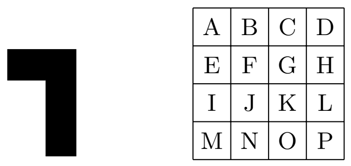

Fig. 2. Part to place and the grid to place on.

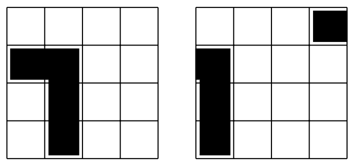

Fig. 3. A valid (left) and an erroneous (right) placement

Transforms For heuristics, it can be useful to branch on which, if any, of the up to 8 transforms of a patch is placed (transforms include rotations and flips of the patches). In order to do this, the unique transforms are generated beforehand in the model set-up code, instead of always generating all 8 transforms and using DFA minimization to handle symmetries. To encode the different rotations, 8 Boolean control values S_{ps} for patch p in each possible transform s are prepended to each placement expression, with exactly 1 of the variables set to 1 for each transform. Note that for simplicity the same number of Boolean variables are prepended, regardless of the number of non-symmetric transforms generated.

Optional placements In Patchwork, not all patches are placed on the board. While it is possible to dynamically add variables and constraints during solving/searching, it is better to set up all constraints for all patches from the start and controlling placement of a patch using reification. Given a patch p with board variables B_p, placement expression R_p for all transforms, a 0/1 control variable U_p indicating if the patch is to be placed, the 8 S_p Boolean variables indicating which transform is used, and a Boolean variable N_p indicating that no transform was used, the following constraint is posted:

$$\texttt{regular}((10R_p) \,|\, (010^*),\ U_p N_p S_p B_p) \tag{2}$$

This is a domain-consistent full reification of the placement constraint; when a patch can no longer be placed the propagation will set U_p to 0. Note that the second value corresponds to no transform chosen. This is so that the Boolean variables $N_p S_p$ can be channeled into a single integer variable with domain 0..8

Usage It is well-known that cumulative usage reasoning can be very effective for packing problems [18]. A first step towards cumulative reasoning is to model resource usage. Usage reasoning *per patch* is added to the model using regular expressions. Variables C^{Σ}_{pc} for column sums and R^{Σ}_{pr} for row sums indicating the

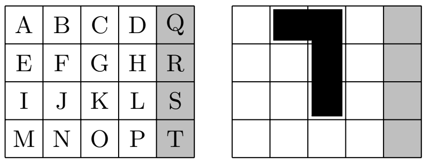

Fig. 4. Grid extended with dummy column, and placement in new grid.

number of ones in each column c and row r for patch p in B_p. Consider again the patch in Figure 4: the first column it is placed in has a single one and the second column has three ones, while it has usage two, one, and one for its rows. This means that C^{Σ}_{p} belongs to the language $0^{*}130^{*}$ and R^{Σ}_{p} belongs to the language $0^{*}2110^{*}$. Concatenating the column (including the dummy column) and row usages into $C^{\Sigma}_{p} R^{\Sigma}_{p}$, all usages for all placements belong to the language $0^{*}130^{*}00^{*}2110^{*}$. Using the same infrastructure for symmetries, pre-fixing with the control variable U_p and transform S_p gives a domain consistent full reification for the usage constraint for a single patch.

5 Placing patches on the board

Armed with a model that expresses the underlying placement model, heuristics are needed that can be used when a player chooses a patch and needs to place it on the board. It is important to remember here that in the context of game playing (both for MCTS-style roll-outs and for traditional tree search), a patch needs to be fully placed, without deciding on any of the other parts concurrently. Also important, there is no possibility of a back-track.

We call the overall heuristic used to generate a placement on the board a *strategy*. The strategy is divided into two independent parts, a *policy* that generates placements of a tile, and an *evaluation* that chooses among different placements. The concept of a placement policy is common in packing literature. The policy is typically incomplete; most policies can not be used to find a guaranteed optimal packing.

5.1 Placement policy

A *policy* in this paper is is a method used to place a tile on a partially filled board in some manner, potentially generating many different placements. Policies can be implemented using standard constraint programming branching and search. Which placement that is used is not something the policy needs to consider. A central goal for a policy is to cheaply produce some good placements of a part. In constraint programming terms, a policy can be regarded as an incomplete large-step heuristic branching without ordering of the alternatives produced.

Recall from Section 4 that a patch p has 0/1 variables B_p representing the placement of that patch, variables C^{Σ}_{pc} and R^{Σ}_{pr} representing the number of occupied squares in each column and row for the patch, and control variables S_p representing the transform of the patch.

Bottom-Left One of the most common heuristics for placement problems is the Bottom-Left heuristic [1] (BL), in which each part is placed in the bottom-most position, and among all bottom-most positions the left-most is chosen. Worth noting is that in the context of a symmetric and bounded board the direction is arbitrary (i.e., Right-Top is the same as Bottom-Left, and so on), compared to the original context where packing was on a roll of material that was infinite in the top/up direction.

To implement the policy, integer variables representing the first occupied row R_p^f (and C_p^f for columns) are needed. For patches that are not placed on a board the sentinel value of one past the end is used. The row variable is defined using the following schema for the patch p using the 8 row sum variables R_{pr}^{Σ} (column variables defined similarly).

$$R_p^f \in \{0..9\},\ Z_{0..8} \in \{0,1\},\ F_{0..8} \in \{0,1\} \tag{3}$$

$$\forall_{i=0..8}\ Z_i \Leftrightarrow R_{pi}^{\Sigma} = 0 \tag{4}$$

$$Z_0 = F_0,\ \forall_{i=1..8}\ F_i \Leftrightarrow F_{i-1} \wedge Z_i \tag{5}$$

$$R_p^f = \sum_{i=0..8} F_i \tag{6}$$

The Z variables indicate if there is no placement in a row, and the F variables define if an index is part of the first run of zero variables. Summing up F gives the first index, or 9 (one past the end) if there is no placement. This is somewhat similar to the decomposition for `length_first_sequence` in the Global Constraint Catalogue [2].

Using these variables standard constraint programming variable/value heuristics can be used to implement Bottom-Left for the placement of patch p. Branchings are added on first C_p^f and then R_p^f trying the smallest value first. A third branching on the B_p variables is added with in-order true-first choices. Searching for the first solution using DFS will give the Bottom-Left placement of the patch in some transformation.

BL/LB Building on the standard BL heuristic, it is easy to extend to a heuristic that places in both Bottom-Left and Left-Bottom order producing two results. Two individual DFS searches are made, one with branching first on C_p^f and then on R_p^f, and one with the order of the branchings reversed. In both cases the B_p branching is kept the same.

Pareto BL Pareto Bottom-Left is a generalization of Bottom-Left, where instead of only finding a solution for a single value in one direction, we try all values in that direction. This corresponds to testing all columns as the minimum column and finding the minimum row placement for each. The minimum row/column values for the placements found $((r_i, c_i))$ form a Pareto front under the natural point wise order relation, thus the name,

In practice, to limit the amount of branching in the beginning and keep solutions heuristically relevant, not all columns are tested for placement. The maximum column with some fixed placement on the board $c_{\max}$ is found before search, and only columns in $0..c_{\max}+1$ are used. For a partial packing that is still close to a corner, this means that non-needed placements close to the other corner are not tested.

Given a set of columns to test, the implementation manually assigns the C_p^f to each values in $0..c_{\max}+1$. After this step, for each column the search proceeds as for Bottom-Left.

Policies based on standard heuristics There is a rich area of research into general heuristics for constraint programming. We use five different general heuristics that are available in Gecode, the system used for implementation. The heuristic values used are:

In Order Use the order of the variables. This is actually quite close to a Bottom-Left style heuristic, but instead of working on the bounding box of the patch, it works on the individual squares.

Size The domain size for variables. Commonly called *First fail*.

AFC The accumulated failure count [4] (also known as *weighted degree*) is the sum of all the times propagators connected to the variable have failed a search tree.

Action Action is the number of times the domain of a variable has been reduced [10] (also known as *activity*).

CHB CHB (Conflict History-based Branching, [9, 16]) uses a combination of domain reduction counts and when failures occur.

The branching is done on the B_p Boolean variables, but for all but the simplest (*In Order*), the heuristic values would not make much sense on the B_p variables: the domain size is always 2 and there is typically not much activity on the Boolean variables. The base version is to use the heuristic value associated with the corresponding B variable that represents all the patches. For *AFC*, *Activity*, and *CHB* variants are also defined that sum the heuristic value over both the corresponding B variable and the corresponding B_t variables for *all* patches t (including the current patch p). The summed variants are called $\sum$*AFC*, $\sum$*Activity*, and $\sum$*CHB*.

For *AFC*, *Activity*, *CHB*, and their summed variants, the heuristic values that are used in the policies are divided by the domain size of the variable, as is common. This is denoted *X/Size* for the measure X.

For all the standard heuristics, equal values use the order of the variables as a tie-breaker.

Every transform The above heuristics all produce a result in some transformation, without actually making any specific choice. For all the heuristics, variants that create placements for all possible rotations of a patch are also tested.

To implement an *every transform* variant for a placement policy, all possible values for the rotation variable U_p are assigned first. For each assignment, the base policy is applied to generate a placement of the patch.

All Finally, just generating all the possible placement and letting the evaluation make the choice is possible. This is simply implemented as a standard DFS search for all solutions when branching over the B_p variables. Note that *All* will naturally always generate solutions for all rotations, so it does not make sens to use the *every transform* modification.

5.2 Placement evaluation

Many of the policies produce more than one alternative. To choose among the various alternatives, an *evaluation* is used that takes multiple placements and returns the "best" one. In constraint programming terms, an evaluation can be seen as an ordering of the alternatives produced by a branching, so that a left-most exploration order would follow the evaluations heuristic. Compared to many constraint programming branchings, the evaluation expects the alternatives to be of equal type; they would not work for classical $c \vee \neg c$ two-way branching such as $x = d \vee x \neq d$.

Each placement evaluated by the evaluation is represented by the full search state with the patch placed and all the other board variables present and all constraints fully propagated. Also, the evaluations have access to the state before the placement, for comparisons.

First and Random The evaluation *First* simply chooses the first alternative. This is not a meaningful evaluation (except for heuristics that only generate one placement), but is interesting as a baseline. Similarly, the *Random* evaluation chooses one of the alternatives at random.

Bottom. Left, and Area A placement of a patch has a maximum extent to the right and to the top. The *Left* evaluation chooses the placement with minimum right extent, while the *Bottom* evaluation chooses the minimum top extent. The *Area* evaluation measures the increase in the bounding box of the fixed placements, promoting placements that are kept tight to a corner.

Propagation Guided Global Regret When placing a patch, propagation will remove possible placements for other patches. This is a valuable signal on how many possibilities we have left, and can be used to guide the search.

We call this the *Propagation Guided Global Regret*, since it uses propagation to give an indication of the effect of a choice globally. More formally, we define it as follows. Let the original variables for the whole board be B, and the variables after a placement and propagation of that placement be B'. The expression B_{ij} represents the square at indexes i and j, and $|B_{ij}|$ represents the domain size of the variable. The patch to place is named p. The heuristic value is defined by the following summation.

$$\mathrm{pggr}(B, B', p) = \sum_{i=0}^{8}\sum_{j=0}^{8} \begin{cases} 0 & \text{if } |B_{ij}| = 1 \\ 0 & \text{if } B'_{ij} = p \\ |B_{ij}| - |B'_{ij}| & \text{otherwise} \end{cases} \tag{7}$$

Regret is a well-known heuristic in constraint programming, and is usually defined as the difference between the minimum and next to minimum (and similarly for maximum) value in a domain. This is mostly useful when there is a direct connection between variable domain values and some optimization criteria. However, the concept of regret is more general than the typical single-variable domain centric value optimizing view. Here we lift the concept to be with respect to keeping possible future options open.

In a sense, propagation guided global regret is the anti-thesis of Impact based search [13], where variable/value pairs are chosen based on the amount of propagation that they trigger historically. The context here is very different though, with the amount of propagation used is a negative signal. In addition, it is used for the current assignment using look-ahead, and not based on statistics of historic values.

6 Experimental evaluation

This section reports results of experimental evaluation to clarify how the different strategies formed by combinations of the policies and evaluations introduced perform.

6.1 Implementation and Execution Environment

The implementation consists of a game state implemented in C++17 with the placement model implemented using the Gecode [6] constraint programming system version 6.2.0. The code is single-threaded only. While it would be possible to use multi-threaded search, single-threaded execution gives a simple and more stable evaluation. The implementation is available at `https://github.com/zayenz/cp-mod-ref-2019-patchwork`. All experiments are run on a Macbook Pro 15 with a 6-core 2.7 GHz Intel Core i7 processor and 16 GiB memory.

For the learning heuristics (those that use *AFC*, *Activity*, and *CHB*) the statistics they are based on are collected throughout the whole experiment, giving them their best possible chance of learning interesting aspects.

6.2 Core packing problem

The central aspect in the game play is that a sequence of patches are chosen and placed on the board. Each placement must be done without knowing which future patches will be placed. A pure packing problem that captures this is to order all the patches, and to test placing each patch in turn, incrementally building up a patchwork.

In table 1 results are shown for testing the same 1000 random orders of parts on 119 strategies. The strategies are combinations of policies with *Some* or *Every* transformation matched with different evaluations. Each strategy has four metrics. For each metric, the best value is marked with dark blue, those within 1% of the best value are marked with medium blue, and those within 5% are marked with light blue background. The metrics are

Area The mean amount of area placed after all patches have been placed. This is the main metric in the game, and is a natural metric on the power of a strategy to make good placements without knowing what patches to try next.

Streak The mean number of patches placed before the first failure. This is a measure of how much the strategy manages to keep options open, but does not necessarily correspond to the amount of area placed.

Time The mean time spent making a single placement of a patch in milliseconds.

Alts The mean number of alternatives produced for each patch.

Note that for strategies that only create one alternative, the evaluation does not matter, but is tested for completeness. The evaluation *ReverseRegret* (choosing the maximum instead of minimum regret) is added to show the behaviour when explicitly going against the intuition that keeping options open for packing is a good idea.

For all learning heuristics, only the version with the heuristic value divided by domain size is tested. We do not show the results for $\sum AFC$, $\sum Activity$, and *CHB* since they were slightly worse than their corresponding summed/non-summed variants. The full data is available on request.

6.3 Results of packing experiments

The most striking observation, is that using propagation guided global regret is clearly the most important factor in maximizing the amount of area placed. The fact that regret is best when trying all placements is a strong indication that it is an evaluation that can truly guide the placement, and not just choose among a set of probably good placements. The reverse of regret is clearly the worst among all evaluations, validating the assumption that regret is a good measure.

Among the other evaluations, it is worth noting that when using a heuristic that places first bottom then left, it is better to use an evaluation that agrees with the direction placements are made in (that is, for Bottom-Left it is better to make choices based on bottomness than leftness).

Given a reasonable evaluation (that is, not *First*, *Random*, nor *ReverseRegret*), the policies based on packing heuristics are better than generic constraint programming heuristics. The most interesting stand-out here is *In Order*, since it combines a very simple constraint programming definition and speed of evaluation with actually being similar in effect to classical packing heuristics.

In implementing game tree search, the time used for expanding nodes is crucial to get reasonable performance, since there is typically a hard limit on

		In Order		Size		AFC/Size		Action/Size		∑CHB/Size		BL		BL/LB		Pareto BL		All
		Some	Every	Some	Every	Some	Every	Some	Every	Some	Every	Some	Every	Some	Every	Some	Every	Every
First	Area	76.51	72.25	66.49	65.95	73.73	71.68	74.97	71.70	74.98	72.16	76.12	72.28	76.12	72.28	76.12	73.75	76.51
	Streak	15.79	14.06	12.22	12.50	14.50	13.72	15.52	14.15	14.85	13.72	15.62	14.46	15.62	14.46	15.62	14.88	15.79
	Time	25.37	86.90	24.22	83.72	24.72	87.86	25.16	88.90	25.01	88.90	162.30	316.35	217.67	501.64	722.31	1669.19	1815.18
	Alts	1.00	2.99	1.00	2.95	1.00	2.97	1.00	3.03	1.00	2.97	1.00	3.04	1.94	6.02	3.64	9.62	71.86
Random	Area	76.51	72.25	66.49	66.18	73.73	71.63	74.97	71.50	74.98	72.08	76.12	72.35	75.43	71.91	72.72	70.62	69.49
	Streak	15.79	14.26	12.22	12.27	14.50	13.60	15.51	14.02	14.85	13.71	15.62	14.36	15.66	14.18	14.19	13.10	12.29
	Time	26.03	89.09	24.15	82.85	24.76	87.68	25.11	88.47	25.05	88.42	161.86	317.08	210.55	497.82	1208.40	2793.32	1197.45
	Alts	1.00	3.02	1.00	2.91	1.00	2.98	1.00	3.02	1.00	2.96	1.00	3.02	1.94	5.99	4.87	11.22	51.41
Left	Area	76.51	72.79	66.49	66.05	73.73	72.03	74.99	71.91	74.97	72.48	76.12	73.47	76.11	73.47	75.93	74.47	75.08
	Streak	15.79	14.08	12.22	12.18	14.50	13.70	15.52	14.22	14.85	13.97	15.62	14.91	15.62	14.91	15.51	15.13	14.95
	Time	25.24	88.13	24.13	82.60	24.75	88.46	25.11	89.28	24.98	89.49	162.11	303.51	217.95	447.45	726.92	1127.38	1657.31
	Alts	1.00	3.00	1.00	2.91	1.00	2.98	1.00	3.03	1.00	2.97	1.00	3.05	1.94	6.05	3.60	7.95	65.69
Bottom	Area	76.51	72.25	66.49	65.95	73.73	71.68	75.00	71.74	74.98	72.16	76.12	72.28	76.12	72.28	76.12	73.75	76.51
	Streak	15.79	14.06	12.22	12.50	14.50	13.72	15.54	14.15	14.85	13.72	15.62	14.46	15.62	14.46	15.62	14.88	15.79
	Time	25.30	86.58	24.13	83.87	24.67	87.80	25.13	89.22	25.03	88.71	162.03	315.70	217.42	501.39	722.04	1668.19	1815.77
	Alts	1.00	2.99	1.00	2.95	1.00	2.97	1.00	3.03	1.00	2.97	1.00	3.04	1.94	6.02	3.64	9.62	71.86
Area	Area	76.51	72.79	66.49	66.05	73.73	72.03	74.98	71.91	74.98	72.48	76.12	73.47	76.11	73.47	75.93	74.47	75.08
	Streak	15.79	14.08	12.22	12.18	14.50	13.70	15.51	14.24	14.85	13.97	15.62	14.91	15.62	14.91	15.51	15.13	14.95
	Time	25.26	88.03	24.16	82.82	24.67	88.14	25.12	89.41	25.07	89.45	161.79	303.60	217.66	447.02	727.14	1129.46	1652.75
	Alts	1.00	3.00	1.00	2.91	1.00	2.98	1.00	3.03	1.00	2.97	1.00	3.05	1.94	6.05	3.60	7.95	65.69
Regret	Area	76.51	77.13	66.49	69.67	73.73	75.11	75.00	76.23	74.98	75.75	76.12	77.20	77.21	78.21	77.86	78.36	78.40
	Streak	15.79	16.21	12.22	13.76	14.50	15.36	15.56	15.99	14.85	15.52	15.62	16.24	16.39	16.76	16.43	16.65	16.56
	Time	25.29	96.33	24.23	89.96	24.71	95.29	25.06	98.15	25.03	96.50	161.95	355.38	223.62	606.60	1426.38	3550.62	1464.55
	Alts	1.00	3.17	1.00	3.03	1.00	3.12	1.00	3.21	1.00	3.11	1.00	3.17	1.94	6.46	4.76	12.39	59.48
ReverseRegret	Area	76.51	68.91	66.49	63.34	75.14	68.97	74.12	68.54	74.78	69.69	76.12	68.75	73.34	67.57	67.46	64.70	62.73
	Streak	15.79	12.82	12.22	11.03	15.46	12.68	14.90	12.39	14.96	12.34	15.62	12.80	14.93	12.32	11.59	10.57	9.49
	Time	26.64	84.63	24.81	79.56	26.24	84.72	25.72	83.81	25.58	83.63	164.97	278.77	187.41	404.47	1174.86	2116.79	1195.69
	Alts	1.00	2.92	1.00	2.82	1.00	2.93	1.00	2.90	1.00	2.87	1.00	2.92	1.94	5.72	4.94	9.82	52.05

Table 1. Results for combinations of policy and evaluation for the core packing problem. Best values for each metric are indicated with blue, darker being better.

the deliberation time of an agent. It is clear that many of the strategies take much more time than would be feasible. Some simple profiling of the code indicates that there is some overhead in the implementation in how many clones are generated of the Gecode search spaces, which could potentially be optimized. Parallel execution is also a clear potential for improving time.

In game play, combinations of strategies can be used. For example, when implementing MCTS it is common to use different strategies for the tree expansion and for the roll outs. For tree expansion, quality of moves and strong ordering is important, while roll-outs have very strong speed requirements. A combination of a more expensive tree expansion such as *BL/LB-Every + Regret*, and a simple and fast roll-out strategy such as *In Order-Some + First* could be useful.

In conclusion, the most important decision is to use propagation guided global regret to choose among possible placements, with the choice of policy guided by the time-requirements needed.

7 Conclusions

This paper has introduced the use of constraint programming for representing parts of a game state for the game Patchwork. The model represents a packing of an unknown subset of polyominoes, that are to be chosen during game tree search. The packing model extended and improved a previous polyomino packing model based on `regular` constraints. The use of constraint programming simplified the task of implementing the packing part of the game state, with a high-level specification.

To guide the search, several new strategies were developed for placing patches. Placement policies inspired by classical packing literature are shown to be good. For choosing among different placements, the concept of propagation guided global regret was introduced and shown to be very effective in guiding search towards good placements of patches.

Future work This paper has focused on the strategies used for placing polyominoes when implementing the game Patchwork. The natural next step is to also apply the model in game playing agents. An investigation into the relative complexity of implementing the packing model without the support of a constraint programming system would also be interesting. In particular, something like global propagation guided regret would be very hard to implement by hand.

Acknowledgments

Thanks to Magnus Gedda, Magnus Rattfeldt, and Martin Butler for interesting and informative discussions on games and search methods. Thank the anonymous reviewers for their comments that helped improve this paper.

References

1. Baker, B.S., Coffman Jr., E.G., Rivest, R.L.: Orthogonal packings in two dimensions. SIAM J. Comput. **9**(4), 846–855 (1980). https://doi.org/10.1137/0209064, `https://doi.org/10.1137/0209064`
2. Beldiceanu, N., Carlsson, M., Rampon, J.X., Demassey, S., Petite, T.: Global constraint catalogue (2014), `http://sofdem.github.io/gccat/gccat/index.html`, Accessed on 2019-05-01
3. Board Game Geek: Toplist (2019), `https://boardgamegeek.com/browse/boardgame`, [Online; accessed 2019-07-09]
4. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: de Mántaras, R.L., Saitta, L. (eds.) ECAI. pp. 146–150. IOS Press (2004)
5. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games **4**(1), 1–43 (2012)
6. Gecode team: Gecode, the generic constraint development environment (2018), `http://www.gecode.org/`
7. Gedda, M., Lagerkvist, M.Z., Butler, M.: Monte carlo methods for the game kingdomino. In: 2018 IEEE Conference on Computational Intelligence and Games, CIG 2018, Maastricht, The Netherlands, August 14-17, 2018. pp. 1–8. IEEE (2018). https://doi.org/10.1109/CIG.2018.8490419, `https://doi.org/10.1109/CIG.2018.8490419`
8. Lagerkvist, M.Z., Pesant, G.: Modeling irregular shape placement problems with regular constraints. In: First Workshop on Bin Packing and Placement Constraints BPPC'08 (2008), `http://www.gecode.org/paper.html?id=LagerkvistPesant:BPPC:2008`
9. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Exponential recency weighted average branching heuristic for sat solvers. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. pp. 3434–3440. AAAI'16, AAAI Press (2016), `http://dl.acm.org/citation.cfm?id=3016100.3016385`
10. Michel, L., Van Hentenryck, P.: Activity-based search for black-box constraint programming solvers. In: Proceedings of the 9th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. pp. 228–243. CPAIOR'12, Springer-Verlag, Berlin, Heidelberg (2012)
11. Nijssen, P., Winands, M.H.: Monte carlo tree search for the hide-and-seek game Scotland Yard. IEEE Transactions on Computational Intelligence and AI in Games **4**(4), 282–294 (2012)
12. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace [20], pp. 482–495
13. Refalo, P.: Impact-based search strategies for constraint programming. In: Wallace [20], pp. 557–571. https://doi.org/10.1007/978-3-540-30201-8_41, `https://doi.org/10.1007/978-3-540-30201-8\_41`
14. Robilliard, D., Fonlupt, C., Teytaud, F.: Monte-carlo tree search for the game of "7 Wonders". In: Workshop on Computer Games. pp. 64–77. Springer (2014)
15. Rosenberg, U.: Patchwork (2014)
16. Schulte, C., Tack, G., Lagerkvist, M.Z.: Modeling and Programming with Gecode (2019), corresponds to Gecode 6.2.0

17. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
18. Simonis, H., O'Sullivan, B.: Search strategies for rectangle packing. In: Stuckey, P.J. (ed.) Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5202, pp. 52–66. Springer (2008). https://doi.org/10.1007/978-3-540-85958-1_4, `https://doi.org/10.1007/978-3-540-85958-1\_4`
19. Szita, I., Chaslot, G., Spronck, P.: Monte-carlo tree search in Settlers of Catan. In: Advances in Computer Games. pp. 21–32. Springer (2009)
20. Wallace, M. (ed.): Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings, Lecture Notes in Computer Science, vol. 3258. Springer (2004)