Proceedings of the Eighteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2022)

Tiered State Expansion in Optimization Crosswords

Adi Botea,[1] **Vadim Bulitko**[2]

[1]Eaton
[2]Department of Computing Science, University of Alberta, Edmonton, AB, Canada

Abstract

Crosswords puzzles continue to be a popular form of entertainment. In Artificial Intelligence (AI), crosswords can be represented as a constraint problem, and attacked with a combinatorial search algorithm. In combinatorial search, the branching factor can play a crucial role on the search space size and thus on the search effort. We introduce tiered state expansion, a completeness-preserving technique to reduce the branching factor. In problems where the successors of a state correspond to the values in the domain of a state variable selected for instantiation, the domain is partitioned into two subsets called tiers. The instantiation of the two tiers is performed at different times, with tier 1 first and tier 2 in a subsequent state. Before a tier-2 instantiation occurs, its set of applicable values can shrink substantially due to constraint propagation, with a corresponding reduction of the branching factor. We apply tiered state expansion to a constraint optimization problem modeled on the Romanian Crosswords Competition, empirically demonstrating a substantial improvement. Tiered state expansion allows finding a full solution, with an average CPU time of up to 1.2 minutes, to many puzzles that would otherwise time out after 4 hours.

1 Introduction

Crosswords puzzles are a very popular form of entertainment. We address a problem inspired by the Romanian Crosswords Competition and introduced in the search literature by Botea and Bulitko (2021). The task is to fill a crosswords grid with words and black cells. Some words, called thematic words, give points. The objective is to obtain as many points as possible. The application has a decades-long history of annual national-level competitions. The problem is challenging to AI, which lags behind the performance of top human contestants. The problem can be represented as a constraint optimization and tackled with a search algorithm. As usual in a search, the branching factor (i.e., the number of successor states to a parent state) can greatly impact the search space size and search time.

We present *tiered state expansion*, an approach to reducing the branching factor. We adopt a few assumptions that characterize an important and broad class of constraint optimization problems. Specifically our states are defined by a finite set of variables with a finite discrete domain each. A state corresponds to an instantiation of zero or more variables. Successors of a state are defined by selecting an uninstantiated variable and instantiating it with some/every of its possible values. Legal states need to satisfy a set of constraints specified in the definition of the problem instance. An objective function maps states to a solution cost (lower is better) or score (higher is better). Constraint propagation can gradually shrink domains of uninstantiated variables based on values of already instantiated variables.

For example, in a crosswords application, variables can correspond to slots, where a slot is defined as a horizontal or vertical sequence of white cells bordered at each end by either the margin of the grid or by a black cell. The domain of such a variable is the set of words that fit into that slot. At a given point in time, a slot (variable) can be instantiated with a word (value) or uninstantiated. A state comprises all slots with their corresponding assignment at that time (i.e., either filled with a word or uninstantiated).

Tiered state expansion partitions the current domain of a variable selected for instantiation into two subsets called *tier 1* and *tier 2*. The two tiers are defined so that a tier-1 subset is likely to contain values that could lead to higher-quality solutions, and a tier-1 subset is preferably significantly smaller than the current domain before splitting. When the variable is selected for expansion for the first time, only tier-1 values are used to generate successor states. An additional, special successor state called a *placeholder successor* leaves the variable uninstantiated but flags it so that any future instantiations (i.e., in a state down in the subtree of the placeholder successor) will use only tier-2 values. Thus, tier-1 and tier-2 instantiations for a given variable are separated during the search and other variable instantiations can occur between those two. By the time tier-2 instantiations would be performed, the number of remaining tier-2 values can be much smaller due to constraint propagation. This can result in a significant reduction of the actual branching factor.

Our algorithmic contributions are applicable to constraint optimization problems that match the assumptions outlined earlier in this section. We require only partitioning a variable domain into two tiers, with tier 1 preferably small, and with tier-1 likely to contain some values that would lead to high-quality solutions.

Our ideas are easy to understand and implement. They can be implemented as extensions to a search algorithm, or as new constraints in a constraint satisfaction problem. For simplicity, we adopt the former approach.

Tiered state expansion is implemented in WOMBAT, an existing state-of-the-art crosswords puzzle solver. Experiments demonstrate substantial speed and memory gains. Consequently, tiered state expansion allows to fully solve puzzle instances that, with standard expansion, would fail after a four-hour search. We made our code and experiments available to accelerate progress in this challenging problem.

2 Related Work

The literature distinguishes between solving crosswords puzzles and several variations of generating crosswords grids. In the former, a problem instance is defined by a grid with black cells but no letters and a list of clues. The task is then to fill the grid with words that match the clues (Littman, Keim, and Shazeer 2002; Ernandes, Angelini, and Gori 2005; Ginsberg 2011; Chen et al. 2022).

The problem of *crosswords grid generation* takes as input a list of words and a grid with only black cells. The task is to fill the grid with words from the list (Mazlack 1976; Ginsberg et al. 1990; Botea 2007; Anbulagan and Botea 2008). The problem has recently been extended with a score function (Botea and Bulitko 2021) and with the additional task to automatically generate black cell configurations as well (Bulitko and Botea 2021). Similarly to Botea and Bulitko (2021), we address a crosswords grid generation problem with a score function and with the black cells given as input. This is what we call the *optimization crosswords problem* in this paper. It is a stepping stone towards the full Romanian Crosswords Competition problem where black cells should be added as part of constructing a solution.

Our work is complementary to the approach by Botea and Bulitko (2021). They perform multiple fast searches, grouped into two phases with the aim that such searches collectively are faster than a standard search. A phase-one search aims at quickly finding a promising partial solution. A phase-two search sets the initial state to the partial solution discovered in phase one, rather than starting from an empty initial state. In contrast, our contribution reduces the branching factor in a search.

Botea and Bulitko (2021) focused on finding high-score black cell configurations for an optimization crosswords problem and mentioned tiered state expansion in passing, as a readily available feature turned on in their experiments. In this paper we describe and evaluate the concept of tiered state expansion in depth.

A simplified version of The Romanian Crosswords Competition problem was posed in the 2018 XCSP Competition (Lecoutre and Roussel 2019). Thirteen instances are featured in the competition, with square grids whose sizes range 3×3 to 15×15. Eight instances remained unsolved (Lecoutre and Roussel 2019). Audemard, Lecoutre, and Maamar (2020) use the same representation as a testbed for using segmented tables to encode constraints.

Domain-independent AI planning has seen the application of concepts such as helpful actions (Hoffmann and Nebel 2001) and preferred operators (Helmert 2006). The idea is to partition the actions applicable in a state into two subsets, with one subset being heuristically considered likely to contain an action that would result in a (good-quality) solution. In the Fast Forward system, unhelpful actions can be pruned away, at the cost of losing the search completeness (Hoffmann and Nebel 2001). In Fast Downward, preferred operators can get a higher priority at expansion, compared to a non-preferred operator (Helmert 2006). A key difference from our work is that tier-2 successors never get generated as children of the original parent state. Instead, their instantiation is considered in another state, deeper in the search tree, at which point the number of tier-2 successors can greatly be reduced due to constraint propagation.

3 Problem Definition

We address the same problem as Botea and Bulitko (2021). It is modeled as a type of constraint optimization obtained from a constraint satisfaction problem (CSP) by adding a score function on partial or full solutions.

Definition 1. A *constraint satisfaction problem (CSP)* is a tuple $\mathcal{P} = \langle X, D, C \rangle$, where:

- $X = \{v_1, \ldots, v_n\}$ is a collection of variables;
- $D = \{D_1, \ldots, D_n\}$ is a collection of finite domains with D_i corresponding to variable v_i;
- $C = \{C_1, \ldots, C_m\}$ is a collection of constraints. Each constraint C_j is a pair $\langle t_j, R_j \rangle$, where $t_j \subset X$ is a subset of p variables and R_j is a p-ary relation on the corresponding subset of domains.

We define D^* as an extension of D that contains a special symbol $\perp$ in each domain for variables with no actual value assigned yet. This allows to represent partial assignments.

Definition 2. An *optimization CSP* is a tuple $\langle \mathcal{P}, f \rangle$, where $\mathcal{P}$ is a CSP and $f : D^* \rightarrow \mathbb{R}^+$ is a score function.

An assignment $s \in D^*$ is *consistent* if it violates no constraint. A consistent partial assignment is called a *partial solution*. A consistent full assignment is a *full solution* (or *solution* for short). A solution is *optimal* if no solution has a higher score.

4 Romanian Crosswords Competition

In this work we apply our ideas to the Romanian Crosswords Competition, a problem introduced in search literature by Botea and Bulitko (2021). We describe it below to make the paper more self-contained.

The Competition is an annual contest with human participants started in 1965. The task is to create a 13×13 grid filled with words and at most 26 black cells. The input includes two lists of words (dictionaries), called the *thematic list* and the *regular list*. The thematic list changes every year and is on the order of a few hundred words. The regular list contains approximately 135 thousand Romanian words. If the two lists overlap, the common words are considered to be thematic and are removed from the regular list. Black cells cannot have common edges but they are allowed to have common corners. White cells have to form a cardinally connected region. Black cells cannot create semiclosures – configurations of black cells such that adding one more black cell would partition the open/white area of the grid into two or more areas with the disconnected areas having more than one open cell each.

Given a configuration of black cells, a *word slot* is a set of contiguous white cells in a row or a column where each end of the slot is adjacent to either the border of the grid or a black cell. Slots of length 1 can be filled with any letter. Slots of length 2 can be filled with any combination of two letters but no two-letter combination can be repeated in the grid. Slots of length 3 or higher can be filled with words from the two lists. For simplicity we say that singleton letters are words of length 1 and two-letter combinations are words of length 2. We treat them as regular words unless such a combination is in the thematic list.

Words of length 2 or higher cannot be repeated in a grid. So-called families of words are forbidden (e.g., writer and writing cannot both be placed on the same grid).[1] All cells in a (full) solution must contain a letter or a black cell. The score of a grid is the sum of the lengths of all thematic words.

Constructing a competition grid requires both adding black cells and words. Similarly to Botea and Bulitko (2021), we assume that the black cells are given as input and address only the task of filling in the words. With the black cells given the remaining problem of filling in the words can be modelled as an optimization CSP. Apart from the optimization component (score function) this is a standard example of a CSP. Word slots are variables and their initial domains are all words of the corresponding length.

5 Background: Optimization CSP as Search

A standard representation of a CSP as a search problem defines states as consistent partial or full variable assignments $s \in D^*$. A fully instantiated state (i.e., a full solution) is a goal state. Frequently the root state is the empty assignment (i.e., with all variables uninstantiated). The successors of a partial-assignment state s are obtained by selecting an uninstantiated variable from s and instantiating it with part or all values in the domain. Each value generates a successor state obtained from the parent state s by instantiating the selected variable with that value. Constraint propagation can shrink the domains of uninstantiated variables. We call a search space generated in this way a *CSP search space*.

It is well known that the depth of a CSP search space is bounded by the number of the variables and that a CSP search space has no duplicate states. The latter property can be shown with a recursive argument: As the successors of the root state are generated by instantiating a *single* variable with *different* values the subtrees of distinct successors of the root state never overlap. The same argument can be applied recursively to states deeper in the tree proving the claim.

In CSPs depth-first is a popular search strategy given that such algorithms are light on memory requirements. When a problem has an optimization component depth-first branch and bound, or shorter DFBnB (Land and Doig 1960), can be used to seek optimal solutions. Besides DFBnB we evaluate a best-first search algorithm, Weighted A* (Pohl 1970), in this paper. Best-first search requires memory to store the states in the Open list[2] which can be prohibitive. However, one of the benefits of our contribution is a reduction of the memory requirements, mitigating this drawback.

In this work we aim at maximizing a score rather than minimizing a cost. In Weighted A* the evaluation function is $f(s) = g(s) + w \times h(s)$ with $w \leq 1$. Solutions found with such an algorithm have a score no smaller than wC^* where C^* is the optimal score. Function g is the contribution to the score achieved so far in the current state and h is an admissible (optimistic) estimation of the additional contribution to the score that could be achieved on top of the current state.

6 Our Approach: Tiered State Expansion

The main contribution of this paper is tiered state expansion. We begin with the intuition and illustrate it with a hand-traceable example. We then present the algorithmic details.

6.1 Intuition

The goal of tiered state expansion is to reduce the branching factor in search. Tiered state expansion assumes that successors of a state can be partitioned into two tiers. Instead of generating all successors at once when expanding a state tiered state expansion generates all tier-1 successors and a single placeholder successor instead of the actual tier-2 successors. That immediately reduces the branching factor since there are usually many tier-2 successors.

However, merely *delaying* generation of tier-2 successors would not necessarily deliver savings overall. Tiered state expansion works because when the search comes to generating actual tier-2 successors, the state with the placeholder variable instantiation will have other variables instantiated with actual values which, via constraint propagation, will reduce the number of valid tier-2 instantiations to the original variable. This means that fewer actual tier-2 successor states will be generated at that point in the search.

Delayed problem-solving is a common technique in decision-making, both human and AI, motivated by the hope that the future will reduce the number of possibilities faced in solving a problem. In fact, sometimes the problem may disappear entirely. For instance, consider tiered expansion in best first search, where states get expanded in the order of their priority (the f-value). When tier-2 values have little or no contribution to the overall score a placeholder value assigned to a variable makes little or no contribution to the heuristic value (and consequently to the priority in the Open list) of the state. This is because a future instantiation of that variable can use only tier-2 values. Thus, states with placeholder values have a low priority to expansion. Thus

[1]We do not observe the family-of-words constraint as no mapping of words into their families is available to us.

[2]While generally best-first search also requires memory to store the Closed list, in CSP search this is not necessary due to the following properties: (i) absence of duplicate states eliminates the need for duplicate detection; and (ii) as soon a goal state is discovered a solution is readily available, with no need to reconstruct a path backwards from the goal to the root state.

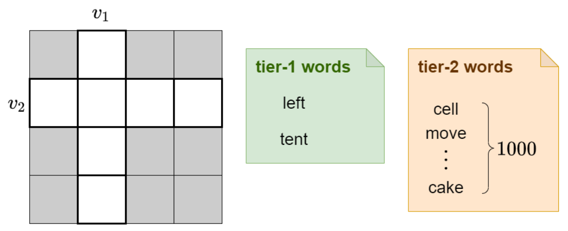

Figure 1: State s with slots v_1 and v_2, and the two word lists.

the best-first search will likely keep expanding states whose variables have tier-1 values and possibly may never even get to expanding the state with placeholder values by the time a goal state is reached. In this way generating successors with tier-2 values is not just delayed but avoided entirely.

In this work tier-1 values correspond to thematic words and tier-2 values are regular words.

6.2 A Toy Example

Consider the two-slot crosswords puzzle in Figure 1. Its state s has two variables: v_1 (the vertical word slot of length 4) and v_2 (the horizontal word slot of the same length). Suppose the thematic dictionary has two 4-letter words, left and tent, whereas the regular dictionary has 1000 4-letter words.

For simplicity, assume that the search algorithm is A*. Suppose the search expands the state s and picks the variable v_1 to instantiate. With the standard state expansion $2 + 1000$ successor states will be generated (Figure 2). Each tier-1 successor will have the g-value of 4 since a 4-letter thematic word was put it. The h-value is also 4 since there is a possibility of putting a 4-letter thematic word in the slot v_2. Thus each of the two tier-1 successors will have the f-value of $4 + 4 = 8$. Each of the 1000 tier-2 successors will have a g-value of 0 since regular words do not contribute to the solution score. Their h-value is 4 when a thematic word can be placed on the horizontal slot and 0 otherwise.

With the tiered state expansion the 1000 successors with tier-2 instantiations for the variable are replaced with a single placeholder successor which has no word instantiations but a Boolean flag indicating future tier-2 instantiations for that variable (shown by highlighting the slot in yellow in Figure 3). This reduces the branching factor from 1002 to 3 with corresponding savings in time and Open list.

The placeholder successor is placed in the Open list with the g-value of 0 (since only tier-2/regular words will be eligible for placing in the slot v_1) and the h-value of 4 since there is still a possibility of filling the other slot v_2 with a thematic word. So the f-value of the placeholder successor is $0 + 4 = 4$ which is below the $f = 8$ of the two tier-1 successors. Consequently the search may find a goal state before ever expanding the placeholder successor, thus never generating any of the 1000 tier-2 instantiations of v_1 at all.

Alternatively, the placeholder successor state may come up for expansion later in the search process. At that time either v_1 or v_2 needs to be selected for instantiation. Variable v_1 would generate 1000 successors, whereas v_2 would generate only 3 successors (two with thematic words and one

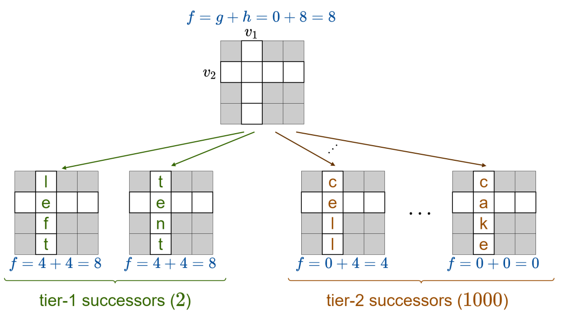

Figure 2: Standard state expansion.

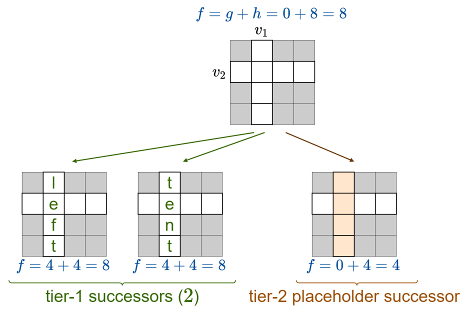

Figure 3: Tiered state expansion.

with a placeholder for the variable v_2). According to a popular variable-selection strategy, variable v_2 is thus preferred as it generates fewer successors. After instantiating variable v_2 with a word such as tent, constraint propagation immediately excludes from the domain of v_1 all tier-2 words that do not have e as the second letter (Figure 4).

6.3 Algorithmic Details

In tiered expansion we assume that, given the domain D of a variable v, it is possible to partition D into two subsets $D = T_1 \cup T_2$ where T_1 is the tier-1 subset of values and T_2 is the tier-2 subset of values. Note that given a variable v its domain $D = T_1 \cup T_2$ can shrink as search progresses along a path due to constraint propagation. When we refer to sets D, T_1 and T_2 for a variable v we mean the snapshots *at the corresponding state in the search*, unless specified otherwise.

Consider a state s and a variable v selected for instantiation. A standard expansion of s would generate $|D|$ successors. In contrast, we separate v's instantiation in two tiers. To do so we extend the state definition with a set of Boolean flags (i.e., binary variables) f_v, one for each variable v. That is a state becomes $s = (\mathbf{v_1}, \dots, \mathbf{v_n}, f_{v_1}, \dots, f_{v_n})$. In the initial state flags are set to **false**. Flag values are copied from a parent to a successor, unless explicitly changed in a successor as shown in this section.

Tier-1 expansion generates $|T_1| + 1$ successors. The first $|T_1|$ successors instantiate v with one tier-1 value each. The $(|T_1| + 1)$-th successor p in tier-1 expansion leaves v uninstantiated but sets the flag f_v to **true**, telling that any future instantiation of v (i.e., in a descendant of p) should consider only tier-2 values. We call the state p a *placeholder suc-*

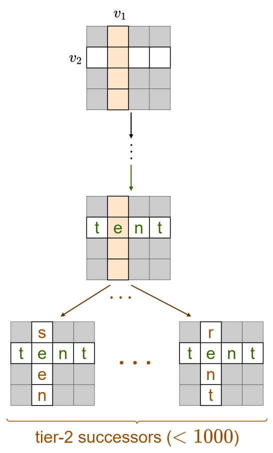

Figure 4: Delayed expansion of a placeholder state.

cessor since the variable v is left uninstantiated, as a placeholder for future instantiations with tier-2 values.

For a given variable v tier-1 expansion and tier-2 expansion can be separated in time with a number of other expansions involving other variables between them. A key advantage is that prior to the time when tier-2 expansion for variable v occurs, constraint propagation stemming from the instantiations of other variables can significantly reduce the part of T_2 still available for instantiating v. This can significantly reduce the effective branching factor of the search.

When constraint propagation performed in a state s reduces the tier-1 subset of a variable u to the empty set the flag f_u is set to $\mathbf{true}$ in state s. This will make the heuristic presented later in this section more accurate, ensuring that variable u will not contribute to the heuristic score.

Algorithm 1 shows tiered state expansion, with tier-1 expansion on lines 5–9, and tier-2 expansion on lines 11–13.

When expanding a state, the selection of the variable to instantiate (line 2 in Algorithm 1) can greatly impact search performance. A popular strategy is to give priority to a variable that leads to fewer successors. Benefits include a smaller branching factor in search and an early detection of dead ends (a variable with an empty domain). In a standard search, this strategy is equivalent to defining $r(v) = |D^v|$, where D^v is v's domain, and preferring variables with a smaller $r(v)$. In our approach we adapt the strategy as follows. For a variable v with domain $D^v = T_1^v \cup T_2^v$, define $r(v)$ as $|T_1^v| + 1$ if $f_v = \textbf{false}$ and $|T_2^v|$ otherwise. We then prefer variables with a lower $r(v)$.

6.4 Evaluation Function

We use the evaluation function $f(s) = g(s) + w \times h(s), w \leq 1$ for tiered state expansion, where h is an admissible heuristic. Given a state the heuristic optimistically assumes that all uninstantiated slots with a non-empty T_1 set will be instantiated with a thematic word. Uninstantiated slots with an empty T_1 set contribute 0 to the heuristic value with the benefit that the heuristic is penalized as soon as an uninstantiated slot gets an empty tier-1 set, before the slot is actually

Algorithm 1: TIERED STATE EXPANSION

```
   Input: State $s$
   Output: Set of successor states $S$
1  $S \leftarrow \emptyset$
2  Select variable $v$ for instantiation
3  Let $D = T_1 \cup T_2$ be the domain of $v$
4  if $f_v$ = false then
5      for $\mathbf{v} \in T_1$ do
6          Generate successor $c$ by instantiating $v \leftarrow \mathbf{v}$
7          $S \leftarrow S \cup \{c\}$
8      Generate successor $p$ by setting $f_v \leftarrow \mathbf{true}$ and
         $D \leftarrow \emptyset \cup T_2$
9      $S \leftarrow S \cup \{p\}$
10 else
11     for $\mathbf{v} \in T_2$ do
12         Generate successor $c$ by instantiating $v \leftarrow \mathbf{v}$
13         $S \leftarrow S \cup \{c\}$
14 return $S$
```

instantiated with a tier-2 word. More formally, for a state $s = (\mathbf{v_1}, \ldots, \mathbf{v_n}, f_{v_1}, \ldots, f_{v_n})$, define

$$1_i^H(s) = \begin{cases} 1 & f_{v_i} = \textbf{false} \text{ and } \mathbf{v_i} = \perp \\ 0 & \text{otherwise} \end{cases}$$

Define ℓ_{v_i} as the length of the word slot v_i. In other words, this is the contribution of the slot (variable) v_i if it is instantiated with a thematic word. The heuristic is defined as

$$h(s) = \sum_{i=1}^{n} 1_i^H(s) \times \ell_{v_i}.$$

As we address a maximization problem admissibility means that $h(s)$ is *not lower* than the best possible score obtainable via the uninstantiated slots.

Function g is the score achieved with already instantiated thematic words. Define

$$1_i^G(s) = \begin{cases} 1 & \mathbf{v_i} \text{ is a thematic word} \\ 0 & \text{otherwise} \end{cases}$$

Then

$$g(s) = \sum_{i=1}^{n} 1_i^G(s) \times \ell_{v_i}.$$

7 Empirical Evaluation

We implemented tiered state expansion in WOMBAT, a freely available high-performance C++ solver for crosswords optimization. In the evaluation function we set $w = 0.5$. In tests we used two sets of puzzles Random and Evolved, each with 1024 puzzles. A puzzle is a 13×13 grid with 26 pre-initialized black cells. The configuration of black cells differs from puzzle to puzzle. All puzzles use the thematic dictionary published for 2021, with 445 words, and a regular Romanian dictionary, with 134277 words. Random puzzles were generated by randomly placing 26 black cells on the grid, and keeping instances with a valid configuration of black cells (according to the rules presented in Section 4). Evolved puzzles were previously synthesized with a separate project, based on deep learning and evolution, and aimed at obtaining puzzles with higher scores than random puzzles. Figure 5 illustrates one evolved and one random puzzle.

Our code and data are available at https://www.dropbox.com/s/13pdcqzfaanmyca/wombat-aiide-22.tgz?dl=0.

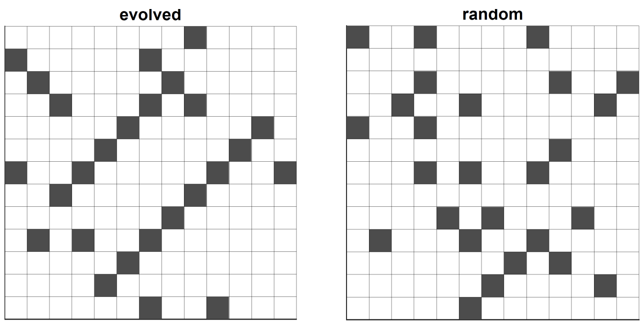

Figure 5: Sample puzzles used in our experiments.

The rest of this section is structured as follows. We give a performance comparison between DFBnB and Weighted A* (WA*), both with tiered state expansion turned on. This will establish that Weighted A* performs significantly better in our tests. Then, we evaluate the impact of tiered state expansion in WA*, on evolved and random puzzles, respectively. A similar analysis for the impact of tiered state expansion on DFBnB is left for a future report.

7.1 Weighted A* and DFBnB

Weighted A* and DFBnB are compared in Figure 6. The horizontal axis is the search time up to one hour and the vertical axis is the average score at that time, over the 1024 puzzles at hand (random or evolved). A score can correspond to a full solution, if available, or a highest-score partial solution available at that time otherwise. Except for very small timeouts (e.g., a few minutes), WA* consistently outperforms DFBnB, with a more pronounced lead for the evolved puzzles. The size of the Open list in WA* is moderate, growing to an average of 59378 states in random puzzles and 7256 states in evolved puzzles, after one hour of searching.

7.2 Impact of Tiered State Expansion in WA*

To evaluate tiered state expansion we considered three conditions: `standard/standard` with standard expansion of all states, `tiered/standard` where the root state uses tiered expansion but all other states use standard expansion and `tiered/tiered` where tiered expansion is used for all states. We ran two sets of experiments, one with the resources set to 20 minutes and 4GB RAM, and the other with 4 hours and 16GB RAM, as listed in Table 1. When WOMBAT ran out of memory on a puzzle (e.g., with tiered expansion turned off) we removed that puzzle from all three conditions. In all other cases, we considered the score of a full solution, if found, or the best partial score of the states encountered in the search otherwise (the choice will be clear from the context).

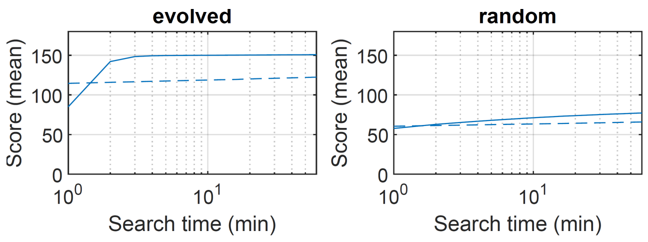

Figure 6: WA* (solid line) versus DFBnB (dashed line).

For evolved puzzles Table 1 shows statistics averaged over 817 puzzles. WOMBAT with the condition `standard/standard` and 20 minutes and 4 Gbytes of memory limits did not find a full solution for any of the 817 puzzles due to the time limit. Furthermore, its partial solutions scores are low, with an average of 18.6 and the maximum of 69. Switching the root state expansion to tiered (condition `tiered/standard`) has a moderate effect, increasing the maximum score to 74 and the average to 19.1.

Switching all state expansions to tiered (condition `tiered/tiered`) has a major effect. WOMBAT found full solutions for 810 out of 817 (i.e., 99.1%) of the puzzles. The average partial/full solution score increased to 148.1 and the maximum to 168. Note that while WOMBAT in the first two conditions ran against the time limit of 20 minutes on all 817 puzzles, the average time to find a full solution in the `tiered/tiered` condition was 1.2 minutes/puzzle (with the minimum of 8 seconds and the maximum of 6.3 minutes).

The ratio of the number of states visited to the number of states expanded is the effective branching factor (BF) of the search for a given puzzle. Table 1 lists the averages for the three conditions. For a given time limit, tiered expansion allows WOMBAT to expand on average many more states than standard expansion (4.9×10^3 versus 457) because the effective branching factor is much lower: 1.5 versus 91.4. Each generated successor requires non-trivial processing such as constraint propagation. Consequently, the effective branching factor significantly impacts the state expansion numbers affordable within the time limit.

As mentioned, the branching factor is reduced significantly in the `tiered/tiered` condition. The reduction is achieved by tiered state expansion via two mechanisms. The more direct mechanism is the reduction illustrated in Section 6. The second mechanism works by allowing the search to progress deeper within a given time limit. States deeper in the search tend to have lower branching factors as more variables are then instantiated, allowing constraint propagation to remove many choices.

In summary, switching all state expansions from standard to tiered yielded full solutions for nearly all puzzles and improved the average partial/full score by almost 8 times.

Given that with the standard state expansion (i.e., conditions `standard/standard` and `tiered/standard`) WOMBAT showed a limited performance with the 20 minutes time limit, we will now turn to the results with the higher resource limits (4 hours and 16 Gbytes of memory). Even with the additional resources WOMBAT in the `standard/standard`

Condition	Evolved puzzles						Random puzzles					
	Score	**Time**	**Sol**	**Exp**	**Visited**	**BF**	**Score**	**Time**	**Sol**	**Exp**	**Visited**	**BF**
ss (20m/4Gb)	18.6	–	0	457	28K	91.4	11.4	–	0	605	44K	292.3
ts (20m/4Gb)	19.1	–	0	1.1K	16.8K	18.1	12.2	–	0	726	37.9K	191.3
tt (20m/4Gb)	**148.1**	**1.2 m**	**810**	4.9K	6.2K	**1.5**	76.6	**9.2 m**	3	74K	93.9K	**1.3**
ss (4h/16Gb)	24.1	–	0	11.9K	193.1K	23.0	12.0	–	0	9.4K	310.1K	124.9
ts (4h/16Gb)	24.1	–	0	11.5K	207.1K	21.2	12.8	–	0	10.9K	296.2K	95.5
tt (4h/16Gb)	**148.1**	**1.2 m**	**810**	29.9K	31.2K	**1.5**	**83.2**	1.1 h	**10**	876.6K	1.1M	**1.3**

Table 1: WOMBAT on evolved and random puzzles, with conditions `standard/standard (ss)`, `tiered/standard (ts)` and `tiered/tiered (tt)`. Time and memory resources are set to 20 minues and 4Gb RAM (top), and 4 hours and 16Gb RAM (bottom). We show the average partial or full score (Score), average time to find a full solution (Time), instances fully solved (Sol), average expanded nodes (Exp), average visited nodes (Visited), and average branching factor (BF). Best values are bolded.

and `tiered/standard` conditions failed to fully solve any of the 817 puzzles. The average partial score had a minor increase to 24.1 and the numbers of expanded and visited states increased due to the longer running time.

Even when the expansion strategy is fixed (e.g., `standard/standard`), the branching factor can decrease when more resources are available (e.g., from 91.4 to 23.0). This is because additional resources allow to reach greater depths in the search tree and, as mentioned earlier, states at deeper levels tend to have a smaller branching factor due to constraint propagation.

With the random set of puzzles in use, fewer puzzles allow for full solutions and partial scores are lower. Consequently only 88 out of the 1024 puzzles had partial/full solutions for all six configurations of conditions and resource allocations. On those 88 puzzles the `tiered/tiered` condition with 20 minutes and 4 Gbytes found 3 full solutions which raised to 10 puzzles with 4 hours and 16 Gbytes. No full solutions were found in any condition with standard expansions.

Recall that Table 1 presents statistics for the subset of 88 random puzzles where *all* three conditions produce at least a partial score. We will now look at problems fully solved in an individual condition. On the full set of 1024 random puzzles `tiered/tiered` with 20 minutes and 4 Gbytes fully solved 26 puzzles with an average time of 7.7 minutes. With 4 hours and 16 Gbytes it fully solved 118 puzzles with an average time of 1.3 hours.

Similarly Table 1 presents statistics for a subset of 817 evolved puzzles where all three conditions produce at least a partial score. Removing that constraint we see that on the full set of 1024 evolved puzzles `tiered/tiered` with 20 minutes and 4 Gbytes fully solved 1012 puzzles with an average time of 1.05 minutes. With 4 hours and 16 Gbytes it fully solved 1013 puzzles with an average time of 1.07 minutes. Thus, most evolved puzzles are solved very quickly, with an average time of just above a minute.

Remarkably the results show that tiered expansion is more effective for higher quality puzzle instances (i.e., instances that allow for higher score solutions). This is important in optimization problems where high-quality solutions, rather than just any solutions, are desired.

The fact that evolved puzzles allow higher scores, compared to the random ones, appears to be one of the key features to explain why evolved puzzles are easier to solve. Consider two puzzles, the first with a higher optimal score, and the second with a lower one. Since the latter has a lower optimal score, and we work with admissible heuristics, the heuristic evaluations of states in the search state of the second puzzle will cover a smaller range of values than the states in the search space of the first puzzle. As such, the second puzzle could have more states where the heuristic is the same, thus making it more difficult for the search to distinguish between truly promising states and states that will lead to a deadend. A deeper study of this question would be interesting and we leave it as future work.

8 Conclusions

In combinatorial search the branching factor can have a major impact on the search effort. We have introduced tiered state expansion, a technique to reduce the branching factor. Our technique applies to a generic class of constraint optimization problems. We applied this to a challenging problem inspired by the Romanian Crosswords Competition. Empirical results demonstrate substantial performance gains. In a few minutes Weighted A* with tiered expansion fully solved problem instances that were unsolvable in four hours with the standard expansion.

In future work we plan to explore tiered state expansion to additional problems, such as graphical models and scheduling. In terms of defining the tiers, a generic recipe would be to heuristically rank successors of a parent state when a heuristic evaluation function is available. Then we can split the set of successors into tier 1 and tier 2 subsets. Choosing the splitting point can consider criteria such as a point where a sudden drop in the heuristic evaluation from one successor to the next is observed. In addition we plan to further advance AI capabilities in the Romanian Crosswords Competition, an application where top human performance remains significantly stronger than AI.

Acknowledgments

We appreciate support from Compute Canada.

References

Anbulagan; and Botea, A. 2008. Crossword Puzzles as a Constraint Problem. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, 550–554.

Audemard, G.; Lecoutre, C.; and Maamar, M. 2020. Segmented Tables: An Efficient Modeling Tool for Constraint Reasoning. In *Proceedings of the European Conference on Artificial Intelligence*, 315–322.

Botea, A. 2007. Crossword Grid Composition with A Hierarchical CSP Encoding. In *Proceeding of the CP Workshop on Constraint Modelling and Reformulation*.

Botea, A.; and Bulitko, V. 2021. Scaling Up Search with Partial Initial States in Optimization Crosswords. In *Proceedings of the Symposium on Combinatorial Search*, 20–27.

Bulitko, V.; and Botea, A. 2021. Evolving Romanian Crossword Puzzles with Deep Learning and Heuristic Search. In *Conference on Games*.

Chen, L.; Liu, J.; Jiang, S.; Wang, C.; Liang, J.; Xiao, Y.; Zhang, S.; and Song, R. 2022. Crossword Puzzle Resolution via Monte Carlo Tree Search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 35–43.

Ernandes, M.; Angelini, G.; and Gori, M. 2005. WebCrow: a Web-Based System for Crossword Solving. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1412–1417.

Ginsberg, M. L. 2011. Dr.Fill: Crosswords and an Implemented Solver for Singly Weighted CSPs. *Journal of Artificial Intelligence Research*, 42: 851–886.

Ginsberg, M. L.; Frank, M.; Halpin, M. P.; and Torrance, M. C. 1990. Search Lessons Learned from Crossword Puzzles. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 210–215.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Land, A. H.; and Doig, A. G. 1960. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3): 497–520.

Lecoutre, C.; and Roussel, O. 2019. Proceedings of the 2018 XCSP3 Competition. *CoRR*, abs/1901.01830.

Littman, M. L.; Keim, G. A.; and Shazeer, N. M. 2002. A probabilistic approach to solving crossword puzzles. *Artificial Intelligence*, 134(1-2): 23–55.

Mazlack, L. J. 1976. Computer Construction of Crossword Puzzles Using Precedence Relationships. *Artificial Intelligence*, 7(1): 1–19.

Pohl, I. 1970. Heuristic Search Viewed as Path Finding in a Graph. *Artificial Intelligence*, 1(3): 193–204.