Artificial Intelligence 335 (2024) 104192

Contents lists available at ScienceDirect

Artificial Intelligence

journal homepage: www.elsevier.com/locate/artint

A crossword solving system based on Monte Carlo tree search

Jingping Liu [a,*], Lihan Chen [b,*], Sihang Jiang [c], Chao Wang [d], Sheng Zhang [e], Jiaqing Liang [c], Yanghua Xiao [c,*], Rui Song [e]

[a] *East China University of Science and Technology, Shanghai, China*
[b] *Beijing Institute of Control Engineering, Beijing, China*
[c] *Fudan University, Shanghai, China*
[d] *Shanghai University, Shanghai, China*
[e] *North Carolina State University, Raleigh, United States of America*

ARTICLE INFO

Keywords:
Crossword puzzle
Monte Carlo tree search
Language understanding

ABSTRACT

Although the development of AI in games is remarkable, intelligent machines still lag behind humans in games that require the ability of language understanding. In this paper, we focus on the crossword puzzle resolution task. Solving crossword puzzles is a challenging task since it requires the ability to answer natural language questions with knowledge and the ability to execute a search over possible answers to find an optimal set of solutions for the grid. Previous solutions are devoted to exploiting heuristic strategies in search to find solutions while having limited ability to explore the search space. We build a comprehensive system for crossword puzzle resolution based on Monte Carlo Tree Search (MCTS). As far as we know, we are the first to model the crossword puzzle resolution problem as a Markov Decision Process and apply the MCTS to solve it. We construct a dataset for crossword puzzle resolution based on daily puzzles from The New York Times with detailed specifications of both the puzzle and clue database selection. Our method achieves state-of-the-art performance on the dataset. The code of the system and experiments in this paper is publicly available: https://www.github.com/lhlclhl/CP.

1. Introduction

With the remarkable development of artificial intelligence, current intelligent systems have surpassed humans in various computationally demanding games, such as GO [38] and Atari [27]. However, when it comes to games that involve language understanding ability, intelligent machines still fall behind human capabilities. A notable example is the American Crossword Puzzle Tournament 2021,[1] where the AI participant Dr. Fill [11] demonstrates superior speed compared to the human player, but performs less accurately, making three errors while the human players solve all the puzzles perfectly. If AI participants can achieve superiority over humans in games that require language understanding and knowledge application, it will significantly contribute to the advancement of artificial intelligence as a whole.

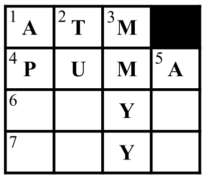
In this paper, our focus is on solving crossword puzzles, which pose a unique challenge requiring both computational power and language understanding ability for machines. A crossword puzzle usually comprises a grid of white and black squares, along with a corresponding set of clues. Each clue corresponds to a series of consecutive white squares in a row or column, and players are

* Corresponding authors.
E-mail addresses: jingpingliu@ecust.edu.cn (J. Liu), lhc825@gmail.com (L. Chen), tedsihangjiang@gmail.com (S. Jiang), cwang@shu.edu.cn (C. Wang), szhang37@ncsu.edu (S. Zhang), l.j.q.light@gmail.com (J. Liang), shawyh@fudan.edu.cn (Y. Xiao), rsong@ncsu.edu (R. Song).

[1] https://www.crosswordtournament.com/2021/index.htm.

https://doi.org/10.1016/j.artint.2024.104192
Received 12 July 2023; Received in revised form 16 July 2024; Accepted 20 July 2024

Available online 25 July 2024



Crossword Grid

Across	Down
1. $ dispenser	1. Pit-__
4. Adidas alternative	2. Spanish pipe
6. What the club gets without playing	3. Credit-card exp. date format
7. 1992 Robin Williams movie	5. Two-time loser to DDE

Puzzle Clues

Fig. 1. A toy example of an unfinished crossword puzzle, constructed by us with real clues from published crossword puzzles. In the grid, each slot is hinted by a clue and starts with a square whose top-left corner is indexed by the corresponding clue number. Words reading from left to right are hinted by the clues in the "across" column (e.g., `PUMA` at "4-across" is hinted by the clue *Adidas alternative*), and those from top to down are the "down" column (e.g., `MMYY` at "3-down" is hinted by the clue *Credit-card exp. date format*). Words end when reaching the border or a black square.

tasked with filling words into these slots based on the given clues. The words intersect at a shared letter within the grid, with black squares serving as separators between slots. An example crossword puzzle is depicted in Fig. 1. The objective of the game is to find the solution, fulfilling three key criteria: 1) each slot must be filled with a word, 2) every two words crossing in the grid are not conflicted, and 3) each word in the slot must align with the semantics of its corresponding clue. Consequently, machines need to possess two essential abilities to solve crossword puzzles effectively:

- They require significant computational power to search through the vast combination space of words. In addition to the words in the given vocabulary, any letter sequence can potentially be a valid "word" in a crossword puzzle. For example, in Fig. 1, the answer to "3-down" is `MMYY`, which is not a conventional word. Therefore, any letter can potentially fill a square. In a typical crossword puzzle, the grid contains approximately 200 white squares, resulting in a total search space of 26^{200} for finding a solution.[2]
- Machines must possess the ability to understand the semantics of the natural language. This understanding is crucial for accurately evaluating the matching between words and clues, which enables the selection of the best solution. Clues in crossword puzzles are often designed to be implicit, making them challenging even for humans to comprehend. For instance, a clue such as "`$ dispenser`" requires advanced language understanding capabilities in machines to decipher its meaning. Hence, a machine's language understanding ability needs to be fully developed to effectively handle such clues.

Many efforts have been devoted to exploiting the above two abilities of machines to solve CPs. On one hand, to address the challenge of exploring the enormous search space, many methods develop search algorithms based on fixed heuristic strategies to find a better solution [20,10]. These strategies, such as Weighted A* [31] and limited discrepancy search [11], remain unchanged for different puzzles and do not dynamically adjust subsequent search strategies based on experience gained from previous search results. Hence, they may not adapt well to different situations and are not robust when heuristics make mistakes. On the other hand, previous methods leverage machine language understanding to aid in CP resolution. This involves generating candidates for each slot based on the corresponding clue to narrow down the search space [10]. Additionally, the automatic evaluation of candidate solutions during the search is constructed based on the degree of matching between clues and words. Mainstream approaches [3,28,37,42] often employ ranking models to evaluate the words for each slot. However, these solutions exhibit some notable shortcomings. First, previous candidate generation strategies have difficulty in balancing effectiveness, where each slot's ground truth is appropriately ranked in the word list retrieved by the algorithm based on the corresponding clue, and efficiency, measured by the time required for the retrieval process. Some methods achieve effective results (e.g., candidate generation based on many submodules [20]) but sacrifice efficiency (e.g., 7-10 minutes are needed for candidate generation in a total 15-minute solving time). Others prioritize efficiency, but this comes at the cost of effectiveness. Second, existing automatic evaluation methods often lack robustness in selecting the best filling solution. The candidate words are generated from diverse sources, and different sources may require different ranking mechanisms. As a result, a single ranking mechanism fails to adequately evaluate the solutions.

We propose a novel framework for solving the crossword puzzle based on the Monte Carlo Tree Search (MCTS) algorithm [6]. For the immense search space, MCTS provides adaptive strategies that can asymptotically improve the search process to find the perfect solution. Building upon the MCTS framework, we introduce an action space reduction strategy to address the challenge of large action spaces. Additionally, we propose a variance control mechanism in value estimation to enhance policy stability. Moreover, to further narrow down the search space, we develop an efficient candidate generation approach that incorporates multiple external sources. In addition to the clue database, we leverage dictionaries [25,23], knowledge bases [41], and pre-trained language models [9] to generate candidates. This approach ensures both effectiveness and efficiency in candidate generation. Finally, to provide a robust evaluation of candidate solutions during the search process, we design a reward function that combines signals from different sources. This reward function incorporates various features, including prior knowledge (e.g., word frequency) and semantic features (e.g., semantic relatedness to the clue). The reward function leverages the benefits of data-driven models and can be easily tuned based on system performance. In summary, our contributions can be summarized as follows:

[2] This value is determined using the calculation method outlined in [11]. It is a fairly loose upper bound because many slots are filled with common dictionary words, while only a small number contains unusual strings. Notably, due to the considerable variability in answer value distribution among different puzzles, it is difficult to provide a precise upper bound.

- *Novel formalization.* We formalize the task of solving CP as a Markov Decision Process, departing from previous methods that treated CPs as a constraint satisfaction problem (CSP) [20,11]. This new formalization allows us to apply algorithms that can obtain asymptotically improved policies in sequential decision-making, such as trial-based heuristic tree search (THTS) [16] or reinforcement learning [39]. These methods have been proven effective in exploring large search spaces, offering new opportunities for CP resolution. To the best of our knowledge, we are the first to propose this formalization and utilize MCTS to solve CPs.
- *Systematic solution.* We build a comprehensive system for crossword puzzle resolution. Since CP resolution requires the integration of multiple machine abilities, it is essential to combine and adapt different models and algorithms into a unified system. We make significant efforts to make it happen in three aspects. First, in order to accommodate MCTS to the CP task, we propose an action space reduction strategy to optimize the search procedure and a variance control strategy to improve the tree policy. Second, for the purpose of efficient and effective retrieval of candidate words from external data sources, we build a candidate generation module with a strategy complementary to the search algorithm and several efficient submodules to retrieve candidate answers from multiple sources. Third, to provide robust evaluation for possible solutions, we design a novel reward function that combines multiple ranking models from different sources. We also optimize this reward function based on the system's performance.
- *Empirical evaluation.* On one hand, we construct a dataset for crossword puzzle resolution based on daily puzzles from The New York Times (NYT). Compared to the datasets used in previous works [10,28], our dataset has three main distinctions. First, we provide detailed specifications of the puzzle selection. Second, we set the clue database as a fixed setting in the dataset. Seen clues, defined as those with known answers derived from previously played CPs, are crucial to the performance of this task. To ensure fair comparisons with different methods, we fix them in the dataset as the training set or clue database for experiments. Third, our dataset is publicly available. We conduct extensive experiments to evaluate the effectiveness of our method. The results indicate that our method outperforms the current state-of-the-art Berkley Crossword Solver [42]. The experiments also validate the effectiveness of all modules and designs in our system, including the search algorithm, candidate generation, and reward function.

The rest of this paper is organized as follows. Section 2 discusses the related work of this paper. Section 3 formulates the problem and provides an overview of our solution for CP resolution. Section 4 introduces the MCTS algorithm for CP resolution and presents the improved strategies in MCTS, including variance control for upper confidence bound (UCB), action space reduction, and default policy. Section 5 gives the candidate generation procedure (i.e., clue-dependent and clue-independent candidate generation) to produce high-quality candidate words for each slot. Section 6 describes the reward function used to determine the quality of solutions for a CP. Section 7 details the construction of the dataset used for CP resolution. The experimental results are reported in Section 8, and our conclusion and future work are discussed in Section 9.

The previous version of this paper has been published at ICAPS 2022 [7]. In this new version, we make several key improvements to enhance the overall quality and effectiveness of our approach. The specific differences between the submitted manuscript and its previous version are as follows. (1) To generate high-quality candidate words for each slot, we introduce three new modules, including dictionary retrieval, knowledge base retrieval, and slot-filling retrieval (Section 5.1). (2) To learn an effective reward function, we incorporate the scores produced by three newly added retrieval modules as ranking features (Section 6.1). (3) We add a new test set with puzzles from the ACPT 2021 tournament and conduct comparisons between our method and new state-of-the-art methods [42]. The results demonstrate that our method outperforms the competitors (Section 8.2). (4) We perform an analysis of the prior probability for the default policy and explore its impact on the performance of the MCTS system (Section 8.4). (5) To study the impact of each module in clue-dependent generation on a single clue, we compare our proposed method with multiple alternatives to verify the effectiveness of the proposed method. More importantly, we conduct extensive experiments to explore the impact of module combinations on the search system (Section 8.5).

2. Related work

Related work on crossword puzzle resolution can be divided into two categories: search-based methods and clue-understanding methods.

2.1. Search-based methods

Traditional search strategies, such as branch and bound, suffer from the problem of "early mistakes" [12], where it becomes challenging to backtrack from errors that occur at shallow levels of the search. To alleviate this problem, many heuristic search algorithms have been adopted to solve CPs such as A* search [20,10] and limited discrepancy search (LDS) [11]. Those methods construct heuristics to evaluate the effectiveness of search states and modify the search order to prioritize the exploration of more potential ones. Although those search algorithms mitigate the problem, they heavily rely on exploiting the heuristic strategy and lacks the ability to explore alternative options. MCTS provides a trade-off between exploitation and exploration, making it a valuable approach for domains represented as sequential decision trees, such as games [38,32] and certain NLP problems [21]. Nested Rollout Policy Adaptation (NRPA) [36] on MCTS has been applied to crossword construction tasks, specifically for filling words in crossword grids without considering clues. Berkley Crossword Solver [42] combines belief propagation (BP) and local search as the search algorithm to find the puzzle solutions. However, there are currently no applications of MCTS for solving crossword puzzles that involve understanding and utilizing clues.

2.2. Clue understanding

Clue understanding involves generating candidates based on clues and evaluating the degree of matching between the clues and potential words. To generate accurate candidates, traditional efforts often rely on complex expert systems with multiple submodules [20,10]. However, these heavy candidate generation modules are impractical, as CP problems usually have a time limit that the search and candidate generation share. Hence, the following work [11] employs basic candidate generation strategies, including clue matching, part-of-speech analysis, dictionary usage, abbreviation rules, and filling blanks with common phrases. It focuses more on the search algorithm, particularly limited discrepancy search, to achieve a better overall performance. Nevertheless, with advancements in natural language processing techniques [24,9,45,15] and knowledge bases [1,41], it is now possible to enhance the search performance by incorporating additional candidate generation modules. These modules leverage the power of natural language processing and knowledge bases to generate more accurate and diverse candidates. Regarding the evaluation of clue-word matching, traditional methods rely on manually-tuned scoring functions to assess the level of compatibility between clues and words [10,11]. However, these scoring functions have limited effectiveness when dealing with large-scale data. More recent approaches employ data-driven methods, such as learning-to-rank models [3,28,37], to train the scoring function. Berkley Crossword Solver [42] adopts a dense retriever based on neural question-answering models as the scoring function. Nonetheless, ranking models primarily optimize the ranking of candidates for individual clues, and they may not directly evaluate the entire assignment for all clues, often disregarding grid constraints. Hence, in this paper, we introduce a novel approach that utilizes a global reward function in combination with multiple ranking models obtained from diverse sources.

3. Overview

In this section, we first define the problem of crossword puzzle resolution. Then, we formalize the task of solving CPs as a Markov Decision Process and propose a novel solution.

3.1. Problem definition

Each CP, in this paper, is formalized as a quadruple $\langle X, C, D, F\rangle$. $X = [x_1, ..., x_q]$ is an array of variables, where each variable x_i $(i = 1, ..., q)$ refers to a word slot to be filled (e.g., "1-across"). Each word slot x_i corresponds to a clue c_i which is a constant. $C = [c_1, ..., c_q]$ is the array of clues. D is the domain set of all sequences of letters, which includes dictionary vocabularies (e.g., `puma`) and other strings (e.g., date format `MMYY`). F represents the grid constraints, including cross consistency (e.g., in Fig. 1, the second letter of the "3-down" word `MMYY` equals to the third letter of the "4-across" word `PUMA`), and length constraint (e.g., in Fig. 1, the "1-across" word is a 3-letter word).

To represent the solution of a CP, we define an *assignment* of X as $W = [w_1, ..., w_q]$, where each w_i is a word filled for the corresponding word slot x_i. Each w_i is a word in the domain (i.e., $w_i \in D$), or unfinished (i.e., $w_i \in \Delta$). Δ is the set containing all the unfinished words with at least one blank (i.e., an unfinished grid is also an assignment). For example, the "6-across" of the assignment in the grid of Fig. 1 is an unfinished word with 3 blanks and belongs to Δ. The constraint F is formulated as an indicator function of W: if all the words in W satisfy the grid constraints F, $F(W) = 1$; otherwise, $F(W) = 0$. CP problem is thus formulated as:

$$W = [w_1, ..., w_q] \text{ s.t. } \begin{cases} w_i \in D, \quad i \in [1, ..., q] \\ F(W) = 1 \\ w_i \;\text{ satisfies }\; c_i, \quad i \in [1, ..., q] \end{cases} . \tag{1}$$

The first is the completeness constraint. The second is the grid constraint F. The last constraint involves the semantic matching of words in W with their corresponding clues in C. Since it is difficult to measure the third constraint in Eq. (1) using a black-or-white method, we transform this "hard" constraint problem into a "soft" constraint problem. We design a function $r(w_i, c_i) \in \mathbb{R}$ to quantify the matching of each clue-word pair, where $w_i \in D \cup \Delta$ and $c_i \in C$. A higher value of $r(w_i, c_i)$ indicates a stronger matching between w_i and c_i. As these constraints are independent of each other, our objective is to find an assignment that maximizes the sum of rewards across all clues, which is formulated as:

$$\arg\max_{W} \sum_{1 \le i \le q} r(w_i, c_i) \quad s.t. \;\; w_i \in D \;\; and \;\; F(W) = 1. \tag{2}$$

3.2. Markov decision process formalization

As a sequential decision-making problem, the solving of a CP can naturally be formalized as a Markov Decision Process (MDP). An MDP has mainly four components, including the set of *states* S, the set of *actions* A, *transition function* $T(s, a) \in S$, and *reward* $R(s) \in \mathbb{R}$, where $a \in A, s \in S$. In a CP game, each state $s = [s_1, s_2, ..., s_q] \in S$ corresponds to an assignment W that satisfies the grid constraint, i.e., $F(W) = 1$. For convenience, we use s_i to denote w_i when no conflict is introduced. An action $a = \{(x_i, v) | 1 \le i \le q, v \in D\} \in A$ includes two components: selecting a word slot x_i and filling it with a word v. We define the action space A_s for state s as the actions that satisfy the current grid constraint of s. The state s is *terminal* if $A_s = \emptyset$. The deterministic mapping from a state-action pair to another state is denoted as the transition function $T(s, a)$, and the *cumulative reward* $R(s)$ for each state s is computed by the average score of the reward $r(s, c)$.

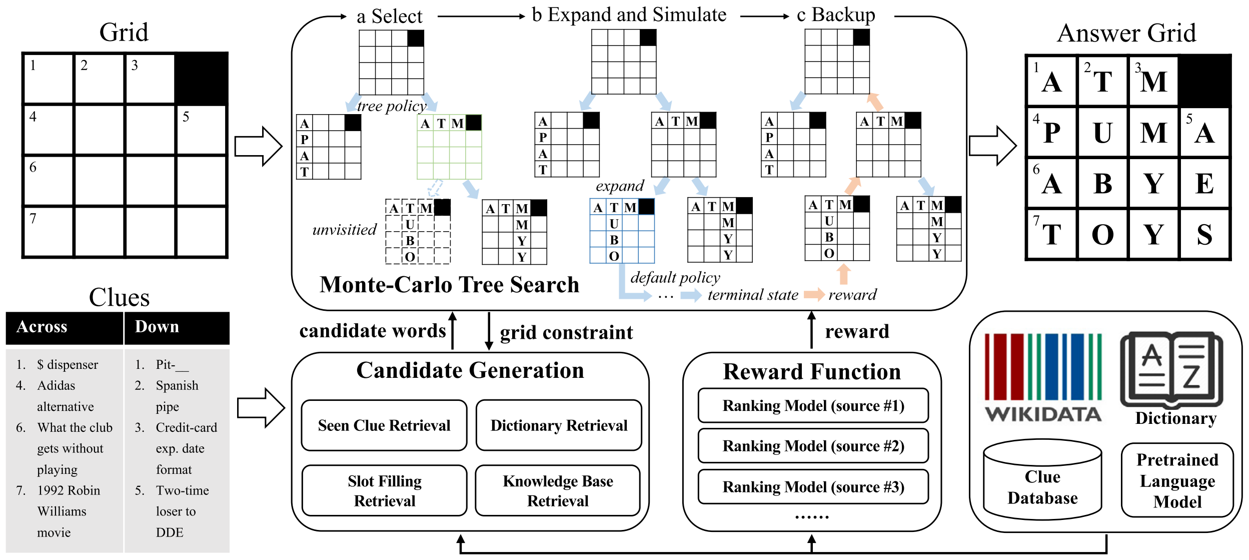


Fig. 2. The framework of our crossword puzzle resolution system based on MCTS. The algorithm iteratively executes three steps to build a search tree and update the best solution. We design a candidate generation and a reward function module for specific problems of CPs.

$$R(s) = \frac{1}{q} \sum_{1 \le i \le q} r(s_i, c_i). \tag{3}$$

3.3. *Our solution*

A common approach to solving a Markov Decision Process (MDP) is to obtain a policy, which represents the strategy for taking actions in each state, by iteratively selecting the action that maximizes the expected accumulated reward [4]. To acquire such a policy, a search must be performed on all the states. However, in many real-world applications, the state space is too large to perform a complete exploration. To address this challenge, the Monte Carlo Tree Search (MCTS) algorithm has gained popularity by incorporating random sampling into the policy construction process, proving to be effective [6]. This effectiveness has motivated us to apply MCTS to solve CPs. By simulating the game iteratively, MCTS produces a policy with a search tree. In the search tree, each node corresponds to a state of MDP, and each edge corresponds to an action taken from the state of a node to the state of its child node. The framework for solving the CP problem with MCTS is presented in Fig. 2. The algorithm follows a three-step process to construct the search tree iteratively. In the *selection step*, it recursively selects child nodes from the root node with a tree policy based on statistical information gathered from previous simulations. In the *expansion and simulation step*, an unvisited state is added to the tree and a default policy based on a prior distribution P is utilized to simulate actions from that state. The simulation ends up with a terminal state and a reward is returned. In the *backup step*, the obtained reward is used to update the statistics within the nodes of the search tree.

4. Monte Carlo tree search for crossword puzzle resolution

In this section, we introduce our MCTS algorithm for crossword puzzle resolution. First, we describe the overall design of the algorithm. Then, we elaborate on specific designs of several modules in the algorithm to solve the CP problem.

4.1. *Algorithm overview*

As shown in Algorithm 1, our MCTS algorithm conducts iterative game simulations to continuously construct a search tree and simultaneously refine the optimal solution until a predetermined time limit is reached. Notably, each node of the search tree corresponds to a state and holds two values: $N(\cdot)$, which denotes the number of times the node has been visited, and $G(\cdot)$, corresponding to the cumulative rewards obtained from playouts that traversed through the state, spanning from current states to terminal states. In each iteration, the algorithm executes the following three steps to construct a comprehensive search tree.

Selection (lines 8-13). Starting at the root node, a tree policy is recursively applied to select children until an expandable node is encountered. A node is considered expandable if it has unvisited children. The tree policy is to select an action maximizing the UCB [6,38] to trade off the actions with high estimated cumulative rewards against those less-visited ones. To compute the UCB in the tree policy, we adopt the widely-used P-UCT algorithm [38,35], $\mathrm{UCB}(s,a) = Q(s,a) + \lambda\sqrt{N(s)}P(s,a)/(1+N(s'))$, where λ is a hyper-parameter used to determine the extent of exploration and $P(s,a)$ is the prior probability of taking an action a on the state s.

Algorithm 1: Monte Carlo Tree Search for CP.

```
Input: CP $\langle X, C, D\rangle$, reward function $R$, prior probability $P(s,a)$, transition $T(s,a)$, postprocessing function $Post(\cdot)$, and upper confidence bound function UCB$(s,a)$
Output: A state of largest rewards $bestS$
1  $maxR \leftarrow -\infty$
2  $bestS \leftarrow null$
3  $G \leftarrow \{s_0 : 0\}$
4  $N \leftarrow \{s_0 : 0\}$
5  $Parent \leftarrow \{s_0 : null\}$
6  while within time limit do
7      $s \leftarrow s_0$
8      $E \leftarrow \{a : a \in A_s \ \& \ T(s,a) \notin G\}$
9      while $E = \emptyset$ do
10         $a \leftarrow \arg\max_{a' \in A_s} \mathrm{UCB}(s,a')$
11         $s \leftarrow T(s,a)$
12         $E \leftarrow \{a : a \in A_s \ \& \ T(s,a) \notin G\}$
13     end
14     $a \leftarrow \arg\max_{a' \in E} P(s,a')$
15     $s_t \leftarrow T(s,a)$
16     $Parent \leftarrow Parent \cup \{s_t : s\}$
17     $s \leftarrow s_t$
18     while $s_t$ is not terminal do
19         Sample $a \in A_{s_t}$ with $P(s_t,a)$
20         $s_t \leftarrow T(s_t,a)$
21     end
22     $G \leftarrow G \cup \{s : 0\}$
23     $N \leftarrow N \cup \{s : 0\}$
24     while $s$ is not null do
25         $N(s) \leftarrow N(s) + 1$
26         $G(s) \leftarrow G(s) + R(s_t) - R(s)$
27         $s \leftarrow Parent(s)$
28     end
29     $s_t \leftarrow Post(s_t)$
30     if $R(s_t) > maxR$ then
31         $maxR \leftarrow R(s_t)$
32         $bestS \leftarrow s_t$
33     end
34 end
35 return $bestS$
```

Initially, this search control strategy prefers actions with high prior probability and low visit count, gradually shifting towards actions with high action value as the process continues asymptotically.

Expansion and simulation (lines 14-21). The expansion step starts at an expandable node with state s and adds an unvisited child of that node to the search tree. If there are multiple unvisited children, the selection is based on the prior probability P. The simulation step reaches a terminal state s_t from the expanded child by iteratively applying the default policy according to P and obtains a cumulative reward $R(s_t)$.

Backup (lines 22-28). In the backup step, we update the statistics of nodes $(G(s), N(s))$ along the path from the expanded node to the root with $R(s_t)$.

After the three steps above, the search tree is updated and a potential assignment corresponding to s_t is obtained. Subsequently, we implement a postprocessing procedure $Post$ [11] (line 29) on s_t and update the best assignment (lines 30-33). The core idea of $Post$ involves enumerating the elimination of each word from the fill, substituting it with the most suitable word selected by our method for the respective slot. This strategy has been proven to be effective in rectifying a small number of incorrectly filled letters.

In our CP problem, it is common for different sequences of actions to result in the same assignment. This means that different nodes in the search tree may correspond to the same state. In a deterministic MDP, a standard MCTS typically stores statistics to nodes or edges, resulting in the statistics of a state being distributed across multiple corresponding nodes. However, when estimating the expected rewards of a node, the statistics for the corresponding state are not fully utilized, leading to potential estimation bias. To address this, we aggregate the statistics of nodes that correspond to the same state. This aggregation enhances the tree policy by reducing the bias in estimating the expected reward for a given action a on the state s. With this approach, the search tree can be seen as a directed acyclic graph (DAG). It is worth noting that similar strategies are also introduced as Monte Carlo Graph Search (MCGS) in some recent works [19,8]. However, in this paper, we refer to it as a variant of MCTS named G-MCTS since they share the same procedure.

4.2. Variance control for UCB

In the selection step, we adopt the P-UCT algorithm to compute the UCB. Within this algorithm, the action value Q can be easily estimated by the statistics available in the tree. Specifically, we determine $Q(s,a) = G(s')/N(s')$, where $s' = T(s,a)$ represents the next state. However, it is important to note that the estimated Q value is subject to high variance due to various puzzles (obscure clues may get lower rewards) and different states (subsequent states usually get lower Q-value). To address this problem, we propose utilizing the state value of s, denoted as $V(s)$, which is calculated as $G(s)/N(s)$. This state value serves as a normalization for the aforementioned term. In this paper, the UCB is defined as follows:

$$\text{UCB}(s,a) = \frac{G(s')N(s)}{G(s)N(s')} + \lambda \frac{\sqrt{N(s)}P(s,a)}{1+N(s')}. \tag{4}$$

The first term can be viewed as the advantage of the action. Although the range of Q in UCT algorithms should be in $[0,1]$ [23], our strategy works well in practice by tuning a proper value for hyper-parameter λ. This strategy shares similar ideas of setting λ to state or action values [2,5,16]. The key distinction here is that we present the variance control as a normalization mechanism, thus we can further improve the performance by tuning λ.

4.3. Action space reduction

In the expansion and simulation step, the algorithm needs to comprehensively explore all possible actions for each state, ensuring thorough node expansion. However, the action space is huge since each action must consider every word v in the immense domain D for each word slot. This expansive action space would cause the MCTS algorithm to become trapped in exhaustive action enumeration at shallow levels. To address this issue, we need to optimize the efficiency of action exploration. It is time-consuming to even locally consider each word v in D, let alone the exploration for selection. Hence, it becomes essential to generate high-quality candidate words for each slot, prioritizing their exploration within the search algorithm. To this end, we use clues in the CP and the grid constraint F to narrow down these candidates (Detailed information on this process is described in Section 5). After generating candidates, the word selection domain D (size about hundred thousand) for each slot x is reduced to a much smaller candidate set D_x (size about hundreds). However, despite this reduction, the action space remains extensive for the exploration in the selection step, as hundreds of words need to be selected for each of the hundred slots.

To further mitigate the challenge, we propose an action space reduction strategy. First, we rank candidate words v for each clue c_i with the reward function $r(v, c_i)$. Then, we only include the filling of the word with the highest rank for each clue in the action space. Besides, for a non-terminal state s, we design an *additional action* to take other words into consideration.

$$A_s = \{(i, \arg\max_v r(v, c_i)) | 1 \le i \le q \wedge s_i \in \Delta\} \cup \{\alpha\}. \tag{5}$$

The additional action α makes a special transition from the current state to another state with the same assignment. The new state eliminates the highest-rank candidate for each clue and will consider filling the rest of the candidates as actions. Note that the additional action is recursively performed, meaning it systematically considers all candidate words for each clue. As the highest-rank candidates are eliminated, the rank-2 candidates become the highest rank in the new state. The new state also associates with an additional action eliminating the current highest-rank candidate. With the additional action design, the definition of the state is technically changed to the assignment plus the number of top-ranked candidates eliminated. Due to this strategy, the state should consider the number of elimination actions. So a state is represented as $[s_1 \dots s_q]$ plus $[n_1 \dots n_q]$ where n_i is the number of higher-ranked responses (that satisfy the grid constraints) eliminated for clue i.

4.4. Default policy

In the simulation step, the default policy is to use the prior probability distribution to take an action from the action space for each state. As mentioned in Section 4.3, the action space A_s of the state s includes the actions that fill the highest-rank candidate word to each word slot and the additional action. Inspired by [11], a good strategy to fill the word is to maximize the difference between the first-largest and the second-largest score, which is usually called *regret* [18]. So, we define the score for a prior heuristic strategy for an action $a \in A_s$ as:

$$H(s,a) = \begin{cases} r(v, c_i) - max2_{v'} r(v', c_i), & \text{if } a = (i, v) \\ 0, & \text{otherwise} \end{cases}, \tag{6}$$

where $max2(\cdot)$ means the second-largest score. Hence, the prior probability $P(s,a)$ is then defined as the Boltzmann distribution based on the heuristic H:

$$P(s,a) = \frac{e^{(H(s,a)/\tau)}}{\sum_{a' \in A_s} e^{(H(s,a')/\tau)}}, \tag{7}$$

where τ is a hyper-parameter to trade off the exploitation against exploration on the actions. When τ gets smaller, the default policy tends to exploit the actions with a high value of H, and when τ gets larger, the default policy tends to explore the actions uniformly.

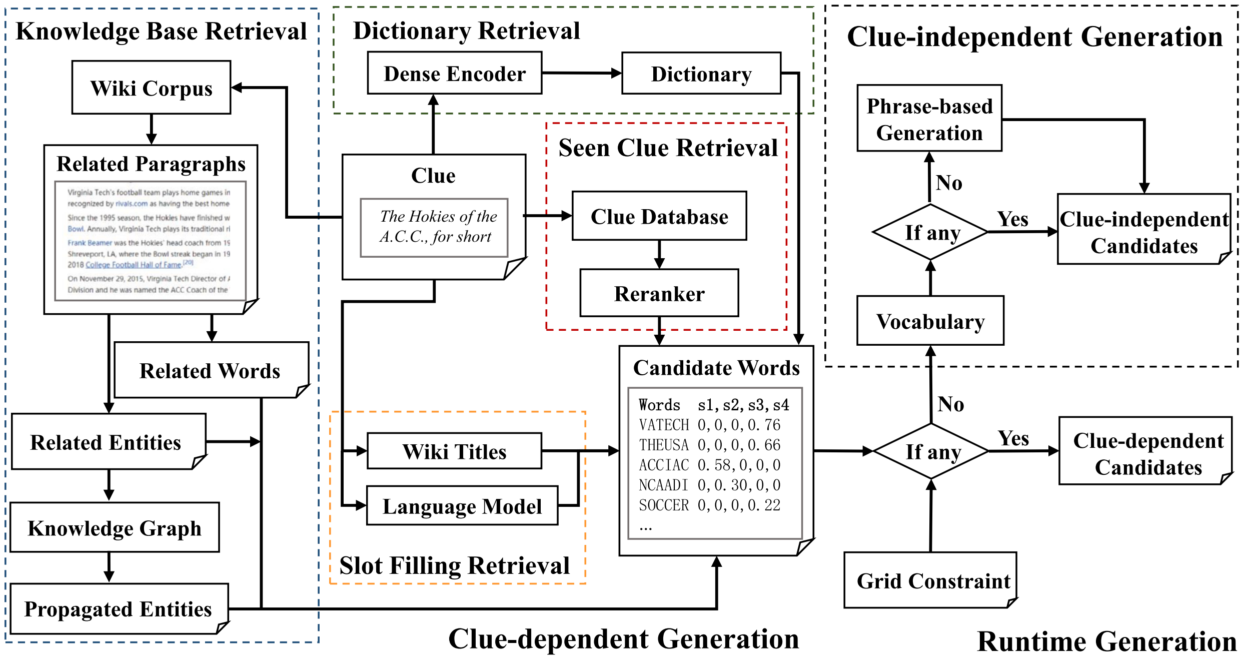


Fig. 3. Our candidate generation is divided into two categories, the clue-dependent generation, and the clue-independent generation. The former generates quality but limited candidates before the search starts, including *seen clue retrieval*, *dictionary retrieval*, *knowledge base retrieval*, and *slot filling retrieval*. The latter generates comprehensive but unreliable candidates, including *vocabulary retrieval* and *phrase-based generation*. They work during the game when none of the previous candidates satisfy the grid constraint.

5. Candidate generation

As mentioned in Section 4.3, candidate generation is expected to be effective in producing high-quality candidate words for each slot while consuming time as little as possible. To achieve this goal, we propose a systematic workflow inspired by human behavior in playing a CP game. When a human engages in a CP game, they typically begin by filling in answers for which they have the highest confidence based on the clues provided. As more words are resolved, they consider the grid constraints to eliminate answers that are difficult to infer from the given clues. Additionally, they use the grid constraints to make educated guesses for answers even without specific clues.

Based on these observations, we categorize the candidate generation modules into two groups: *clue-dependent generation* (CDG) and *clue-independent generation* (CIG), as illustrated in Fig. 3. CDG is executed before the search starts. During the search, the algorithm filters the CDG candidates according to grid constraints. In situations where none of the CDG candidates satisfy the grid constraints, the CIG module is activated. This process of generating candidates during the search is referred to as *runtime generation*. CDG consists of four modules: seen clue retrieval, dictionary retrieval, knowledge base retrieval, and slot-filling retrieval. CDG excels in retrieving high-quality candidates but may suffer from a lower recall, potentially missing some good candidates. To improve recall, CIG is employed, especially when CDG fails to generate any candidates. CIG generates candidates that adhere to the grid constraints by leveraging vocabularies or phrases. While CIG retrieves candidates that are less reliable, it ensures comprehensive coverage. It first searches for a valid candidate word within the vocabulary. If no match is found, CIG attempts to generate a multi-word compound phrase that satisfies the grid constraint.

5.1. Clue-dependent candidate generation

Seen clue retrieval This module generates candidate answers for the query clue by returning the answers of similar seen clues already present in the database. For instance, given the query clue “Some Catholic gift shop purchases”, the clue retrieval module searches the database for a similar clue, such as “Cathedral gift shop items”. It then uses the answer associated with the retrieved clue, such as “ROSARIES”, to fill in the slot. To achieve this goal, we design a two-stage strategy. First, we query the puzzle clue to retrieve related clues from the database and return the corresponding answers with the proper length. For each puzzle clue, we select the top-ranked γ_1 answers of seen clues with the highest BM25 score as candidate words. Here, BM25 is a bag-of-words ranking function that scores a set of documents (clues in the clue database) based on the query terms (clues in the CP) according to their relevance [34]. We then re-rank these candidates to achieve more accurate retrieval results. The dominant signal for ranking these candidates is the clue similarity with seen clues, referred to as S_{seen}. To perform this matching task, we adopt the Siamese [14] architecture with the pre-trained language model (e.g., BERT [9]) as the text encoder for calculating the similarity between two clues. In the Siamese architecture, we encode the query clue c_q and database clue c_d with two identical BERT encoders, i.e., BERT(c_q) and BERT(c_d). Then, we use the encodings of "[CLS]" tokens as the vectors to represent the two clues, $\mathbf{v}(c_q) = \text{BERT}(c_q)_{[CLS]}$ and $\mathbf{v}(c_d) = \text{BERT}(c_d)_{[CLS]}$. Finally, we calculate the matching score with the dot product of the two vectors, $S_{seen} = \mathbf{v}(c_q) \cdot \mathbf{v}(c_d)$. However, it is much harder for neural models to match the clues because they are usually purposely paraphrased and ambiguous. Hence, we create a dataset of similar clues from the seen clue database and fine-tune the BERT encoder on it using contrastive loss [43,17]. The contrastive loss is defined as $L_{infoNCE} = \sum_{c,c' \in \mathbb{S}} \frac{exp(\mathbf{v}(c)\cdot\mathbf{v}(c'))}{\sum_{c'' \in \mathbb{C}} exp(\mathbf{v}(c)\cdot\mathbf{v}(c''))}$, where $\mathbb{C}$ is the set of clues in the clue database and $\mathbb{S}$ is the set of similar clue pairs. Due to the high cost of manually constructing datasets, we employ distant supervision [26] to train our model. Distant supervision is a learning scheme in which the model is learned given a weakly labeled training set, where the data is automatically labeled based on heuristics. In this paper, we define our labeling heuristics as follows: if clue x and clue y ($y \neq x$) shared the same answer, then x and y are considered similar clues.

However, the labeled data generated through distant supervision is prone to noise due to the weakness of the assumption. There are two main challenges. First, words often have multiple meanings, leading to situations where clues x and y may describe different senses of the same answer word. For example, the word "NIKE" can answer both "Adidas alternative" and "Goddess of victory", but the former clue refers to the sports brand and the latter refers to the name of a Greek goddess. Second, there are numerous ways to describe a word, so clues x and y might capture different aspects of the word. For instance, clue "Purr producer" and clue "Curiosity victim" both refer to the word "CAT" but they have totally different meanings. To address these issues, we propose two strategies to strengthen the assumption, respectively.

First, we introduce an additional condition requiring that clues x and y should exhibit textual similarity. Specifically, for a given clue x, we retrieve related clues from the database based on the ζ_1 highest BM25 scores. We then select those clues that share answers with x as similar clues. Second, we consider the overlap of shared answers. The underlying idea is that the more answers two clues have in common, the more likely they are to be similar. For each clue x in the clue database $\mathbb{C}$, we define D_x as the answer set of clue x. To quantify the confidence that clues x and y are similar, we introduce a metric as follows:

$$conf(x, y) = \frac{\sum_{z \in D_x \cap D_y} g(z)}{\sum_{z \in D_x \cup D_y} g(z)}, \quad g(z) = \log \frac{|\mathbb{C}|}{|\{x' \in \mathbb{C} : z \in D_{x'}\}|}, \tag{8}$$

where $g(z)$ denotes the inverse frequency of answer z. The more clues an answer has, the more likely it is polysemous or has many different aspects, and the less influence it will have on the similarity of the two clues. We select clue pairs (x, y) with $conf(x, y) \geq \zeta_2$ as similar clues. In addition to the above two strategies, we also adopt the data augmentation tool nlpaug [22] to generate more similar clues. For example, for the clue "Start of old hymn", we obtain an augmented new clue "Start of old anthem" as its similar clue for training. The hyperparameters ζ_1 and ζ_2, as well as the statistical information on the process of constructing the dataset, are detailed in Section 8.5.1.

Dictionary retrieval This module is designed to retrieve candidate words from a dictionary by comparing the similarity of their glosses (definitions or explanations) to the given clue. The rationale behind this module is that many clues, such as "Misbehaving toddler", can be matched with the definition (e.g., "a very troublesome child") of a word (e.g., "terror") in the dictionary. When the clue provides a definition of a word, it becomes a standard reverse dictionary task [13]. However, according to previous works [45], the application of the reverse dictionary model to the clues-answering task suffers from problems of domain shift and low vocabulary size. That is to say, since the style of the CP clues differs from the training sets of reverse dictionary models, and the vocabulary of the answers is much larger than the vocabulary that reverses dictionary models can predict, the performance of reverse dictionary models can be very poor, which we will prove in the experiments (Section 8.5.1). We find that an unsupervised dense retrieval method [15] with word2vec [24] encoding works well in practice. Specifically, we encode the gloss of each word v in the dictionary by computing the average GLoVE [30] embedding of its constituent words. Similarly, we encode the query clue c_i using the same method. We then retrieve words whose gloss encoding exhibits similarity to the clue encoding, selecting γ_2 candidates with the highest similarity score $S_{dict}(v, c_i)$ between the clue c_i and the gloss of the candidate word v in the dictionary. We adopt dense retrieval for the dictionary but not for the clue database since the quantity of dictionary gloss (0.1M) is much less than clues in the database (4M) and dense retrieval is not suitable for the large clue database due to its lower efficiency.

Knowledge base retrieval This module retrieves candidates from the encyclopedia and knowledge base. Previous works [20] only retrieve candidates from the encyclopedia ignoring the structured knowledge base. However, many clues are relevant to entities in the knowledge base. For example, a real clue "*The Hokies of the A.C.C., for short*" suggests VATECH as the answer, which is the abbreviation of an entity *Virginia Tech*. The athletic team of this entity is named *Hokies* and participates in the *Atlantic Coast Conference (ACC)*. The entities mentioned in the clue can be easily found from existing knowledge bases.

Hence, we propose a relatively efficient method to retrieve candidate words with the help of an encyclopedia and knowledge base. The basic idea is to retrieve candidates from relevant texts in Wikipedia articles, extract linked entities in the context, and further expand more related entities through the relations in Wikidata [41] (a widely used knowledge base). Given a clue c_i, we determine the candidate words in relevant texts retrieved from Wikipedia articles with BM25. For each retrieved text p relevant to c_i, we extract three types of information, words and phrases $K_W(p, c_i)$ from the context, linked entities $K_E(p, c_i)$ from the context, and entities $K_P(p, c_i)$ propagated by Wikidata relations based on $K_E(p, c_i)$. We provide an example to clarify the acquisition of K_E and K_P. Consider the query clue "The Bulldogs of the N.C.A.A.". The module retrieves the Wikipedia page for "N.C.A.A." (National Collegiate Athletic Association) and obtains the linked entity "University of Georgia" in the description, which is added to set K_E. Simultaneously, it identifies the entity "Georgia Bulldogs" by retrieving the article title related to "Bulldog" and propagates to obtain the entity “University of Georgia” for the set K_P through the Wikidata relation <“Georgia Bulldogs”, “university”, “University of Georgia”>. These two signals can be combined to reinforce the confidence in the entity “University of Georgia”. Moreover, the abbreviated name “UGA” can be obtained through entity information in Wikidata. Finally, we aggregate words in $K_W(p, c_i)$, entity names in $K_E(p, c_i)$ and $K_P(p, c_i)$, and then obtain the top γ_3 candidates v according to the following score.

$$S_{kb}(v, c_i) = \sum_p [B(v, K_W(p, c_i)) + \eta_E B(v, K_E(p, c_i)) + \eta_P B(v, K_P(p, c_i))], \tag{9}$$

where $B(v, Set)$ equals 1 for $v \in Set$ and 0 otherwise. We tune the hyper-parameters η_E and η_P in the validation set to achieve the best mean reciprocal rank.

Slot filling retrieval This module is designed to generate candidates for fill-in-the-blank type clues. When the blanks belong to commonly used phrases, such as “*Here’s the __*”, neural language models like BERT can easily generate suitable words, such as “Thing”. To achieve this, we replace the blank in a given clue c with a mask token and input it into BERT. We then retrieve candidates by selecting words w with the highest restored probabilities $P_{lm}(w|c)$.

However, when the blanks correspond to long-tailed phrases, such as “*‘__ the Man’ (2006 film inspired by ‘Twelfth Night’)*”, language models often struggle to predict the answer “She’s”. This is because language models are suitable for predicting commonly used phrases, but struggle with predicting less common ones. We observe that these blanks are often hinted at by the name of an entity in Wikipedia. Hence, we complement the language model generation by retrieving words from article titles in Wikipedia. First, we consider the clue c as a query to retrieve a related set of titles E_c. Then, we define the probability of retrieval for each title $e \in E_c$ as:

$$P_{retr}(e|c) = \frac{score(e, c)}{\sum_{e' \in E_c} score(e, c)}, \tag{10}$$

where $score(e, c)$ is the BM25 score of title e as document and clue c as the query. Finally, we obtain the words v in titles with a consistent context to the blank in the clue:

$$P_{context}(v|e, c) = \frac{con(v, e, c)}{min(1, \sum_{v' \in e} con(v', e, c))}. \tag{11}$$

$con(v, e, c) = 1$ when the context of word v in e is consistent with the blank in c, it equals 0 otherwise. Specifically, the context is consistent when the left and right words of v equal those of the blank. We retrieve candidates by obtaining words v with high probabilities of

$$P_{name}(v|c) = P_{retr}(e|c) \cdot P_{context}(v|e, c). \tag{12}$$

We combine the above two metrics to retrieve γ_4 candidates with

$$S_{sf}(v, c) = \eta_{sf} \cdot P_{lm}(v|c) + (1 - \eta_{sf}) \cdot P_{name}(v|c). \tag{13}$$

We tune the hyper-parameter $\eta_{sf} \in [0, 1]$ in the validation set to achieve the best mean reciprocal rank.

5.2. Clue-independent candidate generation

Vocabulary generation When none of the candidates from clue-based generation satisfy the grid constraint, we retrieve candidates directly from a large vocabulary. The vocabulary contains all the answer words in past puzzles, all the words, and forms from the Wiktionary [23], and all the titles from Wikipedia pages within a limited length (maximum grid size). Specifically, we retrieve candidate words from the vocabulary that satisfy the grid constraints (i.e., cross consistency and length constraint as defined in Section 3.1). If the number of candidates meeting these constraints is too high (beyond the predefined hyperparameter γ_5), we judge this module as ineffective in narrowing down candidates, thus returning an empty set. Otherwise, if the number of candidates is less than γ_5, we return the retrieved set of candidates.

Phrase-based generation In situations where none of the candidates from the vocabulary satisfy the hard constraint, the remaining options for the fill often involve phrases with multiple words that are not included in the vocabulary. For example, the answer for “*End of the verse*” is `FINDAWAYTOBEATTHEWRAP`, which represents a long phrase “*find a way to beat the wrap*”. This kind of clue is hard to answer and we usually leave it at last when many crossed words are filled and we use the grid constraints to “guess” the phrase. Notably, we generate only one candidate in this module for two reasons. First, the candidates generated through guessing are not highly reliable and do not deserve many tries. Second, the grid constraints should be strong enough to limit the available options. In this module, we adopt a segmentor [20] to generate the candidate phrase $v \in D$ that satisfies the current grid constraints. The segmentor partitions a sequence of letters into a phrase with the maximum logarithm probability $S_{phrase}(v)$. For instance, the slot may have been filled with other intersecting words during the search, forming constraints such as `FI_DAWA_TO_EATT_EW_AP`. To generate the answer, the segmentor identifies the best partition with the highest phrase probability, i.e., `FI_D/A/WA_/TO/_EAT/T_E/W_AP`. The phrase probability is calculated as the product of word probabilities produced by the unigram language model, where the probability of each word is independent of each other, i.e., $P(\textit{find a way to beat the wrap}) = P(\texttt{FIND}) \cdot P(\texttt{A}) \cdot P(\texttt{WAY}) \cdot P(\texttt{TO}) \cdot P(\texttt{BEAT}) \cdot P(\texttt{THE}) \cdot P(\texttt{WRAP})$. More formally, given a letter sequence l where some are letters and some are placeholders, we need to find a

Table 1
Specifications of features used in the reward function.

Type	Feature	Description
Clue-aware features	S_{seen}	Score of seen clue retrieval
	S_{dict}	Score of dictionary retrieval
	S_{kb}	Score of knowledge base retrieval
	S_{sf}	Score of slot filling retrieval
	W2V	Word2vec similarity of word and clue
	POS	POS tag matching of clue and word
Clue-free features	S_{vocab}	Whether the word is in the vocabulary
	S_{phrase}	Score of phrase-based generation
	Occur	Log # of occurrences in clue database
	Unigram	Log unigram counts

sequence of words $U = [w_1, ..., w_k]$ with maximum probability, $P(U)$, and the sequence of words should match the letters in l. Under the unigram assumption, the objective is to find a sequence of words with the maximum probability

$$\arg\max_{U} \sum_{i=1,..,k} \log P(w_i) \quad s.t. \quad M(l,U) = 1, \tag{14}$$

where $M(l,U) = 1$ denotes the sequence of words matches the constraint in l. The probability of words $P(w)$ can be calculated by their frequency

$$P(w) = \frac{Count(w)}{\sum_{w'} Count(w')}, \tag{15}$$

where $Count(\cdot)$ is the number of occurrences of the word in the corpus. Since the problem has an optimal substructure, we obtain the best word sequence via dynamic programming.

6. Reward function

In this section, we introduce the design of the reward function specifically tailored for the MCTS algorithm. The objective of our algorithm is to identify an assignment that maximizes the accumulated reward described in Eq. (3) for a CP. Unlike games with easily defined rewards, such as a simple win or lose outcome in games like GO, the reward in a CP relies on the degree of matching between words and clues, requiring careful definition. Designing the reward function poses two main challenges. First, a robust reward function necessitates a strong language-understanding ability for the machine, as even obscure and vague clues can be challenging for humans. Hence, we need to combine multiple features while controlling the noise introduced by specific features, as not all features are consistently effective. Second, a good reward requires evaluating candidate words from different sources simultaneously, since computing them separately makes it difficult to compare them with each other through a standard metric. Hence, we need to design a global reward function combined with multiple ranking models from different sources.

6.1. Ranking features

The essence of the reward function $r(s_i, c_i)$ is evaluated by the matching degree between a word and a clue. We introduce various features in Table 1 that effectively capture how well a word aligns with a clue. These features used in our algorithm are divided into two categories: clue-aware and clue-free features. Clue-aware features measure the matching degree between the clue and the word and are effective for explicit or easily understandable clues. Clue-free features assess the quality of the word itself, independent of the clue, and are useful for obscure clues. For clue-aware features, in addition to four retrieval scores obtained from clue-dependent candidate generation (as described in Section 5.1), we incorporate two additional features: POS tag matching degree and word2vec similarity. The idea behind the POS tagging feature stems from the observation that crossword clues often provide information about the answer word's part of speech (POS). For instance, the answer for the clue "*Doing some canoodling*" is `SMOOCHING`, while the answer for the clue "*Works too hard on public transport*" is `OVERTRAINS`. Hence, we train a text classifier to predict the POS tags of words based on their corresponding clues. To accomplish this, we leverage distant supervision [26] using clues from past puzzles and POS tag information from the dictionary. Additionally, we utilize word2vec [24] to measure the similarity between the candidate word and the clues. Specifically, we calculate the similarity between the vectors of the candidate word's vector and the average vector of the clue, which serves as another clue-aware feature. For clue-free features, we consider four aspects: a binary score $S_{vocab}(v)$ to indicate whether the word (string) $v \in D$ is in the vocabulary (1) or not (0), the score produced by phrase-based generation (illustrated in Section 5.2), the number of occurrences (denoted as *Occurrences*) in the clue database, and the unigram count (denoted as *Unigram*) of the word.

6.2. A ranking model baseline

A typical solution is to use the scores provided by a ranking model trained on the ground-truth clue-answer pairs as the reward function $r(w, c), w \in D, c \in C$ for answers to a single clue [28]. To achieve this, we leverage the aforementioned features to train a ranking model denoted as $f(w, c; \theta)$, which assesses how well a candidate word w matches a clue c. For the optimization of our ranking model, we employ the ListMLE algorithm [44]. In practice, for a given clue c, we compute the negative log-likelihood of the ground truth (answer) y_c from a list of candidate words W_c, and minimize it to learn the model parameters:

$$L(f;\theta) = -\sum_{c}\sum_{w\in W_c} y(w,c)\log\frac{e^{(f(w,c;\theta))}}{\sum_{w'\in W_c} e^{(f(w',c;\theta))}}, \tag{16}$$

where $y(w,c) = 1$ if w is the answer of c, and $y(w,c) = 0$ otherwise. Then we define the reward function as the trained ranking model, i.e., $r(w,c) = f(w,c;\theta)$.

6.3. *Reward function based on multi-source ranking*

However, the above solution of using the ranking model as the reward function is not optimal for the assignment evaluation due to its inability to handle data from different sources effectively. First, different features have varying relevance in different sources. When ranking candidates from CDG, the retrieval scores hold dominance, while other features may introduce noise. Conversely, when ranking candidates from vocabulary, the retrieval scores become irrelevant. Second, a single model cannot fit the data from diverse sources in a unified manner, as the information contained within these sources differs significantly. While a data-driven approach excels at fine-tuning rankings for a large volume of samples, it struggles to learn a global strategy for candidates from which source is more important. Thus, we design a novel reward function for matching the word s_i and the clue c_i, which is defined as:

$$r(s_i, c_i) = \phi(s_i, c_i) + \omega_1\rho(s_i, c_i) + \omega_2 I(s_i), \tag{17}$$

where ϕ and ρ are two ranking models for answers sourced from the CDG and CIG, respectively. The third item is a reward for words found in the vocabulary (or a punishment for the words not found in the vocabulary), denoted as $I(s_i) = max(S_{vocab}(s_i), S_{phrase}(s_i))$. Hyper-parameters ω_1 and ω_2 (both greater than zero) are utilized to control the scaling of rewards for different sources. Both ϕ and ρ are neural networks trained on data from different sources. In general, neural networks with more parameters could bring higher performance. However, the complex networks would negatively impact search efficiency. Hence, we adopt simple but effective one-layer linear models as neural networks in practice. ϕ exclusively utilizes the four features from CDG, while ρ utilizes the remaining features except for S_{vocab} and S_{phrase} mentioned in Table 1. We train the ϕ using data samples of clue-answer pairs obtained from the clue database in the training set. For the training ρ, we construct a dataset by running a basic MCTS without the reward of ρ on the training set and obtain the data samples from the candidate generation of the vocabulary source.

7. Dataset construction for crossword puzzle resolution

There is currently no public evaluation benchmark (including puzzles for testing and a clue database for training) for CP problems. Some previous methods [20,10] tested their approach on self-selected puzzles, which are not publicly available. Since the criteria for selection are not specified, it is hard to compare these methods. While the American Crossword Puzzle Tournament (ACPT) puzzles have been used for testing by some methods [11,42], they have two limitations as a public benchmark. First, the ACPT test puzzles are insufficient for comprehensive evaluation. Each ACPT only includes seven puzzles, with over half of them being easy. Second, it is difficult to compare the different methods in a fair manner, since the clue database is not specified for evaluation. In practice, the source of the clue database significantly influences the difficulty of the test puzzle, because the more seen clues are in a CP, the easier it gets. Hence, it is crucial to construct a dataset with puzzles and a clue database fairly selected.

We construct a dataset based on NYT daily crossword puzzles from 1/1/1994 to 11/30/2020 and online clue databases. Because the online clue database is updated in real time, one problem in constructing such a dataset is the leakage of the new puzzles. When players are playing crossword puzzles, the clue database they can use should only contain clues from crossword puzzles in the past time. To address this issue, we set a split date. Clues collected before the split date are used for the dataset, while puzzles after the split date are used for testing.

According to this criterion, we construct our test set by setting the split date as 9/26/2020. After splitting data, we obtain 66 puzzles from NYT Daily puzzles. We filter out non-standard puzzles whose grids contain special rules or multiple letters in one square. This filtering results in 57 standard crossword puzzles, which we refer to as *standard test set*. In this setting, we only use the clues before 9/26/2020 in the database as our data source. We aggregate clues from two clue databases before the date. We remove duplicates and result in a clean clue database of 4,568,786 unique clues with associated answers. Note that the clue database is a part of the dataset, all comparative methods can exploit this database and use the feature of the matching degree with seen clues. The size of grids in our dataset ranges from 15×15 to 21×21, with the total size of words ranging from 66 to 140. Except for the *standard test set*, we additionally select a *hard test set* from it to show the capability of our algorithm. The motivation behind this selection is that many puzzles in the dataset are too simple and can be easily resolved because most of the clues can be found in the clue database, even though we set the split date. The construction process of the hard test set is as follows. For each clue in the puzzle from the standard test set, we use it as a query to retrieve related clues from our built database and rank the answers of retrieved clues through the relevance score (BM25) between the query clue and retrieved clues. We select the top 50 candidates for each clue and consider the answer corresponding to the clue as “recallable” when it occurs in the 50 candidates. Intuitively, a puzzle is considered easier if it has more “recallable” answers. We then calculate the ratio of “recallable” answers for each puzzle in the standard test set and select puzzles with a ratio of no more than 80%. This process results in a set of 24 puzzles denoted as the hard test set.

Table 2
Statistics of datasets for crossword puzzle resolution used in our experiments.

Dataset	# Puzzles	# Seen Clues	Split Date
validation set	85	4,510,058	06/18/2020
standard test set	57	4,568,786	09/25/2020
hard test set	24	4,568,786	09/25/2020
ACPT 2021	7	6,782,248 [42]	-

Table 3
Specifications of data used in our system.

Category	Sources	Quantity
Puzzle	https://www.xwordinfo.com/	9,967
Clue	https://www.crosswordgiant.com/	3,094,710
	http://www.otsys.com/clue	2,866,925
Knowledge Base	Wikidata [41]	306,198,423
Corpus	Wikipedia	6,271,708
Dictionary	WordNet 3.0 [25]	117,791
	Wiktionary [23]	999,905
Vocabulary	Various sources	10,047,262

We also construct a validation set for tuning hyperparameters and training ranking models. In this setting, we set the split date as 6/18/2020. The 100 puzzles from 6/18/2020 to 9/25/2020 are used as the training set, and we only utilize the clues before 6/18/2020 in the database as our data source. In addition, to compare with recent state-of-the-art crossword puzzle solvers, both human and AI, we also include ACPT 2021 puzzles as one of our test sets. For the clue database selection of ACPT 2021, we use the CrosswordQA dataset [42] as the clue database. The specifications of all the datasets used in our experiments are presented in Table 2.

8. Experiments

In this section, we first introduce the setup of our experiments. Next, we compare our method to existing state-of-the-art approaches as well as the best human player, demonstrating the effectiveness of our approach. Finally, we conduct a thorough analysis of each module in our system, illustrating its rationale.

8.1. Experimental setup

Data sources We provide a comprehensive list of the data sources utilized by our system for crossword puzzle resolution in Table 3. In addition to the puzzles and clues themselves, our system leverages a diverse array of external sources, including Wikidata triples, Wikipedia articles, and multiple dictionaries.[3] The expansive vocabulary employed by our system is derived from a combination of answers from past puzzles, dictionary words, and Wikipedia titles.

Metrics To assess the effectiveness of crossword puzzle resolution methods, we employ three metrics: the ratio of correct words (W-Acc) and correct letters (L-Acc) based on the ground-truth grid, along with the perfect puzzle (P-Puz) accuracy, which requires answering every clue in the puzzle correctly. Since the crossword puzzles are required to be finished in a limited time, we conduct experiments using two different time limits for testing, i.e., 100 seconds and 1000 seconds. For the 100-second time limit setting, we perform our algorithm 5 times and average the results to reduce the bias due to the randomness of MCTS. For the 1000-second time limit setting, we run our algorithm only once due to the relative stability of long-term performance and the cost involved in running this setting multiple times for 60 puzzles. For the experiments in Section 8.3, 8.4, and 8.6, we focus on the analysis of the behavior of the search algorithm. Thus, we adopt a simplified version of our method with seen clue retrieval as the only module in clue-dependent candidate generation, and run each method in settings of two different time limits (100 s and 1000 s). For the search experiments in Section 8.5, we focus on the analysis of candidate generation modules and their influence on the search algorithm. We then run each method in only the 1000-second time limit, since additional candidate generation modules take more time and leave over little for the search in the setting of the 100-second time limit.

Parameter setting We perform hyper-parameters tuning on the 100 CPs from the validation set. The process takes approximately two days to complete. The values of hyper-parameters we use are as follows: $\lambda = 0.2$, $\tau = 0.08$, $\gamma_1 = 50$, $\gamma_2 = 10$, $\gamma_3 = 30$, $\gamma_4 = 10$, $\gamma_5 = 200$, $\eta_E = 2$, $\eta_P = 0.02$, $\eta_{sf} = 0.04$, $\omega_1 = 0.7$ and $\omega_2 = 15$. Regarding the models ϕ and ρ, we find that deeper models tend to yield better performance in terms of ranking accuracy. However, they come with the drawback of being computationally inefficient during the

[3] We extract Wiktionary, Wikipedia, and Wikidata from 20210320 dumps.

Table 4
System comparison on ACPT 2021. The scores are computed with the ACPT rules and the numbers in parentheses represent the incorrect letters.

Competitors	Total	Individual Puzzles						
		# 1	# 2	# 3	# 4	# 5	# 6	# 7
BCS QA + Dr. Fill	12825 (3)	1280 (0)	1670 (0)	2075 (0)	1385 (0)	1620 (1)	2055 (0)	2740 (2)
Human Winner	12810 (0)	1255 (0)	1620 (0)	2000 (0)	1360 (0)	1690 (0)	1980 (0)	2905 (0)
BCS QA + BP + LS	13065 (1)	1280 (0)	1670 (0)	2075 (0)	1385 (0)	1620 (1)	2055 (0)	2980 (0)
Ours	**13185 (0)**	1280 (0)	1670 (0)	2075 (0)	1385 (0)	1765 (0)	2055 (0)	2955 (0)

search process. Our experiments are conducted on a computer with a CPU of Intel i7-8700K@3.70GHz, GPU of GTX 1080Ti, RAM of 32 GB, and system of Ubuntu 18.04. We implement our system using Python 3.7, and the search process is executed within a single thread. On average, our MCTS algorithm achieves a runtime of 20-40 iterations per second, depending on the puzzle size. To handle knowledge base queries, we utilize MongoDB,[4] and for text retrieval, we rely on Elasticsearch [40], both running on the local machine. The runtime memory of our algorithm consumes approximately 70% of the total system memory. This memory usage mainly includes vocabularies (about 11 GB), Elasticsearch (about 4 GB), Word2vec (about 2 GB), neural networks (about 2 GB), and others (about 3 GB). The runtime memory for the search tree (0.02 GB[5]) is negligible since we only store the statistics of the nodes on the tree. We do not cache other states generated during the simulation because the visited number N and cumulative reward G of nodes are only considered and start counting when the nodes are expanded to the tree and the states encountered during the simulation are disposable. Note that actions for each state can be cached to accelerate the search. However, we only cache the actions of the nodes on the search tree and do not cache those of states during simulation because they will be out of the memory in our environment.

8.2. System comparison

In this section, we evaluate the overall performance of our system on crossword puzzle resolution. We conduct experiments on two different test sets: the ACPT 2021 tournament puzzles and our built standard test set. For a fair comparison with the state-of-the-art methods, we uniformly use the CrosswordQA dataset [42] as the clue database. Compared to our dataset, the ACPT 2021 tournament has a different evaluation setting. It consists of 7 puzzles of varying difficulty levels. Each puzzle is assigned a specific time limit for completion. The time limits for the 7 puzzles are as follows: 15, 25, 30, 20, 30, 30, and 60 minutes, respectively. Scoring in the tournament takes into account both accuracy and speed, following the criteria outlined below[6]: 1) 10 points for every correct word the player entered across and down. 2) A bonus of 25 points for each full minute the player finished ahead of the suggested solution time — BUT reduced by 25 points for each missing or incorrect letter (but not beyond the point the bonus returns to zero). 3) A bonus of 150 points for each completely correct solution.

In the evaluation of ACPT 2021, we compare our system with the state-of-the-art Berkley Crossword Solver [42]. There are two versions of the crossword solver. The first version, BCS QA + Dr. Fill, is an improved version based on Dr. Fill [11]. It adopts Dr. Fill's search algorithm (LDS) with their proposed QA method, and achieves the performance over the human winner in ACPT 2021. The second version, BCS QA + BP + LS, substitutes the search algorithm with belief propagation (BP) and local search (LS), achieving better performance on the ACPT 2021 dataset. In addition, to adapt the scoring rules, our algorithm follows a strategy that automatically stops when the best solution remains unchanged for 30 seconds during the search. However, to prevent premature stopping due to the inherent instability of the MCTS algorithm, we establish a minimum search time for each puzzle. Specifically, we set a minimum of 1 minute for puzzles 1, 2, 3, 4, and 6, and 3 minutes for puzzles 5 and 7. This ensures that the algorithm has sufficient time to explore potential solutions before making a decision. The results of our evaluation are presented in Table 4. Analyzing the data, we have the following observations: 1) Our algorithm outperforms all competitors in terms of total score, indicating its superior effectiveness. In particular, our method correctly solves all puzzles. 2) Baselines hold a distinct advantage over human players in terms of efficiency, with AI competitors quickly resolving simpler puzzles and achieving higher overall scores. However, when it comes to accuracy, they are beyond comparison with human players. Our algorithm, on the other hand, equals human accuracy while being faster. 3) Our method shows slightly slower efficiency in puzzles 5 and 7 compared to BCS QA + BP + LS,[7] likely because these puzzles are more complex and require our algorithm to spend additional time on search space exploration.

In the evaluation of our dataset, we compare our system with the current state-of-the-art method, i.e., BCS QA + BP + LS. The results are reported in Table 5. We can observe our method also outperforms BCS QA + BP + LS under both the 100-second and

[4] http://www.mongodb.org/.

[5] The number of expanded or visited nodes in the search tree is determined by the iteration count, which is about tens of thousands.

[6] https://www.crosswordtournament.com/info/brochure.htm#scoring.

[7] Based on the scoring rules, our method trails behind BCS QA + BP + LS by 2 minutes on puzzle 5 and 1 minute on puzzle 7. Here's the breakdown of the calculation: For puzzle 5, assuming BCS QA + BP + LS correctly solves the puzzle, their total score would be 1620 for the puzzle, plus 20 (for two correct words), plus 25 (for one correct letter), plus 150 (for a completely correct solution), totaling 1815. This score is 50 points higher than our score of 1765, corresponding to a 2-minute advantage since 25 points are equivalent to a 1-minute speed advantage. On puzzle 7, their score of 2980 exceeds ours by 25 points, or exactly 1 minute, with our score being 2955.

Table 5
System comparison on the standard test set. The number in the lower right corner represents the standard deviation calculated by running the dataset five times. The standard deviation for all metrics of the baseline algorithm is 0 because the baseline is a deterministic algorithm, without any stochastic decisions. The standard deviation of P-Puz typically exceeds that of W-Acc and L-Acc due to the more pronounced randomness in P-Puz. For instance, a shift from 99% to 100% in L-Acc represents a mere 1% increase. However, for P-Puz, this enhancement could signify a substantial leap from being nearly correct to entirely correct, constituting a 100% change.

Time Limits	100 s			1000 s		
Metrics	W-Acc	L-Acc	P-Puz	W-Acc	L-Acc	P-Puz
BCS QA + BP + LS	90.10	91.06	82.46	99.56	99.84	89.47
Ours (MCTS)	**99.61±0.32**	**99.87±0.10**	**91.58±1.31**	**100.00**	**100.00**	**100.00**

Table 6
Search algorithm comparison on the standard test set. The standard deviation for all metrics of the baseline algorithms is 0 because they are deterministic, without any stochastic decisions.

Time Limits	100 s			1000 s		
Metrics	W-Acc	L-Acc	P-Puz	W-Acc	L-Acc	P-Puz
Webcrow (WA*)	36.21	47.16	0.00	36.21	47.16	0.00
WA*+Heuristic	63.34	77.06	22.81	63.43	77.12	22.81
Dr. Fill (LDS)	88.04	93.99	43.86	89.69	95.17	45.61
Ours (MCTS)	**94.05±0.14**	**97.30±0.15**	**58.95±1.40**	**97.04**	**98.81**	**70.64**

1000-second time limit settings. In particular, our method successfully completes all puzzles given sufficient search time. In the 100-second time limit setting, BCS QA + BP + LS shows significantly inferior performance. This is due to their deep QA model for answer retrieval, which usually takes 1-3 minutes to complete, resulting in exhaustion of the time limit with little or no time left for search. In contrast, our efficient clue-dependent candidate generation module only takes 10-20 seconds to complete, leaving a much longer time for search in the constrained time limit setting. However, the total search time of BCS QA + BP + LS is less than ours. BCS QA + BP + LS always terminates before reaching the 1000-second time limit because they adopt a local search strategy with fewer states to search.

8.3. Search algorithm comparison

In this section, we conduct a comparative analysis of our MCTS algorithm against other search algorithms used for solving CPs on the standard test set with our built clue database. Since the source codes of previous works on crossword puzzle resolution are not available, we re-implement the search algorithms, keeping the candidate generation and scoring function of our method as baselines. It is important to note that these re-implemented baselines are solely used for comparison purposes and may not necessarily represent the best-performing systems reported in the respective papers. The baselines are as follows:

- Webcrow (WA*) [10]. Webcrow adopts a weighted A* search algorithm for grid filling. In our re-implementation, we remove the web search module due to its time-consuming nature and potential test set leakage, as we only consider clues available before a specific date. In the A* search, the heuristic for a state s is defined as $h(s) = \sum_{1\le i\le q} max_{w\in D_{x_i}} r(w, c_i)$, which is the reward sum of all the possible best filling for each slot.
- WA*+Heuristic. We enhance the WA* algorithm used in Webcrow by incorporating the heuristic from Dr. Fill as a search order. Specifically, the heuristic of Dr. Fill is defined on a state-action pair (s, a), where $a = (i, v)$. The heuristic is calculated as $h'(s,(i,v)) = \max_v h(T(s,(i,v)) - max2_{v'} h(T(s,(i,v'))$, which is the reward difference of the best and the second best filling for the slot i. The search with the heuristic of Dr. Fill will explore the action with the highest h' value first.
- Dr. Fill [11]. It adopts LDS [12] to solve the search problem in CP. Since Dr. Fill is a closed-source algorithm and many implemental details are missing, we only re-implemented its heuristic strategy and the search algorithm, combined with our candidate generation module and reward function.

The experimental results are presented in Table 6. The results demonstrate that our method outperforms all competitors on the dataset in terms of W-Acc, L-Acc, and P-Puz, illustrating the effectiveness of the adopted MCTS approach. In particular, our MCTS-based method achieves a W-Acc that is 7.35% higher, an L-Acc that is 3.64% higher, and a P-Puz that is 25.61% higher than Dr. Fill (LDS) at the 1000 s time limit setting. In addition, a good heuristic can significantly improve the performance of the A* search algorithm. It is important to note that baselines such as [28] are not considered in this experiment because they focus on ranking the candidates for individual clues. We will discuss their scoring strategy with the ranking model in the following experiments.

Table 7
Ablation study of different improved designs in our MCTS algorithm on the hard test set.

Time Limits	100 s			1000 s		
Metrics	W-Acc	L-Acc	P-Puz	W-Acc	L-Acc	P-Puz
Ours	91.20±0.46	96.15±0.28	**25.00**±1.67	94.88	**98.04**	**42.50**
Impact of G-MCTS						
ST	80.64±0.89	89.26±0.54	4.17±2.04	80.26	89.09	4.17
Impact of Variance Control for Tree Policy						
w/o VC	90.20±1.02	95.53±0.63	19.17±1.83	92.89	96.99	33.33
$\lambda = Q/V$	88.61±0.34	94.75±0.16	15.83±1.97	94.21	97.75	33.33
$\lambda = 1$	87.04±0.76	93.52±0.54	16.67±1.75	93.38	97.25	29.17
$\lambda = 0.5$	89.06±0.25	94.50±0.34	19.17±1.58	94.15	97.45	41.67
$\lambda = 0.2$	90.31±0.26	95.73±0.34	**25.00**±2.11	**94.89**	97.99	37.50
$\lambda = 0.01$	**92.26**±0.86	**96.84**±0.65	20.83±1.90	93.76	97.35	41.67
Impact of Action Space Reduction						
AA	46.40±1.15	64.42±0.93	0.00±0.00	47.76	65.03	0.00
NA	91.31±0.92	96.22±0.70	21.67±1.49	94.15	97.36	37.50
Impact of Prior Probability						
UP	68.75±0.89	79.46±0.65	4.17±1.29	82.88	90.73	12.50
GS	89.47±0.31	95.38±0.20	15.00±2.17	89.80	95.32	20.83
Impact of Post						
w/o *Post*	90.94±0.94	95.68±0.81	24.17±1.39	94.31	97.60	39.17

8.4. Study of improved strategies in MCTS

Next, we provide a detailed analysis of the improvement strategies of MCTS in Section 4. This analysis includes G-MCTS, variance control for the tree policy, action space reduction, and prior probability for the default policy. For the specific analysis, we utilize the *hard test set* with our built clue database to evaluate the effectiveness of these strategies in this section.

Impact of G-MCTS We perform experiments to validate the effectiveness of our special design in sharing statistics among nodes with the same state within our MCTS algorithm. With this design, the conventional search tree transforms into a DAG. To demonstrate the effectiveness of our design, we establish a standard MCTS algorithm (referred to as **ST**) as a baseline, which stores statistics at each node. As depicted in Table 7, our MCTS design yields significant performance enhancements compared to the standard version. For example, our method achieves notable improvements of 10.56% in W-Acc, 6.89% in L-Acc, and 20.83% in P-Puz compared to the standard MCTS in the 100 s setting. More importantly, as the time limit is extended from 100 s to 1000 s, our method naturally exhibits even more pronounced improvements. However, the performance of the standard MCTS remains unchanged, failing to benefit from the increased time allocation. This highlights the superiority of our approach in delivering better performance over time.

Impact of variance control for tree policy We propose a variance control technique for the UCT algorithm to address the issue of variant estimated Q-values in the CP problem. To demonstrate the effectiveness of this design, we conduct extensive experiments in this section. For comparison, we utilize the standard P-UCT algorithm for the tree policy (without variance control, denoted as w/o VC) with the exploration factor λ of UCB re-tuned on the validation set. We also explore other strategies, including setting the exploration factor to the action value [2] ($\lambda = Q/V$) and the state value [16] ($\lambda = 1$). Additionally, we experimented with different λ settings ($\lambda = 0.01, 0.2, 0.5$) to examine their impact on the results. The results are presented in Table 7. We can see the performance drops without our variance control technique, which proves the effectiveness of our design. Regarding the influence of the exploration factor, a lower level of exploration ($\lambda = 0.01$) encourages the algorithm to achieve better short-term assignments but limits its overall performance in the long run. Our chosen setting ($\lambda = 0.08$ tuned on the validation set) can be regarded as a trade-off between short-term exploitation and long-term exploration. To further illustrate the effectiveness of our strategy, we analyze its runtime behavior during the search. Specifically, we select a puzzle where both versions of our method yield the same results (W-Acc of 97.2%). The reward-iteration curves for MCTS without and with our variance control strategy are depicted in Fig. 4. It is evident that MCTS with our variance control strategy exhibits faster convergence and more steady improvement throughout the iterations.

Impact of action space reduction In our MCTS algorithm, we design an action space reduction strategy that involves limiting the action space to filling in the highest-rank candidate word for each clue and an additional action to consider the rest. This strategy embodies an essential trade-off between exploitation and exploration. On one hand, the strategy that maximizes exploitation involves considering only the highest-rank candidates, while on the other hand, the strategy that maximizes exploration entails considering all candidate words simultaneously. In this section, we conduct experiments to demonstrate the effectiveness of this trade-off strategy by comparing it with two alternative approaches:

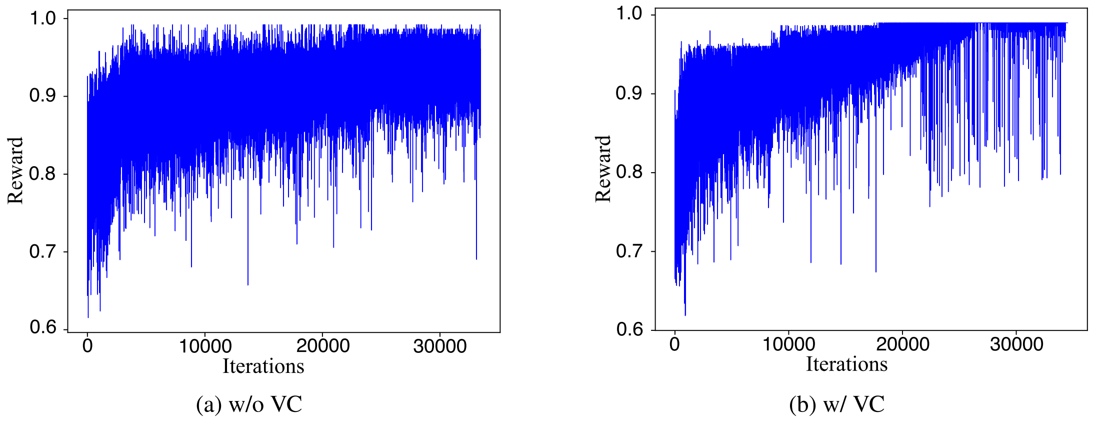


Fig. 4. Reward-iteration curve comparison between the algorithms without and with the variance control strategy.

- All Actions Included (AA). This baseline considers filling all the candidate words as actions in MCTS. It is a strategy that emphasizes exploration.
- No Additional Action (NA). This baseline only considers the highest-rank candidate for each clue without the additional action. This strategy is inclined to the exploitation of the heuristic.

The results presented in Table 7 show the effectiveness of the trade-off in our action space design. It is obviously a bad idea to consider all the candidates as actions. Besides, the additional action in our design brings performance gains, particularly as the search time grows. This is because the lack of exploration in the NA strategy limits its performance in the long run.

Impact of prior probability In our MCTS algorithm, we design a prior probability $P(s,a)$ based on a heuristic, which plays a crucial role in the expansion and simulation steps. In this section, we conduct experiments to study the influence of the prior probability on the performance of MCTS. The influences of the prior probability are twofold. First, the value $P(s,a)$ provides a prior strategy to select relatively reliable actions without any simulation statistics. Second, the randomness introduced during the simulation step creates a trade-off between exploiting the prior probability and exploring other potential options. To demonstrate the effectiveness of our use of prior probability in these two aspects, we compare our method with two alternative designs:

- Uniform Prior Probability (UP). We change the prior probability to a uniform probability.
- Greedy Simulation (GS). We select the action maximizing the prior probability in the simulation step, which is full exploitation of the prior probability.

The results are presented in Table 7. The performance of UP is notably poor, particularly in a short time limit, but shows significant improvement as the time limit increases. However, there remains a substantial gap compared to our prior probability. This indicates that the value of $P(s,a)$ is very important for the system to obtain a good start. On the other hand, the performance of GS closely matches our method in the 100-second time limit. However, its performance exhibits minimal improvement in the 1000-second time limit. This is because the greedy simulation approach lacks exploration. While it may quickly reach a good assignment based on the heuristic, it struggles to explore potentially better assignments in future iterations.

Impact of post To assess the effectiveness of the postprocessing strategy *Post*, we exclude this component from our method, and the results are presented in Table 7. From the results, we observe that this strategy slightly contributes to the system's performance. For example, our method, which integrates *Post*, exhibits a decrease in W-Acc of 0.26% and 0.57% compared to the method without *Post* at settings of 100 s and 1000 s, respectively. This indicates that although the strategy is simple, it remains necessary for enhancing overall effectiveness.

8.5. *Study of candidate generation*

During the candidate generation process (in Section 5), we propose four candidate answer retrieval modules based on the given clues, including seen clue retrieval, dictionary retrieval, knowledge base retrieval, and slot-filling retrieval. In this section, we will elaborate on each retrieval module with the hard test set and our built clue database, providing a detailed analysis of their impact on the model's performance. Furthermore, we will explore the influence of different numbers of candidate answers returned by these modules on the overall performance of the model.

8.5.1. *Impact of each module in CDG on a single clue*

To validate the performance of each module in CDG for a single clue, we conduct experiments using a dataset comprising 15,800 clues from NYT crossword puzzles between 9/26/2020 and 5/2/2021. Mean Reciprocal Rank (MRR) and Recall@k (R@k) are used as

Table 8
Comparison of different clue-dependent candidate generation modules and methods. We give the overall comparison among the categories as well as the specific comparison among different methods in individual sources.

Category	Method	MRR	R@1	R@5	R@10	R@20	R@50	Time	Coverage
Seen Clue	Textual+Rules	62.14	54.21	71.94	75.80	78.67	81.81	34	100%
	BERT	45.14	36.94	53.97	60.61	67.26	76.53	120	
	BERT-FT	**63.51**	**55.36**	**73.73**	**77.80**	**80.62**	**83.02**	120	
Dictionary	Textual	4.60	3.09	6.10	7.87	9.36	11.21	11	100%
	Dense-W2V	**5.41**	**3.33**	**7.23**	**9.77**	**12.52**	**16.58**	37	
	Dense-BERT	0.76	0.34	1.03	1.30	1.93	3.64	174	
	Dense-BERT-FT	3.71	2.03	5.05	6.80	9.07	13.21	175	
	Reverse Dict.	1.46	0.60	2.10	3.17	4.56	7.42	330	
Knowledge Base	W	11.14	7.75	14.80	17.98	21.36	25.92	181	100%
	W+E	11.62	7.86	15.73	19.34	22.93	28.03	189	
	W+E+P	**11.73**	**7.94**	**15.83**	**19.38**	**22.98**	**28.19**	233	
Slot Filling	Name	1.83	1.41	2.30	2.66	3.11	3.48	18	7%
	LM	0.33	0.18	0.44	0.55	0.74	1.14	53	
	Name+LM	**1.86**	**1.43**	**2.34**	**2.72**	**3.15**	**3.58**	70	

evaluation metrics for the candidate generation modules. The reciprocal rank of a query clue is defined as the multiplicative inverse of the rank of the ground truth in the candidate generation list. MRR is calculated as the average of the reciprocal ranks of all clues in the dataset. R@k is the proportion of cases where the ground truth appears within the first k elements of the generated candidate list. In addition to these metrics, we also present the average time taken per clue in milliseconds (Time) and the ratio of clues for which the modules generated responses (Coverage). The overall results of all the modules are presented in Table 8. We can see that the clue database plays a dominant role in candidate generation performance for single clues. Following that, retrieval from the knowledge base also provides a substantial number of candidates. In the rest of this section, we delve into the performance analysis of each individual retrieval module to gain a deeper understanding of their efficacy and contributions.

Seen clue retrieval This module generates candidate words by retrieving clues from a clue database and then re-ranking them with a BERT model for improved accuracy. To fine-tune this model, we construct a dataset of similar clues using three distinct strategies. The first strategy involves using clues from past NYT daily puzzles as query clues to retrieve related clues from our database, based on the ζ_1 highest BM25 scores. We set $\zeta_1 = 20$, yielding 1,084,260 pairs of similar clues. The second strategy considers all the clues in the database and selects clue pairs sharing answers with confidence (denoted as Eq. (8)) no less than ζ_2 as similar clues. We set the threshold $\zeta_2 = 0.1$, resulting in 504,646 similar clue pairs. The third is the data augmentation strategy, where we use nlpaug[8] to generate 1,612,913 augmented clues pairs. By integrating these three strategies, we build a training dataset comprising 3,201,819 similar clue pairs. To prove the effectiveness of our neural re-ranking model based on fine-tuned BERT (BERT-FT), we compare it with two baseline methods. The first involves combining the BM25 score of retrieved candidates with transformation rules mined from the clue database, such as "X for short = X abbr.", referred to as Textual+Rules. The second employs the BERT model without fine-tuning. The comparison results are presented in Table 8. The results demonstrate that the neural model based on fine-tuned BERT (BERT-FT) outperforms the textual score with rules (Textual+Rules) in re-ranking the candidates retrieved from the clue database. On the other hand, the BERT model without fine-tuning performs poorly, indicating the significance of fine-tuning in improving the re-ranking performance.

Dictionary retrieval To demonstrate the effectiveness of our dense generation approach using WordNet, we compare it with several alternative methods, including textual retrieval with BM25, dense retrieval with BERT [9], and a state-of-the-art reverse dictionary model [45]. For the textual retrieval and unsupervised dense retrieval with Word2Vec (W2V), no supervision is utilized. In the case of the reverse dictionary model, we fine-tuned it using a dataset of past clues from the clue database. There are two options for dense retrieval with a BERT encoder: an unsupervised encoder (Dense-BERT) and a fine-tuned encoder (Dense-BERT-FT). The fine-tuned encoder follows a similar approach to [15], using distant supervision from clue-gloss pairs related through words in the clue database and WordNet. The results presented in Table 8 demonstrate that unsupervised dense retrieval with Word2Vec achieves the best performance among all the competing methods. Besides, we infer that the low recall of the reverse dictionary model may be due to its limited vocabulary. The poor performance of the BERT encoder may be attributed to the noise introduced by the distant supervision, as a clue and a gloss can describe the same word in completely different ways.

Knowledge base retrieval To demonstrate the improvement brought by our knowledge base retrieval approach compared to previous methods that utilize the encyclopedia as the generation source [20], we conduct an ablation study on two key components of our approach: linked entities (E) and propagation (P) in the knowledge graph. The first component involves using linked entities from the

[8] https://github.com/makcedward/nlpaug.

Table 9
Results of different candidate numbers in seen clue retrieval on the hard test set.

CDG hyper-parameters		Metrics		
γ_1	$\gamma_2/\gamma_3/\gamma_4$	W-Acc	L-Acc	P-Puz
30	0	94.00	97.48	40.83
40	0	94.52	97.64	41.67
50	0	**94.88**	**98.04**	**42.50**
60	0	94.31	97.58	40.83
70	0	94.53	97.46	41.67

Table 10
Results of different candidate numbers in dictionary retrieval with fixed $\gamma_1 = 50$ on the hard test set. The numbers in the lower right corner brackets indicate improvements compared to the results obtained with $\gamma_1 = 50$ and $\gamma_2 = 0$.

CDG hyper-parameters		Metrics		
γ_1	γ_2	W-Acc	L-Acc	P-Puz
50	0	94.88	98.04	42.50
	5	95.06	98.00	45.00
	10	**95.25** (+0.37)	**98.07** (+0.03)	**45.83** (+3.33)
	15	94.74	97.79	44.17

context and their aliases to enhance the recall of candidate words. By incorporating this information, we are able to retrieve a broader range of relevant candidates. The second component is the propagation step, where we traverse the knowledge graph based on the linked entities to further enhance the recall. This propagation process allows us to capture additional related candidates that may not have been directly linked in the original context. The results presented in Table 8 indicate the effectiveness of both components. The recall rate is significantly improved when compared to simply extracting words from the encyclopedia (W). However, it is important to note that the increase in recall comes at the cost of lower efficiency. While the efficiency is not greatly reduced compared to the encyclopedia document retrieval, this component itself is time-consuming, and therefore its efficiency is relatively lower compared to other retrieval modules.

Slot filling retrieval For the blank generation, we evaluate the performance of three methods: P_{name} (Name), which utilizes the Wikipedia title name for slot filling, P_{lm} (LM), which utilizes the language model for slot filling, and our hybrid approach combining both methods (Name+LM). From Table 8, we observe that retrieving from the titles of Wikipedia articles achieves higher recall compared to the neural language model generation. The neural language model is trained on a large corpus, in which the knowledge is close to commonsense. However, the blanks for fill-in clues do not usually occur in the corpus and are often related to some rare encyclopedia knowledge. Nevertheless, the combination of the two modules, Name+LM, achieves the best performance. The lower recall in blank generation is due to the fact that it only operates on clues with blanks, which account for approximately 7% of the dataset. As a result, the performance metrics are calculated only on this subset of clues. When considering this subset, the MRR of the blank generation module reaches 27%, even surpassing the performance of the KB retrieval module. Hence, there is no concern about introducing noise with this module. The average time reported is also calculated only on the 7% subset, indicating that this module is efficient. Although using recall and MRR metrics for all clues might seem unfair for modules that only retrieve candidates for a subset of clues, we still use them because they reflect the influence of every single module on the MCTS algorithm.

8.5.2. *Impact of module combinations on search system*

Next, we analyze the impact of the number of candidates produced by different modules in CDG on the performance of the search system. Specifically, we observe the performance with different numbers of candidates γ_1 for seen clue retrieval, γ_2 for dictionary retrieval, γ_3 for knowledge base retrieval, and γ_4 for slot filling retrieval. These experiments are conducted on the hard test set. Based on previous works [28,42], we already know that seen clues are essential for solving CPs. Hence, for this set of experiments, we consider CDG with only seen clue retrieval as the baseline and focused on discussing the remaining three modules.

From Table 9, we observe that the model exhibits optimal performance when it utilizes 50 candidates from seen clue retrieval. Altering this number, either by decreasing or increasing it, adversely affects performance. Based on $\gamma_1 = 50$, we analyze the impact of varying the number of candidates from the other three modules on the model's effectiveness, as detailed in Tables 10, 11, and 12. The analysis reveals that the addition of each retrieval module, when combined with seen clue retrieval, contributes positively to the overall performance. Among these, the knowledge base retrieval module, with $\gamma_3 = 50$, is identified as the most influential, enhancing word accuracy by 1.4 percentage points. Furthermore, we analyze the results of combining different candidate numbers for dictionary retrieval, knowledge base retrieval, and slot filling retrieval, with $\gamma_1 = 50$. The findings, detailed in Table 13, indicate that although each retrieval module boosts performance when combined with seen clue retrieval as mentioned above, combining them is not recommended. This may be attributed to the modules retrieving overlapping candidates. Incorporating additional, less accurate retrieval modules adds minimal value to the selection of ground-truth candidates and significantly increases noise.

Table 11
Results of different candidate numbers in knowledge base retrieval with fixed $\gamma_1 = 50$ on the hard test set. The numbers in the lower right corner brackets indicate improvements compared to the results obtained with $\gamma_1 = 50$ and $\gamma_3 = 0$.

CDG hyper-parameters		Metrics		
γ_1	γ_3	W-Acc	L-Acc	P-Puz
50	0	94.88	98.04	42.50
	30	96.13	98.50	45.00
	40	96.14	98.56	45.00
	50	**96.28** (+1.40)	**98.57** (+0.53)	**45.83** (+3.33)
	60	95.89	98.34	44.17

Table 12
Results of different candidate numbers in slot filling retrieval with fixed $\gamma_1 = 50$ on the hard test set. The numbers in the lower right corner brackets indicate improvements compared to the results obtained with $\gamma_1 = 50$ and $\gamma_4 = 0$.

CDG hyper-parameters		Metrics		
γ_1	γ_4	W-Acc	L-Acc	P-Puz
50	0	94.88	98.04	42.50
	5	94.95	97.96	**45.83**
	10	**95.51** (+0.63)	**98.15** (+0.11)	**45.83** (+3.33)
	20	94.46	97.46	**45.83**

Table 13
Results of combining different candidate numbers for dictionary retrieval, knowledge base retrieval, and slot filling retrieval with fixed $\gamma_1 = 50$ on the hard test set. The results with gray shading are obtained when $\gamma_1 = 50$ and $\gamma_2/\gamma_3/\gamma_4 = 0$. The results with blue shading represent the best performances in Tables 10, 11, and 12. (For interpretation of the colors in the table(s), the reader is referred to the web version of this article.)

CDG hyper-parameters				Metrics		
γ_1	γ_3	γ_2	γ_4	W-Acc	L-Acc	P-Puz
50	0	0	0	94.88	98.04	42.50
50	50	0	0	96.28	98.57	45.83
50	30	10	0	95.71	98.37	44.17
		0	10	96.02	98.52	46.67
		10	10	95.18	97.99	44.17
	40	10	0	96.05	98.50	45.83
		0	10	**96.09**	98.49	**45.83**
		5	5	95.42	98.18	44.17
	50	0	10	96.07	**98.54**	**45.83**
		10	0	95.93	98.43	44.17
		5	5	95.59	98.15	45.00

8.6. *Study of reward function*

In this section, we conduct experiments to explore the impact of the reward function on the performance of MCTS with the hard test set and our built clue database. We compared our designed reward function with previous strategies for scoring solutions in CP. Previous works in crossword resolution typically employed a maximum log-likelihood objective to score the assignment [11,29], which differs from our problem formalization. Our reward function, based on trained neural networks, empirically equals the previous scoring strategy. The problem here is how to obtain such a function to achieve the best performance. To demonstrate the effectiveness of our reward function, we conduct experiments where we systematically eliminated factors, i.e., the CDG ranking model ϕ, the CIG ranking model ρ, and the vocabulary score I, from our reward function and observed their influence on the performance of MCTS. Additionally, we further compare our reward function with other ranking model baselines under the condition that the MCTS algorithm remains fixed.

- w/o ϕ. The reward function only contains ρ and I.
- w/o I. The reward function only contains ϕ and ρ.

Table 14
Ablation study of different reward designs in our MCTS algorithm and reward function comparison on the hard test set.

Time Limits	100 s			1000 s		
Metrics	W-Acc	L-Acc	P-Puz	W-Acc	L-Acc	P-Puz
Ours	91.20±0.46	96.15±0.28	25.00±1.67	94.88	**98.04**	42.50
Ablation study						
w/o ϕ	6.18±0.54	16.50±0.41	0.00±0.00	8.54	18.90	0.00
w/o I	83.66±0.90	92.23±0.52	1.67±0.75	84.35	92.48	4.17
w/o ρ	87.92±0.73	94.80±0.38	11.67±1.83	89.65	95.67	12.50
Reward Baselines						
RMR	85.70±0.50	92.88±0.28	12.50±1.75	89.20	95.00	20.83
IRL	85.15±0.76	92.02±0.45	8.33±1.97	87.98	93.54	29.17

- w/o ρ. The reward function only contains ϕ and I.
- Ranking model as a reward (RMR). This strategy is commonly used in recent works on crossword resolution [3,28]. It directly uses the output of the ranking model as the reward for assignment evaluation. The model is trained using mixed training samples of ϕ and ρ.
- Inverse Reinforcement Learning (IRL). Treating each assignment as a trajectory in the MDP, this problem can be framed as an IRL problem. The objective of IRL is to obtain a reward function that assigns high rewards to demonstration trajectories (good assignments in our case) and low rewards to others. We compare our approach with the Maximum Margin Planning [33] (MMP) algorithm. We run the IRL algorithm on the validation set with the initialized parameters of the ranking model.

The results are reported in Table 14. In the ablation study, we observe that the clue-dependent retrieval component (ϕ) plays a crucial role in our reward function. Without ϕ, the performance of our method significantly drops. However, further analysis of ϕ may not be necessary since the information captured by ϕ is already included in the comparative methods. If we remove the corresponding features in ϕ for those methods, their performances also experience a significant drop. On the other hand, the ranking model ρ is found to be the least influential, but it still contributes to a performance gain of over 3% in word accuracy. In terms of the baseline results, our designed reward function outperforms all the baselines. Notably, the RMR baseline demonstrates that although the learned candidate ranking model provides a more sophisticated ranking for each clue, it serves as a poor reward function. Furthermore, the IRL baseline also exhibits inferior performance. This can be attributed to the fact that negative assignments are often undersampled and biased in the IRL framework.

8.7. Error analysis

In this section, we summarize the two main types of errors based on the analysis of the experimental results. The first type of error is characterized by a low recall rate in candidate generation. The performance of our method is heavily influenced by the confidence of the answers retrieved from the clue database. Given the nature of crossword design, a significant proportion of the clues (around 80% on average) are textually similar to some seen clues in the database, allowing their answers to be retrieved. However, when there are few seen clues or many clues are ambiguous, the performance of our method drops. Notably, the complexity of the puzzle itself does not determine the accuracy. Even large puzzles can be easily solved if the clues are straightforward. The second type of error is attributed to the imperfections in the reward function. In many cases, our MCTS algorithm finds out assignments with larger rewards than the answer but obviously not a good assignment according to humans. This is because the reward function is unable to discriminate the good or bad assignments in some cases.

9. Conclusion and future work

In this paper, we propose a novel framework for solving crossword puzzles using MCTS, introducing several innovative designs including candidate generation and the reward function. To the best of our knowledge, we are the first to apply MCTS to tackle the crossword puzzle problem. In order to facilitate our research, we construct a dataset for crossword puzzle resolution based on NYT daily puzzles with specifications of puzzle selection and clue database construction. We conduct extensive experiments to evaluate the performance of our method and analyze the effectiveness of our specific designs. Our method achieves state-of-the-art performance. Moving forward, there are two key areas for future research. First, we aim to design a more effective candidate generation module. Our experiments reveal that the performance of the system heavily relies on the quality of the seen clue database. Therefore, enhancing the candidate generation process is crucial. Second, we strive to develop a more robust and discriminative reward function. The current reward function can occasionally lead to suboptimal assignments, highlighting the need for further improvements.

CRediT authorship contribution statement

Jingping Liu: Writing – review & editing, Writing – original draft, Methodology, Funding acquisition. **Lihan Chen:** Writing – original draft, Resources, Methodology, Investigation, Data curation. **Sihang Jiang:** Writing – review & editing, Methodology. **Chao**

Wang: Writing – review & editing, Conceptualization. **Sheng Zhang:** Methodology, Investigation. **Jiaqing Liang:** Methodology, Conceptualization. **Yanghua Xiao:** Writing – review & editing, Funding acquisition, Conceptualization. **Rui Song:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (No. 62306112), the Shanghai Sailing Program (No. 23YF1409400), and the Shanghai Pilot Program for Basic Research (No. 22TQ1400100-20). We would like to thank the Knowledge Works Research Laboratory at Fudan University for providing the computing resources and environment.

References

[1] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, Z. Ives, Dbpedia: a nucleus for a web of open data, in: The Semantic Web, Springer, 2007, pp. 722–735.
[2] R.K. Balla, A. Fern, Uct for tactical assault planning in real-time strategy games, in: Twenty-First International Joint Conference on Artificial Intelligence, 2009.
[3] G. Barlacchi, M. Nicosia, A. Moschitti, Learning to rank answer candidates for automatic resolution of crossword puzzles, in: CoNLL, 2014.
[4] R. Bellman, Dynamic programming, Science 153 (1966) 34–37.
[5] B. Bonet, H. Geffner, Action selection for mdps: anytime ao* versus uct, in: Proceedings of the AAAI Conference on Artificial Intelligence, 2012, pp. 1749–1755.
[6] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of Monte Carlo tree search methods, IEEE Trans. Comput. Intell. AI Games 4 (2012) 1–43.
[7] L. Chen, J. Liu, S. Jiang, C. Wang, J. Liang, Y. Xiao, S. Zhang, R. Song, Crossword puzzle resolution via Monte Carlo tree search, in: Proceedings of the International Conference on Automated Planning and Scheduling, 2022, pp. 35–43.
[8] J. Czech, et al., Improving alphazero using Monte-Carlo graph search, in: Proceedings of the International Conference on Automated Planning and Scheduling, 2021, pp. 103–111.
[9] J. Devlin, M.W. Chang, K. Lee, K. Toutanova, Bert: pre-training of deep bidirectional transformers for language understanding, arXiv preprint arXiv:1810.04805, 2018.
[10] M. Ernandes, G. Angelini, M. Gori, Webcrow: a web-based system for crossword solving, in: AAAI, 2005, pp. 1412–1417.
[11] M.L. Ginsberg, Dr. fill: crosswords and an implemented solver for singly weighted csps, J. Artif. Intell. Res. 42 (2011) 851–886.
[12] W.D. Harvey, M.L. Ginsberg, Limited discrepancy search, in: IJCAI (1), 1995, pp. 607–615.
[13] F. Hill, K. Cho, A. Korhonen, Y. Bengio, Learning to understand phrases by embedding the dictionary, Trans. Assoc. Comput. Linguist. 4 (2016) 17–30.
[14] B. Hu, Z. Lu, H. Li, Q. Chen, Convolutional neural network architectures for matching natural language sentences, in: Advances in Neural Information Processing Systems, 2014, pp. 2042–2050.
[15] V. Karpukhin, B. Oguz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, W.t. Yih, Dense passage retrieval for open-domain question answering, in: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, 2020, pp. 6769–6781. Online.
[16] T. Keller, M. Helmert, Trial-based heuristic tree search for finite horizon mdps, in: Proceedings of the International Conference on Automated Planning and Scheduling, 2013, pp. 135–143.
[17] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, D. Krishnan, Supervised contrastive learning, Adv. Neural Inf. Process. Syst. 33 (2020) 18661–18673.
[18] L. Kocsis, C. Szepesvári, Bandit based Monte-Carlo planning, in: European Conference on Machine Learning, Springer, 2006, pp. 282–293.
[19] E. Leurent, O.A. Maillard, Monte-Carlo graph search: the value of merging similar states, in: Asian Conference on Machine Learning, in: PMLR, 2020, pp. 577–592.
[20] M.L. Littman, G.A. Keim, N. Shazeer, A probabilistic approach to solving crossword puzzles, Artif. Intell. 134 (2002) 23–55.
[21] G. Liu, X. Li, J. Wang, M. Sun, P. Li, Extracting knowledge from web text with Monte Carlo tree search, in: Proceedings of the Web Conference 2020, 2020, pp. 2585–2591.
[22] E. Ma, Nlp augmentation, https://github.com/makcedward/nlpaug, 2019.
[23] C.M. Meyer, I. Gurevych, Wiktionary: a new rival for expert-built lexicons? Exploring the possibilities of collaborative lexicography, 2012.
[24] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, Adv. Neural Inf. Process. Syst. 26 (2013) 3111–3119.
[25] G.A. Miller, WordNet: An Electronic Lexical Database, MIT Press, 1998.
[26] M. Mintz, S. Bills, R. Snow, D. Jurafsky, Distant supervision for relation extraction without labeled data, in: Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP, 2009, pp. 1003–1011.
[27] V. Mnih, et al., Playing atari with deep reinforcement learning, arXiv preprint arXiv:1312.5602, 2013.
[28] A. Moschitti, M. Nicosia, G. Barlacchi, Sacry: syntax-based automatic crossword puzzle resolution system, in: ACL, 2015.
[29] M. Nicosia, G. Barlacchi, A. Moschitti, Learning to rank aggregated answers for crossword puzzles, in: European Conference on Information Retrieval, Springer, 2015, pp. 556–561.
[30] J. Pennington, R. Socher, C.D. Manning, Glove: global vectors for word representation, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014, pp. 1532–1543.
[31] I. Pohl, Heuristic search viewed as path finding in a graph, Artif. Intell. 1 (1970) 193–204.
[32] T. Raiko, J. Peltonen, Application of uct search to the connection games of hex, y,* star, and renkula!, AI Mach. Conscious. (2008).
[33] N.D. Ratliff, et al., Maximum margin planning, in: Proceedings of the 23rd International Conference on Machine Learning, 2006, pp. 729–736.
[34] S. Robertson, H. Zaragoza, The Probabilistic Relevance Framework: BM25 and Beyond, Now Publishers Inc., 2009.
[35] C.D. Rosin, Multi-armed bandits with episode context, Ann. Math. Artif. Intell. 61 (2011) 203–230.
[36] C.D. Rosin, Nested rollout policy adaptation for Monte Carlo tree search, in: Ijcai, 2011, pp. 649–654.

[37] A. Severyn, M. Nicosia, G. Barlacchi, A. Moschitti, Distributional neural networks for automatic resolution of crossword puzzles, in: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers), 2015, pp. 199–204.
[38] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al., Mastering the game of go without human knowledge, Nature 550 (2017) 354–359.
[39] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, 2018.
[40] R. Vidhya, G. Vadivu, Research document search using elastic search, Indian J. Sci. Technol. 9 (2016).
[41] D. Vrandečić, M. Krötzsch, Wikidata: a free collaborative knowledgebase, Commun. ACM 57 (2014) 78–85.
[42] E. Wallace, N. Tomlin, A. Xu, K. Yang, E. Pathak, M. Ginsberg, D. Klein, Automated crossword solving, in: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2022, pp. 3073–3085.
[43] Z. Wu, Y. Xiong, S.X. Yu, D. Lin, Unsupervised feature learning via non-parametric instance discrimination, in: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 3733–3742.
[44] F. Xia, T.Y. Liu, J. Wang, W. Zhang, H. Li, Listwise approach to learning to rank: theory and algorithm, in: Proceedings of the 25th International Conference on Machine Learning, 2008, pp. 1192–1199.
[45] L. Zheng, F. Qi, Z. Liu, Y. Wang, Q. Liu, M. Sun, Multi-channel reverse dictionary model, in: Proceedings of the AAAI Conference on Artificial Intelligence, 2020, pp. 312–319.