Evolutionary Optimization of Baba Is You Agents

Christopher Olson
Otto-von-Guericke University
Madeburg, Germany
christopher.olson@ovgu.de

Lars Wagner
Otto-von-Guericke University
Magdeburg, Germany
lars.wagner@ovgu.de

Alexander Dockhorn
Leibniz University Hannover
Hannover, Germany
dockhorn@tnt.uni-hannover.de

***Abstract*—Baba is You is a challenging puzzle game in which the player can modify the rules of the game. This yields a large variety of puzzles and an enormous state space to be searched through. Recently, the Feature Space Search algorithm has shown great results in Sokoban, which apart from the rule modification shares many similarities to Baba is You. It uses multiple heuristics to guide the search into promising regions of the search space. In this work, we are proposing a similar concept for solving Baba is You based on multiple heuristics that are aggregated for guiding a tree-based search process. However, finding parameters for weighting/prioritizing the different heuristics is a non-trivial task. This process is done, by applying evolutionary algorithms for single- and multi-objective optimization. Specifically, we compare the effects of these different optimization schemes on the agents' general level-solving capabilities. In all cases, the agent was able to adapt well to the training and test set with no significant differences among the optimization schemes. Compared with state-of-the-art Baba is You agents our search-based approach shows an improved performance in terms of the number of levels being solved, as well as a reduction in the average time required to solve a level.**

***Index Terms*—Baba Is You, Parameter Optimization, Evolutionary Algorithms**

I. Introduction

Sokoban is a puzzle game in which the player controls a character that pushes boxes around in a warehouse, trying to move them to specific target locations. The player can only push one box at a time, and the boxes cannot be pulled. Sokoban and similar puzzle games present players with many challenging levels to solve. Finding a solution for the more complex levels can be a tough challenge for humans and AI agents alike. Their large state space as well as the long and precise action sequences required to solve a level are hard to overcome while developing AI agents [1]. Sokoban has been proven to be computationally difficult and belongs to the class of NP-hard problems [2].

In this work, we focus on the Sokoban-like game Baba is You (see detailed description in Section II). The game features the unique concept of allowing players to modify the underlying rules over the course of a single level. We developed a search-based agent for solving Baba is You levels making use of multiple heuristics to guide its search process. Each of these heuristics concentrates on another aspect of the game's state and action space and has shown its use in solving a subset of levels. For developing an agent that aims to solve as many levels as possible, we combine these heuristics into a single scoring function. However, weighting these heuristics has shown to be a non-trivial task. Therefore, we made use of several optimization schemes to tune the heuristics' weights.

In this study, we develop a Baba is You agent capable of solving multiple levels efficiently. Our contributions can be summarized by:

- **Creating heuristics for solving Baba is You levels:** We implemented several heuristics taking various game-state measurements into account. These heuristics have been developed based on our own insights and analysis from game-play experience and will be used to guide the tree-based search process of our agent.
- **Developing a performant Baba is You agent:** Solving levels needs to be quick and accurate. In our work, we aggregate several heuristics into a single scoring function. To reduce the agent's computation time, we allow for heuristics to be disabled while aiming to keep the agent's success rate high and the number of iterations during the search low.
- **Comparisons to state-of-the-art agents:** The proposed agent's parameters are tuned using several evolutionary single- and multi-objective optimization algorithms. Comparisons with Bayesian optimization show that the evolutionary algorithms result in slightly better training performance. Our final agent is tested against available submissions of the Keke AI competition.

In Section II, we describe the game Baba is You and the Keke AI Framework which will be used during the remainder of our work. Section III summarizes recent works on Sokoban-like games on which our proposed method will be based. We present our agent and accompanying heuristics in Section IV. The process of optimizing the agent's parameters is described in the subsequent Section V. The resulting agent is compared to the state-of-the-art in Section VI. Finally, in Section VII, we summarize our work and ideas for how it can be further improved in the future.

II. Baba is You and the Keke AI Framework

Baba Is You is a puzzle video game developed by the independent developer Arvi Teikariin. Originally developed as a prototype during the Nordic Game Jam, the game has been widely extended with new graphics, levels, and features until its release on PC and Nintendo Switch in 2019. In the game, the player walks around a grid-like game board to reach a

979-8-3503-1458-8/23/$31.00 ©2023 IEEE

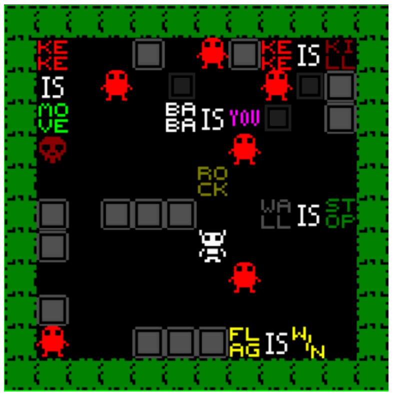


Fig. 1: An exemplary level from the Keke is You AI framework showing multiple objects and rules.

given goal. It shares many resemblances with Sokoban-like puzzle games, with the main difference being that the player is able to change the rules of the game and alter the behavior of the player character and other in-game elements. The game has received widespread acclaim for its innovative gameplay and has won numerous awards.

To better explain the unique concept of the game, we describe the rule modification in more detail. For this purpose, the game consists of a series of levels, each of which presents the player with a game board filled with various words written on tiles representing either objects of the game's world (e.g. Rock, Keke, Flag), concepts (e.g., win or die), or the linking word "is". Each object word is linked to tiles showing a graphical representation of this word. Multiple words can be arranged to represent different rules or properties that can be applied to the game, such as "Keke is You" or "Rock is Push." Rules consist of a combination of three words. The middle word has to be *Is* to form a valid rule. Therefore rules can be subdivided into a prefix and a suffix. The prefix is mostly described by an object part like *Baba Is, Flag Is*. The suffix can result in a feature or action, e.g. *Is Win, Is You, Is Kill*. A classic rule set consists of *Baba Is You* and *Flag Is Win*. Here, the player controls the object "Baba" (a little dog), and touching the flag object (not the word) represents the winning condition. While moving around, the player can rearrange the tiles to modify or invalidate existing rules and create entirely new combinations until the current winning condition is satisfied.

In terms of AI development, Baba Is You represents a challenging problem for search-based approaches. Whereas Sokoban-like games are already known for their large search space, the interactive modification of rules results in even more possibilities. Furthermore, as in many other puzzle games, a single wrong decision can make the puzzle unsolvable, limiting the use of limited-depth search-based algorithms such as the Rolling Horizon Evolutionary Algorithm [3].

To study the development of AI agents, the Keke AI Competition has been created [4]. The associated framework allows the development of agents using javascript aiming to solve a given set of levels. Therefore, each agent receives the level's starting configuration, and a simulator allowing one to anticipate the result of its actions. The computation time of each agent is limited to a maximum time and a maximum number of iterations. Being part of the IEEE Conference on Games 2022, the competition provided a set of baseline agents and attracted multiple entries during its first year. Given a budget of 10 seconds and 10000 iterations, the best agent was able to achieve a 66% win rate on a hidden set of levels according to the leaderboard on the competition webpage.

III. Related Work

Since Baba is You is a relatively new problem in the AI domain, we mainly focused on reviewing related work of Sokoban-like games. Sokoban has a long tradition of being targeted by AI agents [5] and still remains a tough problem due to its long action sequences required to solve some levels. This is a major challenge for search-based approaches since with every move, the search space grows exponentially.

Luckily there often exists not just a single correct action sequence, but multiple sequences to solve a level. For example, in an open room without any obstacles, the agent may take many paths to reach a given destination. On the contrary, there also exist choice points in which a single bad decision could leave the level unsolvable. This is the case when a pushable block is moved into a corner and thus cannot be moved anymore.

There are two types of solving Sokoban-like games. While some Sokoban solvers rely on brute force (exploring many states with low computational costs in between) others make use of heuristics that tend to steer the agent in the right direction at the cost of some computational overhead.

Furthermore, both of these techniques can benefit from integrating expert knowledge that can be used to reduce the search space by decomposing the original problem into several simpler sub-problems. In the context of Sokoban, this has been done by splitting the level into rooms and tunnels [6]. Sadly, effective expert rules for reducing the complexity of Baba is You levels are not yet known.

More generally, abstractions can be used to reduce the complexity of the action and state space [7] by e.g. clustering similar states. More specifically, using constrained clustering techniques [8] would allow expert users to control the abstraction given their own experience with the game. Not all of these techniques ensure completeness (every solvable level can be used) after the abstraction has been applied, but they have shown to benefit search-based approaches in other complex domains [9].

Apart from these exhaustive search agents, other agents have been developed in which the agent mixes the planning and execution process. Here, the available time for the search process is limited and after exhausting the computational budget an action needs to be applied before the agent can continue planning given the new state observation [10]. Alternatively, the search may be reduced to a sampling-based approach such as Monte Carlo Tree Search [11].

In the context of Sokoban, the Feature Space Search (FESS) algorithm [12] has been shown to perform well in comparison to other search-based agents. It has been the first algorithm to solve 90 benchmark problems given a budget of 10 minutes per problem. This has been achieved by using multiple heuristics.

In contrast, the development of Baba is You AI agents is still in its infancy. Simple exhaustive search agents such as depth- and breadth-first search are provided as baseline agents in the Keke AI framework. Additionally, it included a single-heuristic agent that applies a best-first-search algorithm and has been used as a solver for the Baba is Y'all level editor [13]. Developing our own agent, we start with developing multiple heuristics to be aggregated into a single heuristic score for guiding our search-based agent. By doing so, we will allow the agent to deactivate heuristics that have not proven to be beneficial. Thereby, enhancing its search efficiency and level-solving capabilities.

IV. Methodology

We present a method for solving levels in Baba is You. The agent stores game states in a search tree and uses a priority queue to determine the next node to be expanded. Each node contains the following information:

- The score of the node (lower is better)
- The map-state saved as an ASCII representation
- The step sequence as a list
- The parent of the node
- Whether it is a final game state and its respective result
- Whether all player objects have died or not

The priority queue consists of all nodes with unexplored child nodes. The nodes in the queue are sorted by their score, with lower scores being prioritized. At each iteration, the frontmost node in the queue is selected and its child states are added to the tree and the queue. For each of the five available actions (moving in 4 directions and waiting) we make use of the simulator to anticipate the outcome of the next action and calculate the respective information to create a new child node, i.e., checking for victory or defeat, calculating a score for the child node based on various heuristics, and adding the child to the queue if it represents a unique game state. To prevent the agent from getting stuck in a loop, we keep track of previously visited game states and only add a child node to the queue if its game state has not been encountered before. For this purpose, a list of all previously visited map states is maintained and compared with the map state of the child node. Once a winning node has been identified, we return the respective action sequence. Contrarily, if no more player objects exist, because either they have died due to an effect or no more rule of the type "<x>is you" exists, the child will be discarded and continued with the next child or the next iteration. After the new nodes are added, the next iteration is started. This process is repeated until either a solution is found or one of the competition constraints is exceeded (max 10.000 iterations or max 10 seconds for a winning solution).

A. Heuristics

The calculation of the heuristic score is the most complex step. In our agent, we make use of multiple heuristics that are combined into a single scoring function using a linear combination. These heuristics focus on various aspects of the game and are divided into three categories: (1) object type min-/maximizers, (2) distance min-/maximizers, and (3) meta-heuristics. To allow for fine-tuning of the decision-making process, each heuristic was assigned an adjustable weight that determines its influence on the agent's actions.

1) Object type min-/maximizers: The following heuristics count the number of objects in a given category. The idea was that generating helpful objects can be encouraged and harmful objects can be penalized.

Goals: Counts the number of goal objects, i.e., the objects where our agent would win if it manages to reach them. The larger the number of objects, the larger, or smaller (if the optimizer chooses a negative weight), the score will be. This value is calculated using a logarithmic function to encourage the agent to prioritize the first generated goals more highly and subsequent goals less so that it does not solely focus on increasing the goal count even if it has already reached one.

Playable objects: Counts the number of objects that the agent can directly control and multiplies this value by the weight provided by the optimizer. Given a positive weight, this heuristic encourages the agent to keep as many of the player objects alive and to create new player objects (if possible).

Pushable objects: All objects that the agent is allowed to push are counted here. By default, all objects are immovable and can only be pushed if there is a push rule associated with the object in the currently active level.

Automovers: Auto movers are objects that move a field once every iteration. They also tend to be objects that can kill the player. The number of auto movers is counted linearly and multiplied by the specified weight.

Threats: Threats in the game are simply the objects that can kill the player. This heuristic penalizes/rewards the number of threats linearly.

Sinker objects: Objects that move into "sinkers" are destroyed while also removing the sinker object. This means that it is an obstruction for the agent, but it can also be used to destroy other objects and open up new paths to the agent. This heuristic counts the number of sinkers linearly, allowing the optimizer to either punish or reward the creation/destruction of sinker objects.

Stopping objects: Objects can become stopping objects when their object name is combined with the rule "is stop". They prevent all other objects from running through it. They are counted linearly and rewarded/penalized according to their given weight and their amount.

2) Distance min-/maximizers: The following heuristics are typically used to minimize distances to specific object types, e.g., goals. Since their weight could be both negative, they could also be considered to be distance maximizers. In practice, however, we have found that the optimizers have always chosen minimization, which is why we will speak of minimization in the following for the sake of simplicity. The distances are calculated by averaging the Manhattan distance (the number of moves it takes to reach the object) of all player objects to all objects of a given type.

Minimization of the distance to threats: It may seem counter-intuitive, but minimizing the distance between the agent and killing objects has proven effective in testing. This is because threat objects are often guarding important targets, so it makes sense to search in their vicinity.

Minimization of the distance to Points of Interest (POI): This is one of the most important heuristics, as it makes the agent move near interactable objects. The POIs are goals, movable objects, and words, all of which are weighted differently. This heuristic has an additional fourth weight that punishes the agent for not having any players left and is intended to counter the fact that killing all player objects naturally causes the distance to all POI objects to be zero. As mentioned before POIs are often guarded by threats which result in two highly correlated heuristics. However, since there are exceptions to this rule, e.g. a level without any threats, we handle these two separately.

Minimization of the distance to a winning word when there is only one: Inspired by the macro moves from the FESS paper [12], one of our ideas was to build heuristics that only operate in certain situations. This heuristic is only applied when there is only one ”win” word in the level and no victory objects exist yet. Thereby, it helps to guide the agent towards the vicinity of the single ”win” word in order to create a victory condition.

3) Meta-heuristics:

Connectivity: We derived the connectivity heuristic used in FESS [12]. Here, the number of closed rooms, i.e., rooms from which one cannot reach other rooms, is counted and penalized, leading the agent to open up as many rooms as possible. This heuristic is computationally expensive, but it has led to the solving of significantly more levels than without it.

Out-of-plan: As with the connectivity heuristic, this heuristic is also derived from the FESS paper [12] but adapted for Baba Is You. Here, the number of words that can no longer be used is counted and penalized. For the different words, different conditions apply as to when they count as ”unusable”. For instance, the connecting word *IS* is not allowed to be in any corner, while suffixes may be in all corners except the top left because only here they can no longer form a rule. Words that already form a rule are excluded from punishment. The number of words that are ”out-of-plan” can be either rewarded or punished.

Min-/ Maximization of different unique rule combinations: The idea behind this heuristic was initially to enable the optimizers to encourage the creation of new rules that had not yet existed in the level in order to discover possible helpful new combinations. Counter-intuitive to our expectations, penalizing the number of new unique rules was usually preferred by the optimizers.

Rewarding of states with a path to victory: For this heuristic, we developed a divide and conquer algorithm that efficiently checks if there is a direct path to victory from any of the player's objects. If such a path exists, the heuristic rewards/punishes the agent, depending on the chosen weight by the optimizer. Naturally, such a situation is usually rewarded by the optimizer.

4) Combined Heuristic: The final heuristic is the result of weighting above mentioned heuristic components. For a state s and weight vector $w = (w_1, \dots, w_{17})$ the heuristic equals:

$$\begin{aligned} h(s) = {} & w_1 \cdot \textit{NrOfGoals(s)} + w_2 \cdot \textit{NrOfPlayables(s)} \\ & + w_3 \cdot \textit{Connectivity(s)} + w_4 \cdot \textit{OutOfPlan(s)} \\ & + w_5 \cdot \textit{NrOfAutomovers(s)} + w_6 \cdot \textit{NrOfThreats(s)} \\ & + w_7 \cdot \textit{NrOfPushables(s)} + w_8 \cdot \textit{NrOfSinkerObjects(s)} \\ & + w_9 \cdot \textit{NrOfStoppingObjects(s)} + w_{10} \cdot \textit{DistToThreats(s)} \\ & + w_{11} \cdot \textit{DistToWin(s)} + w_{12} \cdot \textit{DistToWords(s)} \\ & + w_{13} \cdot \textit{DistToMovingObjects(s)} + w_{14} \cdot \textit{PlayerIsDead(s)} \\ & + w_{15} \cdot \textit{NrOfUniqueRules(s)} + w_{16} \cdot \textit{DistToWinIfOne(s)} \\ & + w_{17} \cdot \textit{PathToVictory(s)} \end{aligned}$$

Due to the large number of sub-heuristic, we checked for their pair-wise linear correlation using Pearson's correlation coefficient. The strongest correlations have been observed for the pairs $\rho(\textit{NrOfStoppingObjects(s), OutOfPlay(s)}) = 0.74$ and $\rho(\textit{NrOfGoals(s)}, \textit{OutOfPlay(s)}) = -0.55$. Those can be interpreted as (1) given an increasing number of stopping objects, more objects tend to be out of play and (2) the more objects are out of play, the fewer goals we may have. Other combinations had a correlation coefficient of 0.5 or less. Since most heuristics are not strongly correlated we keep them all for the succeeding optimization.

Additionally, we report the computational costs of each heuristic by using our agent to play all levels of the Baba is You benchmark and averaging the time taken to compute a heuristic in each step of the agent's search. Table I shows the average time per heuristic and how much can be saved by excluding one of them. Therefore, we implemented a threshold to allow for disabling heuristics. Any weight that falls within the range $[-0.01, 0.01]$ will become 0 and the heuristic's evaluation will be skipped.

V. Parameter Optimization

In our experiments, we test several optimization schemes to tune the parameters of our agent and thereby improve its performance. For this purpose, we make use of the levels being provided by the Keke AI Competition framework [4]. While the levels of the competition are unknown, we have multiple level sets available. The largest one "full-biy" consists of 184 levels most of which have been generated by the community using the mixed-initiative "Baba is Y'all" level editor [13].

We evaluated agents based on three criteria with decreasing importance, i.e., (1) the number of levels solved by the agent, (2) the average amount of iterations our agent needed to find a solution, and (3) the average computation time. A total of 17 parameters needed to be tuned. Each parameter was bound by the range $[-10, 10]$, while values in the range of $[-0.5, 0.5]$

Heuristic	Time
NrOfGoals	$0.343\mu s$
Connectivity	$86.515\mu s$
NrOfPlayers	$0.186\mu s$
OutOfPlan	$2.829\mu s$
NrOfAutomovers	$0.094\mu s$
NrOfThreats	$0.125\mu s$
NrOfPushables	$0.147\mu s$
NrOfSinkerObjects	$0.087\mu s$
NrOfStoppingObjects	$0.360\mu s$
DistToThreats	$0.281\mu s$
DistToWin, *DistToWords* *DistToMovingObjects*, *PlayerIsDead*	$0.705\mu s$
NrOfUniqueRules	$0.704\mu s$
DistToWinOfOne	$0.164\mu s$
PathToVictory	$105.980\mu s$

TABLE I: Average time for calculating each heuristic. Because the heuristics *DistToWin*, *DistToWords*, *DistToMovingObjects*, and *PlayerIsDead* are evaluated in a single run, we cannot split their time costs.

resulted in the heuristic not being evaluated and instead a value of 0 will be returned (cf. Section IV-A). During parameter optimization we used 50 randomly selected levels of the full level set for training, leaving 134 levels for our test set. During training, each algorithm was given a budget of 2000 iterations and a maximum of 2 seconds of computation time per level. Once any of these thresholds has been reached, the level is considered to be unsolved by the agent. Later during testing, we set the thresholds to 10000 iterations and 10 seconds of computation time per level. This corresponds to the settings used in the competition to make our experiments comparable with previous results. We made use of tighter time constraints during parameter optimization to allow for exploring a larger part of the parameter space and speed up the evaluation of solution candidates in general. A disadvantage of this setup is that due to the tighter time constraints, the optimizer might be inclined to remove a sub-heuristic for speeding up computation time.

For the parameter optimization, we made use of algorithm implementations of the Python package pymoo [14]. The competition framework has been extended by us to be able to send parameterized agents to the javascript-based server app. The code of our experiments and the corresponding results are provided at https://github.com/Razzorior/KekeCompetition. The following sections present details of the optimization schemes used throughout our experiments, being split into subsections for single- and multi-objective optimization algorithms.

A. Single-Objective Optimization

For single-objective optimization we used the algorithms differential evolution (DE) [15], evolutionary strategy (ES) [16], a genetic algorithm (GA) [16], and the Hooke-Jeeves pattern search algorithm (PS) [17]. DE, ES, and GA had a population size of 10 and were run for 20 generations. In total, each algorithm had a budget of 200 fitness evaluations. Due to the high computational effort of our agent (a complex search with a budget of several seconds), we could not compute more generations. For each algorithm, we averaged the results over 5 runs using different seeds.

In our single-objective evaluation, we combined the agent's level-solving rate and the number of iterations to solve a level in a single objective. Since the ultimate goal is to achieve a high solving rate, we multiply it with the maximum number of iterations per level and add to it the remaining iterations. The latter ensures that levels are solved as fast as possible.

$$\mathit{fitness} = \frac{\#\mathit{solved}}{\#\mathit{levels}} \cdot \mathit{max\_iterations} + (\mathit{max\_iterations} - \#\mathit{it})$$

Figure 2a shows the min, mean, and max fitness values per optimization algorithm. Based on the mean result per algorithm, they can be clustered into two groups (GA, DE and PS, ES). Among algorithms of the same group, the average performance of the final population differs just slightly. Comparing all algorithms, the GA and the PS agent reached the highest fitness of 3243 and 3244, respectively. The final population of the GA agent performed slightly better than the other algorithms which is why further experiments will be based on this algorithm. The pattern search performed worst minimal inferior to the evolutionary strategy approach. Overall all algorithms optimized without much noise which also resulted in a small variance in the end results.

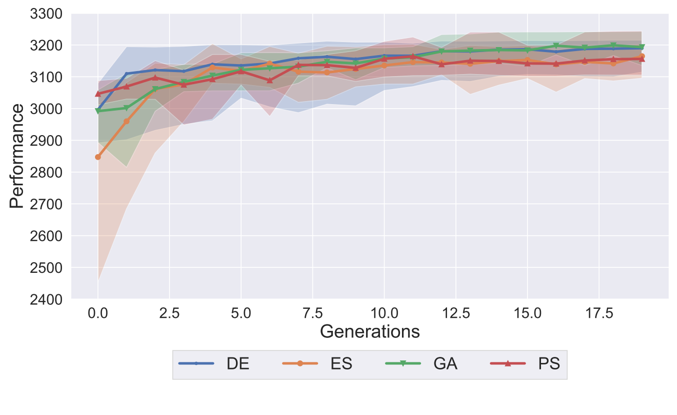
We further compared the result of our evolutionary algorithms with a Bayesian optimization-based approach. Bayesian optimization achieved a maximum performance of 3218 during the first 200 iterations, which is worse than the best solutions found by the GA and PS algorithms in the same amount of evaluations.

Finally, we have taken a look at the best parameter combinations found by each algorithm. For this evaluation, we only took the very best agent per optimization run into account. In total, 20 agents were considered (4 algorithms, 5 repetitions each). Figure 3a shows the distribution of each parameter in the optimized parameter sets. It shows that some parameters have a clear direction ($w_1, w_{10}, w_{11}, w_{12}, w_{13}$), e.g. it is beneficial to maximize the number of goals.

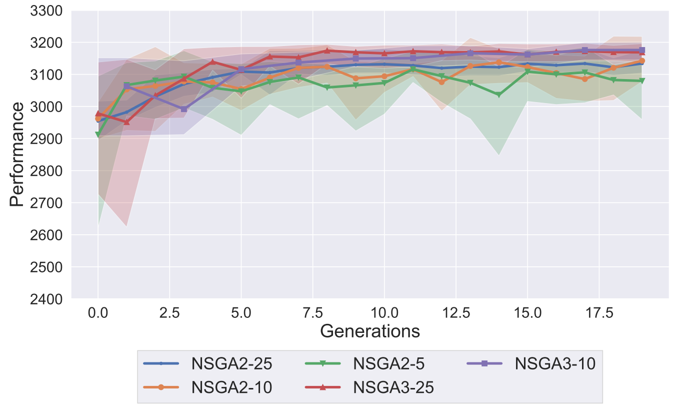
B. Multi-Objective Optimization

In the previous section, we evaluated the number of levels being solved and the number of required iterations by combining them into a single score. This can quickly result in an agent that focuses on levels that can be solved easily, resulting in a local optimum that is difficult to escape. Therefore, the agent may be incapable of finding strategies to solve more complex levels without losing performance on the simpler ones.

To improve the flexibility of our agent we split the level set of our training data into multiple subsets, scoring the performance on each of these subsets separately. For example, a two-objective fitness function has been created by using the win rate of the first 25 levels as the first objective while the win rate of the remaining 25 levels yields the second objective. More objectives can be created by splitting the training level set into even more subsets. In the extreme

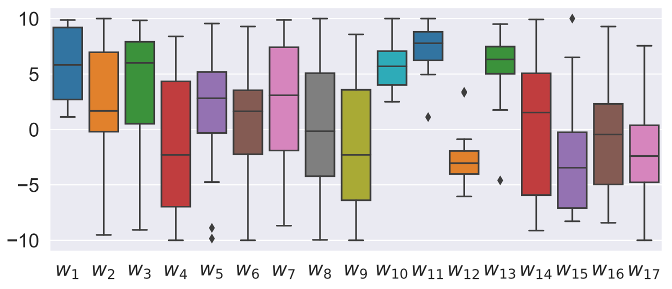


(a) Single-Objective Optimization Algorithms

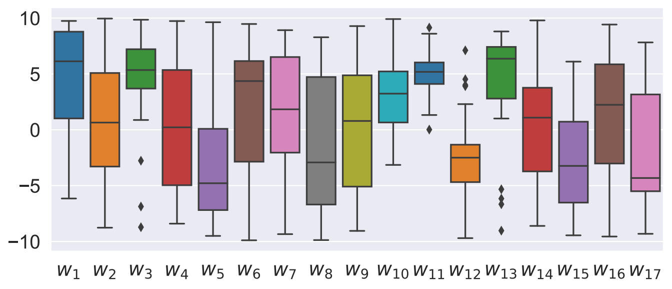


(b) Multi-Objective Optimization Algorithms

Fig. 2: Results of the parameter optimization. Each band shows min, mean and max fitness values of 5 runs per optimization algorithm. For multi-objective algorithms we averaged using the individuals with the highest overall win-rate per generation.



(a) Single-Objective Optimization Algorithms



(b) Multi-Objective Optimization Algorithms

Fig. 3: Distribution of optimized weights per heuristic given the best agents found during parameter optimization.

case, each level represents its own objective. By using multi-objective optimization, we can ensure that we also explore parameter combinations that are solving different level sets. Therefore, allowing the agent to explore multiple solving strategies at the same time. Combining these strategies may result in a well-rounded agent capable of performing well on a variety of levels, rather than just focusing on the easiest ones. To compare with the agents found by the single-objective optimization algorithms, we choose the agent with the best overall win rate as the final solution.

The experiments for the multi-objective approach consisted of the NSGA-2 [18] and NSGA-3 [19], [20] algorithms. The parameters for the different algorithms were chosen so that every algorithm was trained with 200 fitness evaluations so that the results are comparable and no algorithm was preferred. The initialization of each algorithm differs by the number of levels used per objective. For example, 25 levels of the training set were used per objective for the algorithm instance NSGA-2-25, whereas NSGA-2-10 uses 10 levels per objective, respectively. For NSGA-3, we generated reference directions using the Das-Dennis method [21].

NSGA-3-10 had the best mean performance of all the multi-objective approaches whereas NSGA-2-5 had the worst mean performance. However, the end results of the different problem instances lie in a range of 100 points. It can be seen that the noise in the NSGA-2 problem instances is greater than in the NSGA-3 instances. Furthermore, it is shown that the noise within the NSGA-2-10 and NSGA-2-5 instances is the biggest. This confirms our expectations because the NSGA-2 algorithm has problems with more than two objectives.

Once again, we evaluated the best parameter combinations found by each algorithm. This time a total of 24 agents were considered (5 algorithms, 5 repetitions each). Figure 3b shows the distribution of each parameter in the optimized parameter sets. Similar to the parameters of the optimized single-objective agents only a few weights had clear directions $(w_3, w_{10}, w_{11}, w_{13})$.

C. *Comparison of Results*

Taking all optimization algorithms into account, the best mean performance after 200 fitness evaluations has been achieved by the genetic algorithm. It shows slightly better performance than algorithms of both the single- and multi-objective approaches. In comparison between these two, we can see that the variance of single-objective instances is smaller than the variance of the multi-objective ones.

Agent	Training Levels			Test Levels			All Levels		
	Win Rate	Avg. Iter	Avg. Time	Win Rate	Avg. Iter	Avg. Time	Win Rate	Avg. Iter	Avg. Time
$GA_{FullSet}$	**82.0%**	898	$2.13s$	69.4%	1568	$3.41s$	**72.82%**	1386	**3.06s**
$Keke_{Competition}$	72.0%	929	$3.34s$	**72.0%**	**1047**	$3.47s$	72.00%	**1015**	$3.44s$
$BO_{FullSet}$	78.0%	928	$2.45s$	69.4%	1298	$3.37s$	71.74%	1197	$3.12s$
$BO_{TrainSet}$	78.0%	953	$2.42s$	69.4%	1361	**3.32s**	71.74%	1250	$3.08s$
$GA_{TrainSet}$	**82.0%**	**822**	**2.07s**	67.2%	1585	$3.60s$	71.22%	1378	$3.18s$
Default	70.0%	1776	$3.47s$	58.2%	2191	$4.47s$	61.41%	2078	$4.20s$
$Default_{Fixed}$	66.0%	2166	$4.02s$	57.5%	2504	$4.50s$	59.81%	2412	$4.37s$
BFS	62.0%	4121	$4.00s$	54.5%	4774	$4.80s$	56.53%	4597	$4.58s$
DFS	58.0%	1924	$4.55s$	50.7%	2132	$5.34s$	52.68%	2075	$5.13s$

TABLE II: Comparison of the different agents averaged over 10 runs per level set. The best result is highlighted in bold.

VI. Comparison with State-Of-The-Art Agents

For our final comparison against other Baba is You agents, we repeat the optimization process using the GA algorithm and the full budget of 10000 iterations and 10 seconds per level. Furthermore, we compare two versions of our agent. $GA_{TrainSet}$ uses the 50 training levels during optimization and runs for 100 generations, and $GA_{FullSet}$ has access to all 183 levels during optimization and is trained for 50 iterations. Both have been shown to converge at the end of training. As an alternative optimization scheme, we also made use of Bayesian optimization. For comparison, we also report the results of optimizing the parameters based on the set of training levels $BO_{TrainSet}$ and the full level set $BO_{FullSet}$.

We then compare our optimized agents to the baseline agents provided in the Keke AI framework as well as the winning entry of the Keke AI Competition 2022. The framework includes a depth-first search, a breadth-first search, and a more complex agent, called Default. The latter has been used as a solver in the context of the mixed-initiative level editor Baba is Y'all [13]. The Default agent of the framework performs an exhaustive search that uses a heuristic to prioritize its node expansion. The score consists of three functions representing different distance measurements to other objects in the level (similar to the ones in Section IV-A2). The first heuristic calculates the average distance of players to the win object, the second the average distance of players to interactable words, and the last the average distance between players and objects, which have the push trait.

However, we found two major bugs in the implementation of the default agent. The first is related to the distance calculation of the three sub-heuristics. Here it is not taken into account that there are levels in which no goals or pushable objects exist in every state and this is not caught when forming the average distance, resulting in a division by zero. This completely disables the heuristics for these types of levels and turns the agent into a brute-force agent. The second bug has to do with sorting the children in the queue. The standard *.sort()* Javascript function is used here, which converts the elements into strings, then compares their sequences of UTF-16 code unit values. However, this sorts a value of 11 below a value of 2, for example. This bug only affects the sorting of the queue if the map has a certain size so that the average Euclidean distance can be larger than 10 in the first place. Surprisingly, the agent performed worse when fixing these bugs (cf. $Default$ and $Default_{Fixed}$ in Table II), which is why they could be ignored.

The best entry of the KeKe AI Competition, here called $Keke_{Competition}$, generates random solution paths. During the execution of the algorithm, these paths will be mutated as in evolutionary algorithms. The mutation operator splits the action sequence of individuals into multiple sub-strings and swaps them between individuals. The algorithm runs until a solution is found. Because of the nondeterministic nature of the algorithm, we calculated averages over 10 runs to determine the average performance of the algorithm.

Table II shows the mean performance of each agent on the levels of the train and the test set, as well as their aggregated performance over the full set. The standard deviation is mostly negligible since all used algorithms are deterministic. Only slight perturbations in the CPU time caused agents to lose complex levels due to a timeout that they would otherwise have won.

The best agent during our evaluation has been the $GA_{FullSet}$ agent with an overall win rate of 72.82% over all 183 levels. This does not come as a surprise, since it is specifically optimized to solve the given level set. However, the $Keke_{Competition}$ agent, the $GA_{TrainSet}$ agent, and the two BO agents have been close competitors. The $Keke_{Competition}$ agent achieved a win-rate of 72.00%, a difference of 1.5 levels than the best agent, the two BO agents had a win-rate of 71.74%, and the $GA_{TrainSet}$ achieved a win-rate of 71.22%. The *Default* as well as the simple search-based agents achieved win rates ranging from 61.41% to 52.68%.

Moreover, the data of Table II shows that in comparison to the other agents the GA-optimized agents required the least computation time. This especially holds for levels of the training set, in which the proposed agents solved the levels in fewer iterations than other agents. Furthermore, we can see that the performance of our proposed agent degrades when applied to the test levels. Since this holds for both training schemes (FullSet and TrainSet) we assume that this difference can mostly be attributed to the varying difficulty of included levels in the train and test set.

VII. Conclusion

In this work, we proposed a single-heuristic search-based agent for solving Baba is You levels. The agent combined multiple heuristics into a single value. The weighting of these heuristics is optimized by several evolutionary algorithms. To speed up computation time, each sub-heuristic can be excluded entirely from the evaluation. This ensures that complex sub-heuristics are only included if they benefit the agent's performance. Our evaluation showed slight performance improvements to existing Baba is You agents. Its strongest competitor also uses a combined heuristic function with a different search scheme. In the future, it would be interesting to see if the combination of the heuristic-function optimization using other optimization schemes or a more sophisticated combination of the different heuristics could result in an even better agent.

While our agent has already shown to solve many levels of the Keke AI competition, there are different aspects that have room for improvement. In the following, we plan to further expand on the optimization of heuristics. In this work, evolutionary algorithms have been used to tune the weights of a given set of heuristics. Next, we plan to learn entirely new heuristics to further enhance the generality of our agent. Furthermore, combining information on the state space graph or the sub-graph explored by the search algorithm with reinforcement learning techniques may allow the identification of heuristics and abstractions using self-play [22].

At the same time, we want to explore the use of our heuristics in a multiple heuristic search process such as Multiple Heuristic Greedy Best-first Search [23], Independent Multi-Heuristic A [24], portfolio-based search algorithms [25] and the Feature Space Search algorithm (FESS) [12]. Those have already been shown to perform well in their respective domains as long as the set of heuristics has been selected carefully.

References

[1] A. Dockhorn, S. M. Lucas, V. Volz, I. Bravi, R. D. Gaina, and D. Perez-Liebana, "Learning local forward models on unforgiving games," in *2019 IEEE Conference on Games (CoG)*, 2019, pp. 1–4.

[2] D. Dor and U. Zwick, "Sokoban and other motion planning problems," *Computational Geometry*, vol. 13, no. 4, pp. 215–228, 1999.

[3] R. D. Gaina, J. Liu, S. M. Lucas, and D. Pérez-Liébana, "Analysis of Vanilla Rolling Horizon Evolution Parameters in General Video Game Playing," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017, vol. 10199 LNCS, pp. 418–434.

[4] M. Charity and J. Togelius, "Keke AI competition: Solving puzzle levels in a dynamically changing mechanic space," in *2022 IEEE Conference on Games (CoG)*. IEEE, 2022, pp. 570–575.

[5] T. Virkkala, "Solving Sokoban," Ph.D. dissertation, Ph. D. dissertation, Masters thesis, University Of Helsinki, 2011.

[6] A. Botea, M. Müller, and J. Schaeffer, "Using abstraction for planning in Sokoban," in *International conference on computers and games*. Springer, 2003, pp. 360–375.

[7] A. Dockhorn and R. Kruse, *State and Action Abstraction for Search and Reinforcement Learning Algorithms*. Springer International Publishing, 2022, (to be published).

[8] M. Schier, C. Reinders, and B. Rosenhahn, "Constrained mean shift clustering," in *Proceedings of the 2022 SIAM International Conference on Data Mining (SDM)*, Apr. 2022. [Online]. Available: https://github.com/m-schier/cms

[9] L. Xu, J. Hurtado-Grueso, D. Jeurissen, D. P. Liebana, and A. Dockhorn, "Elastic monte carlo tree search with state abstraction for strategy game playing," *arXiv preprint arXiv:2205.15126*, 2022.

[10] R. Pascanu, Y. Li, O. Vinyals, N. Heess, L. Buesing, S. Racanière, D. Reichert, T. Weber, D. Wierstra, and P. Battaglia, "Learning model-based planning from scratch," *arXiv preprint arXiv:1707.06170*, 2017.

[11] M. Crippa and F. Marocchi, "Monte carlo tree search for Sokoban," Bachelor's Thesis, Politecnico Di Milano, 2018.

[12] Y. Shoham and J. Schaeffer, "The FESS algorithm: A feature based approach to single-agent search," in *2020 IEEE Conference on Games (CoG)*, 2020, pp. 96–103.

[13] M. Charity, I. Dave, A. Khalifa, and J. Togelius, "Baba is y'all 2.0: Design and investigation of a collaborative mixed-initiative system," *arXiv preprint arXiv:2203.02035*, 2022.

[14] J. Blank and K. Deb, "pymoo: Multi-objective optimization in python," *IEEE Access*, vol. 8, pp. 89 497–89 509, 2020.

[15] K. Price, R. M. Storn, and J. A. Lampinen, *Differential evolution: a practical approach to global optimization*. Springer Science & Business Media, 2006.

[16] R. Kruse, S. Mostaghim, C. Borgelt, C. Braune, and M. Steinbrecher, *Computational Intelligence*. Springer International Publishing, 2022. [Online]. Available: https://doi.org/10.1007/978-3-030-42227-1

[17] R. Hooke and T. A. Jeeves, "Direct Search solution of numerical and statistical problems," *Journal of the ACM*, vol. 8, no. 2, pp. 212–229, Apr. 1961. [Online]. Available: https://doi.org/10.1145/321062.321069

[18] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[19] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, 2014.

[20] ——, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part ii: Handling constraints and extending to an adaptive approach," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, p. 602–622, 2014.

[21] I. Das and J. E. Dennis, "Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems," *SIAM Journal on Optimization*, vol. 8, no. 3, pp. 631–657, Aug. 1998. [Online]. Available: https://doi.org/10.1137/s1052623496307510

[22] M. Schier, C. Reinders, and B. Rosenhahn, "Deep reinforcement learning for autonomous driving using high-level heterogeneous graph representations," in *International Conference on Robotics and Automation (ICRA)*, 2023.

[23] M. Helmert, "The fast downward planning system," *J. Artif. Int. Res.*, vol. 26, no. 1, jul 2006.

[24] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev, "Multi-heuristic a∗," *The International Journal of Robotics Research*, vol. 35, no. 1-3, pp. 224–243, Aug. 2015. [Online]. Available: https://doi.org/10.1177/0278364915594029

[25] A. Dockhorn, J. Hurtado-Grueso, D. Jeurissen, L. Xu, and D. Perez-Liebana, "Portfolio search and optimization for general strategy game-playing," in *2021 IEEE Congress on Evolutionary Computation (CEC)*, 2021, pp. 2085–2092.