The FESS Algorithm: A Feature Based Approach to Single-Agent Search

1st Yaron Shoham

Google Research
Tel Aviv, Israel
yaronsh@google.com

2nd Jonathan Schaeffer
Computing Science
University of Alberta
Edmonton, Canada
jonathan@ualberta.ca

***Abstract*—This paper introduces the Feature Space Search algorithm (FESS) that projects a single-agent application search space onto an abstract space that is defined by features of the domain. FESS works in this smaller space by directing the search from the projected initial state towards the projected final state. A state transition in the feature space maps to one or (usually) many moves in the domain space. Each of these transitions leads to a change in one or more features, with the FESS algorithm favoring transitions that make progress towards the projected final state. Each feature can be thought of as being a heuristic, and FESS is providing multi-objective guidance from the feature space to the search in the application domain.**

FESS is demonstrated using the challenging single-agent problem of Sokoban. For over twenty years, numerous approaches have been used to try solving the 90 problems in the standard benchmark test suite. FESS, using a set of domain-specific features, is the first program that solves all 90 problems. Further, although the experimental standard is to allocate 10 minutes of solving time per problem (900 minutes), the FESS approach solves the entire test set in less than 4 minutes.

***Index Terms*—heuristic search, single-agent search, path planning, Sokoban**

I. Introduction

Several decades of research into single-agent search have produced impressive results on a number of illustrative (sliding-tile, Rubik's Cube) and real-world (path-finding, job-shop scheduling, robotics) applications. The A* algorithm and variants on it have received the most research attention [1]. However, A* has limitations (memory) and variants to address this also pay a price (time). Exponential growth in the search tree is a limiting factor.

A* implementations use a single heuristic value to guide the search. For many applications, this is sufficient. However, as is well known, these algorithms can get stuck in plateaus – regions in the search space where the heuristic is incapable of differentiating between nearby states [2]. These plateaus can have a dramatic impact on performance. An appropriate approach to minimize this problem is the use of multiple heuristics. If they are well chosen, ideally orthogonal to each other, then the weakness of a heuristic in a search region may be offset by the strength of another in the same region. Single-agent search algorithms can incorporate multiple heuristics by maximizing or combining their values. What if the search considered the impact of multiple heuristics separately, trying to improve the value of all of them?

This paper introduces the Feature Space Search algorithm (FESS) that projects a single-agent search application onto an abstract space that is defined by features of the application domain. Each feature maps to a numerical assessment and can be thought of as being a heuristic. However, instead of combining feature values into a single evaluation, features are viewed as a multi-dimensional search space. FESS provides multi-objective guidance to the search in the application domain (e.g., [3]). The start and goal states are projected onto the feature space. Within the feature space, a transition usually reflects many moves in the domain space. The FESS algorithm rewards states that have progressed in the feature space by biasing their further exploration in the domain space.

While high-level transitions happen in the feature space, a search tree in the original domain space is maintained. In the traditional A* approach, the search is guided by the $f = g + h$ estimate of the solution cost from each node (g=cost from start to current state; h=estimated cost from current to goal state) [4]. In contrast, in FESS each move has a weight (priority) that reflects how promising it is to lead to a solution (with no regard to the solution cost). The feature space informs the assignment of weights to moves, being biased towards moves that (eventually) lead to progress in the feature space.

The single-agent domain of Sokoban is used to demonstrate FESS. Most Sokoban implementations use single-agent search with a sophisticated Manhattan distance heuristic and a plethora of enhancements (e.g., [5]). On the standard 90-problem XSokoban test suite benchmark [6], no program has been able to solve the entire set within the 10-minute time allocation per problem (900 minutes) – despite almost 25 years of research on this problem. The FESS-based Sokoban solver discussed in this paper is the first program to solve all 90 problems. Further, this is accomplished using less then 4 minutes of computing time in total.

The major research contributions of this paper include:

- The FESS algorithm for doing a multi-objective search in an abstracted domain space.
- FESS addresses the problems posed by long solutions and by directed moves that may cause dead ends.
- New features (heuristics) for the Sokoban puzzle domain.

978-1-7281-4533-4/20/$31.00

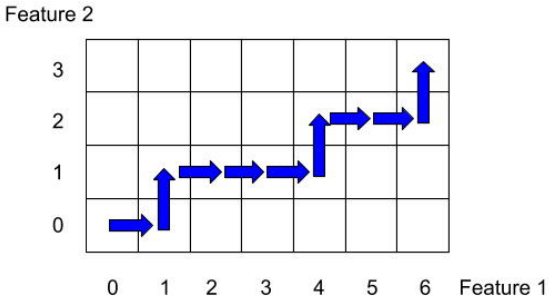


Fig. 1. Feature Space.

- The first program to solve all 90 problems in the standard Sokoban benchmark test suite.

II. FESS

For clarity, we differentiate between the feature space (FS; abstract space; with cells and transitions from cell-to-cell) and the application domain space (DS; actual space; with states and moves from state-to-state).

Consider using A* to solve a sliding tile puzzle problem using the Manhattan distance heuristic. A move consists of sliding a tile in one of four possible directions. Each move will result in the heuristic (estimate of the number of moves to completion) being adjusted up or down by one. This solving method is characterized by the use of a single heuristic, and single moves in the domain space. Manhattan distance can be viewed as a one-dimensional feature space.

Now consider a domain where multiple heuristics, or *features*, are used to guide the search. Figure 1 shows an example of a FS. There are two features with the start state having feature values (0,0) and the goal (6,3). This example illustrates the ideal case where each transition played in the FS brings the problem closer to the goal state in one of the dimensions. A change in feature value could be the result of a single move in the original DS (e.g., as with Manhattan distance), but it may map to many moves. For example, when solving Rubik's cube, a feature could be the number of solved faces of the cube. Transitioning from a state with, say, 0 solved faces to one with 1 solved face is a single transition in the feature space, but likely requires many moves in the original DS.

Traditional single-agent search algorithms combine multiple heuristics into a single value (usually the maximum or a linear combination). Instead features are treated as a multi-dimensional search space. Consequently, FESS does not need an evaluation function (how good the state is) or a heuristic function (how far it is from the goal). The start and goal states of an application are projected onto the FS. The objective of a search is to progress from the projected start to the projected goal cell. In Figure 1 transitioning from cell (4,1) to cell (4,2) should be encouraged, while from cell (4,1) to cell (4,0) discouraged. Note that for difficult problem instances, a straight line to the goal may not possible; diversion to a "poorer" cell might be necessary to find a solution.

Initialize:
Set feature space to empty (FS)
Set the start state as the root of the search tree (DS)
Assign a weight of zero to the root state (DS)
Add feature values to the root state (DS)
Project root state onto a cell in feature space (FS)
Assign weights to all moves from the root state (DS+FS)

Search:
while no solution has been found
  Pick the next cell in feature space (FS)
  Find all search-tree states that project onto this cell (DS)
  Identify all un-expanded moves from these states (DS)
  Choose move with least accumulated weight (DS)
  Add the resulting state to the search tree (DS)
  Added state's weight = parent's weight + move weight (DS)
  Add feature values to the added state (DS)
  Project added state onto a cell in feature space (FS)
  Assign weights to all moves from the added state (DS+FS)

Fig. 2. FESS Algorithm.

The feature space has N features, with each feature taking on one of V_i values. Thus the FS might contain as many as $\prod_{i=1}^{N} V_i$ cells. Transitioning from one cell to the next (good or bad) may require a sequence of moves in the DS. FESS rewards DS states that have progressed in the FS by focusing the search on their continued exploration. *The FS is not searched, but it provides a high-level roadmap for making progress in the DS.*

More precisely, the FESS algorithm maintains both a traditional search tree in the DS and an abstracted projection in the FS. Each state in the tree maps to one cell in the feature space, and the cells in the FS are linked to all of their source nodes in the tree. The tree is initialized with the start state as the root. As new states are considered, they are added to the tree, along with their projected FS values.

FESS works by using the feature space as multi-objective guidance. A cell is selected from the FS, and all nodes in the search tree that project to it are considered for possible expansion. FESS focuses on states in the DS that are likely to make progress in the abstract space. By doing so, it combines the guidance of the abstract space with the actual moves being made in the DS. Intuitively, moves that do not (eventually) lead to a substantive change (i.e., progress in the FS) will gradually lose the algorithm's attention, while those that make progress are rewarded. This is done by giving moves accumulating *weight*. Figure 2 presents the FESS algorithm.

A. Search Tree

The algorithm uses a tree data structure that reflects states explored in the domain space. Each node contains the state, feature values for that state, a list of all possible moves from that state, and weight information for the node and all moves leading from the node. The search explores new parts of the tree by expanding a state. One of the moves from the state is selected and applied, and the resulting state is added to the tree as a new leaf node (if not already in the tree). For this

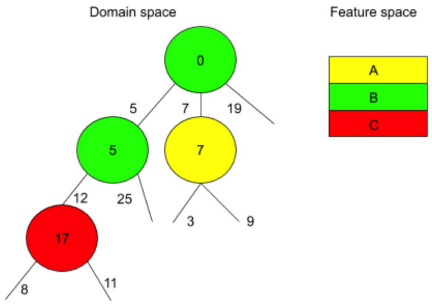


Fig. 3. The DS/FS Relationship Using 3 FS Cells (each shown with a different color).

new node, the feature values, list of new possible moves, and weights are computed and stored.

The search uses a transposition table to avoid analyzing identical states more than once [7]. It also flags a node that has no valid moves – a dead end. If all of a node's children are dead ends, then the node becomes a dead end too (and this can be propagated upwards). The search ignores dead-end nodes.

B. Processing Feature Space Cells

FESS works by going *cyclically* over all non-empty cells in the FS. For each such cell, the algorithm examines all search tree states that project onto this cell. Now the algorithm considers the possible moves from these states, but only moves that were not already expanded. Among all these moves, the algorithm picks the one with the least accumulated weight. The selected move is applied, and the resulting state is added to the tree. The new state is now projected onto the FS.

Cyclically going through the cells in the FS is unusual. This is not a requirement of the algorithm; just an observation that this balances the exploration and exploitation aspects of the search. When newly populated cells are visited, the algorithm exploits its most promising options. When "older" cells are considered, the algorithm explores for alternative routes in the FS. In principle, any selection algorithm will work as long as all cells get coverage.

Figure 3 presents a simple example of FESS iterations. In the DS, the start state and three nodes have been added to the tree. The FS has three cells, and the connection between FS states and DS states is shown by color. A FESS scan starts by choosing cell A. This cell has only one matching state in DS (7). The least weight move is expanded (3), resulting in a new state with accumulated weight of $7 + 3 = 10$. Now FESS continues to cell B. Here there are two states in the DS (0 and 5), and the possible weights of the un-expanded states are $0 + 19 = 19$ and $5 + 25 = 30$. A new node with the minimum accumulated weight of 19 is added below the root. Finally, the scan continues to cell C. In this example, cell C is closer to the projected final state in the FS. FESS expands the node in the DS (17), giving rise to a new node with an accumulated weight of 25 (minimum of $17 + 8 = 25$ and $17 + 11 = 28$). Note that the expanded node (weight= 17) is not close to the root and does not have a small weight. Still, the FESS algorithm chooses it because it looks attractive in the FS. This example illustrates how progress in the feature space translates to move decisions in the domain space.

In the example, the pass through cells A, B, and C resulted in adding three new nodes to the tree. Each new node maps to a cell in the FS: A, B, C, or a new feature combination that has not yet occurred in the search. The FESS algorithm continues by cycling through the cells again and again until a solution has been found.

C. Weights

Some moves in the DS have a greater chance of changing the projected cell in the FS. The simplest example is a move that directly corresponds to a cell transition, preferably in the direction suggested by the FS. Other cases can be moves that improve the chance of a later transition, for example by increasing the number of possible moves. If these useful moves are tested early on, then progress in the FS can be rapid. To implement this, the FESS algorithm assigns weights to moves. Moves with small weights get prioritized by the search, resulting in the early populating of empty cells (leading to their exploration).

Identifying promising moves and assigning weights to them is domain specific. Using weights properly can have a dramatic effect on the search efficiency.

D. Completeness

FESS does not prune any move, unless it has been proven to lead to a dead end. Given a finite search space, FESS is guaranteed to eventually find a solution if it exists.

E. Comparison to Other Algorithms

Conceptually, FESS can be considered a combination of a) a strategic plan (improving features) which takes place in the FS, and b) a tactical search on the state moves that takes place in the DS. It was originally motivated by the problem of long solution lengths and dead ends. For FESS, it does not matter how long the solution is. As empty FS cells become populated, the algorithm ensures they are considered by its repeated scan of the FS. Compare this to breadth-first search, where plans needing a large number of transitions cannot practically be found.

The search may consider a state that a deep analysis might prove leads to a dead end. Even though such states can advance in the feature space, they will eventually stall. Lack of success in making progress results in them gaining weight when they are selected for expansion. Sooner or later, this allows states with lower weights to be selected from the cell.

FESS can be seen as a mixture of several search algorithms. By changing the focus between the cells, FESS can search deeper quickly when progress is being made, much like depth-first search. When progress in the FS is slow, FESS repeatedly revisits cells, trying to expand in the FS to other nearby cells. This systematic search is similar to best-first search. Within a specific cell, FESS prioritizes moves by their weights (best-first). Then it acts like Dijkstra's algorithm, looking in the

DS for the shortest paths from the cell's entry point to states exiting the cell.

Part of the novelty of FESS is that it uses multiple features (heuristics) concurrently. A*, for example, uses a single heuristic value [4]. Even when multiple heuristics are available, they are projected into a single value (usually by taking the maximum). This loses information. The reality for many (hard) problems is that different heuristics may work well in different parts of the search space. For a given problem to be solved, there might be parts of the search where feature-1 is important, but then one needs feature-2 to make progress, and only then can feature-3 direct the search to the goal. FESS considers all features at all nodes.

Multiple heuristics have been used in many contexts. For example, the Multiple Heuristic Greedy Best-first Search (MHG-BFS) approach evaluates every node using multiple heuristics, updating a priority queue (OPEN list) for each heuristic [8]. In contrast, the Independent Multi-Heuristic A* (IMHA*) algorithm conducts multiple independent searches simultaneously, each using a different heuristic [9]. Both of these approaches (and other similar ones) are fundamentally A*-based using a fine-grained heuristic, while trying to produce optimal or near-optimal results (see [10] for a good overview). In contrast, FESS does not use A*, has a single queue, is guided by coarse-valued heuristic, and is not concerned about optimality.

FESS is not without its drawbacks. The algorithm can spend time in trying different axes for advancement; this is inherent in the algorithm. It also spends time revisiting cells (as do some single-agent search algorithms). For easy application problems, a greedy search using a simple evaluation function often finds the solution faster. The algorithm usually produces a non-optimal-length solution. In general FESS is about advancing quickly in the feature space rather than optimizing the solution length. For difficult application problems, finding any solution is usually the primary goal.

III. SOKOBAN

The FESS algorithm is demonstrated using the well-studied domain of Sokoban. The puzzle was invented in 1981 by Hiroyuki Imabayashi. It is a motion planning problem that has been shown to be NP-hard [11]. A search on Google Play returns over 100 implementations of the puzzle. A dedicated group of creative problem designers continually publish new levels to be solved, and a large and enthusiastic community embraces the challenge. The puzzle is used in schools to help develop logic and problem-solving skills [12]. In the research community, Sokoban is an application used for the International Planning Competitions [13].

Figure 4 shows a sample problem (level #1 in the 90-problem XSokoban set [6]). The player can move in four directions, constrained by the walls and obstacles. A problem consists of the player pushing all the boxes (round objects in the Figure) from their start location to the destination locations (red squares). The player, standing next to a box, can push it one square ahead to an empty location.

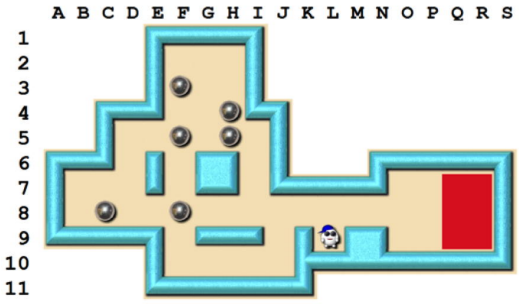


Fig. 4. XSokoban #1.

Solution lengths are commonly several hundred moves long – even longer – and the branching factor can be in the hundreds of moves. This makes it impractical for brute-force search approaches. Further, the push constraint (no pulls) means that the player may move from a solvable state to an unsolvable one (deadlock).

A. Literature

Research using Sokoban up to 2011 is surveyed in [14]. An early attempt to build an automated solver was the ROLLING STONE program [5]. Using a sophisticated Manhattan distance heuristic, it was able to solve 59 of the 90 problems. The TALKING STONES solver does not use a heuristic function [15]. Instead, it considers the solution as a sequence of simpler sub-problems. The program computes the order in which boxes should be packed on targets. Then, each step tries to pack a box according to this plan. The program uses a single feature – the number of packed boxes – in deciding its search strategy. When improving this feature is difficult, the progress of the solver is slow. It can solve 54 levels.

Abstraction is used in the program of Botea *et al.* [16]. The board is considered as a graph of rooms linked by tunnels. For each room, all possible box configurations are computed. Configurations that can interchange with one another define an abstract equivalence class. A search is performed to find a solution using this abstract representation, ignoring the specific box positions on the board. As rooms become bigger, the number of possible configurations grows exponentially. Their approach worked for 10 levels.

In 2017, DeepMind presented their attempts to solve Sokoban problems using deep learning [17]. Recent work has tried using deep reinforcement learning and Monte Carlo Tree Search [18], [19]. Despite these significant efforts, humans are still better at the game. To date, no algorithm has been able to solve all 90 problems within the standard 10 minutes of computing time per problem.

B. Features

Our implementation uses a 4-dimensional feature space.

Packing Feature

This feature counts the number of boxes that have reached a target. The higher the value, the closer one is to a solution. However, for many Sokoban problems the order that the boxes

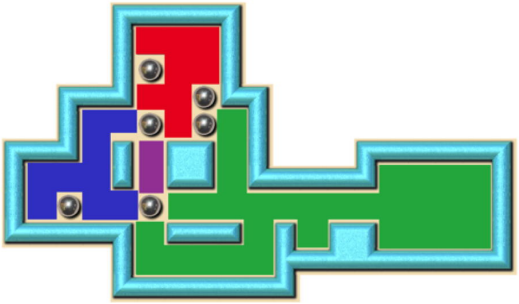


Fig. 5. XSokoban #1 Connectivity.

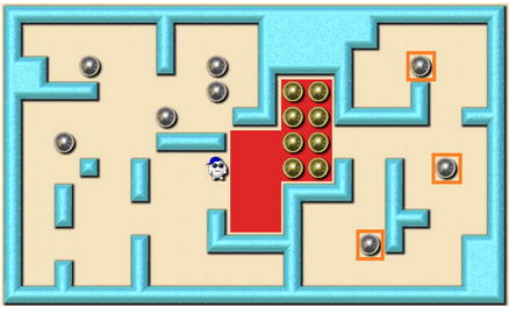


Fig. 6. XSokoban #74 Analysis Showing Out-Of-Plan.

are moved to their destination squares can be critical. A pre-search is done using retrograde analysis [20] to determine a plausible ordering. Hence the actual metric is the number of boxes moved to targets in that particular order. Many Sokoban solvers do some form of packing analysis.

Connectivity Feature

Usually, the boxes divide the board into closed regions. The player can move around freely within a region, but cannot move to another region without first pushing boxes. Figure 5 shows the four regions for Figure 4. A connectivity of one means that the player can move anywhere on the board. Intuitively, it is beneficial to reduce the number of regions, allowing for more move options. Note that connectivity is not monotonically decreasing; some levels require a temporary increase in this heuristic. Connectivity is a new and powerful heuristic for Sokoban solvers.

Room Connectivity Feature

Room connectivity is related to the connectivity idea. Sokoban levels are often composed of rooms linked by tunnels. The room connectivity feature counts how many room links are obstructed by boxes. Generally speaking, reducing this feature increases the access to rooms from more than a single tunnel, and is usually desirable. Room connectivity is a new feature for Sokoban solvers.

Out of Plan Feature

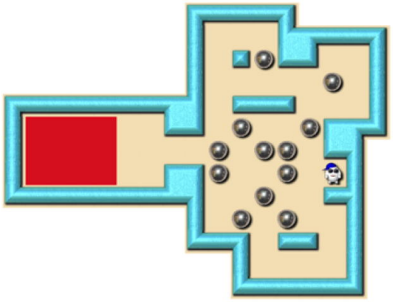
As packing proceeds, some areas of the board may become blocked. Figure 6 shows an example from level #74. A workable packing plan is to fill the targets from right to left. However, if this plan is executed carelessly (as in the Figure), some boxes will become unreachable (the three on the right).

The out-of-plan feature counts the number of boxes in soon-to-be-blocked areas. This number should be minimized. When it is reduced to zero, packing can continue without that risk. Out-of-plan analysis is another new feature used by our solver.

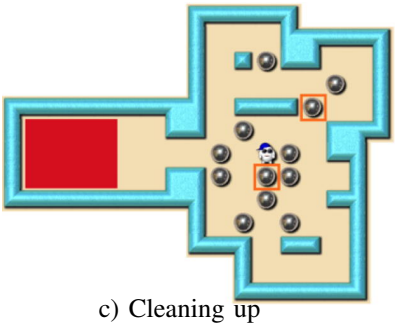
C. Example

Level #5 (Figure 7a) illustrates the advantages of having at least two features. Consider the plan that pushes a nearby box up and then pushes another box down (Figure 7b). This improves the connectivity, allowing boxes to be pushed into the target area (Figure 7c), improving the packed-boxes feature. After having cleared the area this way, the first pushed box can be pushed again (improving connectivity), and the three remaining boxes can be packed.

a) Start



b) Two Pushes (in orange)



c) Cleaning up

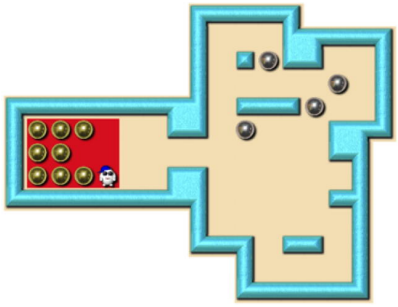


Fig. 7. XSokoban #5

D. Macro Moves

A macro move is a sequence of moves in the domain space. Thus, by definition it has an application-specific component. Using Sokoban as an example, a macro-move is defined as a sequence of moves that push the same box, without pushing

any other box in between. Thus the macro-move describes a push of a box from location (X^1,Y^1) to location (X^2,Y^2). The macro move does not describe the exact order of pushes and player moves needed for this manoeuvre. At a node, all possible macro-moves are generated. In this notation, the solution to Figure 4 (97 pushes; 250 player moves) can be conceptualized as:

- (H,5)-(G,5), preparing a path to the upper room;
- (H,4)-(H,3), opening the upper room;
- (F,5)-(F,7), opening a path to the left room;
- (F,8)-(R,7), packing a box;
- (C,8)-(R,8), packing a box;
- (F,7)-(R,9), packing a box;
- G,5)-(Q,7), packing a box;
- (F,3)-(Q,8), packing a box;
- (H,3)-(Q,9), packing a box.

The first three macros correspond to improving the connectivity feature, and the last six address the packing feature.

Now the solution is much shorter (9 macro moves). The price to pay is a huge increase in the domain-space branching factor. Theoretically, each box can now be sent not just to four neighbor squares but to any empty square on the board.

Our move generator produces macro-moves, and they are treated as the basic move unit in the DS. The rationale is that macro moves have a greater chance of changing something in the FS. For example, a box can be packed to a target from far away, increasing the value of the packed-boxes feature. The use of macro-moves enables fast progress in the FS. It also makes it easier to assign weights, as the macros facilitate changes in the feature values. The downside is that node expansion is expensive (many macro-moves and their feature values have to be computed).

E. Advisors

To address the problem of the large branching factor, we introduce the concept of advisors. Advisors are domain-specific heuristics that aim to pick promising moves. The advisors are implemented using the FESS weighting mechanism: moves that are recommended by advisors are assigned a small weight, and all other moves are assigned a large weight. This allows the solver to focus on relevant moves for feature-space progress despite the large branching factor. Our solver uses seven advisors. Four of them are directly related to improving features (packed-boxes, connectivity, room-connectivity, and out-of-plan), suggesting ways to advance along a specific axis in the feature space. Two additional advisors identify boxes that stand in other boxes' way to targets, and push them away. The last advisor suggests forcing a way into inaccessible areas. Although these are Sokoban-specific mechanics, they all aim to produce new move options; this concept is applicable to any domain.

In our implementation, each advisor is allowed to suggest at most one move. After some experimenting, we set the weight of advisor moves to 0 and the weight of all other moves to 1. Consequently, variations composed solely of advisor moves are explored before other move sequences.

IV. EXPERIMENTS

Experiments were run using an Intel Xeon W-2135 CPU running at 3.70GHz with 64 GB of memory (less than 1 GB of memory was used). Table I gives the results for running FESS on all 90 XSoboban problems. The table columns are:

- #: XSoboban problem number,
- B: Best known solution length (# pushes),
- L: FESS solution length (# pushes),
- E: Number of FESS nodes expanded, and
- T: Time used (in seconds), including pre-processing.

The FESS-based solution is the first to report solving all 90 problems. The *total time* to solve the test set is 229 seconds – an average of 2.5 seconds per problem. The program expands almost 340 nodes per second (E/T), a relatively small number in the single-agent search literature (e.g., [21]). Node expansion is expensive, as it includes adding all that node's children with their feature values to the search tree.

Problem #29 turns out to be the most difficult to solve, building a search tree that is at least five times larger than for the other problems. The solution requires the algorithm to go through high connectivity and room-connectivity values, while the final position has lower values. The features are not as effective as they are for other problems at guiding the search. Still, #29 is challenging for all solvers; being able to find a solution here is an accomplishment.

It is difficult to compare Sokoban solvers, given that they differ in significant ways in terms of their algorithm, heuristics, and definition of an expanded node. For example, ROLLING STONE (and others) use a best-first A* or a depth-first iterative deepening A* search strategy [22], which has different search properties than FESS. TALKING STONE uses a different form of abstraction than in this paper, making it hard to compare search effort. As well it appears that their packing heuristic is not as good as in our implementation. SOKOLUTION solves 81 problems and might be considered a modern version of ROLLING STONE with additional enhancements [23]. TAKAKEN has reported solving 86 (missing #s 29, 50, 66, and 69) but the algorithm has not been published [24].

Table I shows the number of pushes made in the solutions. For each problem the best known solution length is given (B [25]) along with the FESS solution length (L). Generally speaking, FESS tries to find solutions with minimum weight when shifting between cells in the feature space (using mainly advisor moves). Getting search directions from an abstract space and applying them in the domain space guarantees some degree of non-optimality in the solution length. The average solution length found by FESS is roughly 18% longer than the best known result. There are open-source optimizers that can manipulate a given solution to try to reduce the number of pushes. We tried one such tool [26] and saw a 7% reduction in the solution length.

Many Sokoban solvers, SOKOLUTION for example, use brute-force search (many nodes; low cost per node) and they perform quite well. In contrast, FESS searches less by making "intelligent" move selections (few nodes; high cost per node).

#	B	L	E	T	#	B	L	E	T	#	B	L	E	T	#	B	L	E	T
1	97	103	12	0	26	195	243	141	1	51	118	130	22	0	76	204	368	874	2
2	131	153	15	0	27	363	373	81	0	52	421	525	4947	13	77	368	388	1089	2
3	134	150	14	0	28	308	316	39	0	53	186	250	16	0	78	136	146	10	1
4	355	365	24	1	29	164	236	25819	50	54	187	267	21	0	79	174	188	76	0
5	143	147	47	0	30	465	507	240	0	55	120	202	1383	7	80	231	249	39	0
6	110	110	16	0	31	250	342	332	2	56	203	243	24	0	81	173	227	23	0
7	88	168	122	0	32	139	171	1474	4	57	225	283	29	0	82	143	173	21	0
8	230	282	43	0	33	174	230	207	0	58	199	229	28	1	83	194	234	24	0
9	237	245	35	0	34	168	232	4563	8	59	230	270	144	0	84	155	183	29	1
10	512	568	298	3	35	378	414	119	1	60	152	172	17	0	85	329	367	292	1
11	241	303	592	2	36	521	565	4096	11	61	263	301	217	1	86	134	152	17	0
12	212	256	22	2	37	284	434	41	0	62	245	259	22	1	87	233	271	399	1
13	238	262	30	0	38	81	123	134	0	63	431	463	82	0	88	390	440	284	1
14	239	269	24	1	39	672	738	358	1	64	385	399	22	0	89	379	509	2773	14
15	122	252	915	3	40	324	360	4991	12	65	211	227	49	0	90	460	558	581	3
16	186	310	1423	2	41	237	269	1342	7	66	325	459	46	8					
17	213	221	26	0	42	218	416	56	1	67	401	441	467	2					
18	124	162	1049	3	43	146	160	115	0	68	341	363	753	2					
19	302	346	142	1	44	179	201	656	1	69	433	539	41	4					
20	462	528	45	0	45	300	348	63	1	70	333	369	28	0					
21	147	173	51	0	46	247	297	115	0	71	308	340	104	2					
22	324	466	178	2	47	209	225	121	0	72	296	366	94	0					
23	448	466	3787	11	48	200	218	861	3	73	441	469	55	1					
24	544	600	379	2	49	124	188	376	1	74	212	268	2888	9					
25	386	426	1041	5	50	370	460	2580	9	75	295	359	320	1					
																23605	28043	77600	229

TABLE I
XSOKOBAN #1 - #90 (SOLUTION LENGTH, NODES EXPANDED, AND RUNNING TIME).

Which one is better and under what circumstances is an interesting research question.

The FESS results raise two obvious questions: how much of the performance is due to better heuristics, and how does FESS compare to A*-based solutions. To fully answer these questions, one should implement a solver program that combines the new heuristics with a well-understood search algorithm, such as MHGBFS [8] or IMHA* [9]. However, this goes beyond the scope of this paper. That the FESS-based implementation solves all 90 problems – and does so much faster than other A*-based programs – is strong evidence of the FESS algorithm's ability to effectively utilize multiple features.

As a sanity check, an experiment was conducted to compare FESS to A*. In this experiment, only the packing feature was used, and FESS was replaced with Weighted A* (WA*) [27]. The program was able to solve 65 problems, compared to 70 problems when using FESS. Unfortunately, this comparison provides few insights. On the one hand, WA* is forced to find a bounded non-optimal solution; FESS has no such constraint. On the other hand, FESS is deprived of its major strength – using several features to guide the search. Nevertheless, the result is interesting because it shows that the FESS approach can be competitive with A* variants, even when a single feature is being used. Whether this holds for other domains and under what conditions is a question for future study.

V. CONCLUSIONS AND FUTURE WORK

The FESS algorithm uses multiple features in an abstract space to provide guidance to the search in the domain space. Although the algorithm is general, the features are application specific, and this is where the program designer's efforts are concentrated. As in heuristic search in general, a good choice of features is usually critical to success. FESS does not combine features in the way used by typical single-agent search applications. By using multiple features and ensuring that they all have a say in the move selection process, a FESS search avoids the plateau problem often seen in single-agent search/optimization problems.

In the Go-playing program ALPHAGO, a "value network" (position values) and a "policy network" (move selection) benefited from human input [28]. Loosely speaking, this is analogous to our features and advisors. Subsequently, the program became ALPHAZERO by eliminating all human knowledge via machine learning [29]. Ideally, features/advisors should be automatically discovered. This is the ambitious next step for Sokoban solver development.

At its heart, FESS is using multiple heuristics to guide the search. There is nothing inherent in that observation that restricts it to single-agent-search applications. Future work will focus on trying the FESS approach with applications that currently use different search algorithms, such as two-player games.

ACKNOWLEDGMENT

The first author thanks Brian Damgaard and Matthias Meger for their thorough review of the paper and their invaluable suggestions. Thank you to Björn Källmark for permission to use his beautiful "classic" Sokoban skin. Björn's SOKOBAN FOR WINDOWS program was used to produce the diagrams in this paper.

The second author thanks Nathan Sturtevant for his valuable feedback.

REFERENCES

[1] N. Sturtevant, A. Felner, M. Likhachev, and W. Ruml, "Heuristic Search Comes of Age," in *AAAI National Conference*. AAAI, 2012, p. 2186–2191.

[2] M. Asai and A. Fukunaga, "Exploration Among and Within Plateaus in Greedy Best-First Search," in *ICAPS*. AAAI, 2017, pp. 11–19.

[3] L. Mandow and J. De La Cruz, "Multiobjective A* Search with Consistent Heuristics," *Journal of the ACM*, vol. 57, no. 5, pp. 1–25, 2008.

[4] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems*, vol. 4, no. 2, pp. 100–107, 1968.

[5] A. Junghanns and J. Schaeffer, "Sokoban: Enhancing General Single-Agent Search Methods Using Domain Knowledge," *Artificial Intelligence*, vol. 129, no. 1-2, pp. 219–251, 2001.

[6] Sokoban Home Page, cs.cornell.edu/andru/xsokoban.html.

[7] R. Greenblatt, D. Eastlake, and S. Crocker, "The Greenblatt Chess Program," in *AFIPS Fall Joint Computer Conference*, 1967, pp. 801–810.

[8] M. Helmert, "The Fast Downward Planning System," *Journal of Artificial Intelligence Research*, vol. 26, p. 191–246, 2006.

[9] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev, "Multi-Heuristic A*," in *Proceedings of Robotics: Science and Systems*, 2014.

[10] ——, "Multi-Heuristic A*," in *International Journal of Robotics Research*, vol. 35, no. 1-3, 2016, pp. 224–243.

[11] D. Dor and U. Zwick, "Sokoban and Other Motion Planning Problems," *Computational Geometry*, vol. 13, pp. 215–228, 1999.

[12] Susan Miller, "Creating "Sokoban"," courses.elearningmo.org/cf2011/odreams/Sokoban_Lesson_Plan_SM.pdf.

[13] International Planning Competition, 2018, bitbucket.org/ipc2018-temporal/domains/src/master/sokoban.

[14] T. Virkkala, "Solving Sokoban," 2011, weetu.net/Timo-Virkkala-Solving-Sokoban-Masters-Thesis.pdf.

[15] J.-N. Demaret, F. V. Lishout, and P. Gribomont, "Hierarchical Planning and Learning for Automatic Solving of Sokoban Problems," in *20th Belgium-Netherlands Conference on Artificial Intelligence*, 2008.

[16] A. Botea, M. Muller, and J. Schaeffer, "Using Abstraction for Planning in Sokoban," in *Computers and Games*. Springer-Verlag, 2002, pp. 360–375.

[17] R. Pascanu and *et al.*, "Agents That Imagine and Plan," 2017, deepmind.com/blog/article/agents-imagine-and-plan.

[18] M. Crippa, "Monte Carlo Tree Seach for Sokoban," 2018, politesi.polimi.it/bitstream/10589/140141/1/2018_04_Marocchi_Crippa.pdf.

[19] V. Ge, "Solving Planning Problems With Deep Reinforcement Learning and Tree Search," 2018, ideals.illinois.edu/bitstream/handle/2142/101086/GE-THESIS-2018.pdf.

[20] T. Cazenave and N. Jouandeau, "Towards Deadlock Free Sokoban," in *Board Games Studies Colloquium*, 2010, pp. 1–12.

[21] Z. Bu and R. Korf, "A*+IDA*: A Simple Hybrid Search Algorithm," in *AAAI National Conference*. AAAI, 2019, pp. 1206–1212.

[22] R. Korf, "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, vol. 27, no. 1, p. 97–109, 1985.

[23] F. Diedler, "Sokolution," codeanalysis.fr/sokoban.

[24] K. Takahashi, "Takaken," ic-net.or.jp/home/takaken/e/soko.

[25] Sokoban Project, sokobano.de/en/index.php.

[26] B. Damgaard, "YASC: Yet Another Sokoban Clone," sourceforge.net/projects/sokobanyasc.

[27] I. Pohl, "First Results on the Effect of Error in Heuristic Search," *Machine Intelligence*, vol. 5, p. 219–236, 1970.

[28] D. Silver and *et al.*, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature*, vol. 529, p. 484–489, 2016.

[29] ——, "Mastering the Game of Go Without Human Knowledge," *Nature*, vol. 550, p. 354–359, 2017.