Solving Sokoban with curriculum learning

General

This work presents a Sokoban solver based on curriculum learning. It is inspired by the paper “Solving Hard AI Planning Instances Using Curriculum-Driven Deep Reinforcement Learning” by Feng, Gomes and Selman. The solver described in their paper is built upon deep neural networks, and uses a high-end machine with 5 GPUs, and a solution time of 24 hours. In contrast, the objective of this research is to develop a practical Sokoban solver, in terms of architecture used and solution time.

Move prediction and evaluation

We start with a quick recap of the “Solving AI Planning…” paper (it will be referred to as “Feng” for simplicity). Feng suggests an approach in which the solver first tackles *subcases* of easier difficulty. The solver starts by randomizing subcases with only two boxes/targets, and gradually increases the difficulty by solving subcases with more and more boxes.

The program trains both a policy-network and a value-network. The policy network gets a board position and computes the probability of each move. The value-network predicts how many moves are required to solve the level.

Training works as follows: Assume that a solution to a subcase has been found. We will refer to such a solution as an *episode*: a sequence of states + actions (moves) that shift from the initial board position to the final (solved) position.

Given that in state Si of the episode, move Mi was played, the policy network is trained to predict Mi, and the value network is trained to predict (episode-length - i).

When the solver gets a board position, it runs a small search: the policy network stochastically predicts which moves should be examined, and the value-network scores them. The move resulting from the search is applied to the board, and the process repeats until the level is solved or the episode becomes too long.

Note that this learning is AlphaZero style: as more solutions are found, the networks become more accurate; and as the networks become more accurate, more solutions are found. When the networks become proficient enough and solve subcases with a high enough probability, the difficulty is increased by adding more boxes to the subcases.

The two networks trained by Feng are quite large: they are GNNs with 10 layers and an embedding of size 128 for each square. They have different weights for adjacent horizontal and vertical squares. If I understand correctly, then the number of weights in their network is 2*10*128*128 = 327680. Training such a big network requires a long time on a strong architecture. Specifically, they use a 5-GPUs machine for 24 hours. 120 GPU hours are roughly comparable to 1200 CPU hours, for a single Sokoban level.

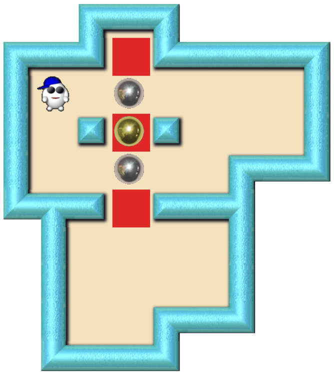
Decision based on similarity

We propose a different approach in this research. Instead of using deep neural networks, we rely on similarities to past episodes in order to make decisions.
When tackling a subcase with K boxes, the solver is given access to all past episodes that solved K-1 boxes subcases. Each past episode finds the position that most resembles the current board position. Then, it suggests playing the same move that was played in the past. We name this method “Similarity based actions”, or in short, SIMBA.

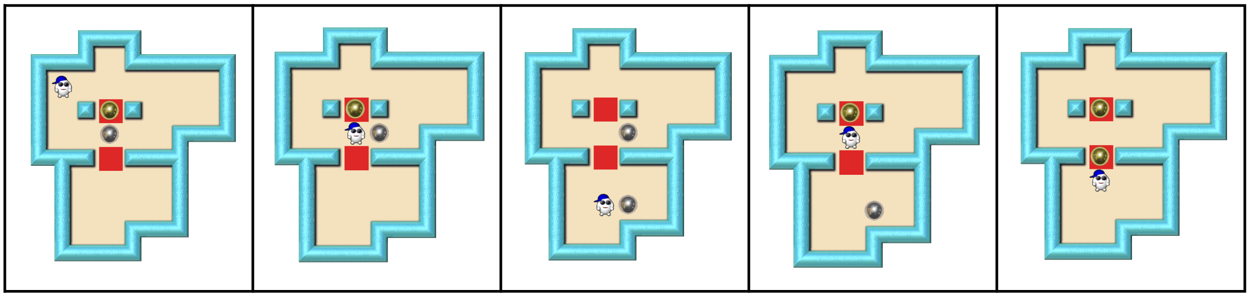
```
Algorithm SIMBA(board B, past episode E)
        foreach state Si in E:
                Similarity[i] = overlap(B, Si)
        BestMatch = argmax(Similarity)
        return E.move[BestMatch]
```

The function “overlap” counts the number of boxes which are on the same squares.

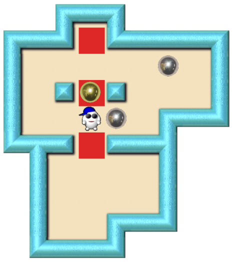
Since this is the core unit of the method, we provide a concrete example. Consider level LOMA04-04:



This level has 3 boxes. An episode solving a subcase with 2 boxes can be:



Suppose that we now try to solve the full problem, and evaluate the board position:

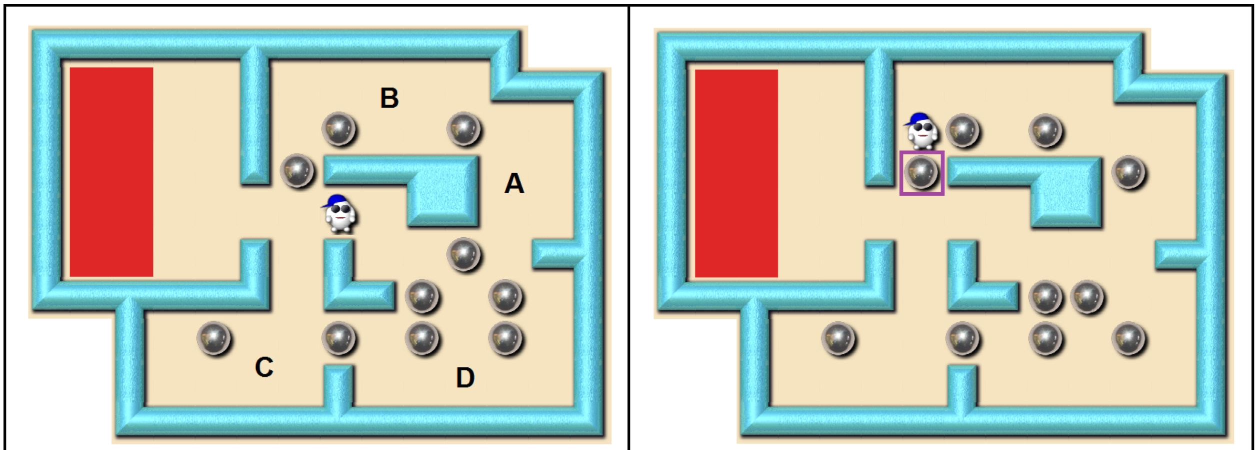


Its overlap values with the episode are: 1,2,1,1,1
Therefore the suggested move is parking the middle box in the lower room.

Note that SIMBA can be implemented to run in a time linear of the episode's length, if overlap is updated incrementally.

Move selection

Given a board position, we apply SIMBA for all past episodes. This results in a distribution of moves to play. Note that some of these moves may be impossible in the current board position; others may be irrelevant due to solving a different problem. The **hope** of the method is that over a large collection of episodes, we should still get a bias for moves pointing us in the right direction.
For a further intuition, consider level XSokoban #2:



XSokoban #2 and its subproblems

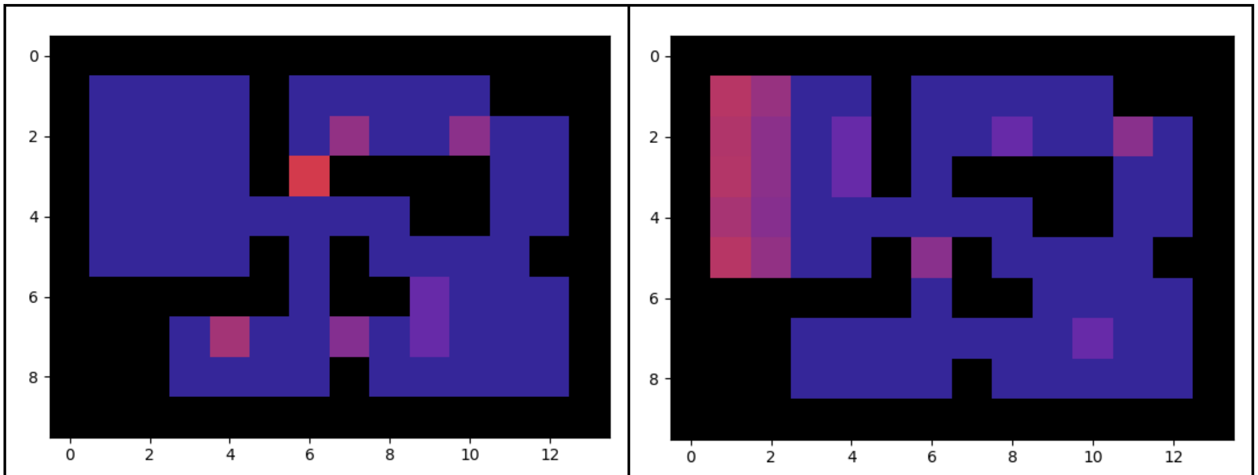
After solving subproblem B

Solving this level consists of solving smaller subproblems: getting into the A zone, getting into the B zone, packing the marked box, and packing the boxes from zones C and D. Getting into the B zone, for example, remains a problem even when there are less boxes in A,C or D. Only when a box from B is removed, the subproblem becomes easier. Thus, we can assume that most subcases will have to deal with the problem of getting into B, with only a small portion of subcases pointing us in irrelevant directions. The same argument can be claimed for solving any of the other subproblems. To sum up, we speculate that if the level can be partitioned into independent subproblems, then most past episodes will point us in the right direction.

Following this approach, we make a count of the moves suggested by the past episodes, and pick the current move with probability proportional to this count. For the sake of generalization, we also assign a small probability to similar moves, going from the same source square or to the same target square.

```
Algorithm GetMoveProbability(board B, past episodes ES)
    initialize Count
    foreach E in ES:
        SuggestedMove = (s,t) = SIMBA(B, E)
        Count[s][t] += 1
        for x in squares:
            Count[s][x] += δ
            Count[x][t] += δ

    compute all possible moves from B
    foreach possible move Mi = (Si,Ti):
        Weight[i] = Count[Si][Ti]
    normalize Weights to a distribution
    return Weights
```

Let us see a concrete example of the count distribution. We consider XSokoban #2 presented above, after the player has entered zone B. For the sake of visualization, we present two distinct counts: of the source square and of the target square.



Source distribution

Target distribution

Looking at the source distribution, we see that the solver rightfully predicts that the best box to push is the one marked in the diagram above. Other box moves could have been useful in subcases with less boxes. For example, in episodes where the marked box is missing, the box next to it can be directly packed instead. Note how the algorithm pays very little attention to boxes in zones A and D. It seems to "understand" which boxes are relevant for the current configuration.

Looking at the target distribution, we see that the solver prefers to pack boxes near the back wall, and even better, in the corners. Note that there's nothing in the algorithm that encodes this useful behavior; it is a simple averaging of successful past episodes.

Some of the source/target squares in the heatmaps do not have boxes in the actual board configuration. The heatmaps serve as a general "strategy" of what should be done in similar scenarios; the second part of "GetMoveProbability" assigns probabilities to the moves that can actually be played.

The heatmaps are presented in a logarithmic scale, in order to show "what the program considers". In practice, any square that is not bright will probably be ignored. In the example position, the algorithm will probably pack the marked box near the back wall, while mostly ignoring any other options.

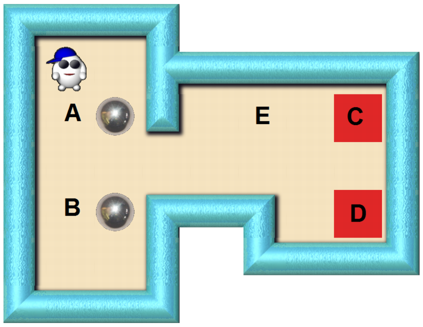
Choosing moves

Choosing moves relies mostly on the probability described above, with two minor enhancements:

```
Algorithm ChooseMove(board B, past episodes ES)
    M = ComputePossibleMoves(B)

    if HasWinningMove(B,M):
        return winning move

    with probability ε:
        return a random move from M

    P = GetMoveProbability(B, ES)
    return a move from M with probability by P
```

First, if there is a move that puts all boxes on targets, then the move is selected. This is useful when initiating the learning process with subcases of a single box, when no past episodes are available. But it also speeds up the solving in other cases. Note that this rule does not use any Sokoban-specific knowledge except for the winning condition.

The second addition is allowing playing a random move with a small probability. Sometimes it essential to play moves that were not observed in past episodes, as exemplified in the level:



When solving this level with a single box, the solver will find the moves
A⇒C, A⇒D,B⇒C,B⇒D
However, solving the version with two boxes requires first parking a box on E. This move can be found using the randomized move selection. Then following the moves suggested by the past episodes will solve the level. In fact, the generalization that we described earlier also allows this randomized behavior, but the probability may be smaller than ε.

In the usual case, there is no winning move and no random move is selected; then a move is selected according to the probability defined by the past episodes.

Now that we have a move selector, it is easy to generate episodes.

```
Algorithm GetEpisode(board B, past episodes ES)
    E = {}
    do MaxSteps times:
        m = ChooseMove(B, ES)
        E.append(m)
        ApplyMove(B, m)
        if BoardSolved(B):
            OptimizeSolution(E)
            return E
    return Failed
```

Optimizing the solution

Sometimes when a board is solved, the solution is longer than it could be. It randomly pushes boxes around the board, until it reaches a familiar configuration and continues to solve. This is a problem for the following reasons:

- When SIMBA executes its argmax, it has more confusing options to choose from
- SIMBA execution is slower
- SIMBA will suggest playing the random moves, and they will be copied to future episodes that will continue to add random moves (similar to garbage DNA)

In order to avoid that, we apply a simple optimization to the solution. Suppose that a box is pushed in step i from A to B, and in step j from B to C. Then we can try to directly push the box from A to C, either in step i or in step j. If this move is possible and all other episode moves remain valid, then the episode is shortened by one move. We repeat the process until no more optimizations can be done.
Note that this optimization does not require any Sokoban specific knowledge and relies solely on the move generator in order to check the validity of moves.

At this point we almost have all the components required to run curriculum learning. Two simple procedures remain:

```
Algorithm GetEpisodes(N_Boxes, N_Episodes, PrevES)
    ES = {}
    while (ES.len() < N_Episodes)
        B = GetRandomBoard(N_Boxes)
        E = GetEpisode(B, PrevES)
        if (E != failed)
            ES.append(E)
    return ES
```

The function GetRandomBoard returns a board with a random subset of N_Boxes boxes and target squares.

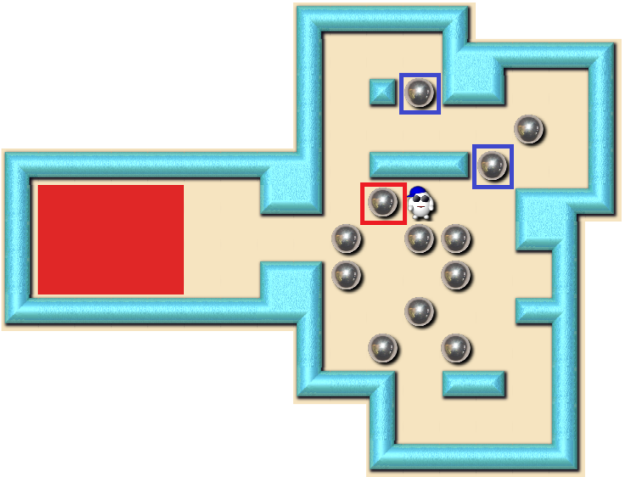
However, this algorithm has a subtle flaw that we will address in the next section. For simplicity, we continue with it for the moment.

Now we can finally learn to solve the level:

```
Algorithm CurriculumLearning(board B, N_Episodes)
    prevES= {}
    N_Boxes = BoxesInLevel(B)

    for k = 1 to N_Boxes
        ES = GetEpisodes(k, N_Episodes, PrevES)
        prev_ES = ES
```

When the solver gets unlucky

The algorithm described so far is indeed able to solve several XSokoban levels. However, sometimes it exhibits strange behavior. If a level has N boxes, then the program easily solves all instances with 1,2,3...N-1 boxes, and then it gets stuck, no matter how much time it is given. In order to understand why, it is helpful to look at an example (XSokoban #5):



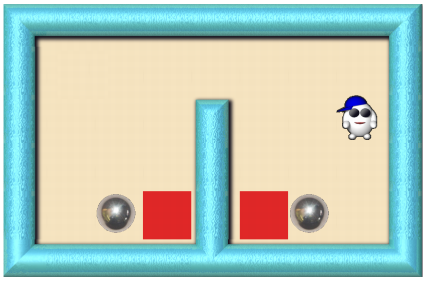
In this position, it appears that a good move would be pushing the box marked in red to the left. This opens access to a new space and allows packing the first boxes. Unfortunately, this move is deadlocking. After playing it, the player cannot enter the upper area anymore. However, if one of the two boxes marked in blue is removed, then the initial move can be safely executed.

We therefore get the following dynamics: When the solver faces subcases with few boxes, the initial move seems to be a good one and solutions are found. As more boxes are added, the solver begins to encounter configurations where the two blue boxes are on board. These cases are not solved and do not result in new episodes. However, occasionally the program is “lucky” and randomizes an initial configuration where one of the blue boxes is missing; the solution is added to the episodes. In short, we get a feedback loop where “lucky” configurations create episodes solved with the initial move, and the initial move encourages episodes with lucky configurations. Unfortunately, when the solver gets to the full problem, it cannot rely on luck anymore and consistently fails.

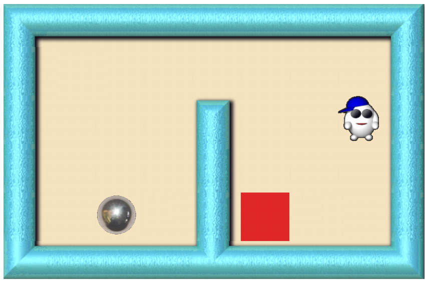
In order to deal with this problem, we will change the GetEpisodes routine. Instead of randomizing the initial board each episode, we begin with a fixed set of board positions. The solver is allowed to proceed to the next step only after solving them. This way, the resulting episodes do not rely on lucky configurations.

```
Algorithm GetEpisodes(N_Boxes, N_Episodes, PrevES)
    ES = {}
    for i = 1 to N_Episodes
        Bi = GetRandomBoard(N_Boxes)

    while (ES.len() < N_Episodes)
        Pick an unsolved board Bi
        E = GetEpisode(Bi, PrevES)
        if (E != failed)
            ES.append(E)
            mark Bi as solved
    return ES
```

We note that some boards may be unsolvable, as is mentioned in Feng's paper and exemplified here:



A solvable level with two boxes

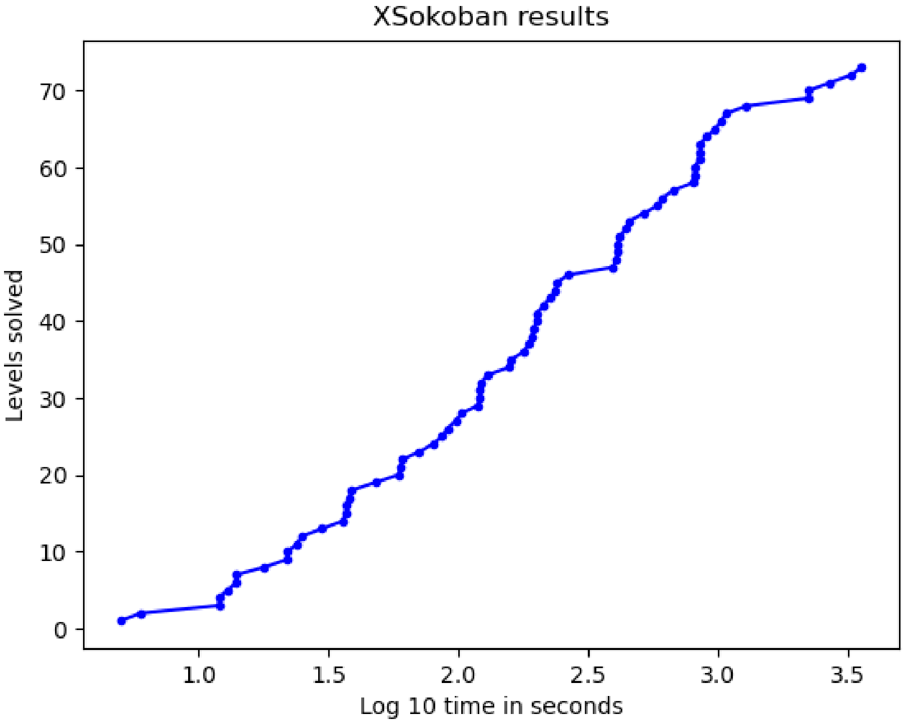


An unsolvable subcase with one box

Therefore in practice we do not require solving all boards, but only a large portion of them (currently, 95%).

Experimental results

The following graph presents the XSokoban results. The program is able to solve 73 levels with a time limit of one hour.



Computing value estimates

The current program does not use any search. Given a board position, it simply calls ChooseMove to play using the policy distribution. In Feng's paper, the program runs a small search before making a decision. The policy network is used for move selection in the search, and the value network estimates the distance to the solution.
So far, we have only described the policy part of our implementation. However, it seems that we can easily produce value estimates: the SIMBA function can simply return
E.len - BestMatch
instead of:
E.move[BestMatch]

Thus the same core unit can serve as a replacement for both the policy network and the value network.

Running a search before making a move means that episode generation will be slower, but each episode will have a higher chance to succeed. In my experiments, the value-based searches underperformed compared to just playing by the policy.

Summary

On the good side

We confirmed Feng's claim that Sokoban levels can be solved by first solving easier cases with fewer boxes. The curriculum learning approach **works**.

We presented SIMBA as a substitute for Feng's deep neural networks. Using this method, we were able to solve Sokoban levels on a single core, at a computational cost that is **1/1200** compared to Feng's experiments. Thus, we believe that a practical Sokoban solver can be obtained.

The solver works without **any handcrafted features** or human designed heuristics. The only thing that it knows is the winning condition of the game, and from there it proceeds to learn how a specific level is solved. This is pretty cool.

Our method is **explainable**. Given a move probability, we can easily explain it by observing the relevant counts and heatmaps. We can even retrieve all past episodes that played this move in order to understand its part in the solution. When using a deep policy network, understanding move probabilities is much more difficult, if possible at all.

On the bad side

The program is painfully slow. Levels that are solved in seconds by the top solvers require several minutes, or remain unsolved.

Solving 73 XSokoban levels is not very impressive. This result is better than “Rolling-stone” or “Talking stones”, but it’s very far from the modern solvers. In fact, the distance seems daunting at this stage.

So far my impression is that the program is able to solve easy levels, and sometimes moderate levels. It doesn’t seem to be able to solve hard levels as reported by Feng. I suspect that the deep-networks are smarter after all.

Appendix - implementation details

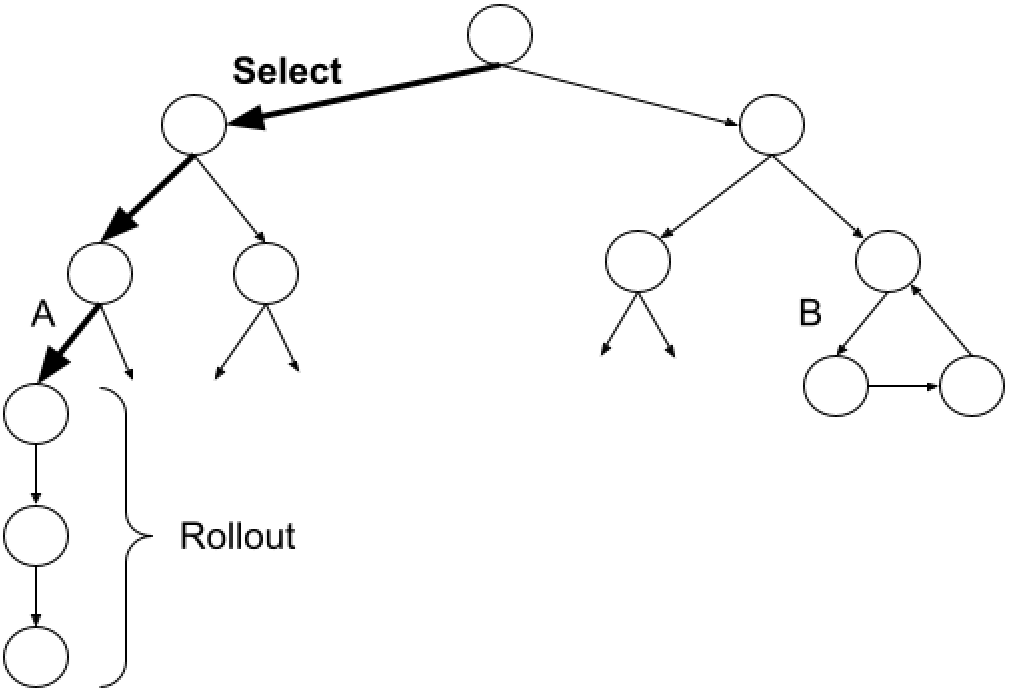
Using a single buffer

In the description so far, the CurriculumLearning procedure used two buffers: Es and PrevEs. In our current implementation, we use a single buffer of all past episodes. When a new board is solved, its solution replaces one of the episodes in the buffer. Specifically, we replace the easiest episode in terms of number of boxes and solution time. Thus the past episodes buffer gradually increases in difficulty, resulting in a smoother transition from K boxes to K+1 boxes.

Search tree

The GetEpisode routine used a single trajectory when trying to solve a board. In practice, we generate multiple trajectories from the root, and save them in a search tree. This has two advantages: first, if a node is visited more than once then there is no need to recompute its moves probabilities. Second, if a node is found to be deadlocked (no possible moves) then this is marked in the search tree and back propagated upwards.

As in MCTS, we use two steps: leaf selection and rollout.



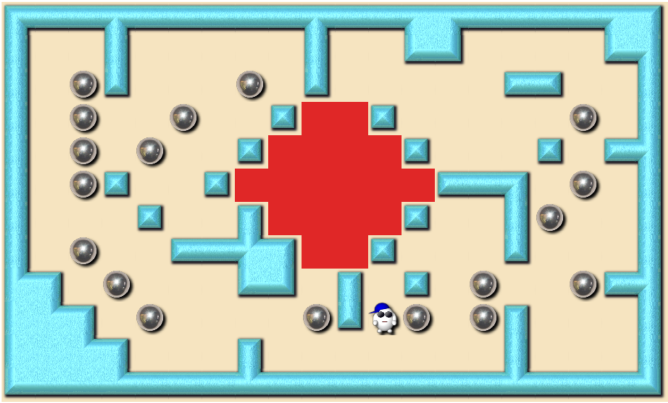
Leaf selection: Traversal of the tree begins at the root and descends using the ChooseMove routine, until it reaches an unexpanded move (leaf A in the figure).
Rollout: A single trajectory is computed starting at the leaf, using the ChooseMove routine. Rollout continues as long as the value function improves.

Sokoban positions are actually an undirected graph, so in some rare cases the leaf selection gets into cycles (see leaf B in the figure). In such cases we use an alternative routine for leaf selection, which picks the leaf with the highest probability of reaching it.

Appendix: some anecdotes of success

The solver is called “Curry”. Although Curry is not very strong, sometimes it has its moments.

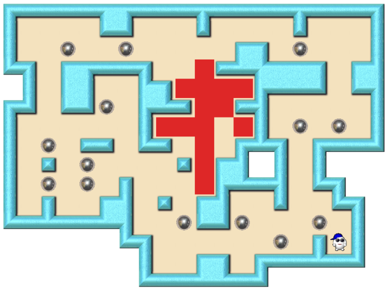
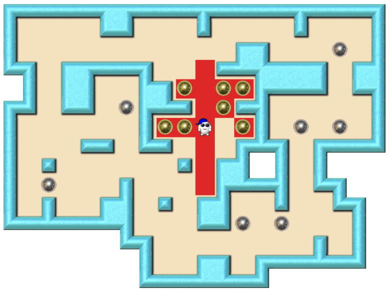
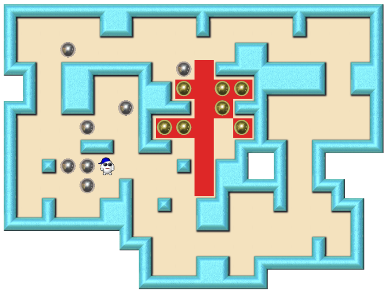
One example is solving XSokoban #71, a level that is missed by the Sokoban solvers Sokolution, JSoko and YASS :



This level requires a careful analysis of the packing order, and it’s quite remarkable that Curry can solve it without a backward search component.

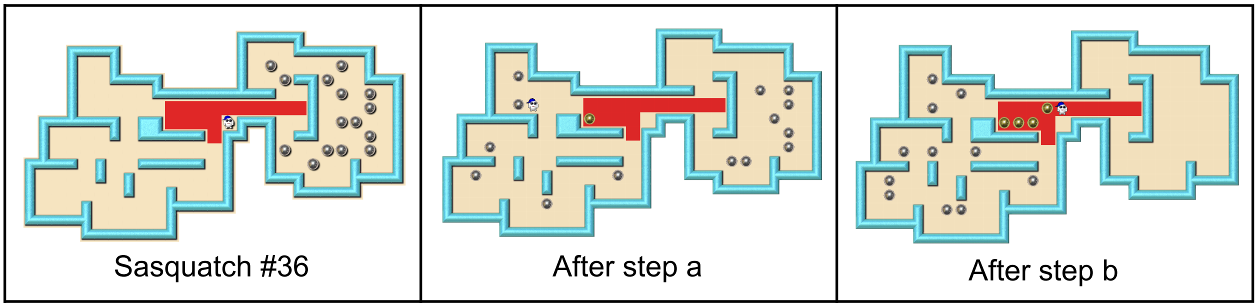
Another surprising example is XSokoban #77, which is missed by the solvers YASS and JSoko. In fact, this level was also difficult for Festival before adding the OOP advisor and the MPDB-2 deadlock detector. Yet, Curry can solve it without using any of these mechanisms.

The level is tricky because the strategy for solving it involves three steps:
a) packing of some boxes
b) moving all boxes from the right side to the left side
c) packing the remaining boxes (this blocks the right side)

 XSokoban #77	 After step a	 After step b

Another example is Sasquatch #36, a level that is missed by Sokolution,Takaken and JSoko. This level is difficult because solving it involves three steps:

a) untying the tight box configuration on the right side
b) sending the boxes to specific parking squares on the left side
c) packing the boxes in a specific order



Sasquatch #36

After step a

After step b

Festival solves this level using a backward search and an advisor that tells it “pull boxes away from targets”. Curry is able to solve the level without a backward search and without any human designed heuristics.