Hybrid informed search Q-Learning for solving Sokoban puzzles

Margarita Geleta
mgeleta@uci.edu
UCI ID: 21804939

Pablo Martín Redondo
pmartinr@uci.edu
UCI ID: 18247656

April 11, 2022

In this report we present our hybrid Reinforcement Learning (RL) Sokoban solver with informed search subroutines. We introduce the game rules and we translate the solving logic into mathematical and algorithmic definitions to define our AI agent. We benchmark A* with our hybrid RL agent, we describe the core algorithm pseudo-codes, data structures and heuristics used in the solvers and provide the time and space complexity of the proposed methods.

1. Introduction

In 1980, Sokoban was created. This Japanese single player game consists on an agent that has to push boxes around a grid and carry them to storage locations. Human beings can solve the levels in minutes, but it is not that easy to solve these puzzles with linear programming algorithms. This fact makes the game a good case of study for the Artificial Intelligence field.

1.1 The Puzzle

Sokoban is played on a two-dimensional grid G of $m \times n$ dimensions. We use the Cartesian coordinate system to locate a position (a cell) in G. Thus, each cell G_{ij} is defined by two coordinates (i, j). Each cell can contain one of the following objects: a player P_{ij}, a k-th box $B_{ij}^{(k)}$, a h-th storage location $S_{ij}^{(h)}$, a wall W_{ij} or nothing, which results in an empty cell $\emptyset_{ij}$ (Figure 1). We have the following assumptions:

1. The number of boxes K equals the number of storage locations, $1 \leq k, h \leq K$;
2. The pairs box-storage can be arranged in any matching combination, i.e., $B_{ij}^{(k)} \wedge S_{ij}^{(h)}, 1 \leq k, h \leq K$;
3. The player can move in four directions ($\uparrow, \downarrow, \leftarrow, \rightarrow$) and can only push one box at a time to storage location and empty cells.
4. Neither the player can move nor the boxes can be pushed into wall cells, $\neg\exists i, j | \exists k(P_{ij} \vee B_{ij}^{(k)}) \wedge W_{ij}$. Therefore, the set of possible values of G_{ij} is defined as:

$$\begin{aligned} &\forall G_{ij} \in \{\emptyset, P_{ij}, B_{ij}^{(k)}, S_{ij}^{(h)}, W_{ij}, \\ &P_{ij} \wedge S_{ij}^{(h)}, B_{ij}^{(k)} \wedge S_{ij}^{(h)}\}, \\ &\forall k, h, 1 \leq k, h \leq K, 1 \leq i \leq m, 1 \leq j \leq n \end{aligned} \tag{1}$$

The puzzle goal is to use the player to push all of the the boxes onto the storage location cells (Equation 2) while minimizing the number of moves.

$$\max \sum_{\substack{k,h \in \{1,\dots,K\} \\ 1 \leq i \leq m \\ 1 \leq j \leq n}} B_{ij}^{(k)} \wedge S_{ij}^{(h)} \tag{2}$$

Where $B_{ij}^{(k)} \wedge S_{ij}^{(h)}$ evaluates to one when a grid cell G_{ij} has a box on a storage location. The sum of these propositions adds to the number of boxes on storage locations, which we want to maximize.

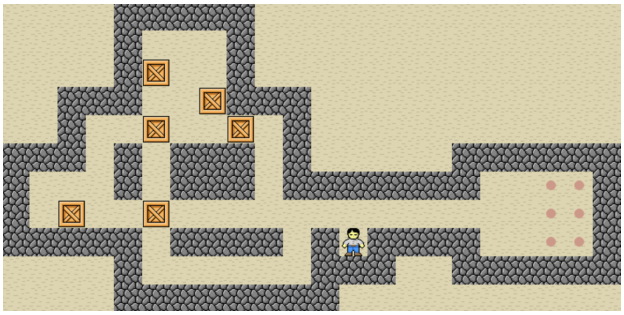

Figure 1: Sokoban sample grid $G_{11\times23}$ with $P_{9,13}$: usually the playing field is surrounded by walls, so that we will always be bounded by walls and cannot reach the edge of the field. We say that the zone outside the wall edge is unreachable by the player.

2. An Intelligent Agent

As it can be guessed, the correct action in each game step may not be trivially obvious. A sequence of actions must be considered in order to achieve the puzzle goal, which is having all the boxes at storage locations with the smallest amount of player moves. For that reason our rational agent needs to undertake a search to find an optimal move at each step. Therefore, we need to provide a clear problem formulation.

Firstly, let us specify the task environment, i.e., the definition of the problem to which our rational agent will try to find a solution. The nature of the task environment is critical for the puzzle solver design [1]. For that aim, we use the PEAS (Performance, Environment, Actuators, Sensors) description (Table 1).

Performance measure	Maximize the number of boxes on storage locations; Minimize the number of moves to win; Minimize the search time
Environment	$G_{m\times n}\|\forall G_{ij} \in \{\emptyset, P_{ij}, B_{ij}^{(k)}, S_{ij}^{(h)}, W_{ij}, P_{ij} \wedge S_{ij}^{(h)}, B_{ij}^{(k)} \wedge S_{ij}^{(h)}\}, \forall k,h, 1 \leq k,h \leq K, 1 \leq i \leq m, 1 \leq j \leq n$ (1)
Actuators	Actions $(\uparrow, \downarrow, \leftarrow, \rightarrow)$
Sensors	Perception of distances between $B_{ij}^{(k)}$ and $P_{i'j'}$ and between $B_{ij}^{(k)}$ and $S_{i'j'}^{(h)}$; Perception of "dangerous" cells or paths, such as corners or deadlocks

Table 1: PEAS description of the task environment.

From Table 1 we can see that the perception of the distances and dangerous zones requires the spatial knowledge of the grid. Thus, the environment is known and fully observable as the agent has access to the complete state of the environment. The selection of the moves is deterministic and the decision may be affected by previous actions, implying that the task environment is sequential, as short-term actions can have long-term consequences. The task environment is semi-dynamic, because the performance measure decreases with time. It is also discrete, because it has a finite number of states. Even though the puzzle is NP-complete [2], the task environment is not the hardest case: it is fully observable, single-agent, deterministic, sequential, semidynamic, discrete and known. As the task environment is known, observable and deterministic, the solution is necessarily a fixed sequence of actions [1].

2.1 The Search Problem

The search of the solution consists in traversing the state space of our search problem. A search problem is defined by: a state space, an initial state, a set of one or more goal states, the actions, a transition model and an action-cost function (Table 2).

In our problem, we define as a "state" at a given point of time to be the grid $G^{(t)}$ at the instant t. In each state $G^{(t)}$ we store the internal elements of the grid, such as the positions of all the boxes, storages, walls and the current location of the player. While the walls and storage locations remain static, the objects that can change are the player and boxes. We say that the state representation is factored, thus, our objects are variables and their locations are values. A puzzle solving episode becomes: $G^{(0)}, a_1, G^{(1)}, ..., a_t, G^{(t)}$ where $a_i \in (\uparrow, \downarrow, \leftarrow, \rightarrow)$. The state space is huge and moving to dangerous zones could jeopardize the puzzle solution. For that reason, as the environment is fully-observable, we need to lock from access those dangerous grid cells or give them a very large cost on access. Dangerous cells are explained in next section. Another important idea is avoid "looping"

State space	A state is the grid $G^{(t)}$ with all of its components at the instant t. he set of all possible states that the environment can be in is the state space.
Initial state	$G^{(0)}$, which corresponds to the grid conditions at the very beginning of the game.
Goal state	$G^{(T)}$ that maximizes Equation 2 and minimizes T.
Actions	$a_t \in (\uparrow, \downarrow, \leftarrow, \rightarrow)$
Transition model*	(1) Player cell → Empty cell: $P_{ij} \wedge (a_t = \uparrow) \wedge \emptyset_{i+1,j} \rightarrow P_{i+1,j}$ $P_{ij} \wedge (a_t = \downarrow) \wedge \emptyset_{i-1,j} \rightarrow P_{i-1,j}$ $P_{ij} \wedge (a_t = \rightarrow) \wedge \emptyset_{i,j+1} \rightarrow P_{i,j+1}$ $P_{ij} \wedge (a_t = \leftarrow) \wedge \emptyset_{i,j-1} \rightarrow P_{i,j-1}$ (2) Player cell → Wall cell: $P_{ij} \wedge (a_t = \uparrow) \wedge W_{i+1,j} \rightarrow P_{ij}$ $P_{ij} \wedge (a_t = \downarrow) \wedge W_{i-1,j} \rightarrow P_{ij}$ $P_{ij} \wedge (a_t = \rightarrow) \wedge W_{i,j+1} \rightarrow P_{ij}$ $P_{ij} \wedge (a_t = \leftarrow) \wedge W_{i,j-1} \rightarrow P_{ij}$ (3) Player cell → Box cell: *Provided there is no wall or box in front of a box in the movement direction.* $P_{ij} \wedge (a_t = \uparrow) \wedge B_{i+1,j} \wedge \neg(W_{i+2,j} \vee B_{i+2,j}) \rightarrow P_{i+1,j} \wedge B_{i+2,j}$ $P_{ij} \wedge (a_t = \downarrow) \wedge B_{i-1,j} \wedge \neg(W_{i-2,j} \vee B_{i-2,j}) \rightarrow P_{i-1,j} \wedge B_{i-2,j}$ $P_{ij} \wedge (a_t = \rightarrow) \wedge B_{i,j+1} \wedge \neg(W_{i,j+2} \vee B_{i,j+2}) \rightarrow P_{i,j+1} \wedge B_{i,j+2}$ $P_{ij} \wedge (a_t = \leftarrow) \wedge B_{i,j-1} \wedge \neg(W_{i,j-2} \vee B_{i,j-2}) \rightarrow P_{i,j-1} \wedge B_{i,j-2}$
Action cost function	For case (1) of the transition model, the cost is +1; for case (2), the cost is $+\infty$, and for case (3), if the position to which the box is pushed is not "dangerous" (or *looping*, when there is a wall in front of the box) then it is +1, otherwise $+\infty$.

Table 2: Search problem definition.

Notes: This limited and explicit transition model will be used by `A*`; in our reinforcement learning approach, our agent will learn a transition model (see Section 2.2).

If we move a player cell into a storage location cell, the behavior for the player will be the same as in (1), the player will move to that cell and $G_{i'j'} = P_{i'j'} \wedge S_{i'j'}^{(h)}$. If the player cell moves into a box cell and there is a wall cell in front of the box in the movement direction, the behavior will be like in (2), i.e., all cells will remain unchanged.

cells, i.e., movements to cells that do not change the state, e.g., moving into a wall. To avoid looping cells we must exclude actions that result in an episode $..., G^{(t)}, a_{t+1}, G^{(t+1)}, ...$ such that $G^{(t)} = G^{(t+1)}$ and $G^{(t)}$ is not a goal state.

2.1.1 Dangerous Cell Detection

We can distinguish between two types of dangerous cells: corners and deadlocks. A "corner" is a cell G_{ij} surrounded by two contiguous wall cells (Algorithm 1). If the player moves into a corner, they can move back to an empty cell, however, if a box is pushed into a corner, it will never be pushed back anymore (Figure 2.c). For that reason, a movement to a corner will be penalized with cost $+\infty$ for A* and will make the episode finish for Q-Learning if a box is pushed there, but the corner itself cannot be excluded from possible player cell's locations.

Algorithm 1 *Algorithm for corner checking at G_{ij}.*

Require: $G_{ij} | 1 \leq i \leq m, 1 \leq j \leq n$
Ensure: $G_{ij} = \emptyset_{ij}$
  function isCorner(G_{ij})
    return $\bigvee_{s \in \{-1,+1\}} W_{i+s,j} \wedge (W_{i,j+s} \vee W_{i,j-s})$
  end function

When we think about a "deadlock" we imagine a cell G_{ij} surrounded by three wall cells (Figure 2.a). In fact, this is just a particular case of a more general definition – a deadlock happens whenever a box is pushed towards a wall and in the perpendicular direction there are corners at both sides and the wall side is contiguous, i.e., with no "doors" (Figure 2.b). If there is a door, then it is not deadlock (Figure 2.d). This algorithm, for checking whether a box is between corners, i.e., case (c), is presented in Algorithm 2.

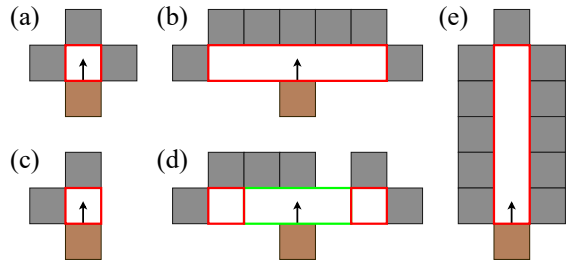

Figure 2: Representation of some dangerous cells. (a) represents the simplest deadlock case, whereas (b) represents a generalization of the deadlock definition; (c) displays a corner; (d) shows a configuration similar to a deadlock from (b) but it is not dangerous because there is a "door" in the wall side; (e) represents an aisle with (a) at the end. Cases (a) and (e) should be pruned.

If a box is pushed into a deadlock, from that state onwards there will not be any solution. While it may be useful to move the player cell into a corner, moving it into a deadlock is always useless. The algorithm for deadlock detection is presented in Algorithm 3.

Algorithm 2 *Algorithm for checking whether G_{ij} is between corners along a contiguous wall.*

Require: $G_{ij} | 1 \leq i \leq m, 1 \leq j \leq n$
Ensure: $G_{ij} = \emptyset_{ij}$
  function isBetweenCorners(G_{ij})
    for s **in** $\{-1, +1\}$ **do**
      if $W_{i+s,j}$ **then**
        return checkSide(G_{ij},s,`horizontal`)
      else if $W_{i,j+s}$ **then**
        return checkSide(G_{ij},s,`vertical`)
      end if
    end for
  end function

Ensure: $s \in \{-1, +1\}$,
  `direction` $\in$ {`horizontal`, `vertical`}
  function checkSide(G_{ij},s,`direction`)
    for d **in** $\{-1, +1\}$ **do**
      if `direction` = `horizontal` **then**
        $j \leftarrow j + d$
        while $(j < n \wedge d = +1) \vee$
          $(j \geq 0 \wedge d = -1)$ **do**
          if isCorner(G_{ij}) **then**
            break
          else if $W_{i+s,j}$ **then**
            $j \leftarrow j + d$
            continue
          end if
          return `false`
        end while
      else if `direction` = `vertical` **then**
        $i \leftarrow i + d$
        while $(i < m \wedge d = +1) \vee$
          $(i \geq 0 \wedge d = -1)$ **do**
          if isCorner(G_{ij}) **then**
            break
          else if $W_{i,j+s}$ **then**
            $i \leftarrow i + d$
            continue
          end if
          return `false`
        end while
      end if
    end for
    return `true`
  end function

Algorithm 3 *Algorithm for deadlock checking at G_{ij}.*

Require: $G_{ij} | 1 \leq i \leq m, 1 \leq j \leq n$
Ensure: $G_{ij} = \emptyset_{ij}$
  function isDeadlock(G_{ij})
    isCorner(G_{ij}) $\vee$ isBetweenCorners(G_{ij})
  end function

Algorithm 4 *Grid pruning algorithm for G*: if there is an aisle with a deadlock, the function `getWayOut`(i, j) returns a position $\emptyset_{i'j'}$ such that $(i' = i \pm 1) \vee (j' = j \pm 1)$, which is the exit "door" from the deadlock.

Require: $G_{m \times n}$
  function PRUNE(G)
    for $i = 1$ **to** m **do**
      for $j = 1$ **to** n **do**
        $(i', j') \leftarrow (i, j)$
        while $\emptyset_{i'j'} \wedge \left(W_{i'+1,j'} + W_{i',j'+1} + W_{i'-1,j'} + W_{i',j'-1} \geq 3\right)$ **do** ▷ (Figure 2.a)
          $W_{i'j'} \leftarrow$ **true**
          if $\neg(W_{i'+1,j'} \wedge W_{i'-1,j'} \wedge W_{i',j'+1} \wedge W_{i',j'-1})$ **then** ▷ If there is a way out
            $(i', j') \leftarrow$ GETWAYOUT(i', j') ▷ Find the door
          end if
        end while
      end for
    end for
  end function

Sometimes, deadlocks are located at the end of aisles (Figure 2.e). Removing the final cell of the aisle transforms the second-last cell into a deadlock of type (a). Therefore, deadlock aisles should be also removed, i.e., be made inaccessible by the player. This pruning can be done iteratively by placing walls in these inaccessible regions. Our proposed algorithm for grid pruning is presented in Algorithm 4.

2.1.2 Tunnel Macros

Other times, aisles are just tunnels that consist of articulation points on the board, i.e., paths that split the grid into two (probably disjoint) pieces [3]. Tunnels can be treated as one square and, thus, explored more directly (Figure 3).

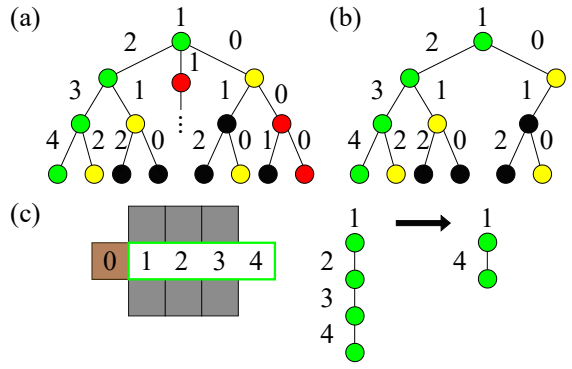

Figure 3: Enhancements in the search: (a) shows an approximate search tree (some paths are omitted for the sake of simplicity) for the (c) pattern; nodes in *green* represent a tunnel path, in *yellow* useless one-step-forward and one-step-backward actions, and in *red* represents actions towards looping cells (e.g., moving into a wall); (b) clearly, the search tree is significantly reduced when we omit looping cells and (c) shows how a tunnel can be converted into a one-step tunnel macro.

2.2 The Solver

For a world with $m \cdot n$ non-obstacle cells and b boxes, there are $m \cdot n \cdot \frac{(m \cdot n)!}{b!(m \cdot n - b)!}$ states; e.g., on an grid $G_{8\times8}$ with a dozen boxes, there are over 200 trillion states [1]. Obviously, we cannot visit all the states, our agent should intelligently check first the paths prone to lead to a solution. A predisposition to check smartly some states before others could come from hints given by some a heuristic function $h(\cdot)$. This strategy is called informed search. A predisposition could also come from experience: our agent could interact with the environment and learn a policy maximizing the rewards. This latter strategy is called reinforcement learning (RL). A new improvement in reinforcement learning is feeding the agent with "hint features" during training [4], which is defining the reward function as a function of $h(\cdot)$.

In this project, we want to benchmark an exhaustive informed search strategy versus a stochastic reinforcement learning method. The Solver structure is defined in Algorithm 5.

2.2.1 Informed search

We have chosen **weighted `A*` search** for our solver's informed search core algorithm, with an evaluation function:

$$f(G^{(t)}) = g(G^{(t)}) + W \cdot h(G^{(t)}) \tag{3}$$

where $g(G^{(t)})$ is the path cost from the initial state $G^{(0)}$ to $G^{(t)}$ (the costs are added from each state transition, see the definition of costs in Table 2); where $h(G^{(t)})$ is the *estimated* cost by heuristics (refer to subsection 2.2.3) of the shortest path from $G^{(t)}$ to a goal state, and W is the *detour index*, $W > 1$.

The Weighted `A*` Algorithm

As we presented before, we are using weighted `A*` for the informed strategy problem-solving algorithm, with some tweaks (presented in Algorithm 6). First of all, we are saving an *identifier* of the state `ID`$(G^{(t)})$

Algorithm 5 *Sokoban Solver*: θ are the hyperparameters for each algorithm. For A*, θ consists of Minkowski distance hyperparameter p and the detour index W. For Q-learning, θ consists of Minkowski p, learning rate α, discount factor γ, greedy ε and decay constant λ.

Require: $G_{m\times n}$
  function SOLVER(G, type, θ)
    $G \leftarrow$ PRUNE(G)
    if type = A* **then**
      $G^* \leftarrow$ A*(G, $\theta = \{p, W\}$)
    else if type = Q-learning **then**
      $G^* \leftarrow Q$-LEARNING(G, $\theta = \{p, \alpha, \gamma, \varepsilon, \lambda\}$)
    end if
    return G^*
  end function

so we do not expand it twice. This identifier represents the position of the player and the boxes (dynamic objects of the state). Secondly, there are some conditions, which are explained next, by which an expanded state is not added to the frontier (see [1]):

1. If we have already visited the state.
2. If the next state is the same as the current (movement to a *looping cell*).*
3. If the next state will is a deadlock.

2.2.2 Reinforcement Learning

For our reinforcement learning approach, we use ***Q*-learning**. Q-learning is a model-free reinforcement learning algorithm, i.e., it learns a direct representation of how to behave. In other words, it learns a direct mapping from state to action, which is given by the *action-utility function* or simply *Q-function*, $Q(G_{ij}^{(t)}, a_t)$. Q-function denotes the expected total discounted (by discount factor γ) reward if the agent takes action a_t and in $G_{ij}^{(t)}$ and acts optimally thereafter [1] (Equation 4):

$$Q(G_{ij}^{(t)}, a_t) = R(G_{ij}^{(t)}, a_t) + \gamma \cdot \max_{a_{t+1}} Q(G_{ij}^{(t+1)}, a_{t+1}) \tag{4}$$

Knowing the Q-function enables the agent to act optimally simply by choosing $\arg\max_a Q(G_{ij}^{(t)}, a_t)$, with no need to look-ahead based on a transition model [1], which is an advantage with respect to A*.

The *Q*-learning Algorithm
After a timestep the agent will choose an action to get to the next state. The Q-function calculates the quality of a state-action pair and the values for that function can be stored in an explicit table which we call Q-table.

First, we initialize the Q-table as empty, which will be filled and updated during training. The training loop looks is: taking an action from the current state, getting a reward, entering a new state-action and updating the Q-table with what is learned. This Q-table, in our implementation, is a hash table of key ID($G^{(t)}$, a_t) and value $Q(G^{(t)}, a_t)$.

During the training, we execute episodes. In these episodes, the agent does an iterative job of taking an action and updating the Q-table. Each episode finishes when the agent gets to a deadlock or when it reaches the goal. The training finishes when we get a policy that can resolve the puzzle (Algorithm 7).

The decision process is as follows:

1. We get a set of next valid actions with the function getAction($G^{(t)}$).
2. Given a parameter ε and the epoch we are at, we apply the ε-greedy policy, i.e., sometimes we execute a random action and sometimes we choose the action given the current sub-optimal policy.
3. We calculate the reward of the selected action with the function getReward($G^{(t)}, a_t$), taking into account whether this action gets us to: a deadlock, to moving a box, to getting closer to a box or to the goal (using "hint features" $h_1(G^{(t)})$ and $h_2(G^{(t)})$, see subsection 2.2.3).
4. We calculate the maximum $Q(G^{(t+1)}, a_{t+1})$.
5. We update the Q-table with the new value (in the function takeAction($G^{(t)}$)):

$$Q(G^{(t)}, a_t) = Q(G^{(t)}, a_t) + \alpha[R(G^{(t)}, a_t, G^{(t+1)}) + \gamma \max_{a_{t+1}} Q(G^{(t+1)}, a_{t+1}) - Q(G^{(t)}, a_t)] \tag{5}$$

2.2.3 Heuristics

Our solvers use the following two heuristics:

1. **Distance heuristic** $h_1(G^{(t)})$: at timestep t, we compute the Minkowski distance of P_{ij} to all of the boxes $B_{i'j'}^{(k)}, 1 \leq k \leq K$ which are not at storage locations yet, and select the distance to the nearest box (Equation 6, Algorithm 8):

$$h_1(G^{(t)}) = \arg\min_k \left(|i - i'|^p + |j - j'|^p \right)^{\frac{1}{p}} \tag{6}$$

2. **Distance heuristic** $h_2(G^{(t)})$: we compute the Minkowski distances of all the boxes to all the

*This could happen when:
(a) There is a wall in the movement direction of the player.
(b) We are pushing a box into a wall.
(c) We are pushing a box into another box.

Algorithm 6 *Weighted A**: The priority queue returns the next $G^{(t+1)}$ with the minimum value of the evaluation function $f(G^{(t+1)})$. `getNextStates`$(G^{(t)})$) returns the 4 possible states $G^{(t+1)}$ given $G^{(t)}$. Finally,the function `isAnyBoxInCorner`$(G^{(t)})$ invokes `isCorner`(G_{ij}) for each G_{ij} where a box is going to be placed.

Require: $G_{m\times n}$
  function A*$(G^{(0)})$
    if isGoal$(G^{(0)})$ **then return** G^0
    end if
    `frontier` ← new(`PriorityQueue`)
    `visited` ← new(`Set`)
    add $G^{(0)}$ to `frontier`
    while ¬ isEmpty(`frontier`) **do**
      $G^{(t)}$ ← pop(`frontier`)
      add id$(G^{(t)})$ **to** `visited`
      if isGoal$(G^{(t)})$ **then return** $G^{(t)}$
      end if
      for $G^{(t+1)}$ **in** getNextStates$(G^{(t)})$ **do**
        if ($\neg G^{(t+1)}$ **in** `visited`) ∧ (¬ equals$(G^{(t)},G^{(t+1)})$) ∧ (¬ isDeadlock$(G^{(t+1)})$) **then**
          add $G^{(t+1)}$ to `frontier`
        end if
      end for
    end while
    return `noSolution`
  end function

storage locations at timestep t and add them. Then we average. (Equation 7, Algorithm 9):

$$h_2(G^{(t)}) = \frac{1}{K} \sum_{B_{ij}^{(k)}} \sum_{S_{i'j'}^{(h)}} \left(|i-i'|^p + |j-j'|^p \right)^{\frac{1}{p}} \quad (7)$$

Additionally, $h_2(G^{(t)})$ can serve as a lower bound on the remaining cost to solve the puzzle. Since there are as many boxes as there are storage locations and every box has to be assigned to a location, this is the formulation of the minimum cost (distance) perfect matching on a complete bipartite graph [3]. In this graph, edges between boxes and storage locations are weighted by the distance between them. Minimum cost perfect matching for a bipartite graph can be solved using *minimum cost augmentation*, also known as "Hungarian method" [5] – given a graph with n nodes and m edges, the cost of computing the minimal cost matching is $\mathcal{O}(nm \log_{(2+m/n)} n)$. Since we have a complete bipartite graph, $m = n^2/4$ and the complexity becomes $\mathcal{O}(n^3 \log_{(2+n/4)} n)$ [3].

The final heuristic is not admissible and is given by:

$$h(G^{(t)}) = h_1(G^{(t)}) + h_2(G^{(t)}) \quad (8)$$

2.2.4 Rapid Random Restart

Another strategy we making our agent "impatient" – sometimes the agent ends up stuck in a search and appears in a hopeless situation where it cannot find any problem-solving path. If the search wastes (relatively) a lot of effort, then probably our agent is on the wrong track and we should switch to another part of the search tree [3]. Rapid Random Restart allows after T timesteps to restart the problem after the search has been unsuccessful, this way saving time and allowing the agent to explore more parts in the search tree.

2.3 Problem-Solving Performance

2.3.1 Completeness

Our informed search solver is guaranteed to find a solution (when there is one) because A* is complete when action costs are $\geq \epsilon > 0$ and the state space is finite.

In contrast, our reinforcement learning solver is not guaranteed to find a solution, but an optimal policy given the rewards. It depends on the stochasticity of the action choices, the definition of the rewards, the training time and the ε, i.e., the probability of taking a random action rather than a greedy one.

If the puzzle has no solution, A* after traversing all possible states and Q-learning after finishing the maximum number of iterations will report failure.

2.3.2 Cost Optimality

A* is cost-optimal when using consistent heuristics [1]. Nevertheless, they may result in more node expansion and, as a result, a slower algorithm. On the other hand, inadmissible heuristics may be sub-optimal but get to a solution faster. Then, if we accept as a solution one in which the agent performs more than the minimal number of steps, this is, accepting this space of solutions as *satisfacing*, we can use inadmissible heuristics to get a solution.

Algorithm 7 *Q-learning algorithm*

Require: epoch= 0, Q-TABLE, $0 \leq \varepsilon, \lambda \leq 1, \alpha > 0, \gamma \leq 1$

```
function TRAIN(nEpochs, nMaxSteps)
    for i = 0 to nEpochs do
        epoch ← i
        EXECUTEEPISODE(G^(0), nMaxSteps)
    end for
end function

function EXECUTEEPISODE(G^(0), nMaxSteps)
    G^(t) ← G^(0)
    steps ← 0
    while ¬ ISGOAL(G^(t)) ∧ steps < nMaxSteps do
        G^(t) ← TAKEACTION(G^(t))
        if ∃i, j ISDEADLOCK(G_ij^(t)) then
            break
        end if
    end while
end function

function TAKEACTION(G^(t))
    a_t ← GETACTION(G^(t))
    R(G^(t), a_t, G^(t+1)) ← GETREWARD(G^(t), a_t)
    G^(t+1) ← GETNEXTSTATE(G^(t), a_t)
    Q(G^(t), a_t) ←Q-TABLE(ID(G^(t), a_t))
    Q(G^(t), a_t) ← Q(G^(t), a_t) + α[R(G^(t), a_t, G^(t+1)) + γ max_{a_{t+1}} Q-TABLE(ID(G^(t+1), a_{t+1}))−Q(G^(t), a_t)]
    Q-TABLE(ID(G^(t), a_t)) ← Q(G^(t), a_t)
    return G^(t+1)
end function

function GETACTION(G^(t))
    validActions ← ∀a_t| GETREWARD(G^(t), a_t) ≠ −1
    probability ← RANDOM(0,1)
    if probability ≤ ε · λ^epoch then
        return GETRANDOM(validActions)
    end if
    return max_{a_t ∈ validActions} Q-TABLE(ID(G^(t), a_t))
end function

function GETREWARD(G^(t), a_t))
    G^(t+1) ← GETNEXTSTATE(G^(t), a_t)
    if ¬EQUALS(G^(t), G^(t+1)) then
        if ∃i, j ISDEADLOCK(G_ij^(t)) then
            return 0
        else if ISGOAL(G^(t+1)) then
            return 100
        else if H1(G^(t+1)) + H2(G^(t+1)) < H1(G^(t)) + H2(G^(t)) then
            return 10
        end if
        return 1
    end if
    return -1
end function
```

This is why we tackle the problem by accepting as a solution any that gets us to $\forall B^{(k)}, S^{(h)} | B_{i,j}^{(k)} \wedge S_{i,j}^{(h)}, 1 \leq k, h \leq K$. Hence, we may be overestimating the cost to get to the solution, but we are getting it in a reasonable amount of time.

For any finite Markov decision process, Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and

Algorithm 8 *Heuristic* $h_1(G^{(t)})$: With this heuristic our agent will prefer a state where a player is located closely to a box to move.

Require: $G_{m\times n}, p$
Ensure: $p \geq 1$
  function H1($G_{m\times n}, p$)
    $h \leftarrow \infty$
    for k=1 **to** K **do**
      $d \leftarrow$MINKOWSKI(P_{ij},$B^{(k)}_{i'j'}$,p)
      if $d < h$ **then**
        $h \leftarrow d$
      end if
    end for
    return h
  end function

Algorithm 9 *Heuristic* $h_2(G^{(t)})$: With this heuristic our agent will prefer a state where boxes are located closer to storage locations.

Require: $G_{m\times n}, p$
Ensure: $p \geq 1$
  function H2($G_{m\times n}, p$)
    $h \leftarrow 0$
    for k=1 **to** K **do**
      for h=1 **to** K **do**
        $h \leftarrow h+$MINKOWSKI($B^{(k)}_{ij}, S^{(h)}_{i'j'}$,$p$)
      end for
    end for
    return $\frac{h}{K}$
  end function

all successive steps, starting from the current state. Nevertheless the time required to find it, may be relatively large.

2.3.3 Time complexity

Even there are some computation wise heavy parts in the code, such as the `prune` function with a $\mathcal{O}(m*n)$ complexity, the main factor is the A* algorithm. Since the problem is NP-hard [2], we could design a verifier that checks a solution in non-deterministic polynomial time.
We are not verifying a solution but doing a search. In a worse case scenario, the branching factor is $b = 4$. Defining d as the depth of the search tree, the worse-case complexity of this A* algorithm is $\mathcal{O}(4^d)$. On the other hand, Q-learning time complexity is $\mathcal{O}(|\mathcal{A}|\cdot 4^d)$, where $|\mathcal{A}|$ is the number of actions [6]. Since in our puzzle $|\mathcal{A}| = 4$, the time of complexity Q-learning becomes $\mathcal{O}(4^{d+1})$.

2.3.4 Space complexity

As we are storing the states in both the `frontier` and the set of visited states, the space complexity is also $\mathcal{O}(4^d)$. In fact, the A* algorithm runs out of memory before than out of time [1].

Regarding Q-learning space complexity, in the worse case it equals to the size of the Q-table. Q-table stores all the state-action pairs, since the number of states is 4^d and we have 4 actions, the space complexity is $\mathcal{O}(4^{d+1})$.

3. Experimental Results

The main result of our experiments is that A* is much faster than Q-Learning [†] solving the problems with a limited branching factor (not too many boxes). Consequently, A* is able to solve more problems than the Q-Learning algorithm. However, when the puzzle grows, both approaches start failing.

A big bottleneck for Q-Learning is that the agent gets stuck in deadlocks that depend on both the agent and the position of K boxes, resulting in the player getting trapped in a portion of the map. To minimize this issue, we use loop detection of those episodes that will never have a solution.

Another bottleneck is memory usage. The state representation of this problem is huge and grows exponentially as seen in Section 2.3.4. As the size of the grid and the number of boxes grow, the program runs out of memory before running out of time.

Due to the fact that A* has shown better performance than Q-Learning alone, we decided to implement it as a subroutine of our Reinforcement Learning algorithm. We called this algorithm: Q-Star (or Q*). The idea is as follows: the valid actions of the agent for the next state are not only $\uparrow, \downarrow, \leftarrow, \rightarrow$, but also A*. This subroutine will do heuristic search and return the state after arriving to the next sub-goal. A sub-goal is defined as moving one single box to one storage. This approach worked better than the standard Q-Learning, but has some issues:

1. If there is not a quick solution, the Reinforcement Learning gets stuck in a step of some episode.

2. Finding a subgoal is not a guarantee of finding a solution. Indeed, as the closest box gets to the closest storage, this situation can be blocking the map and making it impossible to get to a solution.

3. The agent will get rewarded for pushing a box to a storage even if this makes it impossible to get to the final solution.

4. The branching factor grows, as we have even more possible actions.

In our benchmark experiments we have used as default hyperparameters Minkowski $p = 2$ and detour index $W = 200$ for both A* and Q*; for Q* we used a learning rate $\alpha = 0.5$, discount factor $\gamma = 0.6$, greedy $\epsilon = 0.999$ and a constant decay factor $\lambda = 0.999$. We

[†]We implemented our solvers in C++ using compiler-level optimizations. The code is available at our GitHub repository https://github.com/pabloi09/sokoban.

observed that the p for the distance metric has not been significant in run time. All other non-constant hyperparameters have shown slight differences which can be appreciated in Figures 4, 5, 6 and 7 (note the two different scales of the y axis for each of the algorithms). For each hyperparameter value we run 15 experiments on a solvable sokoban puzzle, averaged the execution times and computed the confidence intervals.

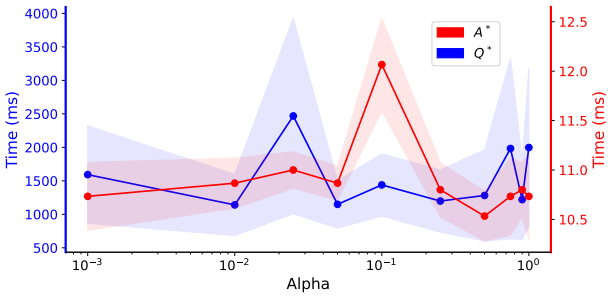

Figure 4: Impact of α hyperparameter in `Q*` with respect to `A*` performance: there is no clear positive nor negative tendency in the values of α.

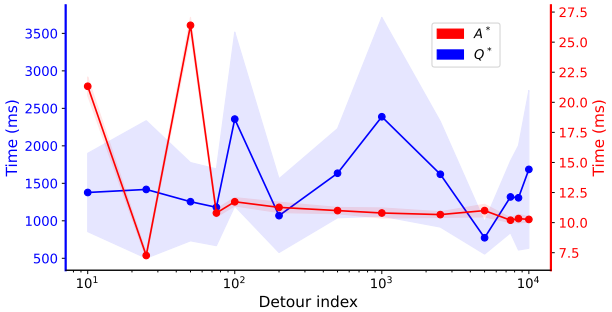

Figure 5: Impact of hyperparameter: better results have been achieved with large values for W, even though `Q*` has certain stochasticity and the variance is larger, averaging shows that the performance is improved when we weight more the heuristics.

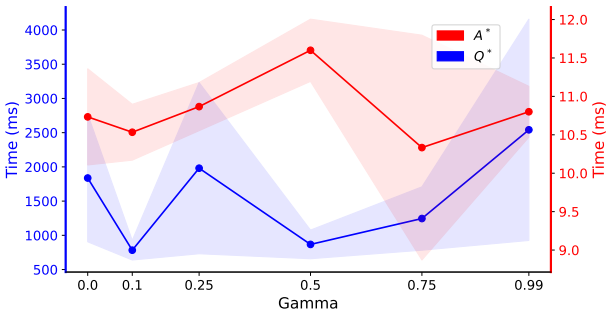

Figure 6: Impact of γ hyperparameter in `Q*` with respect to `A*` performance: results show preference for smaller values of *gamma*. Far-sighted evaluation (large γ) has not worked well in practice.

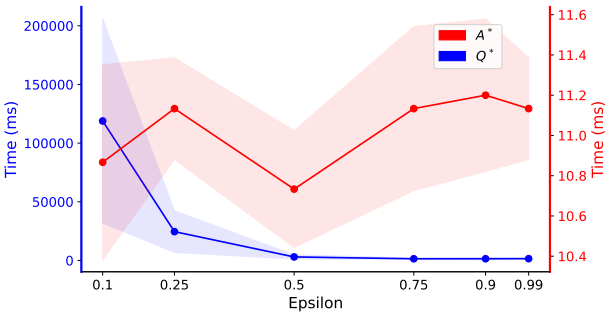

Figure 7: Impact of ε hyperparameter in `Q*` with respect to `A*` performance: large values for ε enforce taking random actions and then slowly switching to greedy actions after sufficiently enough steps. This results clearly shows that randomization is very important for learning the best policy.

Further steps to improve our work include:

1. More advanced loop detection.
2. A simplification of the representation of the states would avoid the memory issues.
3. Perform inverse exploration – explore the map from the solution and try to bring the boxes to their initial positions would prune a lot of states that would become impossible.
4. More fine-tuning on all of the hyperparameters.
5. A more advanced deadlock detection.
6. Establishing a maximum depth for the tree.

4. Conclusions

Solving sokoban puzzles is a NP-complete problem that requires a big computational effort. Simplifications of the problem, good heuristics, well-aimed hyperparameters and state pruning are crucial in order to get to a solution. Our approach has some flaws, and further work is needed in order to solve most of the problems, but this project we have introduced a novel approach that combines both, reinforcement learning and informed search methods.

We believe that a reinforcement learning approach that starts from the end of the problem (inverse exploration) combined with other Artificial Intelligence techniques, such as constraint satisfaction, heuristic search, local search or loop detection; has the potential to improve the current state of our solver.

References

[1] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*, 4rd ed. Pearson, 2016, ISBN: 9780134610993.

[2] D. Dor and U. Zwick, "Sokoban and other motion planning problems," *Computational Geometry*, vol. 13, no. 4, pp. 215–228, 1999, ISSN: 0925-7721. DOI: https://doi.org/10.1016/S0925-7721(99)00017-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925772199000176.

[3] A. Junghanns and J. Schaeffer, "Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock," in *Advances in Artificial Intelligence*, R. E. Mercer and E. Neufeld, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 1–15, ISBN: 978-3-540-69349-9.

[4] Y. Shoham and G. Elidan, *Solving sokoban with forward-backward reinforcement learning*, 2021. arXiv: 2105.01904 [cs.LG].

[5] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955. DOI: https://doi.org/10.1002/nav.3800020109. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nav.3800020109. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109.

[6] S. Koenig and R. G. Simmons, "Complexity analysis of real-time reinforcement learning," in *Proceedings of the Eleventh National Conference on Artificial Intelligence*, ser. AAAI'93, Washington, D.C.: AAAI Press, 1993, pp. 99–105, ISBN: 0262510715.