Programming Assignment 1: Sokoban

Due Jun 7 by 10pm **Points** 100

Released: May 17, 2021

UPDATES:

On May 19, 2021: corrected a typo in the convex downward parabola (XDP) priority formula

On May 20, 2021: clarified that heur_manhattan_distance is to be used to generate your comparison.csv file.

Table of Contents.

1. Critical Warning
2. Introduction
3. Formal Game Description
4. Starter Code
5. Details of Starter Code
6. What To Submit
7. Your Job
8. Weighted A* Variations
9. Anytime Weighted A* Search
10. Anytime Greedy Best First Search
11. Marking Criteria

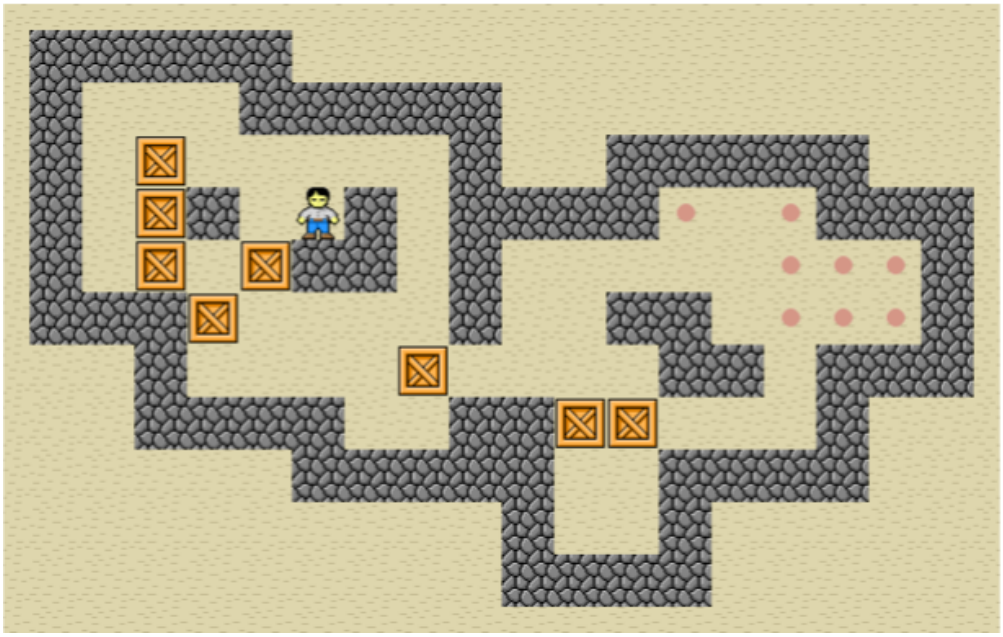

Fig 1. A state of the Sokoban puzzle.

Warning (Please read this)

We are aware that solutions to this problem, or related ones, may exist on the internet. **Do not use these solutions as this would be plagiarism.** To earn marks on this assignment you must develop your own solutions. Also please consider the following points.

- **Do not add any non-standard imports in the python files you submit** (all imports already in the starter code must remain). All imports that are available on teach.cs are considered to be standard.

- **Do not change any of the supplied files except for** `solution.py`
- **Make certain that your code runs on teach.cs using python3**. You should all have an account on teach.cs and you can log in, download all of your code (including all of the supplied code) to a subdirectory of your home directory, and use the command `python3 autograder.py` and test it there before you submit. Your code will be graded by running it on teach.cs, so the fact that it runs on your own system but not on teach.cs is not a legitimate reason for a regrade.
- The test cases used in the autograder reflect the test cases we will use during marking but are not identical. Tests performed by the autograder will help you gauge the final grade you can expect on the assignment. We will also look for certain things in the assignments (e.g., running them through code plagiarism checkers, looking at assignments that fail all tests, etc.). If we have **good reasons** we will change your grade from that given by the autograder either up or down.

Introduction

The goal of this assignment will be to implement a working solver for the puzzle game Sokoban shown in Figure 1. Sokoban is a puzzle game in which a warehouse robot must push boxes into storage spaces. The rules hold that only one box can be moved at a time, that boxes can only be pushed by a robot and not pulled, and that neither robots nor boxes can pass through obstacles (walls or other boxes). In addition, robots cannot push more than one box, i.e., if there are two boxes in a row, the robot cannot push them. The game is over when all the boxes are in their storage spots.

In our version of Sokoban the rules are slightly more complicated, as there may be more than one warehouse robot available to push boxes. These robots cannot pass through one another nor can they move simultaneously, however.

Sokoban can be played online at **this link (https://www.sokobanonline.com/play)**. We recommend that you familiarize yourself with the rules and objective of the game before proceeding, although it is worth noting that the version that is presented online is only an example. We will give a formal description of the puzzle in the next section.

Formal Description

Sokoban has the following formal description. Read the description carefully. Note that our version differs from the standard one.

- The puzzle is played on a rectangle board that is a grid board with N squares in the x-dimension and M squares in the y-dimension.
- Each state contains the x and y coordinates for each robot, the boxes, the storage spots, and the obstacles.
- From each state, each robot can move North, South, East, or West. No two robots can move simultaneously, however. If a robot moves to the location of a box, the box will move one square in the same direction. Boxes and robots cannot pass through walls or obstacles, however. Robots cannot push more than one box at a time; if two boxes are in succession the robot will not be able

to move them. Movements that cause a box to move more than one unit of the grid are also illegal. Whether or not a robot is pushing an object does not change the cost.
- Each movement is of equal cost.
- The goal is achieved when each box is located in a storage area on the grid.

Ideally, we will want our robots to organize everything before the supervisor arrives. This means that with each problem instance, we will want to work toward respecting a computation time constraint. We will start by exploring A* variations and then use the most appropriate one to provide some legal solution to each problem (i.e. a plan) within a time constraint. Better plans will be plans that are shorter, i.e. that require fewer operators to complete.

Your goal is to explore weighting schemes that affect A* and then to use one in order to implement anytime algorithms for this problem. These anytime algorithms will generate better solutions (i.e. shorter plans) the more computation time they are given.

Starter Code

The code for this assignment consists of several Python files, some of which you will need to read and understand in order to complete the assignment and some of which you can ignore. **You can download all the code and supporting as a Zipped file archive by clicking here** **(https://q.utoronto.ca/courses/219334/files/14503438/download?download_frd=1)** . In the archive, you will find the following files.

Files you'll edit andsubmit on Markus:

`solution.py`	Where all of your heuristics and anytime algorithms will reside. Note you must also submit a file called comparison.csv that you will generate.

Files you might want to look at (look but don't modify):

`search.py`	This contains default implementations of search algorithms discussed in class.
`sokoban.py`	This specifies the search to the Sokoban domain, specifically. Some sample problems are contained in this file as well.
`autograder.py`	This is an autograder for you to check your solutions as you develop them. The autograder can be run with the command: `python3 autograder.py`

Note: the autograder and Sokoban environment use python3. Note that on teach.cs the default is python2.7, so you must use the prefix python3.

See the autograder tutorial in Assignment 0 for more information about using the autograder.

An example of the application of the search code to the Water Jugs problem (in the asynchronous lectures) **can be found in this file.** **(https://q.utoronto.ca/courses/219334/files/14520556/download?download_frd=1)** Run $python3 WaterJugs.py for an illustration of the search code's use in that domain.

Details of Starter Code

The file search.py, which is available from the website, provides a generic search engine framework and code to perform several different search routines. This code will serve as a base for your Sokoban solver. A brief description of the functionality of search.py follows. The code itself is documented and worth reading. The file `search.py` contains:

- An object of class StateSpace represents a node in the state space of a generic search problem. The base class defines a fixed interface that is used by the SearchEngine class to perform search in that state space.
- For the Sokoban problem, we will define a concrete sub-class that inherits from StateSpace. This concrete sub-class will inherit some of the “utility” methods that are implemented in the base class.
- Each StateSpace object s has the following key attributes:
  - s.gval: the g value of that node, i.e., the cost of getting to that state.
  - s.parent: the parent StateSpace object of s, i.e., the StateSpace object that has s as a successor. This will be None if s is the initial state.
  - s.action: a string that contains that name of the action that was applied to s.parent to generates. Will be “START” if s is the initial state.
- An object of class SearchEngine se runs the search procedure. A SearchEngine object is initialized with a search strategy (‘depth first’, ‘breadth first’, ‘best first’, ‘a star’, or ‘custom’) and a cycle checking level (‘none’, ‘path’, or ‘full’).
- Note that SearchEngine depends on two auxiliary classes:
  - An object of class sNode (sn) which represents a node in the search space. Each object (sn) contains a StateSpace object and additional details: *hval*, i.e., the heuristic function value of that state and *gval*, i.e. the cost to arrive at that node from the initial state. An *fval_fn* and *weight* are tied to search nodes during the execution of a search, where applicable.
  - An object of class Open is used to represent the search frontier. The search frontier will be organized in the way that is appropriate for a given search strategy.

- When a SearchEngine's search strategy is set to 'custom', you will have to specify the way that f values of nodes are calculated; these values will structure the order of the nodes that are expanded during your search.
- Once a SearchEngine object has been instantiated, you can set up a specific search with: *init_search(initial_state, goal_fn, heur_fn, fval_fn)* and execute that search with *search(timebound, costbound)*.
  The arguments are as follows:
  - `initial_state`; this will be an object of type `StateSpace` and it is your start state.
  - `goal_fn(s)` is a function that returns True if a given state `s` is a goal state and False otherwise.
  - `heuristic_fn(s)` is a function that returns a heuristic value for the state `s`. This function will only be used if your search engine has been instantiated to be a heuristic search (e.g., best first).
  - `timebound` is a bound on the amount of time your code will execute the search. Once the run time exceeds the time bound, the search will stop; if no solution has been found, the search will return False.
  - `fval_fn(sNode,` *weight*`)` defines f-values for states. This function will only be used by your search engine if it has been instantiated to execute a custom search. Note that this function takes in an `sNode` and that an `sNode` contains not only a state but additional measures of the state (e.g., a gval).). The function also takes in a float *weight*. It will use the variables that are provided in order to arrive at an f-value calculation for the state contained in the `sNode`.
  - `costbound` is an optional parameter that is used to set boundaries on the cost of nodes that are explored. This `costbound` is defined as a list of three values. `costbound`[0] is used to prune states based on their g-values; any state with a g-value higher than `costbound`[0] will not be expanded. `costbound`[1] is used to prune states based on their h-values; any state with an hvalue higher than `costbound`[1] will not be expanded. Finally, `costbound`[2] is used to prune states based on their f-values; any state with an f-value higher than `costbound`[2] will not be expanded.
- The output of the search function will include both a solution path as well as a SearchStats object (if a solution is found). A SearchStats object (ss) details some interesting statistics that are related to a given search. Its attributes are as follows:
  - ss.states_expanded, which is a count of the number of states drawn from the Frontier during a search.
  - ss.states_generated, which is a count of the number of states generated by the successor function during a search.
  - ss.states_pruned_cycles, which is a count of the number of states pruned as a result of cycle checking.
  - ss.states_pruned_cost, which is a count of the number of states pruned as a result of enforcing cost boundaries during a search.

For this assignment we have provided sokoban.py, which specializes `StateSpace` for the Sokoban problem. You will therefore ***not*** need to encode representations of Sokoban states or the successor function for Sokoban! These have been provided to you so that you can focus on implementing good search heuristics, variations of A* and anytime algorithms.

The file sokoban`.py` contains:

- An object of class sokobanState, which is a StateSpace with these additional key attributes
  - `s.width`: the width of the Sokoban board
  - `s.height`: the height of the Sokoban board
  - `s.robots`: positions for each robot that is on the board. Each robot position is a tuple *(x, y)*, that denotes the robot's *x* and *y* position.
  - `s.boxes`: positions for each box that is on the board. Each box position is also an *(x, y)* tuple.
  - `s.storage`: positions for each storage bin that is on the board (also *(x, y)* tuples).
  - `s.obstacles`: positions for obstacles that are on the board. Obstacles, like robots and boxes, are also tuples of *(x, y)* coordinates.
- sokobanState also contains the following key functions:
  - `successors()`: This function generates a list of SokobanStates that are successors to a given SokobanState. Each state will be annotated by the action that was used to arrive at the SokobanState. These actions are (*r,d*) tuples wherein r denotes the index of the robot that moved *d* denotes the direction of movement of the robot.
  - `hashable_state()`: This is a function that calculates a unique index to represents a particular sokobanState. It is used to facilitate path and cycle checking
  - `print_state()`: This function prints a sokobanState to stdout.
- Note that sokobanState depends on one auxiliary class called `Direction`, which is used to define the directions that the robot can move and the effect of this movement.

Finally, note that `sokoban.py` contains a set of 20 initial states for Sokoban problems, which are stored in the tuple PROBLEMS. You can use these states to test your implementations.

The file `solution.py` contains the methods that need to be implemented.

The file `autograder.py` runs some tests on your code to give you an indication of how well your methods perform.

What to Submit

You will be using MarkUs to submit your assignment. You will submit:

1. Your modified `solution.py`
2. Your comparisons of A* variations, which you will place in a file called `comparison.csv`

Your Job

To complete this assignment you must modify `solution.py` to:

- Implement a Manhattan distance heuristic (heur_manhattan_distance(state)). This heuristic will be used to estimate how many moves a current state is from a goal state. The Manhattan distance between coordinates (x0,y0) and (x1,y1) is $| x0 - x1 | + | y0 - y1 |$. Your implementation should calculate the sum of Manhattan distances between each box that has yet to be stored and the storage point nearest to it. Ignore the positions of obstacles in your calculations and assume that many boxes can be stored at one location
- Implement a non-trivial heuristic for Sokoban that improves on the Manhattan distance heuristic (heur_alternate(state)). Place a description of your heuristic in the comments of your code.
- Implement three alternative f-value functions (fval_function(sNode, weight), fval_function_XUP(sN,weight) and fval_function_XDP(sN,weight)) to create variations of A* (Weighted A*, Weighted A* (XUP) and Weighted A* (XDP)). Then, compare and contrast the performance of each variation in the context of a custom A* search and log your results in the file *comparison.csv*. Note that each variation will require you to instantiate a SearchEngine object with a 'custom' search strategy and initialize this object with a specialized f-value function. More details are below.
- Implement Anytime Weighted A* (anytime_weighted_astar(initial_state, heur_fn, weight, timebound)). This will be an iterative form of Weighted A* search that can run within a fixed time bound. You may use whatever variant of A* works best for the job. More details are below.
- Implement Anytime Greedy Best-First Search (anytime_gbfs(initial_state, heur_fn, timebound)). This is another iterative search that can run within a fixed time bound. More details are below.

Note that when we are testing your code, we will limit each run of your algorithm on teach.cs to 5 seconds. Instances that are not solved within this limit will provide an interesting evaluation metric: failure rate.

Weighted A* and Variations

Instead of A*s regular node-valuation formula f(node) = g(node) + h(node), Weighted A* introduces a weighted formula:

```
f(node) = g(node) + w * h(node)
```

where g(node) is the cost of the path to node, h(node) the estimated cost of getting from node to the goal, and $w \geq 1$ is a bias towards states that are closer to the goal. Theoretically, the smaller w is, the better the first solution found will be (i.e., the closer to the optimal solution it will be ... why??). However, different values of w require different computation times.

Start by implementing Weighted A* using the f-value function above; this will require you to instantiate a 'custom' Search Engine. When you are passing in an f-value function to initialize the search engine for this problem, you will need to specify the weight. You can do this by wrapping the fval_function(sN,weight) you have written in an anonymous function, i.e.,

```
wrapped fval function = (lambdasN : fval function(sN,weight))
```

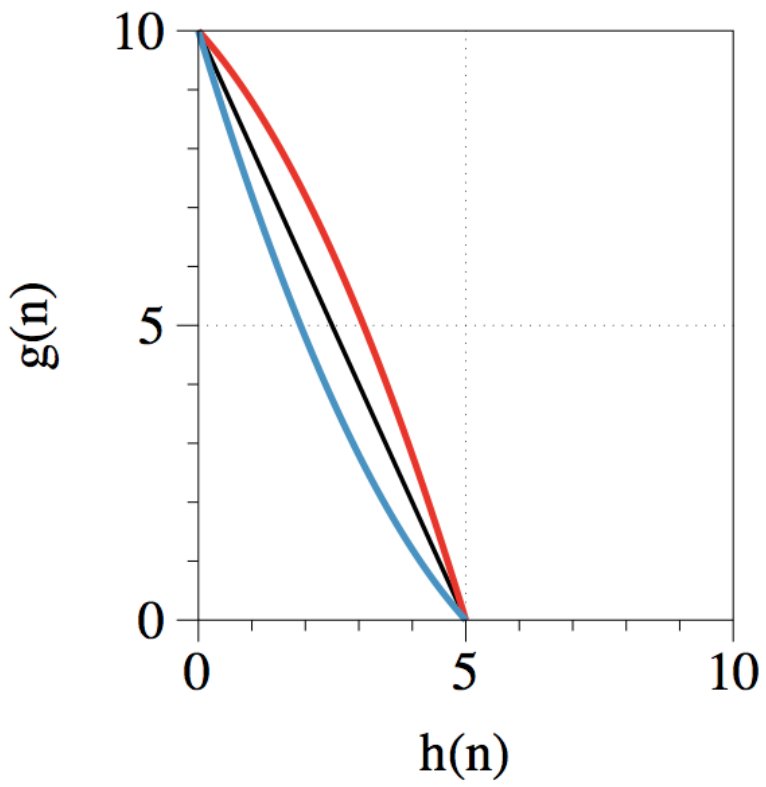

Figure 1. This plot groups g and h-values of paths with the same priority in a Weighted A* search (where the weight is 2). Paths with a priority (or f-value) of 10 would be plotted along the black line for a regular Weighted A* search. By contrast, Weighted A* (XDP) would give the same priority to paths plotted on the red line and Weighted A* (XUP) would give the same priority to paths plotted on the blue line.

At any point during a run of Weighted A*, there may be several paths in the Frontier with the *same* priority (or f-value). Some may have high g-values while others may have high h-values. If we were to plot the g and h-values of all of paths with the same priority in the Frontier, they would lie on a line (as illustrated in black in Figure 1). Note that the sub-optimality these paths admit does not depend on the search's distance from the start state or its estimated proximity to the goal. We could change this, however! We could instead admit more sub-optimality based on how close or far away we think we are from the goal.

This can be done if we create *alternative* f-value functions. There are many, many, many we could try; you can click **here** **(https://www.movingai.com/SAS/SUB/)** for different variations. In this homework, we will explore just a few. In Figure 1, a few weighting schemes are explored. The red line represents g and h-values of paths with the same priority when using a scheme that allows for **more** sub-optimality early in a path's exploration. This f-value function is called the **convex downward parabola (XDP) priority function,** and it is defined as follows:

```
f(node) = (1/(2*w))*(g(node)+(2*w-1)*h(node)+sqrt((g(node)-h(node))^2 +4*w*g(node)*h(node))
```

By contrast, the blue line in Figure 1 represents g and h-values of paths with the same priority when using a scheme that allows **less** sub-optimality early in a path's exploration. This f-value function is called the **convex upward parabola (XUP) priority function,** and it is defined as follows:

```
f(node) = (1/(2*w))*(g(node) + h(node) + sqrt((g(node)+h(node))^2 + 4*w*(w-1)*h(node)^2 ))
```

Your task is to implement the fval_function for Weighted A* as well as these variants. More specifically, encode fval_function(sN,weight), fval_function_XUP(sN,weight) and fval_function_XDP(sN,weight). Then , use your f-value function, the default code base **and your heur_manhattan_distance function** to compare the following variations of A*: Standard A*, Weighted A*, Weighted A* (XUP) and Weighted A* (XDP). Use each variation to solve the first three test problems with following weights: *2, 3, 4 and 5.* Generate a CSV file (*comparison.csv*) to hold your results; this will contain one line for each variant x problem combination. Make sure each line in your CSV respects the following format:

A,B,C,D,E,F

where

A is the number of the problem being solved (0,1 or 2)

B is the A* variation being used (1 = Standard A*,2 = Weighted A*,3 = Weighted A* XUP and 4 = Weighted A* XDP)

C is the weight being used (note however that for standard A*, this weight will be ignored)

D is the number of paths extracted from the Frontier (or expanded) during the search

E is the number of paths generated by the successor function during the search

F is the overall solution cost

Anytime Weighted A* Search

Even with an admissible heuristic, the first solution found by any of the Weighted A* variations may not be optimal when w is anything larger than 1. We can therefore keep searching after we have found a solution in order to try and find a better one. Anytime Weighted A* continues to search until either there are no nodes left to expand (and our best solution is the optimal one) or it runs out of time. Since we have found a path to the goal after the first search iteration, we can introduce a cost bound for pruning in future iterations: if node has a g(node)+h(node) value greater than the best path to the goal found so far, we can prune it.

Implement an anytime version of A* search using the following function stub: *anytime_weighted_astar(initial_state, heur_fn, weight, timebound).* This should be an iterative search that makes use one of your weighted A* variations. Your function should initialize a custom search engine with an f-value function that includes a weight. When a solution is found, remember it and, if time allows, iterate upon it. Change your weight at each iteration and enforce a cost boundary so that you will move toward more optimal solutions at each iteration.

Anytime Greedy Best-First Search

We can also create an anytime version of greedy best first search. This will not depend on a weight! As you remember from class, greedy best-first search expands nodes with lowest h(node) first. The solution found by a greedy algorithm may not be optimal. Anytime greedy-best first search (which is called *anytime_gbfs* in the code) continues searching after a greedy solution is found in order to improve solution quality. Since we have found a path to the goal after the first iteration, we can introduce a cost bound for pruning in subsequent iterations: if a node has g(node) greater than the best path to the goal found so far, we can prune it. The algorithm returns either when we have expanded all non-pruned nodes, in which case the best solution found by the algorithm is the optimal solution, or when it runs out of time. *We will prune based on the g-value of the node only* because greedy best-first search is not necessarily run with an admissible heuristic.

Record the time when anytime searches are called with os.times()[0]. Each time you call search, you should update the time bound with the remaining allowed time.

Marking Criteria

```
(1) Test of Manhattan distance based heuristic (10% max)

(2) Test of the heuristic function in the context of best first search: 5 seconds (based on # solve
d; 20% max)
– 5% for solving more than 2 problems
– >= 10% awarded based on # solved relative to Manhattan distance
– >= 15% awarded based on # solved relative to “better” benchmark
- additional points (up to 5) pro-rated (full marks if entire test set is solved)
- points here are based on # solved, not on solution length

(3) Comparison of weighted A* strategies (20%)
- 5% for correct implementation of each alternative f-value function (15% total)
- 5% for correct search statistics in comparison.csv

(4) Test of the iterative weighted a star function: 5 seconds (based on # solved and length; %25 ma
x)
– 5% for solving more than 2 problems
– >= 10% awarded based on # solved relative to Manhattan distance
– >= 20% awarded based on # solved relative to “better” benchmark
- additional points (up to 5) pro-rated (full marks if entire test set is solved)
- points here are based on both # solved and length of solution

(5) Test of anytime best first search: 5 seconds (based on # solved; 25% max)
– 5% for solving more than 2 problems
– >= 10% awarded based on # solved relative to Manhattan distance
– >= 20% awarded based on # solved relative to “better” benchmark
- additional points (up to 5) pro-rated (full marks if entire test set is solved)
- points here are based on # solved
- deductions if any anytime best first search solution is longer than a regular best first search so
lution
```

GOOD LUCK!