**Project 1: Heuristic Search**

# Deadlines

- Parts 1 and 3 (individual) due [one week after release]
- Parts 1-4 (team) due [two weeks after release]

# 0. Fair Warning

In this project you will apply heuristic search to a genuinely challenging problem. Developing a good heuristic will take creativity, experimentation, and careful analysis. There are likely to be times when you are unsure of how to proceed; you need to leave yourself time to get stuck and then to have a clever idea. Also, this project has a writing component and the quality of your writing will be a significant factor in your grade. You will need time to write, revise, and edit.

In other words **you should start working on this assignment immediately**. Waiting until the last minute (or even the last day!) is not going to work out well.

# 1. Sliding Puzzle Heuristics (10 pts total)

(Individual and team scores will be averaged to find your final score)

We've discussed the 8-puzzle in class. In this part of the project you will explore various heuristics for the 8-puzzle (and sliding puzzles more generally).

If you run

```
python3 slidingpuzzle.py
```

an 8-puzzle will be generated and solved using A* search using several different heuristics. Well…actually none of them have been implemented yet. That's your job! Right now they are all equivalent to the "null heuristic" that always returns 0. Recall that A* using the null heuristic is equivalent to uniform cost search (UCS). Furthermore, when all moves have the same cost, UCS is equivalent to breadth-first search (BFS). So what you see right now is the result of applying BFS to the 8-puzzle.

There are lots of options you can give to the program. Run

```
python3 slidingpuzzle.py –h
```

to see them all. You can give an image file to display the puzzle (e.g. images/octopus.gif), change the dimensions of the puzzle, and even play the puzzle yourself. Most importantly for our purposes, you can use –d to specify how deep the solution is (how many steps it takes to solve the puzzle) and –t to randomly generate multiple puzzles and get average results for the heuristics you will experiment with.

Your job is to implement the following heuristics. In each heuristic class you will implement the `eval` method, which takes a state and returns the heuristic value (a

number that estimates the cost to get from this state to a goal state). The first thing the `eval` method does is to load the given state into `self.problem`, which is a `SlidingPuzzle` object (defined in *slidingpuzzle.py*). You can then use `self.problem` to examine the state.

The methods of `SlidingPuzzle` you may find useful are:

- `getDim()` – returns the number of rows and number of columns
- `isSolved()` – returns `True` if the puzzle is solved
- `getTile(row, col)` – returns the index of the tile at the given position.
- `getPos()` – returns the position of the blank tile as a (row, column) tuple.
- `getBoard()` – returns copy of the board as a list of lists.

You may assume that tile 0 is the blank tile, and that the solution configuration (for the 8-puzzle) looks like this:

```
0 1 2
3 4 5
6 7 8
```

**# Misplaced (2 pts)**

The "# Misplaced" heuristic gives the number of (non-blank) tiles that are out of place. As discussed in class, this heuristic is the optimal solution to a relaxation of the sliding puzzle in which each step can move a tile to any location on the board, regardless of whether that location is occupied.

Since any tile that is out of place must move at least one position, the '# Misplaced' heuristic is admissible. We can also show that it is consistent. Let $s$ be a state, $a$ be an action, and $s'$ be the state reached by taking action $a$ in state $s$. Let $h$ be the heuristic function. The cost of the move is always 1. Note that only one tile has changed position from $s$ to $s'$. If that tile was in place and moved out of place or was out of place and remains out of place, $h(s') \geq h(s)$ so certainly $h(s') \geq h(s) - 1$. If that tile was out of place in $s$ and moves into its correct position, then $h(s') = h(s) - 1$. Thus, $h(s') \geq h(s) - 1$. Since the cost of every move is 1, we see that $c(s, a, s') + h(s') = 1 + h(s') \geq h(s)$. Therefore this heuristic is consistent.

You should implement this heuristic in the `NumMisplacedHeuristic` class in *heuristics.py*.

**Manhattan (3 pts)**

The Manhattan distance between two positions $(a, b)$ and $(c, d)$ is $|a - c| + |b - d|$. It gets its name from the fact that Manhattan is laid out in a grid, so to get anywhere you can't travel in a straight line; you must travel a certain number of blocks across town and then a certain number of blocks up- or down-town. The "Manhattan" heuristic gives the sum of the Manhattan distances between each (non-blank) tile's

position and its position in the goal state. As discussed in class, this heuristic is the optimal solution to a relaxation of the sliding puzzle in which each step moves a tile one square, regardless of whether that square is already occupied.

Note that in the real solution, we must at the very least move each tile into its correct position, and can only do so one square at a time. The Manhattan distance is the minimum possible number of steps it takes to move a tile into position, one square at a time. So this heuristic is admissible. We can also show that it is consistent using a similar argument to the one we used above. Let $s$ be a state, $a$ be an action, and $s'$ be the state reached by taking action $a$ in state $s$. Let $h$ be the heuristic function. Note that only one tile has changed position by one square from $s$ to $s'$. The only way $h(s')$ can be less than $h(s)$ is if that tile moved closer to its intended position. In that case $h(s') = h(s) - 1$. Thus, $h(s') \geq h(s) - 1$. In every other case, $h(s') \geq h(s)$. Since the cost of every move is 1, we see that $c(s, a, s') + h(s') = 1 + h(s') \geq h(s)$. Therefore this heuristic is consistent.

You should implement this heuristic in the `ManhattanHeuristic` class.

### Gaschnig's Heuristic (4 pts)

*Gaschnig's Heuristic* (Gaschnig, 1979) is the optimal solution to a relaxation of the sliding puzzle in which at each step any tile may be moved into the blank space (leaving a blank behind it), not just adjacent tiles. The solution to this relaxed problem can be computed via a greedy algorithm:

1. $n = 0$
2. **while** not in goal state
3.     **if** blank is out of place
4.         move correct tile to the blank position
5.     **else** the blank is in the correct place
6.         move an arbitrary misplaced tile to the blank position
7.     $n = n + 1$
8. **return** $n$

Note that in every step of the real solution, a tile is moved into the blank space. This heuristic gives the minimum number of steps required if in every step a tile is moved into a blank space (with no constraints on which tile that is). Thus, the optimal solution to the full problem must take at least this many steps to reach the goal. Thus this heuristic is admissible. We can also show that it is consistent. Let $s$ be a state, $a$ be an action, and $s'$ be the state reached by taking action $a$ in state $s$. Let $h$ be the heuristic function. Note that only one tile has changed position from $s$ to $s'$. There are two ways that $h(s')$ can be less than $h(s)$. If that tile was out of place in $s$ and moves into its correct position, then note that the greedy algorithm in $s$ would have begun by performing this exact move (moving the correct misplaced tile into the blank space). After that one step, the greedy algorithm is processing state $s'$, so the algorithm performs the same steps in both states. Therefore, the algorithm takes one more step in $s$ than in $s'$, so $h(s') = h(s) - 1$. Similarly, if the tile was out of

place in $s$ and moves to the position where the blank should be, then that implies that the blank was correctly placed in $s$. As such, once again, this is the first step the algorithm would have taken and it will otherwise proceed with the same steps in both states. So $h(s') = h(s) - 1$. Thus, $h(s') \geq h(s) - 1$. In every other case $h(s') \geq h(s)$. Since the cost of every move is 1, we see that $c(s, a, s') + h(s') = 1 + h(s') \geq h(s)$. Therefore this heuristic is consistent.

You should implement this heuristic in the `GaschnigsHeuristic` class. You will probably want to make use of `getBoard` so you can do your own swaps!

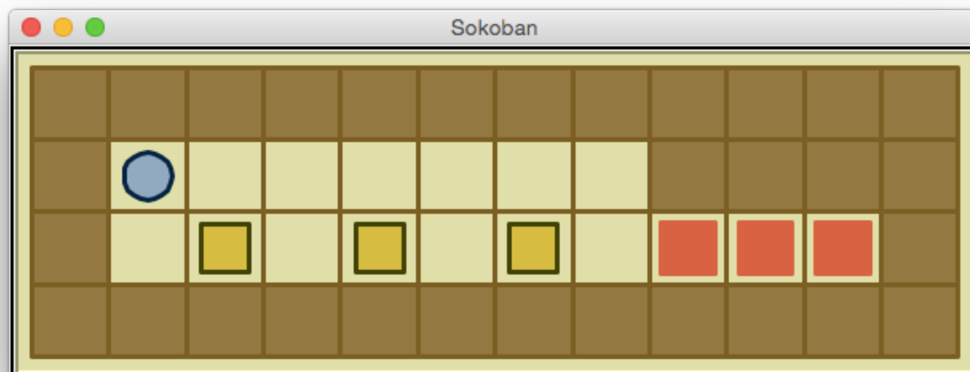
**Combo (1 pt)**

Recall from class that a sound way to combine two heuristics is to return the maximum of the two. In `ComboHeuristic` you should implement the heuristic that returns the maximum of the "Manhattan" heuristic and Gaschnig's heuristic. *Please* do not re-implement the heuristic functions! In the constructor you can create instance variables for the two heuristics, and then use them in `eval`.

Since this heuristic is the max of two consistent heuristics, it is also consistent.

## 2. Sokoban

Sokoban is a puzzle created by Hiroyuki Imabayashi in 1981. The name is Japanese and translates to "warehouse keeper" in English. A Sokoban puzzle is pictured below. In the game, there are boxes (yellow squares), storage locations (red squares), and a robot (blue circle). The rules are quite simple – the robot can move in any of the four cardinal directions (north, south, east, west). The dark squares are impassable walls. When the robot moves, it can push a single box, unless there is something in the way (a wall or another box). Boxes cannot be pulled. The goal is to get all the boxes onto storage locations.



Despite its seeming simplicity, Sokoban can be very difficult to solve. Most Sokoban puzzles still remain out of reach for state of the art AI methods. In this project we'll focus on very simple examples so you should be able to make progress.

We can formulate Sokoban as a search problem. The state consists of the positions of the robot and the boxes. The goal state is the one where all the boxes are stored. There are four actions, one for each direction. There are many ways to measure the quality of a solution. For our purposes, the cost of a non-pushing move is 1, the cost of pushing a box is 2, and the cost of a failed move (e.g. moving into a wall) is 3.

You can get a feel for the puzzle by running

```
$ python3 sokoban.py –p examples/tricky.txt
```

The –p option makes the puzzle playable with the arrow keys. The file *tricky.txt* specifies the pictured Sokoban puzzle. Other simple puzzles are included as well.

## 3. Sokoban Heuristic (10 pts)

(Individual and team scores will be averaged to find your final score)

If you run the program *without* the –p option, it will use A* graph search to find an optimal solution to the given puzzle. When it is done, it will animate the solution and also print out some useful statistics: the number of seconds the search process took, the number of nodes it expanded in the tree, and the total cost of the solution.

Right now the program is using the null heuristic (the heuristic that always returns 0), so it is effectively performing uniform cost search. Your job is to improve performance by developing a better, *consistent* heuristic.

In *heuristics.py* you will find a class called `SokobanHeuristic`, structured just like the sliding puzzle heuristics. You should implement your heuristic in this class. The most important function is `eval`, which takes a state and gives its heuristic value. Note that the first thing the function does is load the state into `self.problem` (a `SokobanPuzzle` object, see *sokoban.py*). That allows you to use `problem` to examine many aspects of the state. The relevant methods of `SokobanPuzzle` are:

- `getDim()` – returns the number of rows and number of columns
- `isSolved()` – returns `True` if the puzzle is solved
- `getItem(row, column)` – returns a character at the given position representing the contents of the square. The characters are:
    - Blank space: `'.'`
    - Wall: `'#'`
    - Empty goal: `'*'`
    - Player on blank: `'+'`
    - Player on goal: '='
    - Box on blank: `'O'`
    - Box on goal: `'o'`

- `getGoals()` – returns a sorted list of the positions of the goals, represented as (row, column) tuples.
- `getBoxes()` – returns a sorted list of the boxes, represented as (row, column, isOnGoal) tuples (the third item is `True` if the box is on a goal).
- `getPos()` – returns the player position as a (row, column) tuple.

Notice also that the heuristic is a class, not just a function. This means that, if you find it helpful, you may pre-compute some quantities in the constructor and store them as instance variables for use in the `eval` function. This might speed up your search if you find yourself calculating the same quantities over and over again. Any pre-processing you do *must* have polynomial worst-case complexity in terms of the number of boxes. In particular note that solving the problem has exponential worst-case complexity, so you *can't* just solve the problem in the constructor!

You may find the provided puzzles helpful to evaluate your heuristic and generate ideas for how you might improve it. You can also make your own examples. Ultimately the performance of your heuristic will be measured on the hardest provided puzzle, *benchmark.txt.* The number of nodes A* expands using your heuristic determines your maximum score:

| Nodes expanded: | Max points: |
|---|---|
| 100,001 – 193,390 | 2 |
| 60,001 – 100,00 | 4 |
| 50,001 – 60,000 | 6 |
| 30,001 – 50,000 | 8 |
| 20,001 – 30,000 (hard) | 10 |
| 8,001 – 20,000 (very hard) | 10+1 bonus |
| ≤ 8,000 (very very hard) | 10+2 bonus |

You will receive fewer than the maximum number of points if:

- your heuristic is not consistent, or not admissible,
- your heuristic is specific to *benchmark.txt* (it should work in other examples),
- you do too much pre-processing (must be polynomial time!),
- your heuristic algorithm/implementation is notably inefficient, or
- your code is notably buggy or poorly written.

Notes and hints:

- Many people have worked on the Sokoban problem. If you take inspiration from existing literature on the subject, make sure you cite your sources!
- If you want to use a well-known algorithm and you find a Python module that can do it, you may use it. However, any non-standard module you import must be included with your submission (do not make me install new packages!). Also make sure you cite your sources.

- If you get a solution with different total cost with your heuristic than the null heuristic, then your heuristic is not consistent, maybe not even admissible. Run the program with the –n option to use the null heuristic instead of yours.
- That said, sometimes inadmissible heuristics yield optimal solutions. You have to analyze your heuristic yourself to be sure!
- Remember that a good way to get an admissible heuristic is to find the optimal solution to a relaxed problem. Consider how you can relax Sokoban to create a problem with a lower cost and easier-to-compute solution.
- Admissible heuristics are often also consistent, especially when derived from a problem relaxation. Start by focusing on admissibility. Once you have one you like, check whether it is consistent and tweak it if necessary.
- Sometimes you can detect that a heuristic is inconsistent by looking at the quantity *total_cost+heuristic* for the states along the solution path. If that ever *decreases* from one state to the next, your heuristic is not consistent.
- If you have an admissible heuristic and want to find a better one, you need to *increase* the heuristic values (i.e. make them more accurate), at least in some states. Think about aspects of the problem you ignored/relaxed away, and try to at least partially take them into account in your heuristic.
- One of the challenging aspects of Sokoban is that there are many states from which no solution can be reached. Can your heuristic take this into account?

# 4. Report (20 pts)

(To be completed as a team only)

In *search.pdf* include the following:

**Sliding Puzzle Heuristic Analysis (10 pts)**

Evaluate the slide puzzle heuristics by running *solveslidingpuzzle.py* on the 8-puzzle with 10 independent trials (–t 10) and with solution depths at 5, 10, 15, and 20 (–d 5, –d 10, –d 15, and –d 20, respectively). Report the average results in two clearly formatted and labeled tables: one for time and one for number of nodes expanded. Then write an email to respond to the following, referring to your result tables:

**To:** you@acollege.edu
**From:** astudent@acollege.edu
**Subject:** Sliding puzzle heuristics

Hey,
I've been working on a sliding puzzle solver but I don't know which heuristic to use. # Misplaced is performing way worse than Manhattan or Gaschnig's. I'm worried I might have implemented it wrong. Is it supposed to be this bad? Why is it so much worse? Also, even though Gaschnig's is definitely worse than Manhattan, Combo is *better* than Manhattan. How can combining a good heuristic with a bad heuristic make a better

heuristic?? I need to pick the heuristic that will solve puzzles the fastest. Which one should I use??? HALP!
-A

## Sokoban Heuristic Description/Analysis (10 pts)

Write a brief description of your heuristic and a clear, convincing, and precise argument that it is consistent. Imagine that your reader is a fellow classmate who has been working on the same problem. Perhaps your heuristic turned out to work better than theirs, and they would like to compare against it in their final project. After reading your description, your classmate should know enough to re-implement your heuristic, *and* be thoroughly convinced that it is consistent. Consider the descriptions of the sliding puzzle heuristics as examples.

If you know that your heuristic is *not* consistent, explain why not and argue that it is admissible. If you know that it is not admissible, explain why not.