

Part 1: Search Project (20 Points)

A lone adventurer, the brave SearchAgent, forges eastward through the mountains with nothing but a map and her algorithmic savvy. Only by a thorough search of her map will she reach her destination without wasting steps.

Introduction

While future projects will tackle realistic present day AI applications, this one focuses on two classic domains in the artificial intelligence community: mazes and the 8-puzzle. After some brief exercises to help familiarize you with Python and the provided code base, the project will consist of building general search algorithms and then applying them to multiple domains. If you are unfamiliar with Python, you should have worked through a Python Tutorial on your own before this project.

The code for this project consists of several Python files which you will need to understand in order to complete the assignment. You can download the code and supporting files as a [hw2.tar.gz archive](http://faculty.tamu-commerce.edu/dharter/tamu/classes/2007fall/csci538/labs/hw2.tar.gz) (<http://faculty.tamu-commerce.edu/dharter/tamu/classes/2007fall/csci538/labs/hw2.tar.gz>).

[mazeworld.py](#) Code for constructing, editing, displaying and searching mazes (Part 1.A).

[search.py](#) Generic search algorithm code (Part 1.A). All your search agents will implement the `solve` method for the empty `SearchAgent` classes in this file. A summary of the search API can be found [here](#).

[util.py](#) Useful data structures for implementing search algorithms (Part 1.A).

[eightpuzzle.py](#) Code for constructing, editing, displaying and searching 8-puzzles (Part 1.B). You will implement the `EightPuzzleSearchProblem` class and two heuristic functions.

What to submit: You will edit portions of `search.py` and `eightpuzzle.py` during the assignment. You should submit them along with *a text or pdf file containing transcripts of your code working and answers to the discussion questions*.

Part 1.A: Maze World

For the first part of your assignment, you'll be writing search agents which solve the maze search problem. The relevant code is in [mazeworld.py](#). A maze is represented as a 2-d grid with a start position 'S' and exit 'E'. There are also obstacles '#' that are impassable and water '~' which is passable but expensive (5 times the step cost of moving through a blank square). For example:

```
----  
|#   |  
|~S E|  
----
```

To create this simple maze, you can type:

```
>>> import mazeworld
>>> simpleMaze = mazeworld.Maze(['#', ' ', ' ', ' '], ['~', 'S', ' ', 'E'])
>>> print simpleMaze
```

The simplest agent in this domain simply moves right until she finds the exit or hits an obstacle and gives up. This simple agent, `SimpleMazeAgent`, is provided in `mazeworld.py`. This agent can find a path through our simple maze.

```
>>> simpleMazeAgent = mazeworld.SimpleMazeAgent()
>>> mazeworld.testAgentOnMaze(simpleMazeAgent, simpleMaze)
Solution cost: 2.0
Number of nodes expanded: 2
Number of unique nodes expanded: 2
```

You can also see a visualization of the maze solution by passing an optional `verbose = True` to `testAgentOnMaze`.

Question 1 (1 point) Write a function that creates the simple maze above and then *edits it* such that the simple right-moving agent cannot reach the exit. Now, write a maze agent that can solve your new maze (hard-coding a sequence of moves is fine). *You need only submit a transcript of the output for this question.*

Now that you've gotten your feet wet, it's time to write full-fledged generic search agents!

Question 2 (3 points) Implement the depth-first search (DFS) algorithm in the `DepthFirstSearchAgent` class in `search.py`. Make your algorithm complete, for example by checking for cycles in the search path (Section 3.5). Test your code on the maze in [maze1.txt](#) as follows:

```
>>> maze1 = mazeworld.readMazeFromFile('maze1.txt')
>>> import search
>>> agent = search.DepthFirstSearchAgent()
>>> mazeworld.testAgentOnMaze(agent, maze1, verbose=True)
```

What is the cost of the solution found by your depth first search agent? Is this the lowest cost solution? If not, what is depth-first search doing wrong?

Question 3 (3 points) Implement the breadth-first search (BFS) algorithm in the `BreadthFirstSearchAgent` class in `search.py`. This time, write a graph search algorithm that avoids expanding any already visited states (section 3.5). Test your code on the maze in [maze1.txt](#) the same way you did for depth-firstsearch (except of course instantiating your BFS agent instead of your DFS agent). Does your BFS search agent find the best solution. Now test your BFS agent on [maze2.txt](#). Does your BFS agent find the best solution for this maze? If not, what is BFS doing wrong, or failing to do? Does your BFS agent expand a substantially different number of nodes than your DFS agent?

Question 4 (4 points) Implement the uniform-cost search algorithm in the `UniformCostSearchAgent` class in `search.py`. Test your agent on [maze2.txt](#). Does it find the best solution? Now test your agent on [maze3.txt](#). How many nodes does it expand in order to get this solution? Looking at the output, does it seem like the agent is expanding unnecessary nodes? Which nodes are wasted exploration?

Question 5 (2 points) Implement the A* search agent in the empty class `ASearchAgent` in `search.py`. You will need to pass a heuristic function into `ASearchAgent` upon construction. The heuristic function should take two arguments: the current state and the search problem. Use the `manhattanDistance` heuristic function provided in `mazeworld.py`. Now, test your A* agent on [maze3.txt](#). How many nodes does it expand compared to the uniform-cost search agent you wrote for question 4? Qualitatively, what is different about the regions explored by the uniform-cost and A* search agents?

Your agent is on a roll! Try [maze 15x15.txt](#), [maze 25x25.txt](#) and [maze 35x35.txt](#) for more fun.

Part 1.B: 8-Puzzle

Upon reaching her goal, SearchAgent happens upon a great computational challenge: the 8-puzzle. Fortunately, her searching abilities make short work of this NP-complete problem.

In this part of the project, you will use the search agents you wrote in part A on a new domain: the 8-puzzle. The 8-puzzle is another straightforward application of search techniques. The 8-puzzle class and methods to manipulate it are provided to you in [eightpuzzle.py](#). However, the puzzle must be formulated as a search problem in order for your search agent to solve it. Solving the 8-puzzle should not require any changes to your search agent.

First, some quick notes on the 8-puzzle implementation. Each instance of the `EightPuzzleState` class represents a particular configuration of the puzzle. Once an instance is created, it should not be changed (`EightPuzzleState` is designed to be an immutable type). Instead, the result function creates a new instance that represents a different configuration: the result of applying the provided move to the current configuration.

The intention of this design is to let instances of the `EightPuzzleState` class serve as states for the search algorithms. Thus, you need not create your own state representation of the 8-puzzle.

Question 6 (3 points) Fill in the missing elements of `EightPuzzleSearchProblem` so that the breadth-first search agent can find a solution. Then, test your agent interactively as follows:

```
>>> import eightpuzzle
>>> import search
>>> puzzle1 = eightpuzzle.loadEightPuzzle(1)
>>> puzzleProblem1 = eightpuzzle.EightPuzzleSearchProblem(puzzle1)
>>> bfs = search.BreadthFirstSearchAgent()
>>> bfs.solve(puzzleProblem1)
```

Will the BFS agent always find the optimal (fewest moves) solution to an 8-puzzle?

Question 7 (4 points) Write the two heuristic functions that appear in the textbook for this problem: misplaced tiles and distance (section 4.2). Compare the performance of A* using each of these heuristics on the provided puzzles.

Extra Credit (2 points) A third heuristic, *Gaschnig's heuristic*, is derived from the problem relaxation that a tile can move from a square A to a square B if B is blank (p. 108). Specify this heuristic and find an efficient way to compute it. Implement the heuristic. How does it compare to the two heuristic functions in question 8 (in terms of A* performance)? Can you quickly write a heuristic function that is always at least as good as the and Gaschnig's heuristics?

Part 2: Constraint Satisfaction Problems (20 points)

Question 2.1 (6 points). Consider the following CSP:

Variables: $\{A, B, C, D\}$

Domain: $\{1, 2, 3\}$

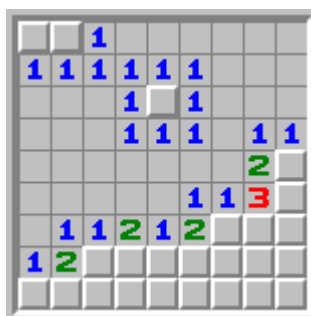
Binary Constraints: $A \neq B, A \neq C, B > C, B < D$

- Draw a constraint graph for this problem and write out the implicit constraints as explicit sets of legal pairs.
- Assuming an alphabetical ordering on both variables and values, show the assignments considered by each step of backtracking DFS with forward checking. After each assignment, indicate the remaining domains of all unassigned variables. E.g., completely naive DFS would search $A=1, A=1 B=1, A=1 B=1 C=1, A=1 B=1 C=1 D=1, A=1 B=1 C=1 D=2, A=1 B=1 C=1 D=3$, then pop back up to $A=1 B=1 C=2$, and so on, but never remove any values from unassigned domains.
- Show the assignments and domains, again with backtracking DFS and forward checking, but now using the MRV and LCV orderings. (Break MRV ties by the degree heuristic, then alphabetically; break LCV ties numerically, smaller values first.) Is this any faster than part (b)? Why or why not?
- Show the assignments and domains, again with backtracking DFS and forward checking, using the original alphabetical ordering, but now using cutset conditioning, with C as a cutset. Is there a better cutset?
- Show the assignments and domains with arc consistency (i.e. AC-3) run as preprocessing at the start and after each assignment.

Question 2.2 (6 points). In this problem, you will prove that every general CSP (constraints on arbitrary subsets of variables) can be transformed into an equivalent binary CSP (constraints only on pairs of variables). You should assume that variables have finite domains. See p. 159 of the textbook for another wording of this problem (and a hint).

- First, show how unary constraints can be eliminated by altering the domains of variables.
- Show that any ternary constraint can be transformed into three binary constraints by introducing a new auxiliary variable.
- Finally, show how a similar approach can be used to transform any n-ary constraint into a set of binary constraints.
- For n-ary constraints, how many auxiliary variables are introduced? What are their maximum domain sizes? How many binary constraints (arcs in the constraint graph) are required?

Question 2.3 (8 points). In the game of Minesweeper, we would like to determine which squares on a grid contain mines. At any given time, we have a partially revealed board, like the one below:



Squares are either marked as revealed (and therefore clear) or hidden (possibly a mine, or possibly clear). All revealed squares show the number of adjacent squares, including diagonals, which have mines (in most programs, zeros are displayed as blanks). At each step, we select a hidden position to reveal. We win if we reveal all clear squares without revealing a mine. We can use CSPs to gather information about unknown squares for a particular state. You can try a game [here](http://gameswizard.com/j_jvmine.html). (http://gameswizard.com/j_jvmine.html)

- Formulate a CSP where a solution is an assignment of the hidden squares to {clear, mine} which is compatible with the observed adjacent mine counts. Assume the number of remaining mines is unknown.
- Sometimes we have to guess. Show a small, simple board configuration for which no hidden square is guaranteed to be clear, and show the set of solutions to the CSP.
- Sometimes we can reason that a certain hidden position is guaranteed to be clear. Show a small, simple configuration which has more than one solution, but where all solutions agree that some currently hidden position is clear. Show those solutions.
- Why will a standard CSP solver not directly tell us where to move next?
- How can we augment the basic DFS-based backtracking constraint solver to tell us whether or not a square is guaranteed to be safe?
- Can we still use forward checking with the augmented DFS-based backtracking constraint solver from part e) ? Why or why not? What about arc consistency?
- [Extra Credit: 2 point] Show a method for correctly deciding which move is safest when no move is perfectly safe.