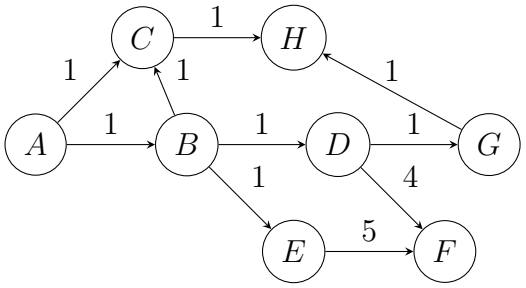
The purpose of this problem set is to gain experience with state-space search methods. You can complete the exercises by either directly marking up this pdf, or by printing, completing, and scanning as a pdf. In the engineering design section you will be formulating and solving three problems using modules from the AIAMA text's website (included in the starter code): heuristic search and the 8-puzzle, a two-player game, and a constraint satisfaction problem. This will require writing three python programs.

You can complete the exercises by either directly marking up this pdf, or by printing, completing, and scanning as a pdf. You should complete the Engineering Design Problems by writing the python code as instructed. The resulting pdf and python files should be uploaded to Canvas via the assignment tab by the due date and time.

Exercises

1. Consider the following graph, with initial state A and goal F, and the heuristic function h.



node, n	h(n)
A	8
B	7
C	3
D	2
E	3
F	0
G	30
H	25

Fill in the order nodes are goal-tested and the contents of the frontier and explored list at each step using the following algorithms (see next page). For nodes in the frontier indicate the f value in parenthesis after the node label, e.g. A(8). Assume nodes are expanded and any ties are broken using alphabetical ordering of the node labels.

(a) (5 points) greedy best-first search

current node	frontier	explored

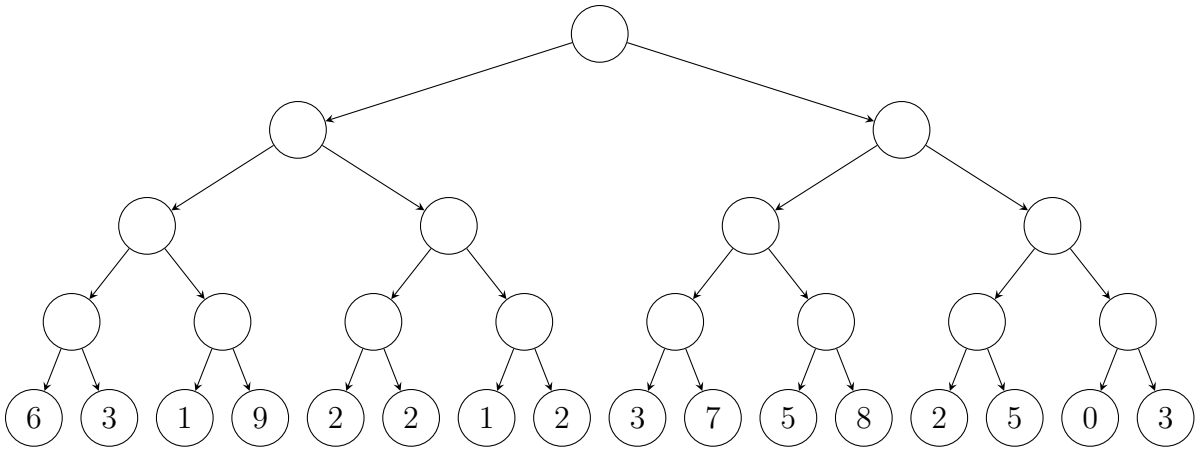
(b) (5 points) A* search

current node	frontier	explored

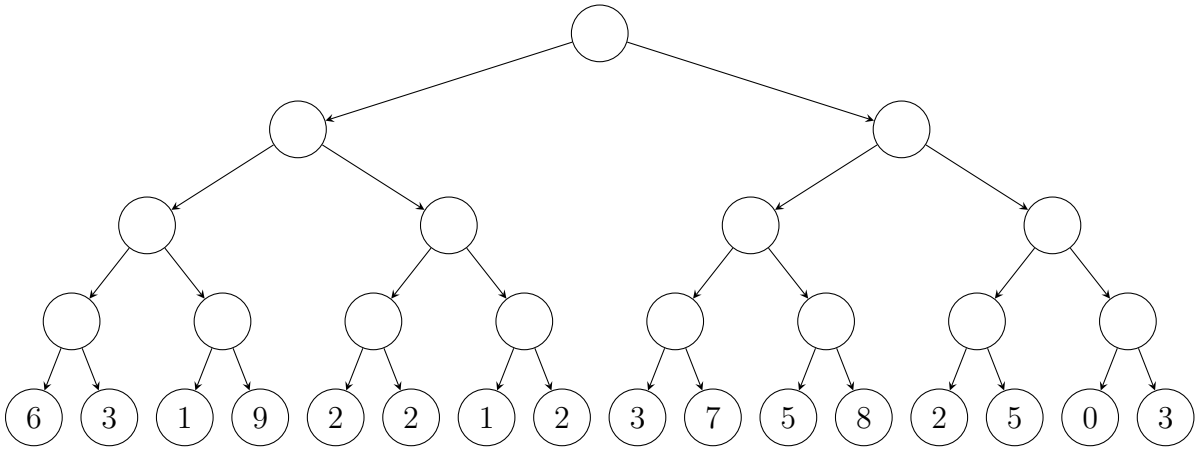
2. (5 points) Is the heuristic in the previous problem admissible? Explain why or why not.

3. (10 points) Given the following game tree, where the terminal nodes contain their respective evaluation function, perform minimax search and indicate which move should be made. Then repeat the search using the $\alpha - \beta$ algorithm and indicate which branches would be pruned (if any) on the tree. Assume MAX goes first.

minimax



$\alpha - \beta$



4. Consider the problem of assembling a mechanism consisting of 10 parts labeled A-J according to the following rules:

   - Part J must be the first part added to the assembly.
   - Part C must be added to the assembly before parts A,D,E,G.
   - Part E must be added to the assembly after part C.
   - Part A and Part B must be added sequentially (one directly after the other) in order

   (a) (3 points) Formalize the problem as a CSP.

   (b) (3 points) Is a solution possible?

5. For the following CSP

   - Variables and Domains:

$$X_1 = \{0, 1\}, X_2 = \{-1, 0, 1\}, X_3 = \{0, 1, 2, 3\}$$

   - Constraints:

$$X_1 \neq X_2, X_2 = X_3, X_3 \neq 3, X_1 + X_2 = 0$$

   (a) (2 points) Apply node consistency.

   (b) (2 points) Then, apply arc consistency (AC-3).

   (c) (2 points) Using your result from part b, is a solution possible?

Engineering Design Problems

To complete the engineering design problems, you will need to download the starter code in the ps1.zip file from the website. Only turn in the files specified by attaching them to your Canvas submission.

6. (10 points) Write a module puzzle8.py that solve the sliding tile puzzle when N=8, i.e. a 3x3 board, using A* search. The main entry point should be the function

   ```
   solve(initial, goal, hname)
   ```

   where `initial` and `goal` are the respective board configurations, and `hname` is a string specifying the heuristic. The `hname` argument is either `'tilesout'` or `'cityblock'`; the function should raise a ValueError exception otherwise. The boards are specified as integer lists of length 9 with indexes mapping onto the board configuration as follows

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

   and each value is an integer 0-8, where 0 indicates the empty slot and 1-8 are arbitrary tile labels. The function should return the value None if no solution exists. If a solution exists then it should be returned as a list of moves interpreted as the index to move the empty tile to. For example in the following python session the returned list means to move the empty to index 1, then to index 0. The exchange with tiles labeled 4 and 1 are implied, but the exchange must be possible or the solution is invalid.

   ```
   >>> import puzzle8
   >>> goal =  [0,1,2,3,4,5,6,7,8]
   >>> initial = [1,4,2,3,0,5,6,7,8]
   >>> puzzle8.solve(initial, goal, 'tilesout')
   [1, 0]
   ```

   You can use the implementations of `astar_search` provided (from the text's website) in the files `search.py` and `utils.py`. You will need to create a subclass of `search.Problem` and override the appropriate methods to use the search function.

   Two implementation hints. The node state must be hashable, so you will need to modify the representation of the boards passed to the solve function. The partial function from the functools module can be used to specialize your heuristic functions with the goal board provided, so that you can pass them to the `astar_search` function.

   A simple set of tests in the included `puzzle8_run.py` can be used during development. It is a large subset of the tests I will run when grading. Feel free to add tests, but do not turn this file in. Turn in the "puzzle8.py" file.

7. (10 points) Consider the game of Nim as follows.

   - There three piles of tokens with initially a given number in each pile.
   - Each turn the player must take at least one token,
   - A player can take multiple tokens, but only from the same pile.
   - The game is won by the player taking the last tiles (none remain after the turn).

   Write a module `nim.py` that defines two Nim playing agents, one using mini-max, the other using alpha-beta. The agent functions, minimax and alphabeta, should each take the current pile allocation as a tuple $(p0, p1, p2)$ and return a tuple of (t, f), where t is the number of tokens to take from pile f (for f in $[0, 1, 2]$). The value of t must be strictly positive and less than or equal to the maximum number of tokens available for pile f (i.e. $0 < t <= p0$ for $f = 0$, etc.). The following python session demonstrates how to call each agent and expected output:

```
>>> from nim import minimax, alphabeta
>>> minimax( (2,3,5) )
(4,2)
>>> alphabeta( (2,3,5) )
(4,2)
```

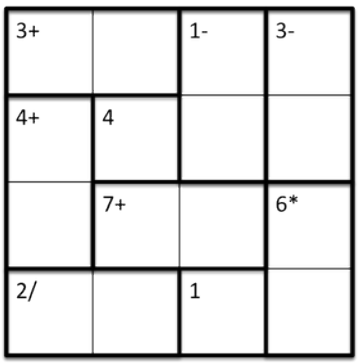
   You can use the implementations of full alpha-beta and mini-max search provided (from the text's website) in the files `games.py` and `utils.py`. You will need to create a subclass of `games.Game` and override the appropriate methods to use the `games.minimax_decision` and `games.alphabeta_full_search` functions correctly. This will require that you read and understand how those functions are implemented.

   A text file with an exhaustive list of games for piles of 5 or less is provided in the file `nim_examples.txt` of the starter code. Turn in the "nim.py" file.

8. (10 points) KenKen is a popular newspaper puzzle similar to Sudoku. The easy version is a 4x4 grid of cells with disjoint subsets of cells forming cages. The rules are

   - each row and column must contain the integers 1-4 non-repeating
   - each cage has associated with it an operation (including a no-op) applied to the cage contents in arbitrary order to produce a target result

   To clarify, here is an example puzzle and its solution.



2	1	3	4
3	4	2	1
1	3	4	2
4	2	1	3

Write a module `kenken.py` containing a function `kksolve` that solves easy (4x4) puzzles, if possible, using backtracking search. The function should take a single argument defining the puzzle and return either the solution as a list of cell values, or `None` if no solution exists. Use forward checking for inference and select unassigned variables using the minimum-remaining-values heuristic.

The puzzle input is specified by a list of cages, with each cage a list formed by a tuple of the target value and operation, followed by the cells in the cage. Cells are indexed 0-15 in row order, meaning cells 0-3 are the first row, 4-7 the second, etc. The operation can be one of `('+','-','*','/',None)` denoting add, subtract, multiple, divide, and no operation respectively. The following python session demonstrates how to call the function and expected output for the example puzzle above:

```
>>> from kenken import kksolve
>>>
>>> puzzle = [[(3,'+'),0,1],
...                [(1,'-'),2,6],
...                [(3,'-'),3,7],
...                [(4,'+'),4,8],
...                [(4,None),5],
...                [(7,'+'),9,10],
...                [(6,'*'),11,15],
...                [(2,'/'),12,13],
...                [(1,None),14]]
kksolve(puzzle)
[2,1,3,4,3,4,2,1,1,3,4,2,4,2,1,3]
```

The `kksolve` function should verify the puzzle specified is valid in the sense that the union of all cages contains all cells, each cage is disjoint with respect to the others, and that the operation and number of cells in the cage is compatible (e.g. a no-op operation can apply to one cell only).

You can use the implementation of `backtracking_search` provided (from the text's website) in the files `csp.py` and `utils.py`. You will need to create a subclass of `csp.CSP` and override the appropriate methods, the critical one being the number of conflicts

(nconflicts). See the documentation and examples in `csp.py`. The Sudoku example will be particularly helpful.

Some implementation hints. Start with a simplified version of the problem using no-ops only as this simplifies your code, but ensures you have the algorithm and call sequences worked out roughly. Note the operations other than None (no-op) are m-ary. For some of the operators this does not matter since they are invariant to permutations, but for others it matters. I'll leave it up to you to figure out which is which.

A simple set of tests in the included `kenken_run.py` can be used during development. It is a subset of the tests I will run when grading. Feel free to add tests, but do not turn this file in. Turn in the "kenken.py" file.