Assignment 3: Tetris Constraint Solving

Created By: Faiz Chaudhry

Due on: DUE-DATE

Introduction:

The game of Tetris was originally created for the Nintendo Entertainment System (NES) by Alexey Leonidovich. From its release to today it has become an extremely popular game with many remakes and spinoff minigames. For this assignment we will be focusing on one particular spinoff minigame, the **Tetromino Puzzle**. As fun as the game is solving it by hand, did you know you can actually write solvers for it? Moreover, it can be represented as a Constraint Satisfaction Problem (CSP). In particular, we can think of the Tetromino pieces to be the variables and the constraint being that the pieces must fit onto the grid without intersecting with each other. For this assignment you will be working on solving Tetromino Puzzles!

Installation:

In order to solve the assignments you will need to install Python 3 along with the Tkinter library (GUI) and TkThread library (for TKinter multithreading). You can install Python 3 through your terminal if you're on unix or through an installer found here `https://www.python.org/downloads/release/python-374/`. To install the Tkinter and TkThread libraries we recommend downloading and using python pip:

- pip install tkinter
- pip install tkthread

Files to Edit:

- tetris_csp.py
- tetromino_puzzle_constraint.py
- csp_algorithms.py

All files are found under the csp directory

Question 1:

The first method you will implement is *build_ domain* found in *tetris_ csp.py*. This is a simple method with the goal being to create a domain for the given variable. Remember that these domains are **not restricted to any constraints**, in other words, they should be all possible positions that the given tetromino variable can take. Don't forget that just like in Tetris, a Tetromino can be **rotated**, thus adding another possible value it can take. Your implementation should **return a list of values in the form ((row, col), rotation)**.

Question 2:

For this question you will be implementing the *check* method found in *tetromino_ puzzle_ constraint.py*. The goal of the method is to confirm, given a list of variables and a dictionary of assignments for said variables, that the assigned values hold true to the constraint. In this case the constraint we are dealing with is that each tetromino can fit onto the grid and that it doesn't conflict with any existing pieces on the grid. Be careful to consider how the pieces will interact with each other as well. Your implementation will **return True if the constraint is satisfied and False otherwise**.

Question 3:

This question asks that you implement the *Back Tracking* algorithm learned in class. It should be implemented in its respected function found in *csp_ algorithms.py*. Remember, this algorithm takes a *DFS* approach to solving for a possible set of assignments which satisfy the constraints given. **Return the first solution found by the algorithm or None if no solution is found**.

Question 4:

Next you will implement the *Forward Checking* algorithm. It should be implemented under its respective method in *csp_ algorithms.py* like the previous question. Remember, when one variable remains unassigned this algorithm will attempt to check forward into said variable's domain and prune values which cannot work with the current assignments. For this algorithm you will find the *forward_ check* function (found in *csp_ util.py*) to be helpful for pruning. **Return the first solution found by the algorithm or None if no solution is found**.

Question 5:

In order to use the GAC algorithm you must first implement the *has_future* method in *tetromino_puzzle_constraint.py*. This method is what allows us to enforce consistency when using the GAC algorithm. Your implementation should take the given fixed pair (a variable and a value for it) and return True if there exists a combination of assignments for the other variables, along with this fixed assignment, that satisfies the constraint. Return False if no such "future" exists for the given (variable, value) assignment. This method's complexity scales with the available domain of each variable. So, depending on the domain sizes of the variables it can take a long time to run.

Question 6:

At last you are ready to implement the *GAC* algorithm mentioned in lecture. Once again it should be implemented in *csp_algorithms.py* under it's respective function. This algorithm focuses on enforcing consistency on the constraints involved in our CSP. You will find the *gac_enforce* function found in *csp_util.py* to be helpful in your implementation. **Return the first solution found by the algorithm or None if no solution is found**.

Testing:

To test your solutions we provide a partial autograder. This autograder **should not** be treated as your final mark as there will be more tests involved during actual marking. To run the autograder simply type 'python autograder.py' or 'python3 autograder.py' (depending on how your environment variables are setup).

Inspiration and References:

The structure of this assignment and some questions are inspired from those found in the assignments provided to students in CSC384 during 2018 (last year). This assignment follows the course material (specifically the lecture slides) of CSC384 during 2018. As the CSC384 course and assignments from 2018 reference concepts taught to students in a similar AI course found at Berkeley, I find it important to include Berkeley as a reference for these assignments as well. To see more information on Berkeley's AI course visit `http://ai.berkeley.edu`.