Project Part 1 Proposal
DUE Friday, Oct. 2 at 11:55 pm

TEAM MEMBERS:

1. Austin Kostreba (kostr009)
2. Sean Lin (linx0486)
3. Sihan Chen (chen2436)
4. Wen Chuan Lee (leex7095)

Problem	Algorithms for Comparison
Sudoku	DFS vs Backtracking DFS (i.e. nodes represent actions and there is a single representation of the board)
n-queens	A* (starting with a completed puzzle) vs Backtracking DFS (i.e. nodes represent actions and there is a single representation of the board)
Sliding puzzle	A* heuristics of Manhattan vs Straight-Line
Towers of Hanoi	BFS vs (Bidirectional OR Iterative DFS)

Problem - Team Member Pairing

Sudoku: Sihan Chen
n-queens: Austin Kostreba
Sliding Puzzle: Sean Lin
Towers of Hanoi: Wen Chuan Lee

Algorithms Implementation

For each, list who will be responsible for implementing the algorithm, and if more than one person is listed, indicate whether this will be in collaboration or independent (e.g. BFS: Nina and Roan independent; A*: Lyra and Manuel collaborative). Not all will necessarily be implemented.

BFS: Wen Chuan Lee (individual)

DFS / Iterative DFS: Wen Chuan Lee (individual), Sihan Chen (individual)

Backtracking DFS: Sihan Chen (individual), Austin Kostreba (individual)

A*: Sean Lin (individual), Austin Kostreba (individual)

Bidirectional: Wen Chuan Lee (individual)

Experiment Hypotheses

For each problem

1. Identify 1 or 2 hypotheses that you are exploring with respect to comparing the 2 algorithms. Where appropriate, use theoretical bounds to support your ideas.
2. Identify the metrics you will use to make the comparison.
3. Identify the test cases (or sets of test cases). Note how different this will look for the various problems. For example, Sudoku puzzles vary in size, the number of pre-filled boxes, and general difficulty, thus you can test along multiple dimensions; whereas n-queens only varies along size, however the A* approach requires an initial (random) configuration, thus there should be a set of same sized test cases with random starting configurations.

You can organize this information in whatever manner you choose.

Sudoku:

1. The backtracking DFS will be better than DFS in both space and time complexity. For the space complexity, DFS has to record the whole board as its state. While for the backtracking DFS, it only need to store the action as its state. And for the time complexity, since DFS has to use deepcopy on each state, it will take more time.
2. I will use time spent
3. First set: Varying the size of the Sudoku puzzles
   Second set: Varying the number of pre-filled boxes
   Third set: Varying the general difficulty of the Sudoku puzzles.

n-queens:

1. A* will perform better than Backtracking DFS. In terms of memory, Backtracking DFS will probably perform better, because it only keeps track of one state and each action, however it still simply goes through the graph randomly until it reaches the end state. A*, however, will start with a board with *n* queens placed, each queen with a number of conflicts, and will try to move the queens such that their number of conflicts decrease.
2. I will use moves made and time spent reaching the solution
3. For A* I will test various n x n sized boards with all the queens in the first column to start, and will pick a size to implement different starting points for the queens with an unchanging board size. For Backtracking DFS I will simply test different board sizes.

Sliding Puzzle:

1. Straight Line heuristic will perform better than Manhattan Distance heuristic. Moving a piece close to its goal one square closer using Straight Line will produce a better heuristic than moving a piece far from its goal closer one square. Moving pieces already close to their goal will result in a faster solution.
2. I will use moves made and time spent.
3. I will test different sizes of puzzles, different puzzle configurations (different initial heuristics), and puzzles that can and can't be solved.

Towers of Hanoi:

1. Bidirectional or Iterative DFS will have a better performance and time complexity than BFS. As BFS has to explore each node at every level before moving on to the next, the memory and time

required will be much longer while DFS will immediately attempt the solution at the lowest depth. BFS will however yield the most optimal number of steps, if there is one.

2. I plan to use Python timer functions to record and calculate the total time to find a solution. In addition I will observe the memory use through ‘top’, a task viewer in Linux when executing both algorithms on puzzles that will take awhile to execute.
3. The test cases will be sets of Tower of Hanoi set ups that have increasing number of plates to shift (hence increasing the number of steps and hardness). This will provide an experimental estimate of the runtime of each algorithm as the problem sizes increase.

Additional Work

If at this point any team member would like to propose additional comparisons, please list the team member and the additional work.

For the **Towers of Hanoi**: should time permit, I would like to implement both the bidirectional and iterative DFS, and then run additional tests comparing those two, or explore another AI search or pruning technique that will further optimize the algorithms used to solve the problem. An example would be to find a particular setup that will cause the generated state to be illegal early in the tree to avoid wasting time traversing down a branch that will not yield a good solution. This different form of pruning will complement the traditional pruning methods and further reduce the possible state space.

For **Sliding Puzzle**: Given enough time, I would like to implement a bidirectional BFS, and/or a more directed A* that mainly focuses on one piece at a time.

For **n-queens:** If time permits, I would like to implement Backtracking DFS with pruning. Normal Backtracking DFS will generate each action without looking to see if it is a viable move or not. If we apply the constraints that another piece cannot be placed in the same row, column, or diagonal as another queen, it would prune the majority of possible moves. I’d like to compare that with A* as well. Another possible method would be to do iterative Deepening DFS (maybe with pruning as well?) with an initial depth limit of n/2 or some other limit that I would choose, maybe even comparing different limits to find the best for this application.

For **Sudoku**: If I have enough time, I would like to implement the bidirectional BFS, bidirectional DFS and maybe some other algorithms to solve the Sudoku puzzle.