Game of Solo Chess

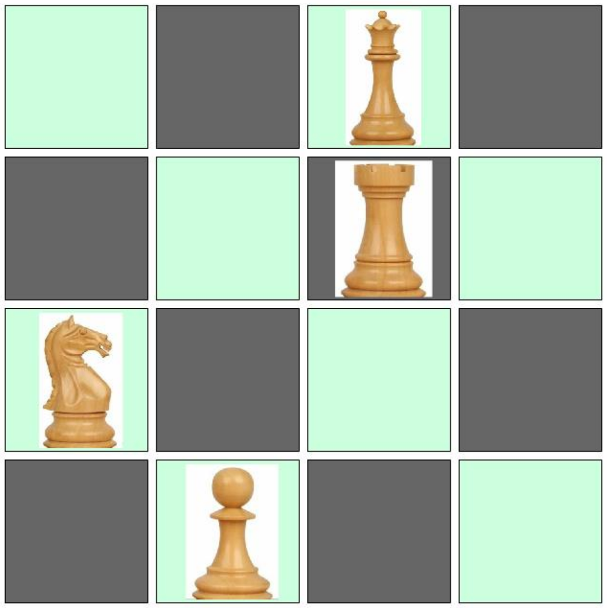
This program implements a popular board game that purports to "train" young minds for the game of chess. The following section describes how it is played.

How the game is played:

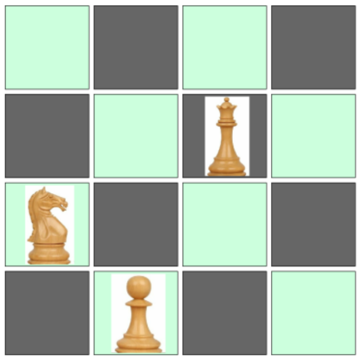
- Use a 4x4 chess board (instead of the standard 8x8)
- Use only 2 pieces each of bishop, knight, pawn, queen, and rook. Color doesn't matter.
- In the initial layout some of these pieces are laid out.
- Start playing with any piece:
    - Rule 1: Every move must kill another piece (using usual Chess rules)
    - Rule 2: Every move may use a different piece
    - Rule 3: When only 1 piece is left on the board, the game is over
- The challenge is thus to play (without violating the above rules) until only 1 piece is left.

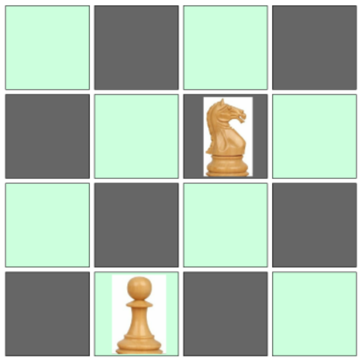
Here is an example:

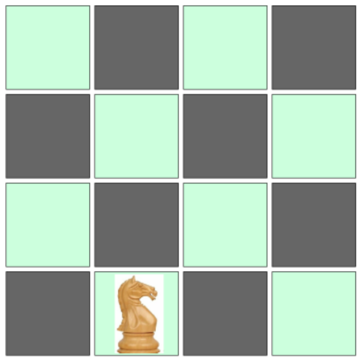
This image indicates one initial board position:



The following moves indicate how the game can be finished:


Queen takes rook


Knight takes queen


Knight takes pawn

Explore the game:

If you want to play with my final program to get a feel for this game, run the Python file specified at the end of the article. Try not to peek at the scripts yet, since we want to design them ourselves below.

1. Start the program. Enter h for help or continue.
2. Pick one of the levels of expertise. The initial layout will be shown.
3. Play the game by making moves: click on two cells to make a move, click on the same cell twice to cancel a move. If the move is invalid, the program will show error.
4. Click on "undo" to retract the last move.
5. Click on "auto" any time to make the program solve the puzzle from the current state.

Click here to download all program files.

Python and CS Concepts Used

When we design this program, we will make use of the following Python and CS concepts. Learn these concepts if you don't know them before proceeding further.

- Algorithms
  - Designing new algorithms
- Arithmetic
  - Expressions
  - Basic operators (+, -, *, /)
  - Advanced operators: mod, floor, ceiling, %, //, etc.
- Concurrency:
  - Multi-threading for timed callback
- Conditional statements:
  - Conditions: YES/NO questions
  - Relational operators (==, <, >, etc.)
  - Conditionals (IF)
  - Conditionals (If-Else)

    - Conditionals (nested IF)
    - Boolean operators (and, or, not)
- Data structures – list
    - List operations
    - Using list as 2-D array
    - List comprehension
- Data structures – tuples
- Data types – basic
    - Integers
- Data types – strings
    - String operations
    - String traversal
- Events
    - Coordinating multiple user events
- Looping (iteration)
    - Looping - simple (for)
    - Looping - nested
    - Looping - conditional (while)
- Procedures
    - User defined (custom)
    - Procedures with parameters and return value
- Program output
    - Text
    - GUI (Tkinter)
- Random numbers
- Recursion
- Sequence
- Testing
    - Automated function testing
- User input
    - Text
    - Click buttons
- User interface elements
    - Label
    - Button
    - Grid
- Variables
    - Simple
    - Local/global scope

Console (text) Version: High Level Design

In this version, we will use a simple text-based user interface. Let's consider how the various features of this program can be separated out as distinct pieces. As usual, we have the front-end that interacts with the user, and the back-end that performs all game functions.

Front-end components:

- 4x4 chess-board:
  - Shows the current layout
  - User input to perform actions such as move, undo.
- 5x2 chess pieces (rook, bishop, pawn, queen, knight)
  - If a cell contains a chess piece, its letter (e.g. 'r' for rook) is shown on the cell

Back-end components:

- 4x4 layout of the board: list of 16 letters: 0 for empty, q, b, k, r, p for pieces
  - The initial layout is taken from a database
  - After every valid (i.e. that satisfy game rules) move (or "undo" operation), the layout is modified.

Objects:

We will distribute the code among the following objects (sprites):

- The "Frontend" object will contain common code for user interaction.
- 5 chess objects: For each chess piece we will have a separate object containing its own logic – which would be fairly similar to each other.
- "Backend" will contain logic to manipulate the 4x4 board (i.e. the 16-item list) as per the game rules. This object will drive the entire game.

We will add methods (procedures/scripts) to these objects as we process each feature idea below.

We may also add more objects as we learn more about the features of the program.

Global data:

List "Board": list of 4 lists each containing 4 items, will hold the current status of the board, 0 for empty cell, r/b/q/k/p (for the respective chess piece)
List "Undo List": will hold the moves made thus far
Integers "From" and "To": locations (1 to 16) for the current move

Feature Idea # 1: The initial setup

When game is started, it should display the initial board positions.

Design:

The game obviously depends on an initial layout. We will use a collection of possible layouts. Each layout could be encoded into a 16-letter string: 0 for blank cell and r/k/q/b/p for a chess piece. For example, the encoding for the board shown at the top of this article would be "00q000r0k0000p00".

We will have 4 collections of such encoded layouts based on level of difficulty: Expert, Advanced, Intermediate, and Beginner.

We could save these 4 collections in 4 separate variables: one for each type. So, for example, "Expert " would contain all expert-level layouts separated by commas.

We will prompt the user to pick a level. The program would then take the appropriate comma-separated string, split it into a list, and pick one of the layouts at random. This layout encoding (16-letter string) would then result into the actual chess layout. This layout will be split it into a 16-item list Board. This list will henceforth determine what each board position looks like (either blank or with a chess piece).

Feature Idea # 2: The move

When user picks two valid pieces, make that move.

Design:

According to the game's rules, every move involves one chess piece taking another. User input for each move will be stored in the global variables "From" and "To". Next, we do the following:

- Validate the move: ask the piece at the first click to check if the move is valid (this feature is implemented later). Proceed only if true.
- Save the move in an "undo" list (for the "undo" feature implemented later).
- Update list Board to show the move.

Each chess piece object will implement its own logic for step 1 as follows:

- For Step 1, each piece will implement a "Validate Move" method, which will verify that the intended move is legal (e.g. rook can only move straight). If it is not, this method will return error to the backend which should then cancel the move. We will implement this method later and for now we will assume the user knows what he/she is doing.

Feature Idea # 3: Undo

User should be able to undo his/her moves.

Design:

For this, we will need to save every move – the best place would be a list. Let us call it "Undo List" – the undo list. We will save in a single string the "from" and "to" locations as well as which piece was taken (killed) by which piece, for example, "2,15,k,p" would mean knight at 2 was moved to replace a pawn at 15.

Next, we will provide an "Undo" command. When user selects this command, the latest element in "Undo List" would be popped and processed. For example, if it is "2, 15, k, p", we need to perform the following actions to undo:

- The knight at 15 came from 2, so we will ask it to move back to 2 (using the "Move" method).
- A new pawn needs to appear at 15. The letter 'p' will be placed the same way as during the initial placement.

Of course, don't forget to update list Board with the new layout since pieces have moved around.

Feature Idea # 4: Play the game

Provide the algorithm to play the game.

Design:

We just need a procedure to drive the overall flow of the game using the features implemented so far. Here is an outline of the procedure: we will call it PlayGameManual because we will have an "auto" version later.

```
Algorithm Play Game Manual
Show the initial board layout.
Ask user what he wants to do: move, undo, or quit
For "undo": call the undo method
For "move" ask for "From" and "To".
Make the move using the "Make Move" method.
If the board is left with 1 piece:
      Declare victory, show the list of moves and quit
Ask user again.
```

Save as Program Version 1

Congratulations! You have completed the basic features listed above. Compare your program with my program below.

Main program: solochess-1.py

Other files:
config_solochess_console.py: global variables for solochess

Feature Idea # 5: Allow valid moves

In feature idea #3 above, we move pieces without checking if the moves follow chess rules. Implement these rules. For example, bishops only travel diagonally.

Design:

In general, we will need a bunch of methods that validate Chess rules for various pieces. Since these would generic functions, we will put them in a separate file and import them into the main program. As will be seen later (in the "advanced" section), some of these functions would also be used by other features.

Fundamentally, we will need an algorithm for each piece that will verify its motion between two cells on the board. We will call these algorithms "Check Obstacle" which will check if the given piece (bishop, etc.) can go from "from" to "to" unobstructed. It would not matter if there was no such piece at "from" or whether "to" is occupied or not. Those additional checks would be up to the caller.

You can see that using this basic "check obstacle" algorithm, we can easily build other useful algorithms, such as:

Is Valid Move:

For a piece at "from" is the position "to" accessible to it? Return true if "check obstacle" returns success and "to" is not occupied.

Is Valid Kill Move:

For a piece at "from" is the position "to" accessible to it? Return true if "check obstacle" returns success and "to" is occupied.

Get Moves:

For a piece at "cell" return a list of all possible empty cells it can move to. (Scan the entire board and call "check obstacle" for each cell.)

Get Kill Moves:

For a piece at "cell" return a list of all possible non-empty cells it can move to. (Scan the entire board and call "check obstacle" for each cell.)

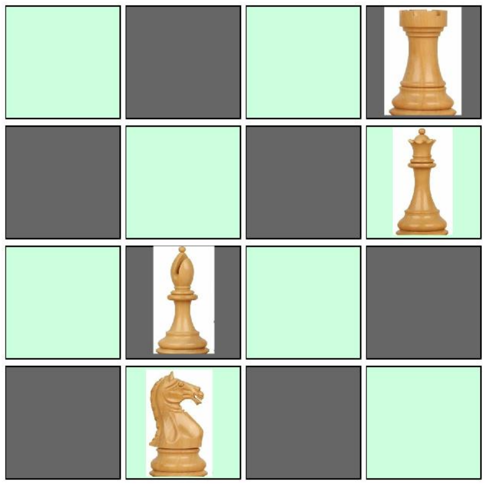
Let us now build the basic "Check Obstacle" algorithm for each piece.

***Step 1**: Check if bishop can go from "from" to "to".*

Design:

Bishop can move as follows:

- Pick any of the 4 diagonal directions (nw/sw/se/sw)
- Move straight until it hits another piece



In the above layout, the bishop can legally move to position 4, 5, 7, 13, or 15 and any other move would be invalid. How do we check that the move picked by the user ("from" to "to") is valid?

After careful analysis, we come up with the following observations:

- If "from" and "to" are diagonally positioned (newrow – oldrow must be the same in value as newcolumn – oldcolumn) it is likely to be a valid move.
- Next we need to find in which direction "to" is situated. This can be found by comparing row/column of "to" with those of "from". For example, if newrow < oldrow and newcolumn < oldcolumn, "to" must be northwest of "from".
- We then scan every cell along that direction: if we reach "to" (and don't overshoot the board) without hitting any "occupied" cell, it is a valid move, else it's an invalid move.

The following algorithm takes care of these observations.

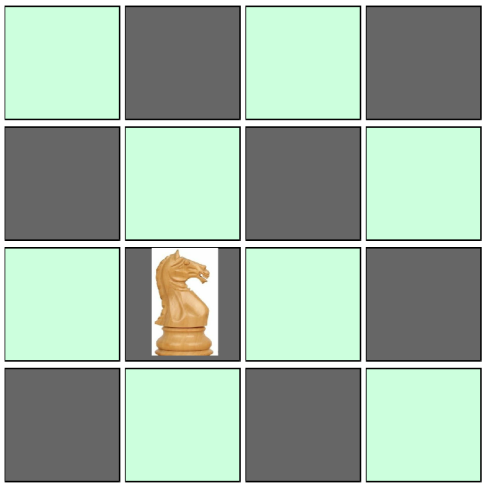
```
Algorithm Check Obstacle for bishop:
Input: from and to (numbers 1 to 16)
R1 and C1: calculate row/column of "from" (1 to 4)
R2 and C2: calculate row/column of "to" (1 to 4)
g1 = R2 - R1
g2 = C2 - C1
If value of g1 and g2 are not equal return False
Determine "direction" of the move by comparing g1 and g2.
      For example: If g1<0 and g2<0 the direction is Northwest.
For each direction:
      Imagine bishop moving from "from" one cell at a time.
      Move until you overshoot the board and check every cell:
            If the cell is "to" return True.
            If the cell is occupied return False. (obstacle!)
```

***<u>Step 2</u>*: Check if knight can go from "from" to "to".**

Design:

Knight can move as follows:

- Pick any of the 4 directions (north/south/east/west)
- Move 2 steps straight
- Move 1 step at the right angle



In the layout above the knight can move to 1, 3, 8, or 16. How do we check that the move picked by the user ("from" to "to") is valid?

After careful analysis, we come up with the following observations:

- It is better to use row (R) and column (C) to do this check.
- There are 8 possible movements as below. (Not all would be valid)
- New positions if it moved 2 steps east, 1 step north or south: (R-1,C+2), (R+1,C+2)
- New positions if it moved 2 steps west, 1 step north or south: (R-1,C-2), (R+1,C-2)
- New positions if it moved 2 steps north, 1 step east or west: (R+2,C+1), (R+2,C-1)
- New positions if it moved 2 steps south, 1 step east or west: (R-2,C+1), (R-2,C-1)
- If we removed the invalid positions (by looking at the new row and column values) we would have a list of valid moves
- We could then check if "to" is one of the valid moves.

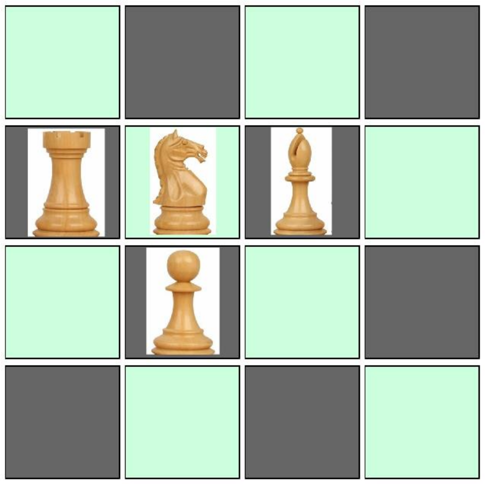
The following algorithm takes care of these observations.

```
Algorithm Check Obstacle for knight:
Input: from and to (numbers 1 to 16)
R = calculate row # of "to"
C = calculate col # of "to"
Create a list L of all valid moves as described above
      Example: (R-1, C+2) is a possible move
      If R-1 is a valid row and C+2 is a valid column
            newposition = ((R-1)-1) * 4 + (C+2)
            Add newposition to L
If "to" is a member of L return True, else return False
```

<u>Step 3</u>: Check if pawn can go from "from" to "to".

Design:

Although pawns move straight up only, while killing they move diagonally. Consider the board as shown below:



In this layout, the pawn can kill the rook or the bishop, but not the knight or any other piece on the board (if there were any). How do we check that the move picked by the user ("from" to "to") is valid?

After careful analysis, we come up with the following observations:

1. "from" is always greater than "to"
2. "from" cannot be less than 5
3. "to" cannot be greater than 12
4. The difference between them can either be 3 or 5.
5. When the difference is 3, "from" cannot be in the last column
6. When the difference is 5, "to" cannot be in the last column

The following algorithm takes care of all these observations. Since we only allow a gap of 3 or 5, items 1 thru 4 are automatically taken care of. (For example, if from = 4 (violating #2) and to = 1, the gap would be 3 and the first "if" below would be violated.)

```
Algorithm Check Obstacle for pawn:
Input: from and to (numbers 1 to 16)
Gap = from - to
If Gap is 3 AND "from" is not in the last column
           Return True
If Gap is 5 AND "to" is not in the last column
           Return True
Return false
```

***Step 4**: Check if queen can go from "from" to "to".*

Design:

Queen can move either as a rook or as a bishop. So we can simply use the bishop and rook algorithms designed earlier.

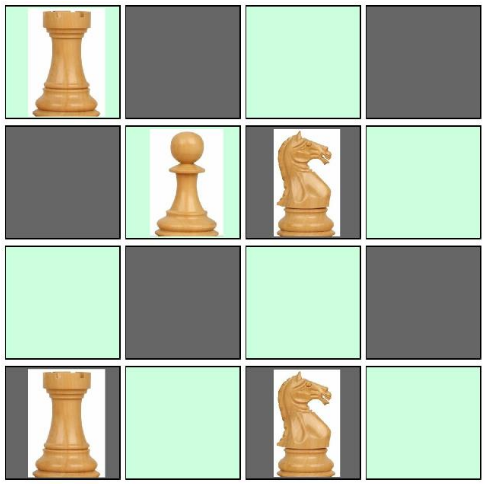
```
We first run the bishop algorithm
      If it returns True, we return True
Next, we run the rook algorithm and return whatever it returns.
```

***Step 5**: Check if rook can go from "from" to "to".*

Design:

Rook can move as follows:

- Pick any of the 4 diagonal directions (N/E/W/S)
- Move straight until it hits another piece



In the above layout, the rook at 13 can legally move to positions 1, 5, 9, 14 and 15 and any other move would be invalid. How do we check that the move picked by the user ("from" to "to") is valid?

After careful analysis, we come up with the following observations:

- If "from" and "to" are positioned straight up/down or sideways (both newrow – oldrow and newcolumn – oldcolumn cannot be non-zero) it is a valid move.
- Next we need to find in which direction "to" is situated. This can be found by comparing row/column of "to" with those of "from". For example, if newrow < oldrow and newcolumn = oldcolumn, "to" must be North of "from".
- We then scan every cell along that direction: if we reach "to" (and don't overshoot the board) without hitting any "occupied" cell, it is a valid move, else it's an invalid move.

The following algorithm takes care of these observations.

```
Algorithm Check Obstacle for rook:
Input: from and to (numbers 1 to 16)
R1 and C1: calculate row/column of "from" (1 to 4)
R2 and C2: calculate row/column of "to" (1 to 4)
g1 = R2 - R1
g2 = C2 - C1
Only one of g1 and g2 must be non-zero. If not, return False
Determine "direction" of the move by comparing g1 and g2.
      For example: If g1=0 and g2<0 the direction is West.
For each direction:
      Imagine rook moving from "from" one cell at a time.
      Move until you overshoot the board and check every cell:
            If the cell is "to" return True.
            If the cell is occupied return False. (obstacle!)
```

Save as Program Version 2

Congratulations! You have completed the features listed above. Compare your program with my program below.

Main program: solochess-2.py

Other files:
config_solochess_console.py: global variables for solochess
chess_routines.py: all chess-related functions

Feature Idea # 6: Automation: Computer solves the board

In the "auto" mode, the computer should attempt to solve the current board. User should be able to specify the auto mode at the start.

Design:

This interesting feature is also the most challenging. We will need to think what it means to automate the game-play.

Consider how a user would play in the manual mode. He/she would be presented with a board and will pick a move (i.e. a piece that can kill another). After this move has been made, he/she is presented with a modified board and may undo the previous move or once again make another move. Thus, you can see the same 2 steps repeated again and again until the game is over. So, we could create a "recursive" approach in which the user would make the "first possible move" – i.e. scan the board from the top-left and look for a piece that can kill another piece. As soon as such a move is found, he/she will make it and then make the recursive call (i.e. present the user with the new board). If the recursive call returns "failure" -- which means the board that we created did not lead to a solution, we will undo the previous move (because that move indeed led us into a dead-end) and make the "next possible move" and try again. If we exhaust all possible moves with the current board, we return "failure".

This approach is called "brute force" or "exhaustive search" because we indeed try all possible moves until gold is struck. This recursive process will drive the automation.

The underlying methods for chess moves (e.g. get all kill moves) have already been implemented in one of the features above.

Here is the algorithm for the main recursive method:

```
Algorithm Play Game Auto
Input: board (our 16-item list)
For every piece on the board:
      Build a list of possible kill moves for this piece
      For every move in this list of moves:
            Make the move and update the board
            If only 1 piece left:
                  Declare victory and return "success"
            End if
            Recursive call with the modified board
            If call returned success, no need to proceed,
                  Return success
            If call returned failure
                  current board did not work, so undo the move and
                  continue to the next move
            End if
      End For
End For
Since there are no more pieces, return "failure"
```

Save as Program Version 3

Congratulations! You have completed all features of the game for the text-based version. Compare your program with my program below.

Main program: solochess-console.py

Other files:
config_solochess_console.py: global variables for solochess
chess_routines.py: all chess-related functions

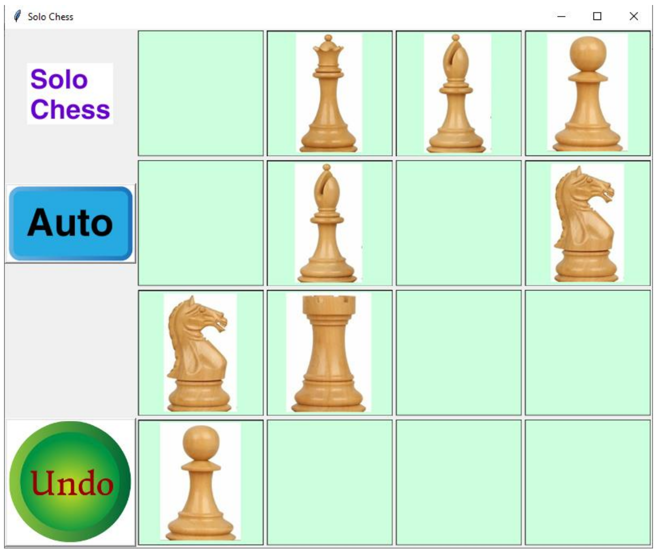
How to play the game:

1. Start the program. Enter h for help or continue.
2. Pick one of the levels of expertise. The initial layout will be shown.

3. Select game mode: auto or manual. If you select auto, the program will solve the puzzle and show the solution.
4. In manual mode, you will play the game by making moves: enter m to make a move, u to retract the last move, or q to quit. The program will show the new board after every move.

GUI Version: High Level Design

In this version, we will use a graphical user interface as we have been showing in this document so far. This is how the final graphical interface will look like. The space between Auto and Undo buttons will be used to display short status messages.



Since we already have most of the game logic in place, we only need to figure out how to replace the text interface with an interface where the board is visible as a graphical 4x4 chessboard and user input comes via clicks.

Let's consider how this new type of front-end can replace the current text-based interface.

Front-end components:

- 4x4 chess-board: The frontend will display the board in a 4x4 grid format and pass user input (which cell was clicked) to the backend. The frontend will always show the current state of the board. We will use a 4x4 grid of Tkinter labels to display the board. We can borrow the design from the tic-tac-toe program we wrote earlier and modify it.
- An additional column of widgets for the following:
    - Simple label showing the game title “Solo Chess”.
    - A click-button “Auto” to initiate automatic play mode.
    - A message area where messages such as “invalid move” can be displayed.
    - A click-button “Undo” to allow user to undo moves.

- So, we will basically have 4x5 grid (4 rows and 5 columns) in which the first column is for the additional widgets and the remaining 4 columns (2 thru 5) are for the chessboard.

Feature Idea # 7: The 4x5 display interface

Draw a 4x4 chessboard grid of clickable cells and an additional column of widgets.

Design:

For this, we can simply borrow an older Python program called "matrix" which provides the functionality of drawing a grid of Tkinter label widgets with grid manager. We will modify it to add the additional column of labels and click-buttons.

Step 1: The chessboard

The chessboard itself does not need to be sensitive to clicks. We will super-impose labels for the chess pieces on top of the chessboard labels. Since, we must have some image for every cell, we will use a tiny "dot" image for blank cells, i.e. cells where no chess piece exists.

For the columns 2 thru 5, each cell (i.e. label widget) will possess row and column numbers so that the backend can associate each visible cell with a location on the game board. When a chess piece is clicked, the clickLabel callback function will send these row/column numbers to the backend via variables.

The content of each cell can be either be blank (actually a tiny dot since we can't really have blank images) or will show the chess pieces rook, bishop, queen, pawn, and knight. We will use appropriate image labels for this purpose.

Step 2: The board state

We already have the list "Board" to represent the chessboard. Each item in this list shows the content of each cell. At the start of the game, the frontend logic will scan this list to decide which image to display in each visible cell.

The board state changes after certain actions in the game: for example, when a move is made or when an "undo" action is taken. To ensure the visible board is in sync, the backend will call the RefreshGrid method every time the board state changes.

Step 3: The additional column of widgets.

As discussed earlier, we will use the first column of the 5-column grid to place these additional widgets. As shown in the picture above, the "Solo Chess" label can be created by using the label widget. "Auto" and "Undo" are button widgets, which will call the "PlayAutoMode" algorithm (described in Feature #6) and the UndoMove routine, respectively.

Feature Idea # 8: Move and Undo

When user clicks on two valid pieces, make that move. Also, when the user clicks on the "Undo" button, retract the most recent move.

Design:

<u>Step 1</u>: The move

According to the game's rules, every move involves one chess piece taking another. Thus, we have a 2-step transaction here: in step 1, user clicks on a piece, and in step 2, he/she clicks either on the same piece (to cancel the move) or on another piece to take it. How can we program this? Here is one possible approach.

When a piece is clicked it will simply save its grid location (1 to 16) in a global variable ("clicked") and inform the backend.

The backend has two global variables: "From" and "To". Initially "From" would have some invalid value such as -1 to indicate that the user is yet to begin the move. As soon as user makes the first click, we save the location in "From". The second time user clicks, "From" would not be -1, so we will know this is the second click. If the second click is the same as the first, we simply cancel the move by setting "from" back to -1. Otherwise, we do the following:

- Validate the move: Send a message to the piece at the first click to check if the move is valid. Proceed only if true.
- Save the move in an "undo" list.
- Update list Board to show the move.
- Update the visible board.

We will implement these requirements as follows:

- We have already taken care of Steps 1 thru 3 in our text-based version.
- For steps 4 and 5, the RefreshGrid method can scan the list "Board" and display images according to the content of each cell.

<u>Step 2</u>: Undo

Undoing would be very similar to making a move. When the user clicks on "Undo", the latest element in "UndoList" would be popped and processed as described in Feature #3 earlier. In addition, we will need to update the frontend UI to reflect the undo operation. As before, the RefreshGrid method can scan the list "Board" and display images according to the content of each cell.

Feature Idea # 9: Highlight move

When user clicks on a cell for a move, highlight the cell to provide visual feedback.

Design:

We will use a different image to show the highlight. Our program presently loads images by looking at which letter is stored in the "board" list, e.g. if it's "r" we load the Rook image. To accommodate the highlighted images (one for each piece) we will upper-case letters, e.g. if it's "R" the highlighted image of Rook would be loaded.

The following algorithm describes all the different things that need to be taken care of for this feature:

```
Algorithm for highlight:
On first click:
   -  highlight the cell (e.g. change p to P),
   -  refresh display
On second click:
   -  if empty cell, do nothing
   -  if same cell, unhighlight the cell (change P to p)
   -  if another non-empty cell:
         o  if valid move: unhighlight the cell (change P to p)
         o  else: do nothing
If "auto" or "undo" clicked, unhighlight the highlighted cell if any
(it is saved in "From" variable)
```

Save as Program Version Final

Congratulations! You have completed all features of the game for the GUI version. Compare your program with my program below.

Main program: solochess-final.py

Other files:
config_solochess.py: global variables for solochess
chess_routines.py: all chess-related functions

How to play the game:

1. Start the program. Enter h for help or continue.
2. Pick one of the levels of expertise. The initial layout will be shown.
3. Play the game by making moves: click on two cells to make a move, click on the same cell twice to cancel a move. If the move is invalid, the program will show error.
4. Click on "undo" to retract the last move.
5. Click on "auto" any time to make the program solve the puzzle from the current state.

Advanced Version

So far we have been using a database of pre-built puzzle layouts. This limits to how many puzzles the user can solve. Can we instead create an algorithm of our own to create a random puzzle layout that is guaranteed to lead to a solution?

Since this is a rather intricate endeavor (as we will see below), we will keep the code it in a separate file.

Feature Idea # 10: Create puzzle layout

Create an algorithm generate a random puzzle layout that is guaranteed to lead to a solution.

Design:

This is an interesting challenge. One idea we can think of is working backwards from a solution. That is, we put a random piece on the chessboard and add more pieces one by one such that every new piece threatens at least one of the existing pieces. That way, we are guaranteed to have a board layout that would lead to a solution (i.e. just one piece left after a series of "kill" moves).

"Adding a new piece" would be a recursive action, because the exact same considerations would apply: the new piece must threaten one of the existing pieces.

Here are some important considerations for our algorithm:

1. We will allow only 1 queen since queen is so powerful it can make the puzzle really easy to solve.
2. We will allow only 2 of other pieces. (This is as per the original board game rules)
3. The board layout can have anywhere from 4 to 9 pieces.
4. Each occupied cell on the board can be tagged as FOOD or KILLER since the piece in that cell would either be killed (and hence be "food") or would take another piece (and hence be a "killer"). Tagging this way will help us decide how to treat every new piece.

Here is the high level recursive approach to create a random puzzle layout.

1. First piece is a random placement of a random piece, cell tagged as FOOD. (Remember: the tag is for the cell, not the piece. Since the game would end with a single piece (no matter what it is) at THIS cell, it makes sense to tag it as "food".)

Recursive steps start here:

2. Pick an occupied cell C1 at random.

3. If C1 is FOOD:
- It cannot be empty, so the piece cannot move out, but can move in, or another piece can kill it.
- Option 1: Move the piece away to C2 and replace by any other piece, tag C2 as KILLER.
- Option 2: Put a new piece at C2 which threatens C1, tag C2 as KILLER.
- Record move C2->C1 in our 'solution'.

4. If C1 is KILLER:
- It can be empty, so it can move in.
- Move the piece away to C2 and replace by any other piece, tag C2 as KILLER.
- Change C1 tag to FOOD.
- Record move C2->C1 in our 'solution'.

5. Recurse until board has required # of pieces.

And here is the detailed algorithm.

Algorithm Get Puzzle Layout

```
Given:
'Pieces': list of available pieces (2 of each, 1 queen)
'board': the 4x4 chess grid
'tags': 4x4 2-D list of tags associated with 'board'

Procedure:
Empty board: Pick a random location C. Place a random piece P. Tag C as Food.
(Use the algorithm "Place Any Piece Any Cell")
```

`Recursive procedure starts from here:`

```
Return success if required # of pieces already added to the board.

L = shuffled list of occupied cells (use algorithm "Shuffle Nonzero Items")
Repeat size of L:
C1 = pop item from L
P1 = piece at C1
If C1 is tagged as FOOD:
   -  Pick one of the options at random.
   -  Option 1:
        P = shuffled list of unused pieces
        Repeat size of P
          P2 = pop item from P
          C = shuffled list of cells from where P1 threatens C1.
            (Use the algorithms we wrote earlier for the chess pieces)
          Repeat size of C
            C2 = pop item from C
            Move P1 to C2, tag C2 as KILLER, Place P2 at C1
            Add move C2->C1 to 'moves'
            Recursive call (to add one more piece):
                  If it succeeds: return success
            Recursive call failed, so undo:
                  Remove P2 from C1, move P1 back to C1
                  Delete move C2->C1 from 'moves'
            Try next item in C
          End repeat
          Try next item in P
        End repeat
```

```
    -   Option 2:
        P = shuffled list of unused pieces
        Repeat size of P
              P2 = pop item from P
              C = shuffled list of cells from where P2 threatens C1.
              (Use the algorithms we wrote earlier for the chess pieces)
              Repeat size of C
                    C2 = pop item from C
                    Place P2 at C2. Tag C2 as KILLER
                    Add move C2->C1 to 'moves'
                    Recursive call:
                          If it succeeds: return success
                    Recursive call failed, so undo:
                          Remove P2 from C2
                          Delete move C2->C1 from 'moves'
                    Try next item in C
              End repeat
              Try next item in P
        End repeat
  End if (If C1 is tagged as FOOD)
  If C1 is tagged as KILLER:
        P = shuffled list of unused pieces
        Repeat size of P
              P2 = pop item from P
              C = shuffled list of locations from where P1 threatens C1.
              Repeat size of C
                    C2 = pop item from C
                    Move P1 to C2; tag C2 as KILLER. Place P2 at C1.
                    Add move C2->C1 to 'moves'
                    Change C1 tag to FOOD
                    Recursive call:
                          If it succeeds: return success
                    Recursive call failed, so undo:
                          Remove P2 from C1, move P1 to C1,
                          Reset C1 tag to the previous one
                          Delete move C2->C1 from 'moves'
                    Try next item in C
              End repeat
              Try next item in P
        End repeat
  End if (If C1 is tagged as KILLER)
  Return failure (since no appropriate piece was found)
```

Save as Program Version Final

Congratulations! You have completed all advanced features of the game. Compare your program with my program below.

Main program: solochess-advanced.py

Other files:
config_solochess.py: global variables for solochess
chess_routines.py: all chess-related functions

make_puzzle.py: functions related to creating puzzle layouts
config_mkp.py: global variables for make_puzzle

Author: Abhay B. Joshi (abjoshi@yahoo.com)
Last updated: 24 June 2020