CSCI 145 Week 5
Assignment 1

Assignment Description

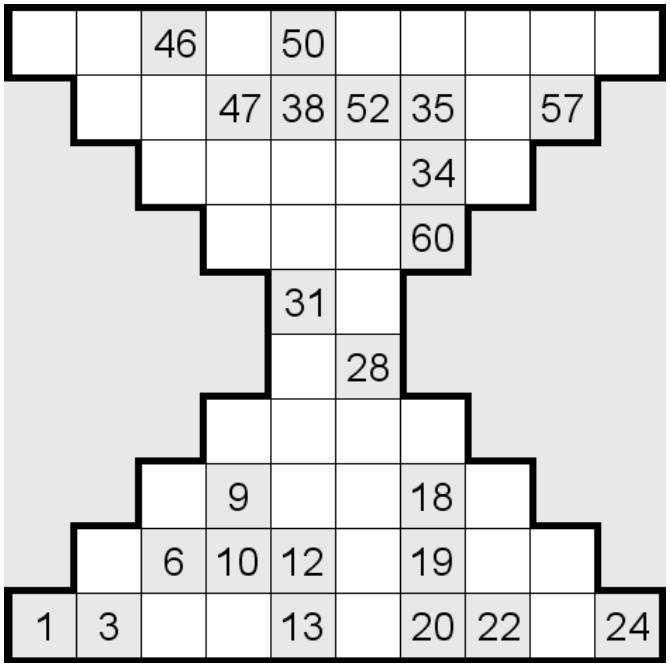
For this assignment you will write a Java program to recursively solve Hidato[1] puzzles. Your program will—with a little help—solve the puzzle and draw a picture of the solution. This assignment is worth 13% of your grade for the course.

Please read the entire assignment before beginning work on it. There is a lot of information in this description, and it is not organized in the order you will want to build your program. In particular, you will want to read the discussion in "Design Decisions" and the order of development in "Suggested Steps", both near the end, before starting to write your program.

Reading Suggestion: Read through the whole assignment and understand what is being discussed in each section. Don't bother to understand all the details. As you are building your program, when you come to the part of the program where a particular section becomes important, then go back and review that section to understand the details.

Background: Hidato

Hidato is a logic puzzle the goal of which is to fill in a grid with consecutive numbers that connect horizontally, vertically, or diagonally. Here is a sample Hidato puzzle (this is a problem from the daily news paper):

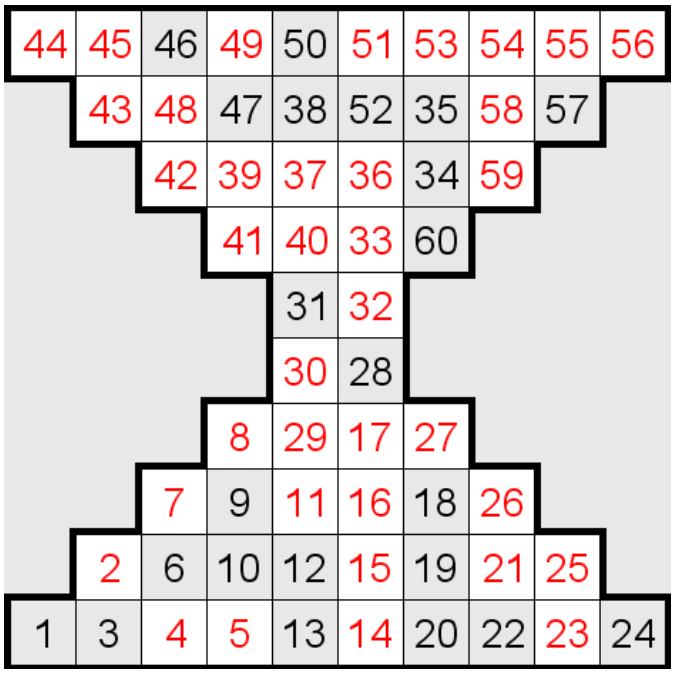


Each of the white boxes must be filled with a number between 1 and 60, the last number in the grid. The numbers in the gray boxes are fixed and must appear at the location shown. For this assignment,

1 https://en.wikipedia.org/wiki/Hidato

you can assume that 1 and the largest number are always given. The goal of the puzzle is to fill in the remaining numbers to create a path from 1 to the last number.

Here is the answer to the puzzle above:



Solution Method

Your Program

Your solution must be in a single Java source file named `Hidato.java`. Your program will require two additional modules: `DrawGrid.java` and `DrawingPanel.java` which I have included with the assignment. The two files I have provided should be placed in the same folder as your `Hidato.java` solution. You do not need to (and, in fact, can not) import either file into your program. However, they will be available to you as if you had imported them.

You should not modify either `DrawGrid.java` or `DrawingPanel.java`. If you feel you need to modify either file, you **must** talk to me first. Otherwise, I will expect that your program will work with the versions I have supplied.

Program Structure and Required Functions

Your main program should like the following "Java/Python-esque" pseudo-code:

```
void main(args) {
    if (!processArgs(args)) {
        print out a usage message
        return
    }
    readGrid(new Scanner(new File(args[0]))
    do any additional setup
```

```
        draw the grid
        if (explore()) {
            print out that the puzzle is solved and iterations
        } else {
            print out that the puzzle is not solved and iterations
        }
    }
```

Based on this code, your program must contain the following three functions:

- `static boolean processArgs(String[] args)` – this function, similar to the corresponding function from Lab 3, processes the command line arguments and does any required setup. The command line arguments for this assignment are described below. This function returns `true` if all the command line arguments are OK.
- `static void readGrid(Scanner input)` – this function reads the input file defining the puzzle, which is described below, and creates the required internal data structures that represent the problem to be solved.
- `static boolean explore()` – this function recursively searches the grid to find a solution. It counts the number of times it is called. This function returns `true` if it can solve the puzzle and `false` otherwise.

The `explore` function uses a recursive backtracking search to solve the puzzle. In the text book, Chapter 12 describes recursion and section 12.5 describes recursive backtracking search. You must implement your solution similar to the solution described in section 12.5 for solving Sudoku. Here is pseudo-code for the algorithm you should to use:

```
boolean explore() {
    increment the number of calls

    // Find the next missing value, that is, the next number that
    // has not already been placed in the grid
    next = findMissingValue()

    // If no missing value was found, we've succeeded. Return
    // true indicating success
    if (next == -1) {
        return true
    }

    // Find the next known value. That is, the next value already
    // placed in the grid
    end = findNextValue(next)



    // start is the value prior to next. This value is guaranteed
```

```
        // to have already been placed in the grid
        start = next – 1

        // Loop through the eight possible adjacent locations
        // to the location of start
        for (nextLoc in adjacentCells(start)) {

            // Test to see if nextLoc is a viable location for
            // the value next. There are three conditions that
            // must be satisfied:
            //   1. nextLoc must actually be inside the grid
            //   2. the grid cell at nextLoc must be available
            //      to have next placed there
            //   3. the grid distance from nextLoc to the grid
            //      location of end must not be too great
            if (nextLoc inside grid
                and grid[nextLoc] is available
                and distance(nextLoc, gridLocation(end)) <= end – next) {

                // Assuming that everything looks good place
                // next at nextLoc and explore again
                place(nextLoc, next)
                if (explore()) {
                    // The search succeeded, return true
                    return true
                }
                // The search did not succeed, remove next from
                // nextLoc and try the next adjacent cell
                remove(nextLoc, next)
            }
        }

        // At this point we have failed, return false indicating failure
        return false
    }
```

There are a number of implications this code. Here are some of them:

- The functions `findMissingValue` and `findNextValue` assume that you can check if a value is already present in the grid.
- For both finding adjacent cells and computing distance between cells, you need to have a way to get the grid location of a value that is known to be in the grid.
- You need a way to find all the grid locations that are adjacent to a given location.
- Given a grid location, you need to be able to check if that grid location is available or already filled or not part of the grid (that is, is permanently unavailable).

Note that this implies that you have to be able to go in both directions between values and grid locations.

In addition to `readGrid`, `processArgs`, and `explore`, you are also required have the following functions as part of your solution:

- `static int findMissingValue()` – this function find the first missing, that is, unplaced, value that needs to be placed in the grid. This function returns -1 if there are no missing values.
- `static int findNextValue(int n)` – this function find the next value after the given value, `n`, that already has a location. It is assumed that this function is only called when it is known that the next value exists.
- `static void place(`*`location`* `cellLocation, int n)` – this functions "places" the value `n` at the given location. It has two responsibilities:
  1. It must call the `fillCell` method, as described in `DrawGrid.java`, below, to get the number `n` displayed on the grid.
  2. It must update your internal data structures to reflect that the given location has been filled with the given value.
- `static void remove(`*`location`* `cellLocation, int n)` – this functions "removes" the current value from the given location. It has two responsibilities:
  - It must call the `clearCell` method, as described below, to clear the number from the grid display.
  - It must update your internal data to reflect that the given location is now empty and available to have another value placed there.

  The extra `int n` argument to `remove` is provided if you need the information to complete update of internal data. You can include it or not as you see fit.
- `static int distance(`*`location`* `loc1, `*`location`* `loc2)` – this functions computes the distance between two locations. The formula for the distance between two location is:

```
Math.max(Math.abs(loc1 row - loc2 row),
         Math.abs(loc1 col - loc2 col))
```

  This is the minimum number of steps required to get from one location to another if there are no obstructions.

Input File

Your program must read a puzzle from a text file. The name of the file will be given as the first command line argument. If the file cannot be opened for input, your program must display an appropriate error message and terminate.

The input file is organized as follows:

1. The first line contains two integers, `h` and `w`, the height and width of the puzzle. The height and width will be between 2 and 20, inclusive. The sample grid, above, has height and width both equal to 10.
2. The remaining lines describe the puzzle. There will be exactly `h` lines. Each line will consist of

exactly `w` words (as defined by Scanner). Each word will be one of

a) `'x'` – indicating that this square is empty—that is, it will never be filled

b) `'.'` – indicating that this square has no value but is available to be filled with a value

c) `n` – where `n` is an integer, `1 <= n <= maxValue,` indicating that this square is pre-filled to the value `n`

Here is the input file (`hidtest3.txt`) for the example puzzle given above:

```
10 10
 .  . 46  . 50  .  .  .  .  .
 x  .  . 47 38 52 35  . 57  x
 x  x  .  .  .  . 34  .  x  x
 x  x  x  .  .  . 60  x  x  x
 x  x  x  x 31  .  x  x  x  x
 x  x  x  x  . 28  x  x  x  x
 x  x  x  .  .  .  .  x  x  x
 x  x  .  9  .  . 18  .  x  x
 x  .  6 10 12  . 19  .  .  x
 1  3  .  . 13  . 20 22  . 24
```

I have supplied ten sample puzzles with the assignment. **You can assume that the input is correctly formatted as described here.** That means you don't have to do any of the error checking (wrong number of lines, missing values, too many values, strings that are supposed to be integers but aren't, etc.) that you would ordinarily be expected to do.

Command Line Arguments

Your program must accept one or two command line arguments. If there are too many or too few arguments or the arguments are incorrect in some way, your program must produce an appropriate error message and exit. This processing is similar to what you did in Lab 3 except that the second command line argument is optional.

The command line arguments are:

1. The name of the file containing the puzzle description. This argument must always be present and must name a readable file. The format of this input file is described in the previous section.
2. A value for the sleep time for the drawing numbers. The `fillCell` method you will call to fill a square on the drawing has a built in delay. You will want to adjust the speed of this delay depending on factors such as the complexity of the puzzle and whether you are debugging or checking to make sure your program works. This argument is optional. If this argument is provided, it must be a non-negative integer. If this argument is not provided, you should use the default delay time that is given by `DrawGrid`. (In the latter case, you don't need to call the `setDelay` method.)

DrawGrid.java

With the assignment I have provided a module, `DrawGrid.java`, that you must use to draw the puzzle and animate the search. (Most of the following description is duplicated from the prologue comments

in `DrawGrid.java`.) To use the class:

1. Place the `DrawGrid.java` and `DrawingPanel.java` in the same folder as your `Hidato.java` program. You do not need an import statement for `DrawGrid`. If you are using jGrasp (or another IDE) to build and run your program, jGrasp will automatically compile these two files. If you are using the command line, you should compile these two files.
2. In your program, call the constructor `DrawGrid(int h, int w)` which creates a grid with the given height (`h`) and width (`w`).
3. For every square in the grid call one of:
   - `fixedCell(int r, int c, int value)` declaring that the cell at coordinates `(r, c)`, `r` = row, `c` = column, `0 <= r < h`, `0 <= c < w`, has the given fixed value. This corresponds to an integer in the input file.
   - `valueCell(int r, int c)` declaring that the cell (square) at coordinates `(r, c)` is a cell where a value is to be filled in. This corresponds to a `'.'` in the input file.
   - `emptyCell(int r, int c)` declaring that the cell at coordinates `(r, c)` will be empty, that is, not part of the grid. This corresponds to an `'x'` in the input file.
4. Once step 2 is complete, call `draw()` to create and display the grid.
5. While searching in explore:
   - In `place()`, call `fillCell(int r, int c, int n)` to indicate that the value `n` should be filled in the cell at `(r, c)`. `(r, c)` must be the location of a value cell from a call to `valueCell` in step 3.
   - In `remove`, call `clearCell(int r, int c)` to clear a previous value from the cell at `(r, c)`. Again, `(r, c)` must be a value cell.

Here is an example of how to use the class (mostly Java):

```
int height = height of grid;
int width = width of grid;

DrawGrid g = new DrawGrid(height, width);

for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        if (this is cell has a fixed value) {
            g.fixedCell(row, col, value);
        } else if (this cell is an empty value cell) {
            g.valueCell(row, col);
        } else {
            g.emptyCell(row, col);
        }
    }
}

g.draw();
```

While searching the grid:

in `place()`: `g.fillCell(r, c, n);` // Places the value n at (r, c)
in `remove()`: `g.clearCell(r, c);` // Removes the value at (r, c)

After filling the cell at `(r, c)` using `fillCell`, `DrawGrid` will delay for a fixed time. A call to `g.setDelay(int delayTime)` will set the delay time to the given number of milliseconds. So, a one second delay after each step is obtained by calling `g.setDelay(1000)`. The default delay is 50 milliseconds.

If you want to see how `DrawGrid.java`, feel free to look at it. This is not required to complete the assignment.

TestData

Eight sample puzzles have been included with the assignment. These should give you more than enough samples to test your code. Here's a brief description of the different grids:

- `hidsimple1.txt` – a simple search. 2 can go in one of two places. Only one of those allows you to place the three
- `hidsimple2.txt` – the mirror image (vertically) of `hidsimple1`
- `hidsimple3.txt` – a simple puzzle with no solution
- `hidsimple4.txt` – a simple puzzle with no solution. There are two location for 2 and one or two for 3 but no place for 4.
- `hidsimple5.txt` – a simple puzzle with a solution. This puzzle in non-square.
- `hidtest1.txt` – a 10x10 puzzle from the newspaper. This one is fairly difficult for both people and the machine as a search (22,715 iterations for my program)
- `hidtest2.txt` – a 10x10 puzzle from the newspaper. This one is fairly difficult for both people and easy for the machine to search (524 iterations for my program)
- `hidtest3.txt` – a 10x10 puzzle from the newspaper. This one is fairly easy for people but difficult for the machine to search (10, 177 iterations for my program)
- `hidtest4.txt` – a 10x10 puzzle from the newspaper. This one is fairly easy for people but potentially really difficulr for the machine to search (337,314 iterations for my program)

I have also include images for the solutions to `hidtest1` through `hidtest4`.

Design Decisions

1. The type of *`Location`*

   The functions `place`, `remove`, and `distance` have parameters of type *`Location`* and the `explore` function must be able to find the location of a value given a value. What is the actual type of *`Location`*? Here are two options:

   a) an `int`. The integer could encode the location as follows: row `r`, column `c` becomes the integer `w * r + c`, where `w` is the width of the grid. This would work well with a choice to use a one dimensional array to represent the grid (see next item). However, `fillCell` and

`clearCell` require `r` and `c`. So, you will have to be prepared to convert between representations. You will probably want to write helper functions that convert an `(r, c)` pair to a *`location`* and a *`location`* to a row number or column number.

b) an `int[2]`, where the first element in the array is `r` and the second is `c`. This is not an ideal solution and you'll end up with a lot of code like `clearCell(loc[0], loc[1])` scattered through your program. If you do this, it is helpful to use the following expression which will create and initialize an array:

```
new int[] {r, c}
```

which will create and initialize a new `int[2]` with values `r` and `c`. Here's an example of how you can use this:

```
return new int[] {r, c};
```

Don't commit to this decision before you consider the next one as well.

2. One or two dimensional grid array

   As discussed in the prior item, you can map a two dimensional grid into a one dimensional array. Do you want to do this or do you want to keep it two dimensional? This decision may heavily influence your choice for the actual type of *`location`*, above. Note, however, that you can use a single `int` for location and still use a two dimensional array for the grid, or vice versa.

3. Finding the location of a value

   There are two possible approaches to mapping from values to locations:

   a) Search the grid for the location of the value.

   b) Keep an extra array indexed by value that gives the location. This array is a one or two dimensional array of `int`s depending on your decision for item number 1.

   The second option (b) makes the code for finding the location simpler at the cost of having to update the array when a new value is placed on or removed from the grid.

Suggested Steps

This is a fairly complicated program with a lot of pieces. If you try to build it all at once, you **will** have problems. You need to build and test it a piece at a time.

While you are doing the assignment, you should consider adding print statements to your program to make sure things are working the way you expect. Make sure you remove or comment out these extra print statements before you turn in your program.

1. Write `processArgs`

   Write the program to process the command line arguments. Put the results into static variables. This should look a lot like the functions from Lab 3.

2. Write `readGrid`

   Having the file that is to be read, read it in and store the structure of the grid in your internal structures. Don't worry at this point about all the structures you might want, just build enough

that you can store the structure of the grid—which squares are empty, available, and already specified.

3. Draw the Grid

   Having the structure of the grid, initialize the `DrawGrid` class, and make the needed calls to `fixedCell`, `valueCell`, and `emptyCell`, and `draw` to get the drawing displayed. You should get something that looks like the first figure.

4. Write `explore` and get it working.

5. After completing step 4, run all the test cases to ensure that everything is working properly. After that, you're done!

References

1. Chapter 12, Recursion, especially, 12.5, Recursive Backtracking.
2. Chapter 7, Arrays, in the textbook. Section 7.5 discusses multi-dimensional arrays.

Coding Standards

Your program should follow the same standards described in Lab 1. For this assignment, the standard: "Use comments at the start of each method to describe the purpose of the method and the purpose of each parameter to the method" is particularly important.

Your program will be graded on conformance to the coding standards as well as correct functionality.

Turn in your program to Canvas

Look for the Assign 1 link on the Assignments Canvas page. You should only turn in your `Hidato.java` source program. Do not turn in either `DrawGrid.java` or `DrawingPanel.java`.

Grading

Your program is due by 11:59pm, Wednesday, February 8. The grading will be:

- 50% for the program working correctly including solving solvable puzzles, handling the input file and command line arguments, drawing the grid and animating the search, and solving puzzles with sufficient speed
- 30% for correctly following the requirements above, including having all the required functions
- 20% for following coding conventions as described in lab 1.

There will be a deduction for not turning in the assignment properly.