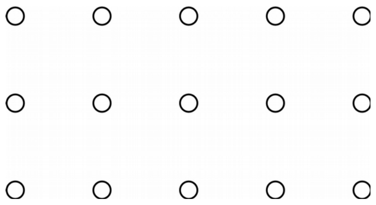
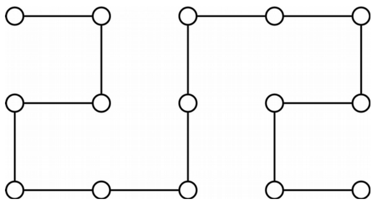
# COSC 2P05 — Assignment 3

## (Hilarious pun related to threading. Maybe a sewing joke?)

Quick! What's your favourite thing in the world?
(You said *combinatorial optimization*, right?)
Combinatorial optimization is exactly what it sounds like: when you have some problem that requires guessing at the best combination of discrete choices, you want to pick the best choice sequence. A classic example is the Traveling Salesman Problem (TSP; where you want to choose cities to visit, such that you spend the least time on the road).
Oftentimes, finding the *optimal* combination (for a sufficiently large problem) might be computationally intractable, so you'll settle for a reasonable guess.
For this assignment, you'll be writing a program with a simple Java Swing GUI, which makes use of *threading*, to optimize a *self-avoiding walk*.

## Self-avoiding Walks

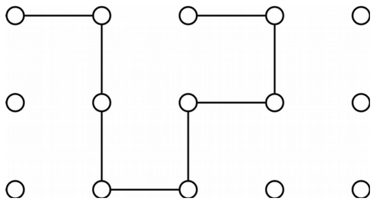
Suppose you had a grid. From some starting point (let's say the upper-left corner), you want to visit as many of those points as possible, using orthogonal paths, without revisiting any vertices (i.e. without crossing your own path). e.g.:

For a true self-avoiding walk, you'll visit each vertex once, without ever revisiting a vertex. As such, if you have N vertices, you can expect a walk of N-1 connections. In practice, this problem can be used as a test for various heuristics and algorithms, in which case you actually score an attempt according to 'how close it got' to ideal.

For your walk, based on some list of instructions (e.g. N/E/S/W, or Forwards/Left/Right, etc.), you'll gauge a candidate solution according to how many vertices it could visit *before* either falling off the grid or hitting an already-visited spot.

Assume the following attempt for a walk: ESSENENW**W**EESWN
As we can see below...



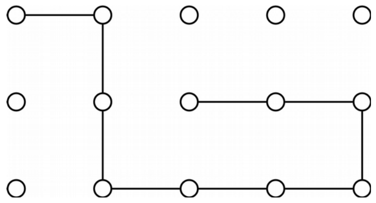
The ninth step would take it to an already-visited vertex, so we stop there. This walk has a 'score' of 9 (the minimum possible score is 1, since you start already on a vertex).

For your program, you'll be starting with some random sequences, and improving upon them. ...how?

## Hill Climbing

A *hill climber* is one of the simplest optimization methods, as it's mostly based on making random alterations to a possible solution, and only keeping those changes if the new version is 'better'.

In the example above, suppose the fifth step had been randomly selected for a change to a new symbol (say, to East, instead of North). The new sequence would be ESSEEENWWEESWN, which would produce:



This version visits 10 vertices instead of the old best (9), so we adopt this new sequence to replace the old. Suppose a further random alteration were to replace the first direction with West. That would obviously fall off the grid immediately, yielding a score of only 1 (far less than the current 10), so the hill climber would reject the new sequence (sticking with ESSEEENWWEESWN, instead).

And... that's it.

How should one create the new sequences based on the old? That's outside the scope of this course (coming soon, in 3P71!). What's described above (picking a single thing to change at a time) is fine enough for this assignment. You could alternatively have some small percentage chance of replacing each direction with a new one (potentially leading to more diverse sequences per iteration). e.g. if there were a 25% chance per-direction, you'd expect, on average, each new sequence to have 3 or 4 modified directions.

## Parallel Hill Climbing

Assuming you're okay with the idea of a hill climber, imagine several of them, working almost entirely independently. This principle is the starting point for some forms of evolutionary computation. For now, consider this:

- If you take the hill climber and put it into a separate `Thread` (defined in its own class), you could then have multiple such worker threads operating in parallel
- You'd need to be careful about resources shared across all threads:
  - The variables storing everything related to the 'current best' is true across all threads, so they all need *access* to those datas
  - The visualization of the 'current best' candidate solution needs to be updated when a new 'best' is found. What's more, that solution can't be modified *while* it's being drawn (if a new best happens to be found part-way through a drawing cycle, it'll need to wait its turn)

## Requirements

Mostly, you just need what's listed above: a Java Swing-based program, that uses a parallel hill-climber to try deriving directions for a self-avoiding walk of some size.

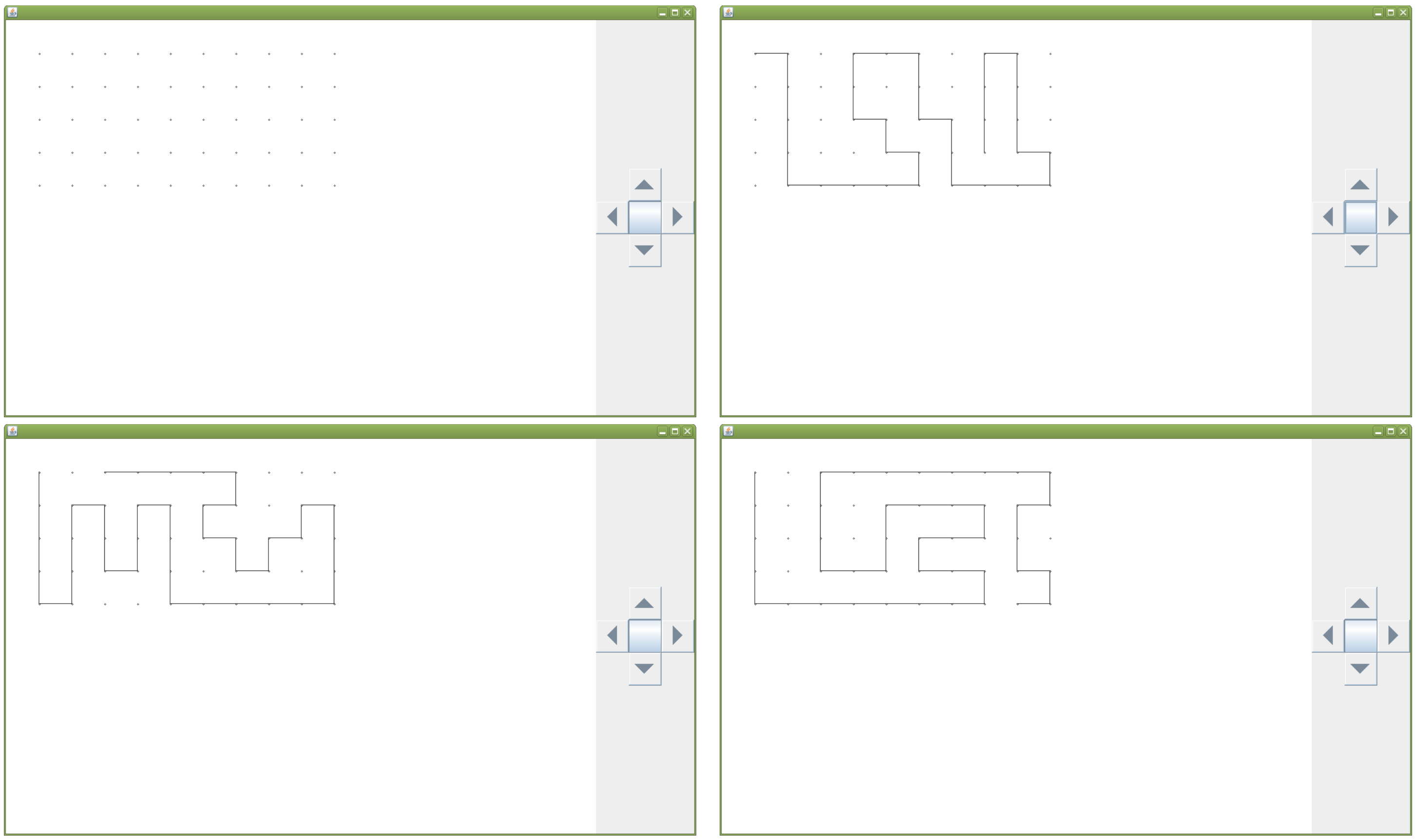
- You'll need the ability to increase/decrease the height and width of the grid
  - Assume a maximum height of 10 vertices, and a maximum width of 16 vertices
  - The minimum size should be 2×2
  - Presumably, it'll start over whenever you change the size
- You should also have some form of suspend/resume
  - It's fine (probably best) to 'start over' whenever it resumes
- It must be *threaded*:
  - Use 4 threads
  - Each thread should use the same algorithm (some hill climber, per above)
- The hill climbers start out with a randomized candidate solution, and each tries to improve that
  - The current 'best solution' across all threads is rendered on the grid
  - You don't need to actually display the score, though you'll obviously need to maintain them internally

## Tips and suggestions

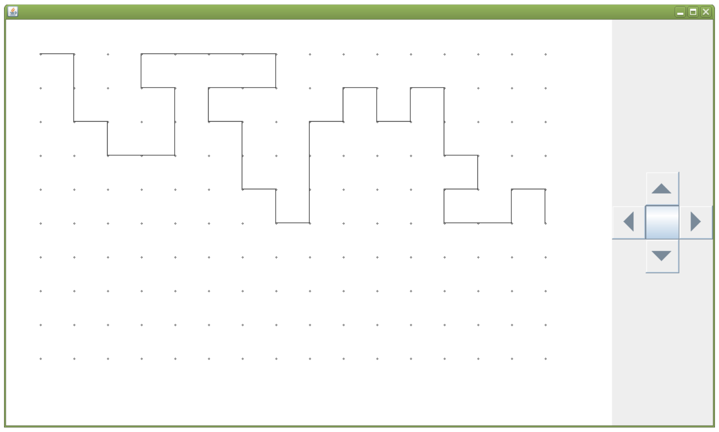
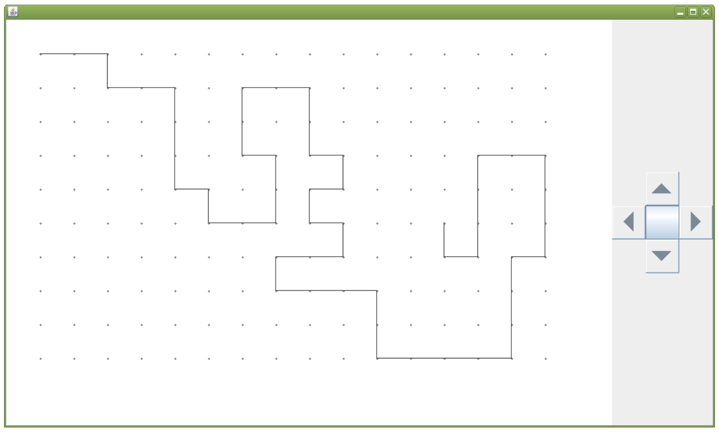
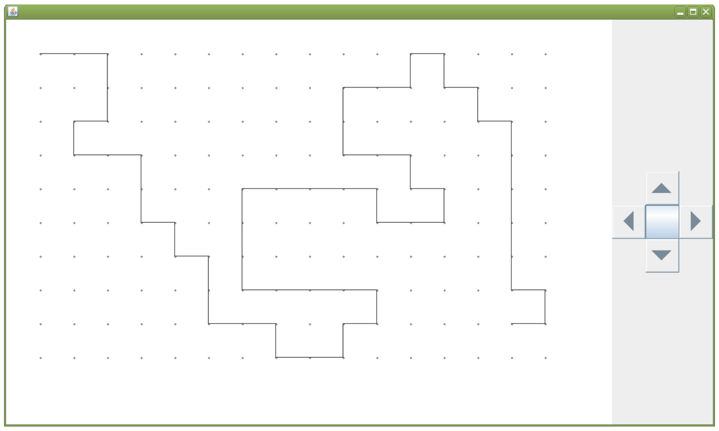
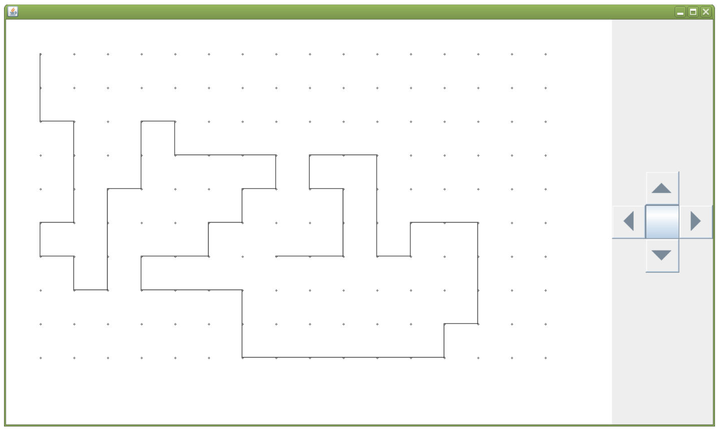
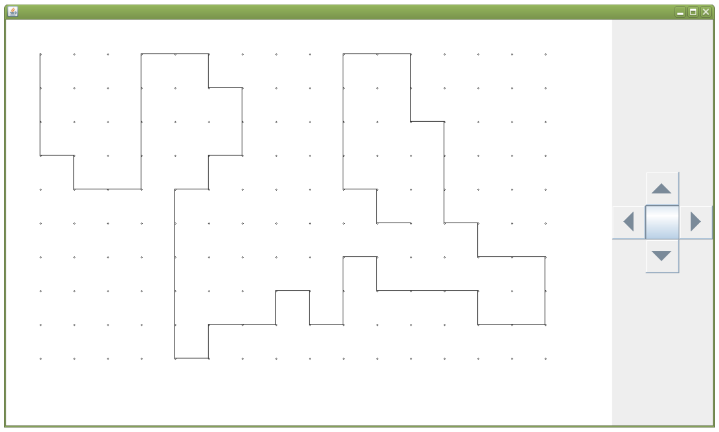
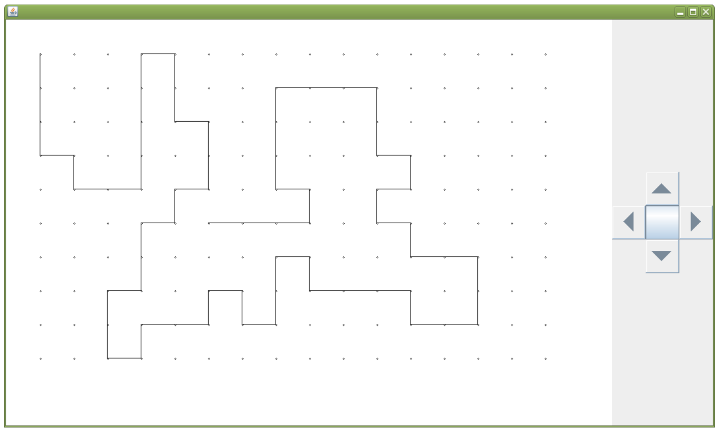
- Seriously, we don't actually care about the scores directly; we just use them to recognize improvements
- There are different possible mechanisms for exploring new solutions. You don't even need to specifically go with one of those mentioned (though those are probably the easiest)
    - Note that there aren't any marks assigned to 'how good' yours is, beyond simply *deriving new solutions*. Pick one you like
- All you need is threads showing changes over time. A *really* bad mechanism won't find many changes (which tend to happen at the very beginning), so if you don't want to make a 'good' mechanism for finding new paths, you can just add a `sleep` to the worker threads so they'll find those few improvements slower
    - If this isn't clear: the marker needs to see that you have threading working, with improving solutions; that's all. If your algorithm doesn't improve much, you can just stretch it out
- *Thread safety* is important! (Hint: *mutual exclusion*! It's worth 2 in the marking scheme for a reason)
- You've been provided a couple extra code examples here:
  https://www.cosc.brocku.ca/~efoxwell/2P05/code/extrasForA2/
  These are *not* for this assignment specifically, but rather just some additional sample code for Swing
    - (Seriously, things like a JOptionPane are really neat, but severely overkill for this!)
- The precise user interface is up to you, so long as you have the option of choosing the grid size, starting/stopping, seeing the 'current best solution', and performing multiple runs per execution of the program. Below is *just one possible sample*

## Possible sample execution

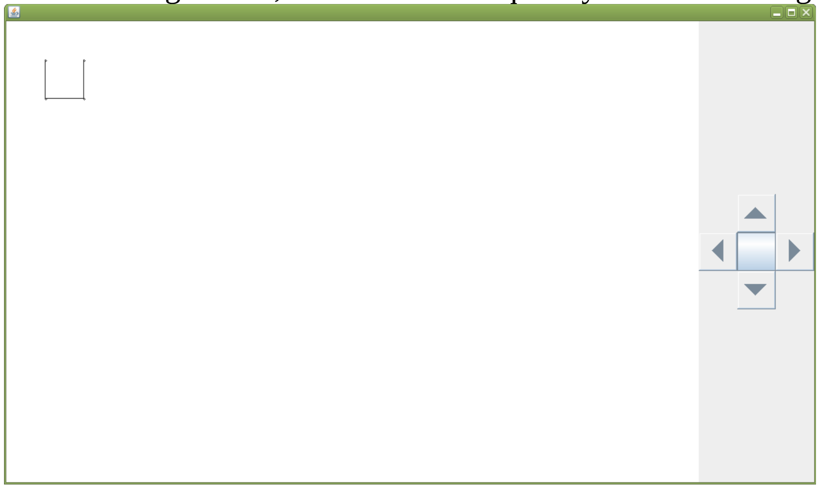
Note that it's a pretty simple interface. (You don't need to use arrows. I just like simplicity when possible)



Next, I tried increasing the grid size and trying again.

I did also do a run with the minimum grid size, but it solves too quickly to be interesting:



## Marking

*Please* note that, outside of things failing catastrophically, or being comically and confusingly wrong (e.g. if you try filing your taxes as your submission), you don't need to worry about requirements not covered below:

- **1** — properly displaying a window with the grid and controls (regardless of solution)
- **1** — control-related functionality: resizing grid, starting/stopping, etc.
- **1** — ability to display a solution on top of the grid
- **1** — ability to derive new solutions
- **2** — adding threading to calculations
- **2** — proper mutual exclusion
- **1** — style and code/design conventions
- **1** — operational correctness (not 'breaking', being able to stop the program, etc.)

Reminder: The marker still needs to know who you are (name and student number)

## Submission

You'll be submitting through Sakai.
Bundle everything up into a single **`.zip`** file, and submit that.
Note: `.zip`.
**Not `.rar`**.
Not .7z.
This is not 1993.

“Everything” includes all of your source files, and project files if necessary. If you're uncertain if the marker will (easily) be able to compile/run it, include instructions.

If there's anything else the marker needs to know, put it into either a `readme.txt` or a `readme.pdf`.