

COMP30050: Software Engineering Project 3

Dr. Tony Veale

Final Report

Due on 29th April 2016

Lamp

Joe Duffin - 13738019

Edwin Keville - 13718661

Niamh Kavanagh - 12495522

Gerard Fogarty - 13303911 (not in attendance)

Contents

Introduction	2
What we produced	3
Overview	3
The Graphical User Interface	4
The Command Line Version	4
Class Diagrams	6
The Gui with Simulated Annealing and the Genetic Algorithm	6
Cli for Simulated Annealing	7
Cli for the Genetic Algorithm	8
Performance Analysis	9
The Genetic Alogrithm	9
The Simulated Annealing alogrithm	10
The Hybrid Algorithm	11
Comparing the Three Algorithms	12
Development Phases/Issues	13
The Design Pattern	13
The Genetic Algorithm	14
The Simulated Annealing Algorithm	16
The Interfaces	17
Development Cycle Conclusion	17
Our Development Model	18
Not sure what needs to go in here	18
Technical Details Of Note	19
Our Design Pattern	19
Callback listeners	19
Inheritance	20
Threads	20
Team Development	21
What went well	21
What didn't go so well	21
Who did what	21
The Team Night Out	22
Burgers and Drama	22
A Sad Day For Lamp	22
The loss of a team member	22

Introduction

As part of our third year software engineering module, all students were required to develop a stochastic search project allocation system. This is to be completed in groups, with the initial framework designed by each individual member through weekly assignments, before amalgamating each member's solution and adding the functionality of two different solution algorithms. The end result produces an optimal allocation of data in a user-friendly format which can be saved as desired.

The project itself involves reading in a list of students and their ranked list of preferences for final year projects within computer science. Some students may have a project preassigned to them, in which case they will have no other preferences and will not be altered in the allocation process. Any students without a preassigned project will then be assigned a project using our chosen algorithms in an attempt to minimize overall student disappointment.

The proposed solution for this mapping of students to projects was developed using both a simulated annealing and a genetic algorithm. The simulated annealing approach provides a singular solution, which it adapts through a series of changes to individual student-project mappings until a good viable solution is achieved. The genetic algorithm creates a population of random solutions, with solutions merging and being culled until a single optimal solution is reached.

To allow our software to be user-friendly, we created a Graphical User Interface and a Command Line Interface. The Gui allows users to load the data file of their choice and run our hybrid algorithm composed of a genetic algorithm populated by simulated annealing solutions, then gives the option to save these results into a spreadsheet format with the energy of the solution displayed at the bottom. The CLI uses command line arguments provided by the user to specify the desired input file, which algorithm to be used, the parameters for each algorithm and the desired format of the output file.

The following report details our software in full, discussing both algorithms used and the hybrid we created of both, a user manual for accessibility, and the entire development process undertaken to create the final project.

What we produced

Overview

In keeping with our core values, we wanted to produce an ultimate solution with lots of modularity and re-usability. We designed our product in such a way that that many different interfaces could use it with ease. This allowed us to produce not only a Graphical User interface (Gui), but a Command line user interface (Cli) as well. If in the future a different user interface, android for example, was needed, it would be very easy to implement.

During the creation of the solution, by either the Gui or Cli, the user is kept informed of the algorithm's progress by means of a status update text area and a progress bar. Any interface which uses our solution will have to implement the associated methods, and the algorithms will update them as necessary.

After a solution is completed, the user is informed of the overall energy of the solution and is given the option to save the solution (in the case of the Cli the solution is automatically saved in the specified output file). Saving creates a tsv file which lists each student along with their assigned project. There is also an indication as to which number preference they received (1 being their 1st preference and 10 being their last preference).

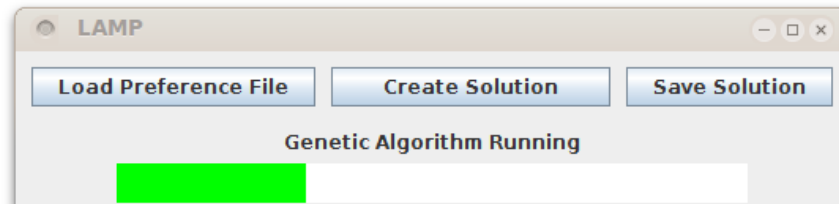
Section of Example Output

	A	B	C
1	Name	Assignment	Ranking
2	Conan the Barbarian	Spotting Plagiarised Essays Economically	2
3	Queen Elizabeth I	Implement a Multimedia Archive	2
4	Loki Laufeyson	3D printing and augmented reality systems in novel vascular models	1
5	Jor El	PlotBot	1
6	Wilkins Micawber	Literature Review Assistant	1
7	Severus Snape	Route Visualization for Indoor Navigation	2
8	Charlie Harper	Reverse engineering of printed stenographic images	2
9	Mr. Darcy	Recommending Movies Using Curated IMDb Lists	1
10	William Bligh	Twitter Network Analysis	1
11	Eric Cartman	Building a 3D room from a Kinect carrying Drone	2
12	Snake Plissken	Review-based Recommendation	1
13	Fletcher Christian	Visualisation of Egocentric Networks	1
14	Red Sonja	Forensic Disk Image Reconstruction from Deduplicated Data Storage	2
15	Mr. Edward Hyde	ASTRA on Android	1
16	Jesus Christ	Biography Reading Media Assistant	1
17	Ebenezer Scrooge	Reversible Watermarking Benchmark	2
18	Alan Turing	Age related interaction differences	1
19	Travis Bickle	Forensic Analysis of P2P Instant Messaging	2

The Graphical User Interface

The simplicity and re-usability of our code was extremely important to us throughout this project, as discussed in our core values. To allow users to easily navigate their way through our software, regardless of its application, we created our Gui following these requirements. The Gui contains only three buttons - load a preference file, create a solution and save the created solution. Once the algorithm begins, a progress bar updates within the Gui to inform the user of the current status. This interface makes use of our hybrid solution - using simulated annealing to create the population for the genetic algorithm. This is discussed in further detail in Performance Analysis.

The Graphical User Interface



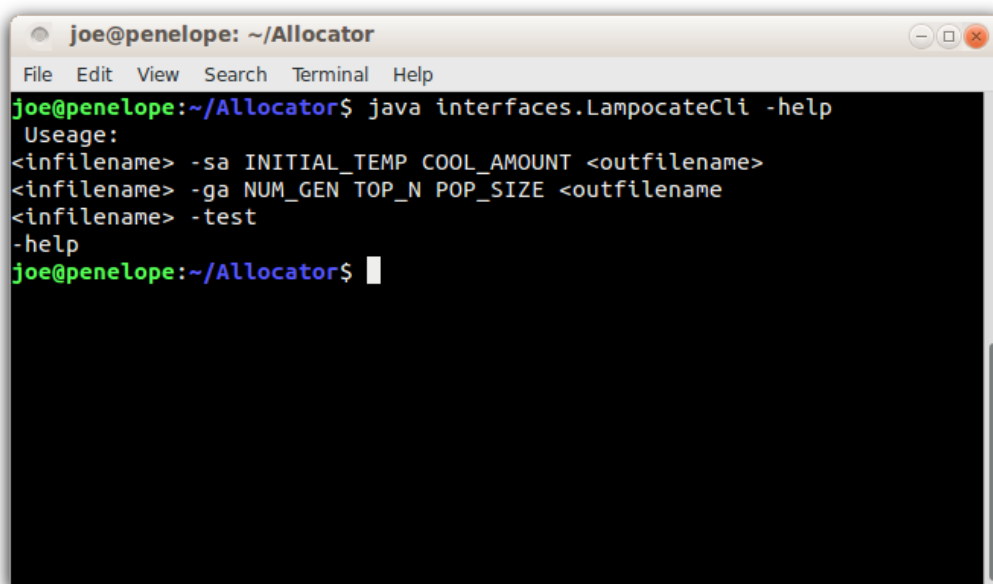
The Command Line Version

This version of our software solution is for the advanced user. It gives the user the ability to specify which of the algorithms to run and under which parameters. These options are passed as command line arguments to the program, as well input and output file names. There is a help option which can be displayed with the "-help" tag.

The data used for performance analysis was created with the command line version of our program with the "-test" tag. This instructs the program to run each of the three algorithms (Simulated Annealing, Genetic and Hybrid) numerous times with varying parameters. The results of these tests are output to three separate tsv files.

To use the Cli, specify "-ga" for the genetic algorithm or "-sa" for the simulated annealing algorithm with the relevant parameters as specified below.

The usage message

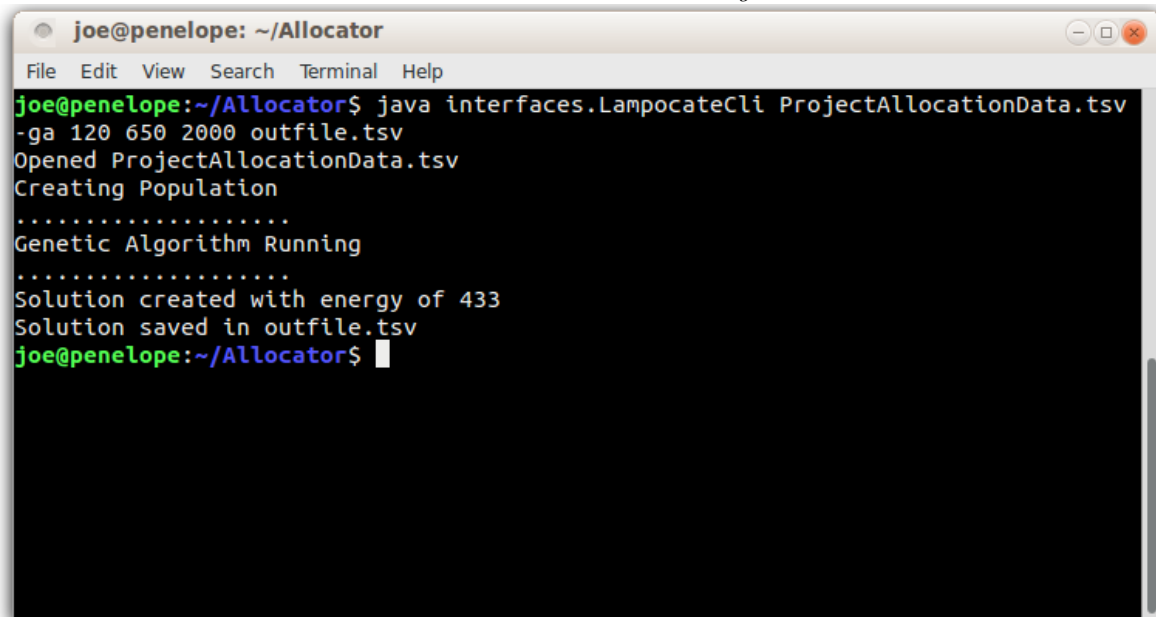


```
joe@penelope: ~/Allocator
File Edit View Search Terminal Help
joe@penelope:~/Allocator$ java interfaces.LampocateCli -help
Usage:
<infilename> -sa INITIAL_TEMP COOL_AMOUNT <outfilename>
<infilename> -ga NUM_GEN TOP_N POP_SIZE <outfilename>
<infilename> -test
-help
joe@penelope:~/Allocator$
```

Example program runs

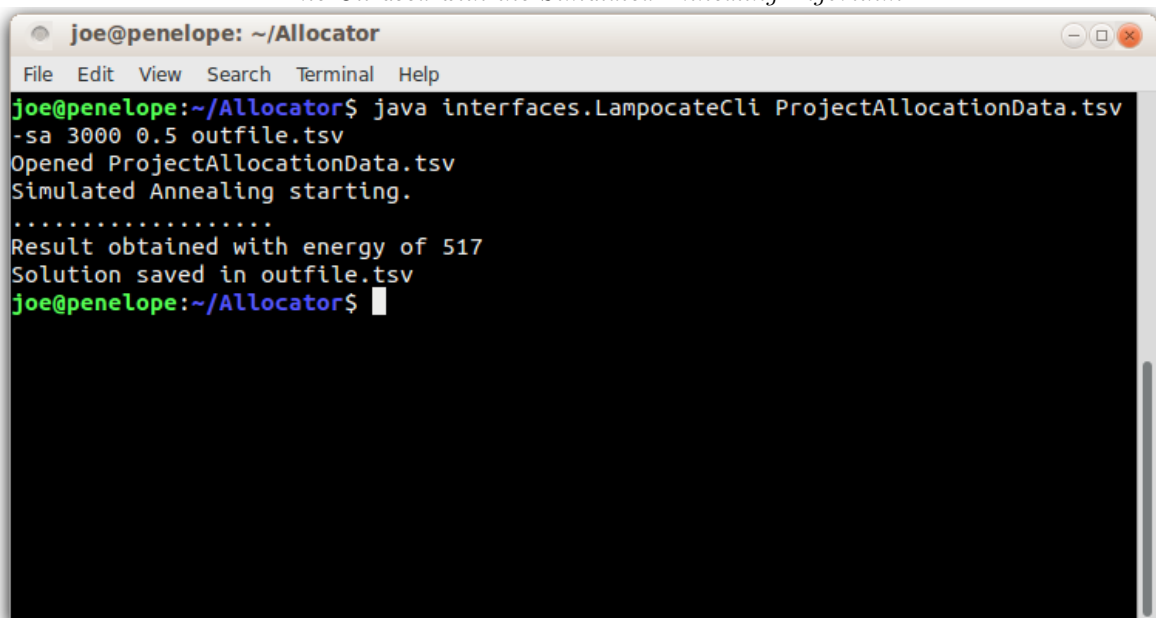
Note: Where the graphic user interface has a progress bar and text field for status updates, the Cli version of our solution uses the console. As the algorithm progresses, the sequence of dots grows in proportion with the algorithms progress and status updates are presented as lines of console output.

The Cli used with the Genetic Algorithm

A terminal window titled 'joe@penelope: ~/Allocator' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:

```
joe@penelope:~/Allocator$ java interfaces.LampocateCli ProjectAllocationData.tsv
-ga 120 650 2000 outfile.tsv
Opened ProjectAllocationData.tsv
Creating Population
.....
Genetic Algorithm Running
.....
Solution created with energy of 433
Solution saved in outfile.tsv
joe@penelope:~/Allocator$
```

The Cli used with the Simulated Annealing Algorithm

A terminal window titled 'joe@penelope: ~/Allocator' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:

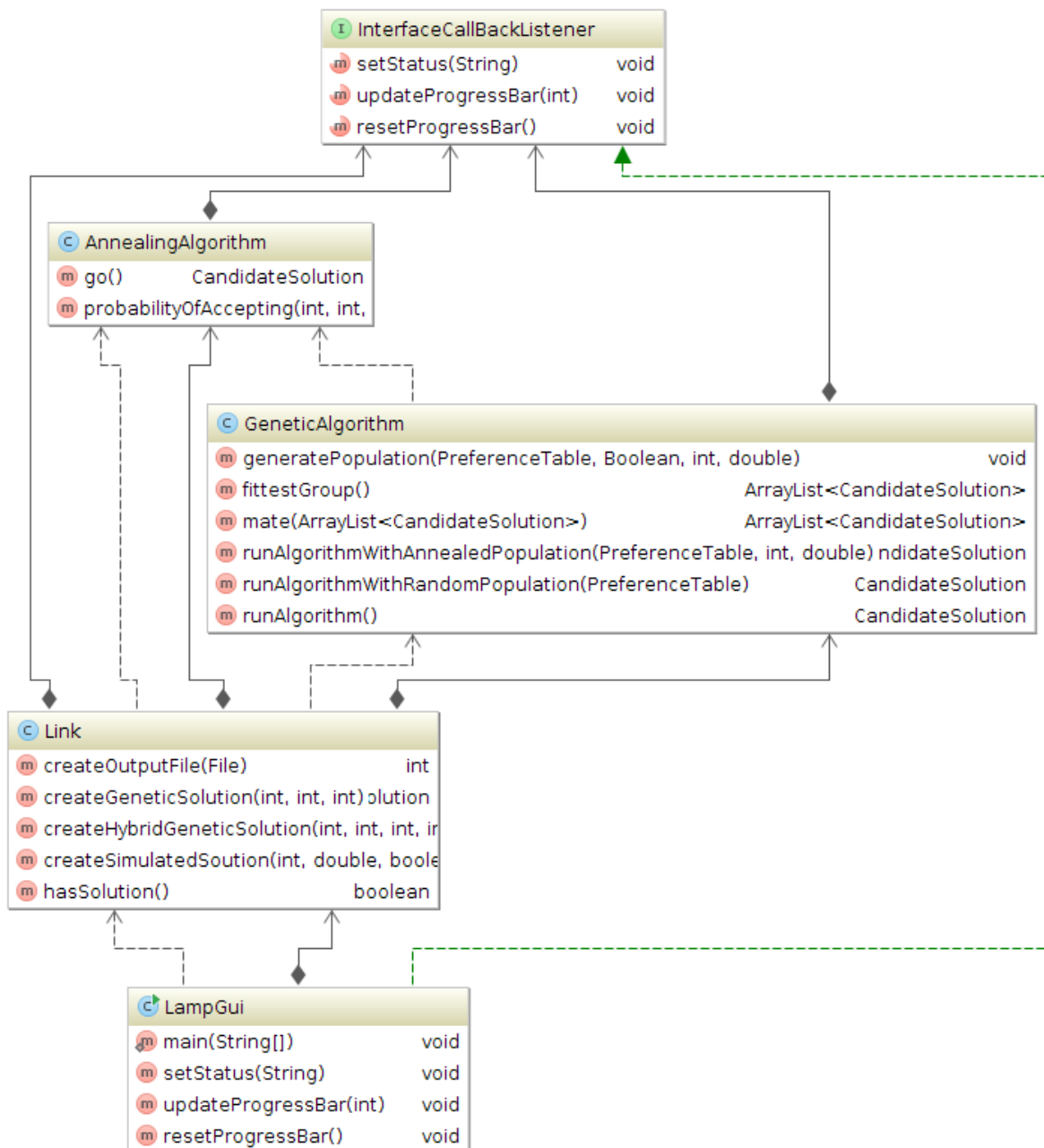
```
joe@penelope:~/Allocator$ java interfaces.LampocateCli ProjectAllocationData.tsv
-sa 3000 0.5 outfile.tsv
Opened ProjectAllocationData.tsv
Simulated Annealing starting.
.....
Result obtained with energy of 517
Solution saved in outfile.tsv
joe@penelope:~/Allocator$
```

Class Diagrams

In this section we present certain case specific arrangements of class diagrams. These demonstrate the modularity of our solution.

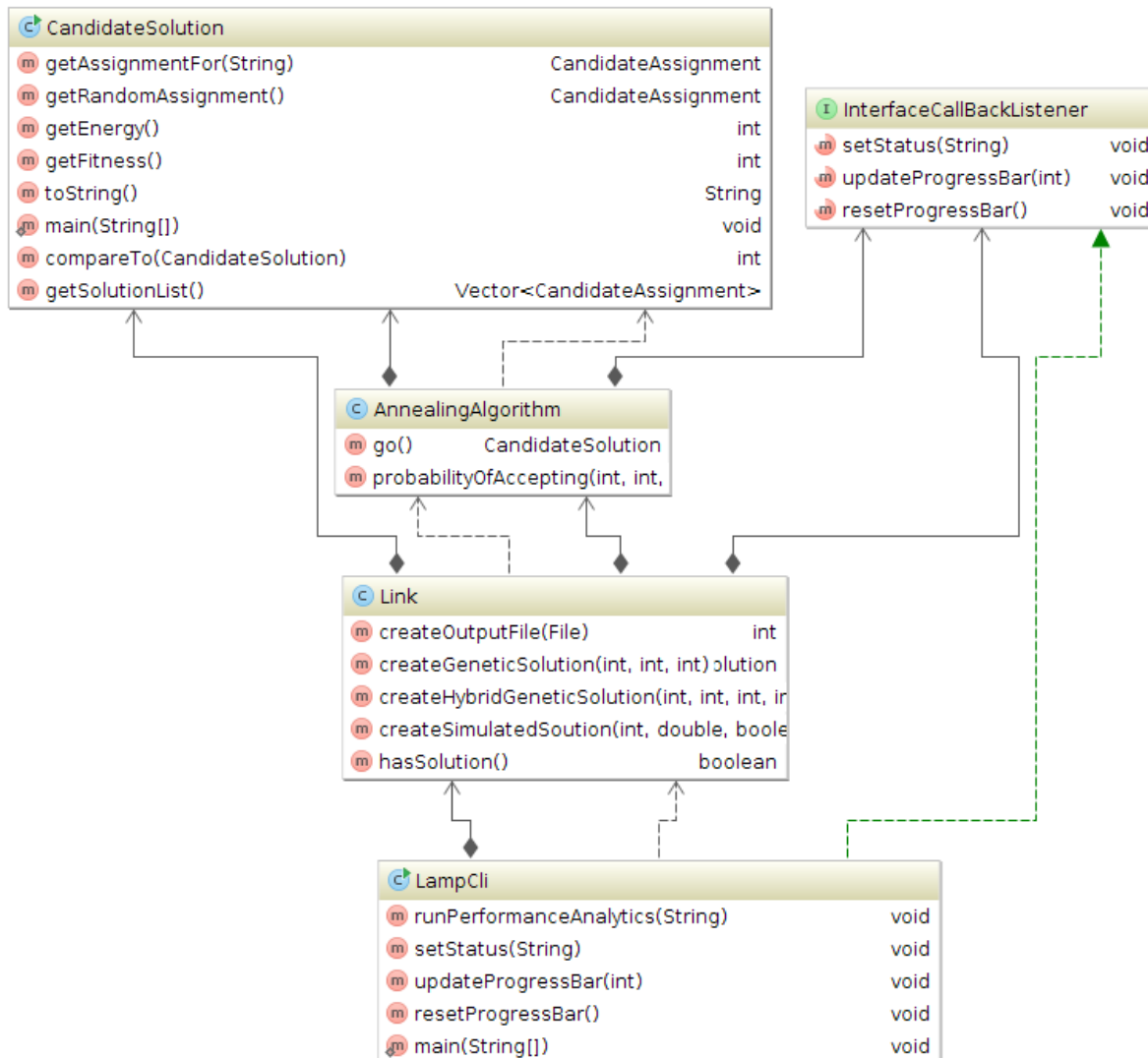
The Gui with Simulated Annealing and the Genetic Algorithm

The Gui relies on the Link to get access to both the Simulated Annealing and Genetic Algorithms. The 1 to 1 relationships between the CallBackListener and the algorithms allow the algorithms to update the status bar and text on the Gui.



Cli for Simulated Annealing

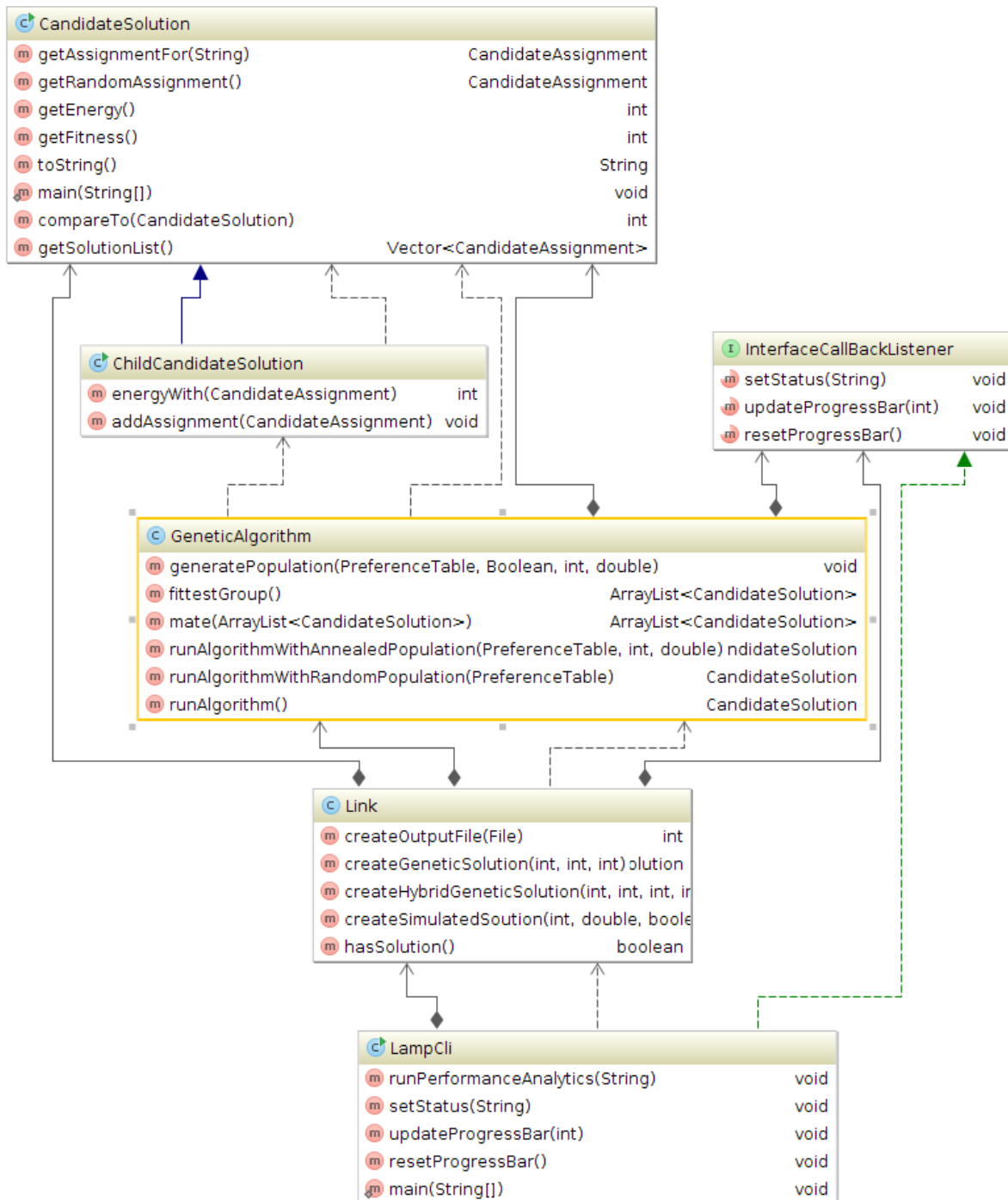
The Cli has a 1 to 1 relationship with the Link, which is shown here to give access to the Simulated Annealing algorithm. This, in turn, has a 1 to 1 relationship with a Candidate Solution which it permutes.



Powered by yFiles

Cli for the Genetic Algorithm

When the user elects to use the Genetic Algorithm, the 1 to 1 relationship between the Link and the Genetic Algorithm is used. In the diagrams below the inheritance relationship the ChildCandidateSolution and the CandidateSolution is evident.



Performance Analysis

Each algorithm has its own set of parameters which are needed and these have a significant effect on the overall energy of the given solution. We wanted our Gui to be a simple no frills program to give the ultimate solution so we removed the option of user input parameters. They are set as constant. In order to determine the best constants we ran overnight tests, subjecting each algorithm to a barrage of parameters. These results can be reproduced using the '-test' tag with in Cli version of our code.

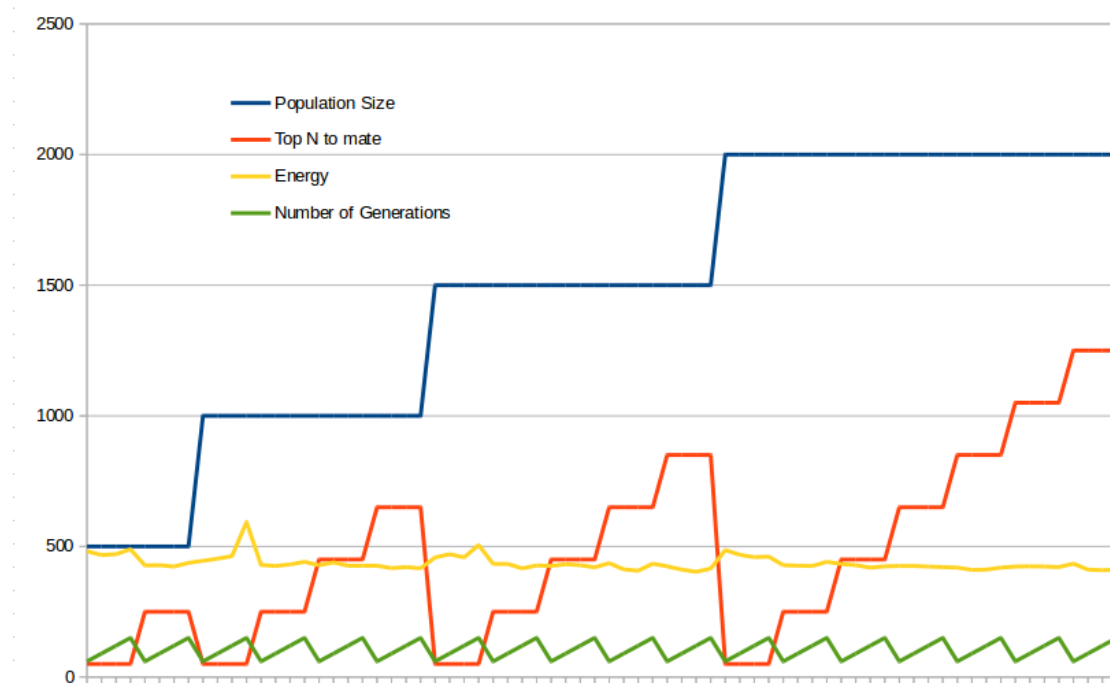
If an advanced user wishes to specify their own parameters, the Cli version of our program will facilitate that.

The Genetic Algorithm

The Genetic Algorithm has 3 parameters:

- The initial population size
- How many of the population to choose for mating (The top N)
- The number of generations the population should evolve through

The overnight test yielded the following graph.



The algorithm was ran with a variety of population sizes, number to mate and number of generations and the best parameters which gave the lowest energy were extracted.

Population Size: 1500

Top N: 850

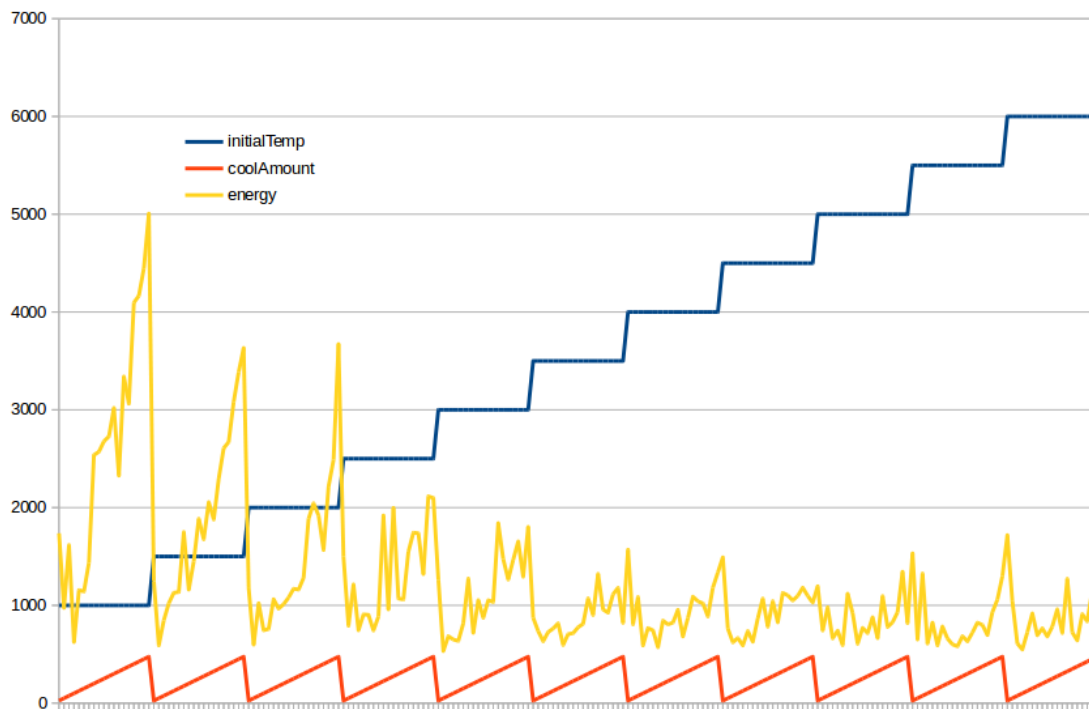
Number of Generations: 120

The Simulated Annealing algorithm

The Simulated Annealing Algorithm has 2 parameters:

- The initial temperature of the solution
- The amount the solution cools on each iteration.

The overnight test yielded the following graph.



The algorithm was ran with a variety of cooling amounts for a variety of initial temperatures. The cool amount has been scaled by a factor of 100 for clarity. The best parameters are shown below.

Initial Temperature 3000

Cooling Amount 0.5

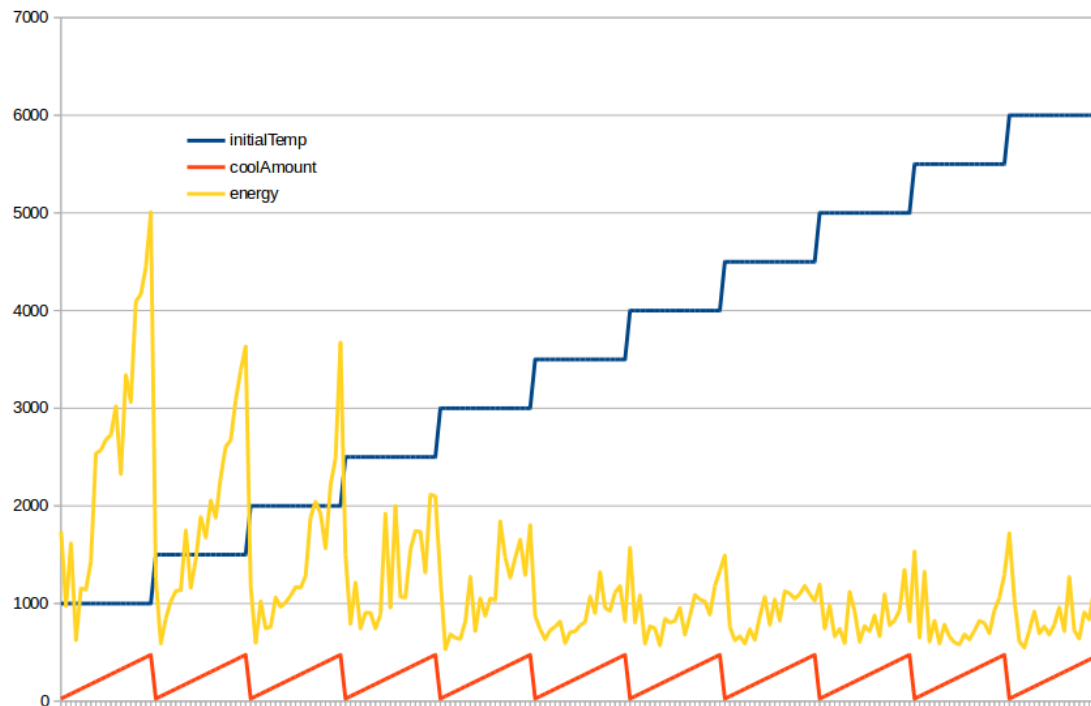
The Hybrid Algorithm

The Hybrid Algorithm uses both algorithms to create a solution. The primary driver is the Genetic Algorithm but instead of a random initial random population, a population of solutions is created using simulated annealing.

The overnight test for this algorithm permuted over all 5 parameters:

- The initial population size
- How many of the population to choose for mating (The top N)
- The number of generations the population should evolve through
- The initial temperature of the solution
- The amount the solution cools on each iteration.

The overnight test yielded the following graph.



The algorithm was ran with all permutations of the 5 parameters and we found these to produce the lowest energy. These are the parameters that we used in the Gui to attain the ultimate solution.

Population Size: 2000

Top N: 650

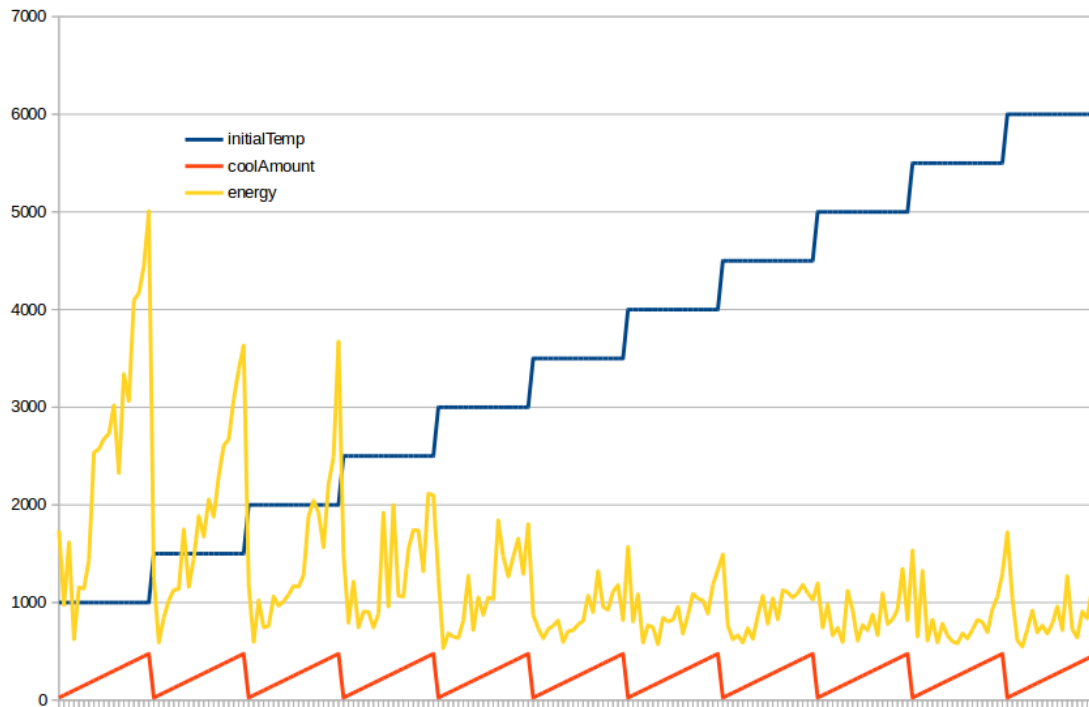
Number of Generations: 120

Initial Temperature 3000

Cooling Amount 0.5

Comparing the Three Algorithms

After attaining the optimal parameters for each algorithm we ran some scripted tests to get an average energy over 1000 iterations and produced the following graph.



Simulated Annealing

We used the optimal parameters defined above

Genetic

We used the optimal parameters defined above

Hybrid Algorithm A

We used the optimal parameters defined in the individual tests from above

Hybrid Algorithm B

We used the optimal parameters defined by the extensive testing of all the permutations of the 5 parameters.

This testing has defined the best set of parameters to use for our Gui.

Population Size: 2000

Top N: 650

Number of Generations: 120

Initial Temperature 3000

Cooling Amount 0.5

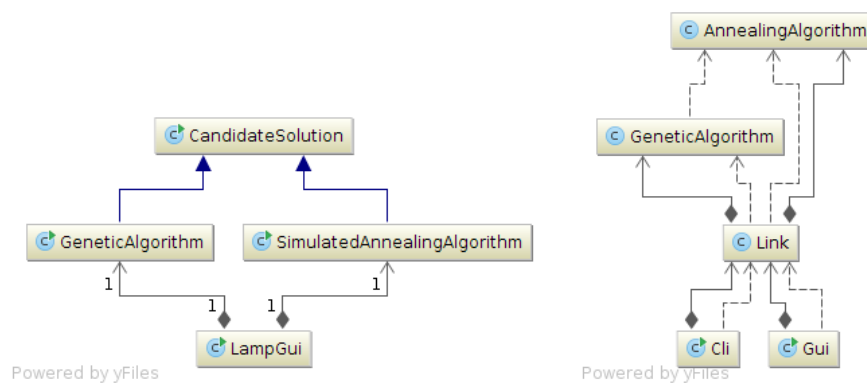
Development Phases/Issues

The Design Pattern

During our first meeting we discussed how our project would be organized, and the overall goals that we hoped to achieve. In keeping with our core values of modularity and re-usability we knew needed a solid design pattern to build upon.

Initially our first thought was that both the algorithms would inherit CandidateSolution and implement extra functionality as necessary. There would be a Gui at the front with instances of both algorithms. Further investigation into the requirements of the project led us see how the first implementation would cause too many road blocks. We settled on a linear design pattern with a very directed data flow from the interface to the link to the algorithm. This is discussed further in the technical details section.

The old and new design patterns



With the overall picture in our heads of the direction we were going we began development the 2 algorithms, the link and the interfaces, with each of us being assigned a particular section.

The Genetic Algorithm

The initial setup

For the genetic algorithm, generations are the key to improving a solution. There were three main steps to consider. Generating a population, mating the best candidates, and replacing the weakest members of the population with the children created from the mating process. This finalized group would then become the population for the next generation. The idea is that each generation will become stronger and yield a good result.

A good result in the scope of our program would be considered one with minimal energy. Energy was chosen as a marker for consistency throughout the program with both the genetic and annealing algorithm referring to it.

Creating the population

Creating a population was a relatively easy step due to the strong framework which we already had in place and which was discussed in our interim report.

generatePopulation is the function which we designed to create an ArrayList of randomly selected CandidateSolutions. Using this type of data structure allowed us to easily amend our population by sorting it and adding/removing members which will become important once the algorithm has run.

This strong basis meant that this function remained consistent throughout the development process and changed very little from beginning to end.

Finding the fittest group

In order for our solution to improve in each generation, we needed to find the best members of the population (or the members which have the lowest energy scores). The Collections class allowed us to compare each member of the population based on their energy simply by implementing a compareTo method within the CandidateAssignment class. Once the population was sorted on this criteria it was simply a matter of selecting the top N elements of the population, with N being an even number, the size of which we would like our fittest group to be. Initially these were set as constants for ease of testing.

Finding the weakest group

With the population already sorted, finding the weakest group was trivial, and ArrayList contained all of the tools necessary for removing the lowest elements and adding in the children created during the mating process.

Mating

Mating was the most complex element of this algorithm and as such was subject to a lot of change throughout the development cycle. Firstly we needed to decide what mating actually meant for this program and how it would ultimately improve the solution.

As a group we discussed the different elements. Taking into account the project specifications and what we ultimately wanted to achieve. We came to the conclusion that the mating should happen on the level of CandidateAssignments as this is where the energy really originates.

First run

Each member of the fittestGroup was paired with their neighbor in the array. This meant that best solutions stayed with the best solutions. The idea being that these would in turn result in the best children.

Our algorithm took the two parents, both of whom are CandidateSolutions, and created a new CandidateSolution child based on the best elements of these parents. The CandidateAssignments within the

parents are always listed in order. This meant that we could iterate through the parents and compare each Candidate assignment.

Additional code was added to the CandidateAssignment class which would allow us access to each student's satisfaction. Whichever student had the better satisfaction score was then used in creating the child. This meant that the child was essentially a combination of the most satisfied students.

Results

This process resulted in a relatively poor score and two large obstacles became apparent.

A large penalty is applied whenever there is a duplication in project selections among students. Our algorithm did not take into account a project allocation already existing within the child solution. So, while the solution was better for the individual student, the overall energy produced by the solution incurred a huge penalty.

The other big problem which occurred was that as our list of fittest solutions was sorted every generation, often the same parents would produce identical children each generation. This meant that our solution became full of duplicates and stopped improving after only a few generations.

Revising the approach

While the overall approach to mating was sound, our implementation was flawed. These situations needed to be handled.

The algorithm was restructured completely while retaining the core ideas. The individual Candidate assignments were still compared but other checks were applied.

We introduced a ChildCandidateSolution class with the function `energyWith()`. This allowed us to check whether the overall energy would improve, preventing larger penalties from incurring.

To prevent the possibility of identical parents producing an identical child, the fittest group was shuffled and a new check was introduced. Before each child can be added to the children list, its energy must be greater than the energy of both parent1 and parent2. All of the children who don't make the cut are drowned in a river and discarded.

Results

Immediately results were much better. With energy dropping from the thousands into the range of 300-400

Mutations

An element of the Genetic algorithm is the idea of mutations. A mutation is a random adjustment made in some way to the solution.

We initially introduced a random student assignment (with adjustable mutation frequency) and moved this onto a random candidate assignment added into the population.

After running numerous tests with both of these possibilities, and various mutation frequencies, we discovered that no improvement to the final solution was made. In fact, the solution often fluctuated resulting in poorer final results.

After discussing the possibilities we decided as a group to remove mutation entirely. While it is an interesting element of the genetic algorithm, it took us further from our goal of achieving a good low energy score.

The Simulated Annealing Algorithm

With the annealing algorithm, we searched for an overall optimum solution as opposed to an actual best solution, we wanted to consider time as a factor. The three main areas for development were the temperature, the cooling amount, the probability of accepting a solution and the algorithm that drives everything.

Approach

The class was first written entirely in pseudo code, which allowed for any logical obstacles to be addressed and overcome before implementing any code. Helper methods were created as required by the pseudo code, with the possibility of these being added to or removed as the program functionality was tested.

The Boltzmann Formula

The algorithm was then written in full. The Boltzmann formula was used to help us find our solution. Annealing works similar to a hill climbing search, however once a local max is found, there is a possibility to keep searching and to find an even greater result. To calculate the probability of the change in solution being accepted, we created a separate method to include the Boltzmann distribution value for the energy and temperature of a certain solution.

$$\textit{The Boltzmann Formula}$$
$$p(\Delta E) = \frac{1}{e^{\frac{\Delta E}{T}}}$$

Parameters

It is here that the Temperature and cooling come into play. The temperature is set and applied to the algorithm, the higher the temperature then the looser the algorithm and the more likely that a bad solution is accepted. The cooling amount is applied to the temperature meaning we are less and less likely to accept a poor answer.

Refinements with Inlining

We initially created a helper method to randomize an element of the solution. Keeping in line with our original ethos and attempting to maintain a good flow with readable code, this evolved to be a very small method. The choice was made to include it in our go method instead. The notion of a go method, meant that while looking through the code, it was easy for any user to track what was happening should any revisions need to be made.

Final Refinements

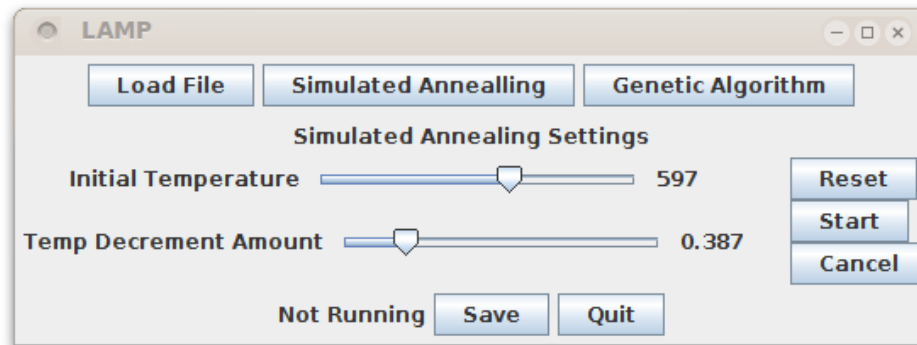
The program initially had constant values for temperature and the amount to be cooled by each time. In testing this, these were altered to determine what combination of variables achieved the best results in under a minutes computation time. This was then altered to allow the user to enter these values as desired through the CLI.

After this was finished, we merely added in functionality to alter the progress bar and allow for status updates as the algorithm begins and ends, before cleaning up the flow of the code and ensuring all variables were global/private as required.

The Interfaces

The first aim of the first Gui was to provide all the functionality of our algorithms. We implemented it with the swing packages and produced a very functional piece of software. After really thinking about the goal of this project we decided this was too much. We wanted to produce an easy to use solution with optimal parameters that were preset and decided upon through thorough testing.

The old and complicated Gui



The realisation of a need for a simple Gui opened the option of a much more functional command line versions of our software. Initially we produced two different interfaces, one for each algorithm. These were beneficial for testing but were ultimately combined into a comprehensive Cli which compliments our user friendly Gui.

Development Cycle Conclusion

One of our mission statements was to create clean, readable code. Once we finished the main coding and testing, we then proceeded to run through the code, being extra critical about its ordering and functionality. The names of the functions were refactored to reflect the modularity of those functions. Encapsulation was refined to ensure elements were public only if required.

More parameters were added to constructors in order to allow adjustments and commands to be input through the CLI and GUI.

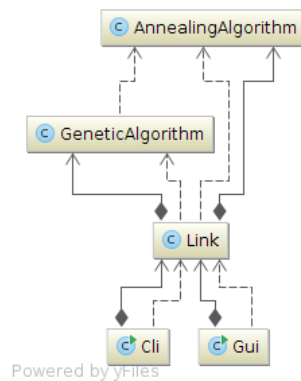
Our Development Model

Not sure what needs to go in here

Technical Details Of Note

Our Design Pattern

In keeping with our core values of simplicity and re-usability we tried to use a very modular design pattern. We identified two distinct types of classes that we created, worker classes and interfaces (the algorithms and the Cli/Gui respectively). An interface invokes methods on a worker. In order to modularise this we created a Link class. This is a pivotal class in our design. Every method call from an interface must go through the link. This gives the ability to pair any interface with any algorithm with ease. Once an interface has access to a Link it can call upon either or both of the algorithms as is necessary. The linear flow of our design pattern is very evident below.



Callback listeners

Independent communication between classes is typically one directional. Usually a class invokes a method on one of the classes that it has references to and a single value is returned upon completion. Sometimes the need for interaction with the calling class is needed during execution of the called method; this is where callback listeners come in. They provide the ability for a method to invoke a method on the calling class.

Use of callback listeners:

1. The listener is ultimately an interface. It declares which methods in the calling class are available to class containing the called method.
2. The calling class implements each of the methods declared in this interface.
3. When creating an instance of a class, whose methods will be called at some point, a reference to the calling class is passed using the "this" keyword to the classes constructor.
4. That class, which needs the reverse communication path, then instantiates an object of type callback listener with the passed reference.
5. This allows method calls back to the calling class during execution and also keeps encapsulation tight as it does not provide the full public interface of the calling class.
6. Callback listeners can also be used with threads to alert the class which created the thread of completion or another significant event.

In our project methods in the algorithm classes are invoked by either of the interface classes (via the Link). We used a callback listener to allow both the Genetic and Simulated Annealing algorithms to invoke methods

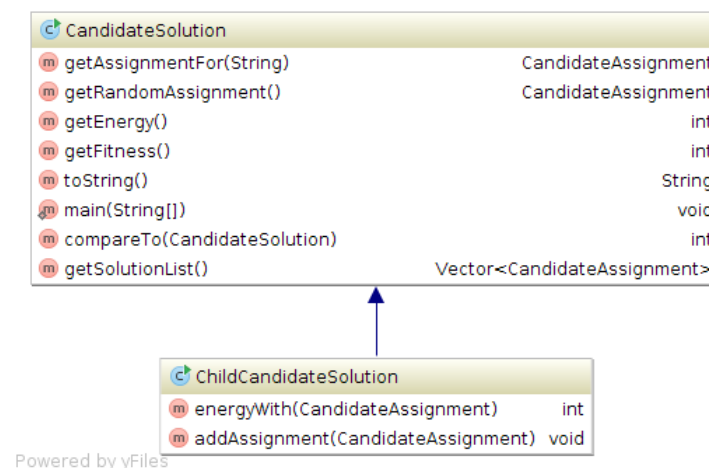
on which ever interface invoked the algorithm. The callback listener is a direct connection straight back to the calling class.

We used the listener to update the progress bar every time 10 percent of the total number of iterations have been completed and also when the status bar needed updating with new information. Our two interfaces, the Cli and Gui, both implement the callback listener and have their own respective versions of the status bar and progress bar.

Inheritance

When building software, in order to efficiently utilise the classes at one's disposal. it is always important to be mindful of potential inheritance relationships. An inheritance relationship gives a subclass access to the methods and fields of the superclass. Inheritance is used when needing to add case specific functionality to a class, i.e. extra functionality that would otherwise bloat other clients of the class.

We identified a need to subclass the CandidateSolution class when creating the Genetic Algorithm. The Genetic Algorithm has to build a CandidateSolution one project at a time and also query the potential fitness of that solution during the building phase. This led to the creation of the ChildCandidateSolution class, which inherits from CandidateSolution, with two extra methods, evident below.



Threads

By default programs run on one main thread. They have one time line, one stream of execution, on which the programs instructions are carried out consecutively. Sometimes, when expensive routines are needed to be called it's advantageous to put them on their own thread, their stream of execution. When the routine is complete is typically not known, but it will complete at some point, (assuming no runtime/logic errors). Callback listeners can be used to signal the main thread of completion or some other significant event.

In our project, the Cli runs in only one thread, and procedurally runs the algorithm and prints the the console to indicate its progress. During execution the user has no interactivity with the program as it's main thread is blocked by the expensive task of computing a solution.

When we designed the Gui it became evident that a separate execution thread was needed as there are two tasks to complete. The JFrame that is the Gui needs redrawing with the progress bar updates and the algorithm has to be executed. We created a separate thread on which the algorithm would run, allowing these two tasks to done concurrently.

Team Development

what we changed
what we liked/didnt,
stuckto/deviated from interim

What went well

the night out and the burgers

What didn't go so well

everything else

Who did what

everything else

The Team Night Out

Burgers and Drama

The development of a large software project involves both individual and group contributions. Many large tasks were broken up into smaller subsections which were divided between the team members. These were researched and brainstormed individually before meeting as a group and developing solutions together.

To keep morale high while we worked through the assignment, we organized a team bonding night between the interim report and the submission of the final report. This took place after a long group coding session in college where the bones of both algorithms were written. We felt that an evening of discussion on topics other than the software project would help keep us connected as a group, which would improve overall work ethic and quality of code produced.

The team night out involved travelling together into town for a large dinner, before returning to UCD campus for a production of *The Nightman Cometh* (an episode of the popular TV show *It's Always Sunny in Philadelphia*) by UCD's DramaSoc. A night off from code development and report writing was well deserved and earned, and allowed for the team to work harder upon our return to college the next day.

A Sad Day For Lamp

The loss of a team member

As a young adult, college can often be a very stressful and time consuming environment to be in. A large amount of students are dropping out of all courses, with 23% of students entering the computer science program in UCD not progressing past their first year. Unfortunately for Team Lamp, we experienced a loss of a member partway through the development process.

Team Lamp started off well, producing strong weekly assignments by each team member. Sadly, we found that as time passed, one member was slowly fading off the radar and was no longer contributing towards the module in both individual assessments and team projects. Gerard Fogarty, our lost Lamp, has unfortunately not been a presence within the team for several weeks.

Team Lamp has persevered through this loss, providing what we feel is still a marketable software solution to this module's brief. Although this module heavily suggested teams of four to produce a solid working solution, we feel that we overcame the obstacle of having only three team members and still maintained a high quality software project and solid written report.

