



Universidade do Porto  
Faculdade de Engenharia  
**FEUP**

# Calendarização de Exames

## *Relatório Final*

Inteligência Artificial  
3º ano do Mestrado Integrado em Engenharia Informática e Computação

Elementos do Grupo

B4\_1

Catarina Ramos - up201406219 - [up201406219@fe.up.pt](mailto:up201406219@fe.up.pt)

Inês Gomes - up201405778 - [up201405778@fe.up.pt](mailto:up201405778@fe.up.pt)

Mário Fernandes - up201201705 - [up201201705@fe.up.pt](mailto:up201201705@fe.up.pt)

21 de Maio de 2017

# 1. Objetivo

O objetivo principal para este trabalho consiste em desenvolver um algoritmo para a otimização na calendarização de uma época de exames. Para este efeito, optamos por adaptar o modelo dos algoritmos evolutivos e o algoritmo “Arrefecimento Simulado” ao nosso tema. A aplicação dos dois algoritmos permitirá a comparação de ambos em termos de eficiência e resultados.

## 2. Descrição

A calendarização de exames é, em qualquer universidade, um tema de bastante relevância devido à quantidade de fatores envolvidos e que tem como objetivo facilitar a vida dos estudantes. Para tal, os estudantes necessitam de se inscrever nas respetivas unidades curriculares por época (normal e recurso). De modo a otimizar a distância entre exames devemos evitar que haja estudantes inscritos em exames que terão lugar em dias consecutivos, bem como maximizar a distância entre exames sempre que haja estudantes em comum, embora sempre dentro da época de exames respetiva. A distância entre exames dá primazia a estudantes sem unidades curriculares em atraso.

Para solucionar este problema serão usados algoritmos de evolução e arrefecimento simulado com dimensões consideráveis. Também será possível visualizar o mapa de exames de um estudante específico, bem como o mapa global de exames, bem como a criação de novo estudantes, exames, inscrição de estudantes a exames, e edição das datas das épocas “Normal” e “Recurso”.

### 2.1. Especificações

Para desenvolver este trabalho foram feitas algumas suposições:

- A época de exames só faz referência a um curso, p.e. MIEIC;
- Nunca há exames sobrepostos, mesmo que não haja estudantes em comum, ou seja, cada slot apenas representa 1 ou 0 Exames;
- Durante toda a época de exames, o horário disponível para marcação começa e termina sempre à mesma hora, p.e. 10h às 18h;
- Não são contabilizados fins-de-semana;
- Exames com estudantes em comum não podem ser feitos no mesmo dia;
- A distância entre exames dá primazia a estudantes sem unidades curriculares em atraso, ou seja, exames do mesmo ano serão mais afastados que exames de anos diferentes.

### 2.2. Esqueleto do programa

O problema foi repartido em várias classes de forma a ter secções distintas, tornando o código bastante mais simples de implementar. Para simular uma época de exames foram criadas as classes *Student*, *Exam* e *Epoch* referentes a estudantes, exames e épocas de exames, cujos objetos estão agregados em vetores na classe *University*. Por

sua vez, cada época terá um vetor de *Subscription* e um *Schedule* que representam as inscrições de estudantes a exames nessa época, e o horário criado. Para a criação desse horário existe também a classe *Algorithm*, cujo objetivo é aplicar os algoritmos aos objetos de modo a obter uma solução o mais otimizada possível. A classe *Algorithm* é a classe mãe das classes *Genetic* e *Simulated Annealing* que aplicam, respetivamente, o algoritmo genético e o algoritmo de arrefecimento simulado.

### **2.2.1. Descrição**

#### **2.2.1.1. IART**

Classe onde se situa o *main()* onde existe a conexão com a interface e base de dados.

#### **2.2.1.2. Student**

A classe *Student* é representada por informações básicas relativas a um estudante, nomeadamente o seu nome e ID.

#### **2.2.1.3. Class**

A classe *Class* é representada por informações básicas relativas a uma disciplina, nomeadamente o seu nome e ID.

#### **2.2.1.4. Exam**

A classe *Exam* é representada por informações básicas relativas a um exame. Além do seu ID e *Class* que representa, tem a duração do mesmo.

#### **2.2.1.5. Subscription**

A classe *Subscription* representa a correlação entre exames e estudantes. Ou seja, cada *Subscription* contém um exame e um estudante, o que significa que o estudante está inscrito nesse exame.

#### **2.2.1.6. Schedule**

A classe *Schedule* é o foco do trabalho e é composto por um vetor de exames, um *map* (estruturas explicadas mais em detalhe na secção [2.3.1](#)), um valor *fitness*, e o valor máximo atingido por este *schedule* na roleta. Estes valores são voláteis enquanto um algoritmo corre. Além destes existe também uma lista de inscrições nos exames desse horário para calcular estudantes em comum, e o primeiro dia de semana desse horário para calcular os restantes, e assim saber se existem exames em fins de semana.

### 2.2.1.7. Epoch

A class *Epoch* é representada por informações básicas relativas a uma época, como nome (Normal ou Recurso) e data de início e fim da mesma. Pelas datas é possível calcular o número de dias dessa época. Além destas informações, ainda agrega um conjunto de inscrições a exames, e o horário final, calculado após a execução dos algoritmo escolhido.

### 2.2.1.8. University

A class *University*, como já foi mencionado, representa uma universidade, pelo que tem um nome e a lista de estudantes, exames e épocas dessa universidade.

## 2.2.2. UML

O Uml de classes que representa a estrutura do programa, à parte da interface, está representado no [anexo E](#).

## 2.3. Algoritmos

Os algoritmos são aplicados a uma época, que por sua vez pertence a uma universidade. A época fornece várias informações desde inscrições de estudantes a exames, número de dias da época, e datas de início e fim da mesma.

### 2.3.1. Representação de Informação

Para se poder aplicar este tipo de algoritmos é necessário organizar as possíveis características que um estado/solução, neste caso um *Schedule*, poderá ter.

Desta forma, uma época tem um conjunto de exames inseridos num vetor que representam um horário. É de supor que, cada dia tem 10 horas úteis para a realização de exames e que cada hora representa uma posição do vetor, referido anteriormente, por ordem de ocorrência. Desta forma, se uma época de exames começar no dia 20, as 10 primeiras posições do vetor seriam reservadas para esse dia, as 10 seguintes para o dia 21, e assim sucessivamente. Também é suposto que apenas existem dias de semana (sábado e domingo são excluídos), pelo que o tamanho do vetor será  $\text{horas\_por\_dia} * \text{dias\_da\_época}$ .

Para além do vetor, existe um *map* que contém, para cada exame a sua primeira posição no vetor. Esta estrutura permite uma organização e cálculo mais eficiente do algoritmo, pois ajuda a saber quais os pontos de crossover e mutação válidos.

Cada exame contém a informação referente sobre:

- A cadeira a qual pertencem, dentro de um conjunto finito de opções associadas à universidade. Exemplo: "IART", "COMP", etc.

- A sua duração, que poderá variar entre 1 ao limite de horas de um dia com variações de 60 minutos. Exemplo: 2:00h , 3:00h , etc.

Outras informações, como o número de estudantes inscritos por exame, são obtidos através da análise das inscrições na época. O cálculo de estudantes em comum também é realizado pelas inscrições.

### 2.3.2. Função de Heurística

A função de heurística é uma função que avalia a qualidade de um estado/solução. Desta forma, toma um papel muito importante nos algoritmos a aplicar porque é através desta que o algoritmo toma determinadas decisões.

A função de heurística, neste caso, será para avaliar um objeto *Schedule*. O resultado desta função será igual à soma dos resultado das funções de heurística aplicadas a cada *Exam* desse *Schedule*.

$$H(Schedule) = \sum_{i=1}^n H(Exam_i), Exam_i \in University$$

Para calcular o resultado da função de cada exame, é feito o somatório da comparação do exame atual com os restantes exames do *Schedule*.

$$H(Exam_j) = \sum_{i=1}^{n-1} Compare(Exam_j, Exam_i), Exam_i \in University \wedge Exam_i \neq Exam_j$$

De cada comparação resulta um número entre 0 e 5 que tem em conta os seguintes parâmetros:

- Existência de estudantes em comum. Se não houver estudantes em comum, a comparação devolve 0;
- Os exames não são no mesmo dia. Se não se verificar a comparação devolve 0;
- Caso as condições anteriores se verifiquem devolve distância \* multiplicador;
  - distância : distância entre as primeiras posições de cada exame no vetor (quanto mais distantes, maior o valor da função de adaptação);
  - multiplicador : número de anos do curso - diferença de anos de cada exame (dá primazia a estudantes sem cadeiras em atraso, visto que, se os exames forem do mesmo ano essa diferença será 0, e o multiplicador ficará igual a 5 (máximo));

Deste modo, quanto maior for o resultado, melhor será o estado/solução.

$$Compare(Exam_i, Exam_j) = \begin{cases} 0, & \text{Subscribers}_i \cap \text{Subscribers}_j = \emptyset \vee [Start_i, End_j] \cap [Start_i, End_j] = \emptyset \\ Distance(Exam_i, Exam_j) * \sqrt{(ExamClassYear_i - ExamClassYear_j)^2} & \text{caso contrário} \end{cases}$$

### 2.3.3. Algoritmo Genético

De modo a otimizar a calendarização de exames, foi escolhido o algoritmo evolutivo. Este algoritmo segue uma série de etapas, mencionadas na secção seguinte.

#### 2.3.3.1. Especificações

Este algoritmo tem como base os conceitos: população, cromossoma e gene. Uma população é um conjunto de cromossomas de tamanho variável e os genes são os vários atributos que compõem um cromossoma.

Tendo em conta a representação da informação na secção 2.3.1, os genes são representados pelos exames que, por sua vez, representam o horário (cromossoma).

A chamada função fitness utilizada neste algoritmo, está descrita na secção 2.3.2.

Para adaptar e otimizar o nosso problema com base em algoritmos evolutivos serão necessários os seguintes estágios:

1. Formação de uma população inicial aleatória com tamanho variável, introduzido pelo utilizador.
2. Cálculo da função de adaptação para cada indivíduo da população (ver 2.3.2);
3. Escolher a população seguinte com 2 critérios:
  - a. seleção elitista para os 3 melhores indivíduos
  - b. criação de uma roleta com a população e seleção de indivíduos face a geração de probabilidade aleatórias.
4. Método de cruzamento uniforme com um ponto de crossover com probabilidade de 90%.
5. Mutação com probabilidade de 5%.
6. Voltar ao ponto 2 até atingir um determinado número de iterações do algoritmo, variável e escolhido pelo utilizador.

#### 2.3.3.2. Aplicação do algoritmo

De modo a aplicar o algoritmo evolutivo foram iniciadas algumas das etapas mencionadas na especificação do presente relatório, entre as quais:

##### A. Criação da população inicial

População criada de acordo com as inscrições recebidas pelo objeto *Epoch*. Pelas inscrições são calculados os exames dessa época. Como já foi referido no ponto anterior, um *Schedule* contém um *vetor* de posições livres e um *map* para identificar a primeira posição de cada exame. Para cada indivíduo da população é criado um vetor e um map aleatórios. Assim, teremos as seguintes etapas para cada exame recebido:

1. Cálculo de todas as posições livres desse schedule para a colocação do exame. São retirados:

- a. fins de semana
    - b. slots já preenchidos por outros exames
    - c. slots onde a duração do exame excederá o dia
    - d. slots onde a duração do exame irá sobrepor outros exames
  2. Se o vetor de posições possíveis estiver vazio, o horário é impossível, e deve ser gerado um novo.
  3. É gerado um número aleatório que escolherá qual a posição dentro do vetor de posições possíveis.
  4. Os slots são preenchidos com o exame. O *map* também é atualizado com uma nova entrada <Exame, primeira posição no vetor>;
- Após colocar todos os exames, teremos o *Schedule* aleatório, e prosseguimos para o seguinte até completar a população.

#### B. Função fitness

A função fitness utilizada é a descrita na secção 2.3.2.

#### C. Seleção da população seguinte

Para proceder à seleção da população são consideradas 2 fases:

- Seleção elitista: seleção dos 3 melhores (com fitness function mais elevada) indivíduos da população para passarem diretamente para a geração seguinte.
- Seleção com roleta:
  - atribuir a cada indivíduo uma percentagem na probabilidade de selecção de acordo com a sua fitness function. Um indivíduo com menor fitness function terá uma probabilidade melhor de ser escolhido.
  - gerar valores aleatórios
  - o indivíduo escolhido é aquele cuja probabilidade gerada se encontra no seu intervalo de seleção da roleta.

#### D. Crossover

Dada a nova população, são escolhidos indivíduos para cruzar com probabilidade de 90%. Cada indivíduo, dentro dos escolhidos, cruza com o seguinte, exceto em caso de número ímpar da população. Nesse caso o último indivíduo não cruza. O cruzamento é feito ao nível do *map*, pois para cada indivíduo a sua chave está igualmente ordenada, apenas diferem os seus valores. Ao transcrever este *map* para o *vector* com os slots, podemos gerar combinações inválidas. Neste caso, é gerado um vetor de posições possíveis para o exame. De seguida é escolhida uma das posições aleatoriamente. Em caso de não haver posições possíveis, a fitness function desse horário passar a ser zero.

#### E. Mutação

Para proceder à mutação é criado um vetor de probabilidades aleatórias para cada exame de todos os indivíduos, ou seja,  $n^\circ$  indivíduos \*  $n^\circ$  de exames. Para cada probabilidade inferior a 5%, é aplicada uma mutação a esse exame. A mutação consiste em gerar uma nova posição para esse exame. De modo a não gerar indivíduos impossíveis, é gerado primeiro um vetor de posições possíveis e é escolhida aleatoriamente uma dessas posições. Se não houver posições possíveis, o indivíduo não é mutado.

### **2.3.4. Algoritmo do Arrefecimento Simulado**

Este algoritmo simula o processo de arrefecer um metal liquefeito até solidificar com a variação da sua temperatura ao longo do tempo.

Numa fase inicial do algoritmo, os vários estados obtidos são ainda muito instáveis, podendo estes, divergir da solução ótima. Conforme o tempo passa, a “temperatura” vai ser cada vez mais reduzida garantindo uma maior estabilidade dos estados, rejeitando cada vez mais, estados que contrariem um rumo à solução ótima.

#### **2.3.4.1. Especificações**

Este algoritmo tem como base os seguintes estágios:

1. Estado corrente = Estado inicial
2. Temperatura = variação de temperatura;
  - a. Se Temperatura = 0, então termina e estado corrente é a solução;
3. Novo estado = estado com base em variação dos parâmetros do estado inicial tendo em conta a temperatura atual;
4. Calcular resultado a partir da função de heurística;
  - a. Estado corrente = Novo estado, se o resultado for melhor do que o do estado corrente;
  - b. Estado corrente = Novo estado, com probabilidade  $e^{-(\text{variação de energia/temperatura})}$ 
    - i. Se probabilidade falhar então volta ao ponto 3.
5. Voltar a 2.

A função de heurística utilizada encontra-se na secção 2.3.2.

#### **2.3.4.2. Aplicação do algoritmo**

De modo a otimizar e adaptar as diferentes etapas especificadas na secção 2.3.4.1 a este tema, foram tomadas algumas medidas em relação às etapas de:

##### **A. Estado Inicial**

O estado inicial é gerado aleatoriamente da mesma forma que o algoritmo genético gera indivíduos aleatórios para a população aleatória (ver secção 2.3.3.2 ponto A)



B. Redução da variável temperatura

Temperatura reduzida de forma linear em  $x$  de acordo com um parâmetro passado na instanciação da classe *SimulatedAnnealing*.

C. Encontrar um novo estado

Para gerar um novo estado, aplica-se variações ao estado corrente. Aplicar uma variação é o equivalente a aplicar uma mutação (ver secção 2.3.3.2 ponto D) num determinado exame.

De forma a simular o conceito inicial do algoritmo (ver secção 2.3.4), em que inicialmente os estados encontram-se num estado instável, supusemos que, quanto mais instável é um estado maior são as diferenças entre esse estado e o próximo. Tendo isto em conta, o número de variações efetuadas no estado corrente para gerar novos estados segue a determinada fórmula:

$$NVariations = \frac{Temperature}{5} + 1$$

As variações serão executadas em exames de forma aleatória.

D. Função de Heurística

A função de heurística utilizada encontra-se descrita na secção 2.3.2.

E. Aceitação de um novo estado

Foi introduzido o parâmetro *acceptance*, passado na instanciação da classe *SimulatedAnnealing*, que serve como um fator que é multiplicado pela temperatura quando se calcula a probabilidade de um estado pior do que o atual ser aceite.

Esta implementação foi necessária devido ao grau de grandeza dos valores da função de heurística, que ronda entre os  $10^5$  e os  $10^6$ . Assim sendo os valores da temperatura teriam que ser muito elevados e pouco realistas. Os valores aconselháveis para o parâmetro *acceptance*, com uma *temperature* = 50, estão contidos no intervalo  $[0, 50]$ , em que o limite máximo representa a aceitação máxima e vice-versa. É de notar que, optando por um valor de *acceptance* = 0, o algoritmo não irá aceitar de forma a alguma estados como no caso indicado acima. Neste caso, podemos verificar que o algoritmo terá um comportamento idêntico ao algoritmo *Hill Climbing*.

Pondo isto em prática, a probabilidade do estado ser seleccionado como próximo estado corrente é calculado através da fórmula:

$$AcceptanceProbability = \frac{H(NewSchedule - CurrentSchedule)}{Temperature * Acceptance}$$

### 2.3.5. Estatísticas e Resultados

Tendo em conta os diversos resultados apresentados nos dois algoritmos, foi nos possível, estatisticamente, chegar a algumas conclusões.

Em relação a uma auto-análise feita ao algoritmo *Simulated Annealing*, chegamos à conclusão que, tendo em conta os dados estatísticos obtidos e apresentados no [anexo A](#), o melhor valor para o parâmetro *acceptance* é de  $acceptance \in [10, 20]$ , para uma  $temperature = 50$ . Este valor é o mais indicado de forma a obter, de forma otimizada, um melhor valor de fitness final tendo em conta o tempo de execução do algoritmo e as percentagens de *Worst Schedules* aceites e rejeitados. É também relevante dizer que, como os parâmetros *temperature* e *acceptance* estão diretamente relacionados para o cálculo da aceitação de um *Worst Schedule*, apenas efetuamos algumas estatísticas tendo em conta a variação deste último parâmetro. Desta forma, e ao mesmo tempo, também nos foi possível comparar entre este algoritmo e o algoritmo *Hill Climbing* com  $acceptance = 0$ .

Em relação a uma auto-análise ao algoritmo *Genetic*, chegamos a algumas conclusões a partir de dados estatísticos obtidos e apresentados no [anexo B](#). Concluímos que para  $population = 50$ , é aconselhável para a execução do algoritmo um número de  $repetitions = 80$ . Para as probabilidades de cruzamento retiramos as conclusões de que os melhores valores seriam:  $CrossoverProbability = 90\%$  e  $MutationProbability = 5\%$  por apresentarem uma maior diferença entre o fitness de entrada e o melhor fitness encontrado.

Possíveis conclusões sobre a comparação entre os dois algoritmos estarão descritas na [secção 4](#).

## 2.4. Interface

Este projecto utiliza uma GUI, executada com auxílio do software *Qt*, facilitando a leitura de informação e a interação com o programa. Todas as figuras aqui apresentadas pertencem ao anexo D.

### 2.4.1. Acções do Utilizador

Quando o utilizador executa o programa é deparado com o menu da [figura D1](#). Aqui o utilizador pode escolher mudar a sua base de dados, prosseguir com o programa, ou sair.

Quando o utilizador escolhe “Menus” depara-se com o menu da [figura D2](#). Neste menu o utilizador poderá:

- ver a lista de estudantes da universidade ([figura D3](#))
- criar novo estudante ([figura D4](#))
- ver lista de exames da universidade ([figura D5](#))
- criar novo exame ([figura D6](#))
- ver lista de exames seguida dos alunos inscritos à mesma ([figura D7](#))
- editar épocas ([figura D8](#))
- ver horários já gerados ([figura D9](#))

- gerar novos horários ([figura D10](#))
- voltar a trás

Aquando a listagem dos estudantes, o utilizador poderá clicar no seu nome e verificar o horário desse estudante. Ao clicar sobre o nome do estudante, será redirecionado para a [figura D9](#), onde poderá escolher a época e assim verificar o seu horário.

Para criar um novo estudante o utilizador apenas terá de indicar o nome do mesmo, como demonstrado na [figura D4](#).

Na listagem de exames o utilizador poderá visualizar todos os exames em registo, bem como o ano correspondente e a sua duração (ver [figura D5](#)).

Para criar um novo exame, o utilizador apenas necessita de inserir o nome da cadeira associada, ano e duração exame (ver [figura D6](#)).

Para verificar as inscrições dos estudantes ao exames, bem como adicionar e remover inscrições de estudantes, podemos clicar em “enrollments”, aparecendo a imagem da [figura D7](#), que para cada exame tem a lista dos estudantes inscritos, por época.

Existe também a possibilidade de editar a data de início e fim de uma época, como é possível verificar na [figura D8](#). Sempre que é feita uma mudança, todas as datas são comparadas de modo a que seja impossível mudar para épocas sobrepostas.

A [figura D9](#) representa os últimos horários gerados relativos a cada época. Cada tabela apresenta uma semana e em cada dia é possível ver os exames que irão decorrer nesse mesmo. Após gerar um horário é possível visualizar as estatísticas da geração carregando no botão “Show statistics”.

Por fim, a [figura D10](#) apresenta a interação que o utilizador tem com o programa para correr os algoritmos genético e de arrefecimento simulado. Neste ecrã é possível alterar os parâmetros de entrada que fazem com que a geração se dê de maneiras diferentes. Após correr um dos algoritmos, o utilizador é redirecionado automaticamente para o ecrã de visualização de horários.

## 2.5. Base de dados

O programa utiliza uma base de dados *sqlite* local para carregar e guardar informação, facilitando bastante a inserção de dados e ao mesmo tempo tornando-o bastante mais genérico.

A base de dados contém informação sobre estudantes, exames, épocas e inscrições.

Para este trabalho são disponibilizados dois ficheiros para serem carregados: *iart1.db* e *iart2.db* representados pelos nomes “big database” e “small database”. Ambos contêm as mesmas épocas e exames. No entanto o primeiro apresenta apenas uma dezena de estudantes e inscrições, o que permite a fácil interpretação de resultados. Por sua vez, o segundo, inclui duas centenas de estudantes e mais de 500 inscrições, proporcionando dados mais relevantes para a elaboração dos algoritmos, no entanto com mais dificuldade para a verificação dos mesmos.

### 3. Execução do programa

Relativamente ao produto final, este é um ficheiro executável que, junto com os seus recursos e algumas bibliotecas, permite ao utilizador utilizar o programa com apenas double click no ficheiro, sem necessidade de instalar qualquer programa.

Após executar a aplicação, o utilizador depara-se com o menu inicial e poderá seleccionar a base de dados desejada, bem como utilizar todas as funcionalidades descritas.

### 4. Conclusões

De forma a tomar conclusões sobre qual dos algoritmos é mais eficaz, foram escolhidos os melhores valores para os parâmetros dos algoritmos em questão, a partir de determinadas conclusões obtidas na [seção 2.3.5](#), e feita uma análise ao comportamento dos dois algoritmos. Esta análise encontra-se no [anexo C](#).

Comparando os dois algoritmos utilizados, concluímos que o tempo de execução do algoritmo *genetic* é consideravelmente superior ao do *simulated annealing*. No entanto, os valores obtidos no *genetic*, estabilizam mais rapidamente do que no segundo algoritmo, podendo até assumir que um número de *iterations* = 60 seria o mais indicado. Isto pode se dever ao facto de, neste primeiro, estarmos a fazer alterações a um conjunto de *schedules* e não a um só, como é o caso do *simulated annealing*. Como se mantém um grupo de elitistas no primeiro algoritmo, conseguimos proporcionar melhores cruzamentos e a passagem garantida destes elitistas para a próxima geração. À medida que a população atinge melhores valores na função fitness, acaba também por ser cada vez menor a probabilidade de haver alterações significativas para um aumento do mesmo na próxima geração.

Em suma, podemos concluir que tanto um algoritmo como o outro proporcionam bons resultados. No entanto, se o factor de número de iterações para atingir um bom resultado for relevante, o algoritmo *Genetic* seria o mais aconselhável. Por outro lado, se o tempo por um factor a considerar, o *Simulated Annealing* seria melhor.

## 5. Recursos

Slides teóricos da cadeira :

[https://web.fe.up.pt/~eol/IA/1617/ia\\_.html](https://web.fe.up.pt/~eol/IA/1617/ia_.html)

Interface:

<https://www.qt.io/>

Base de dados:

[https://www.tutorialspoint.com/sqlite/sqlite\\_c\\_cpp.htm](https://www.tutorialspoint.com/sqlite/sqlite_c_cpp.htm)

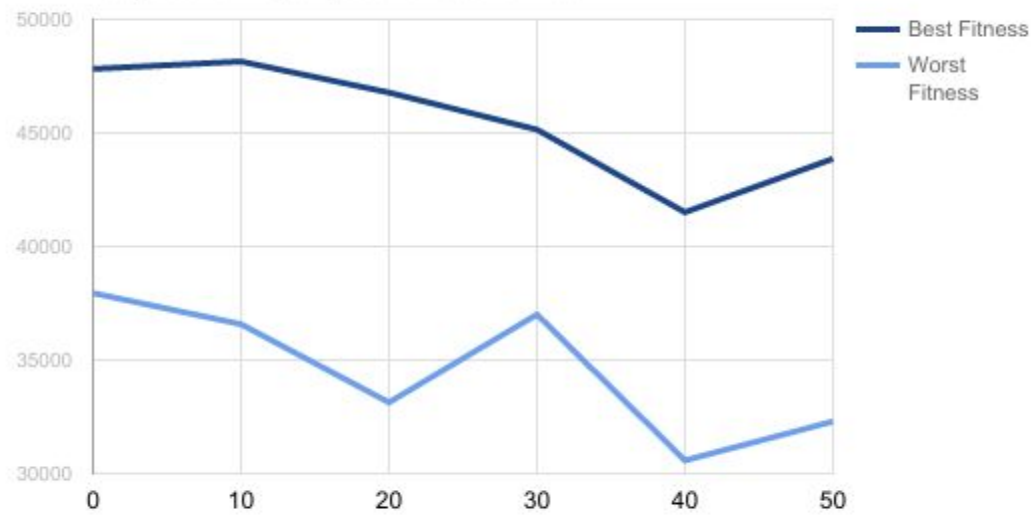
<http://www.dreamincode.net/forums/topic/122300-sqlite-in-c/>

Outros:

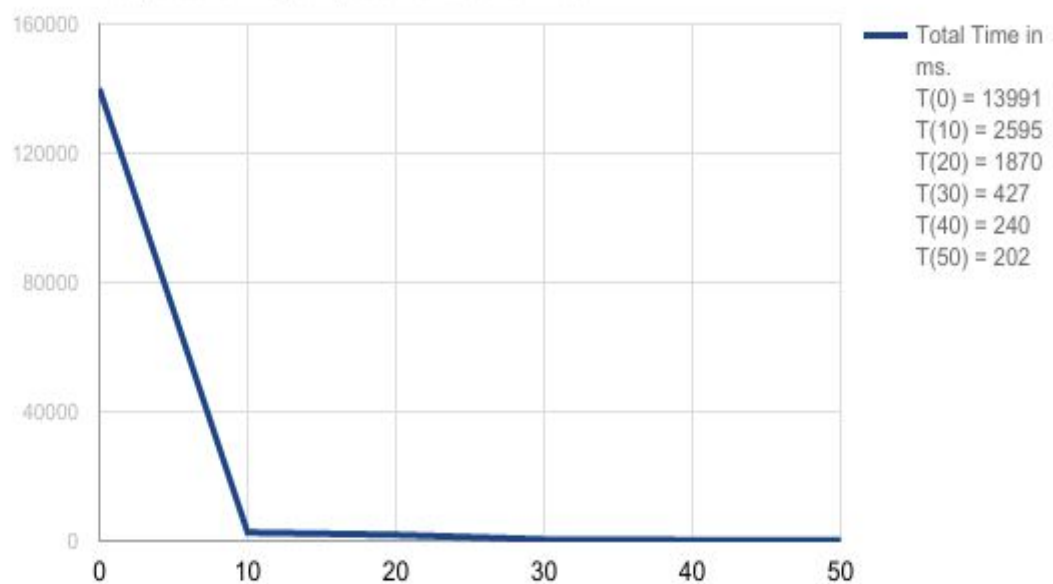
<https://www.codeproject.com/Articles/23111/Making-a-Class-Schedule-Using-a-Genetic-Algorithm>

## Anexo A

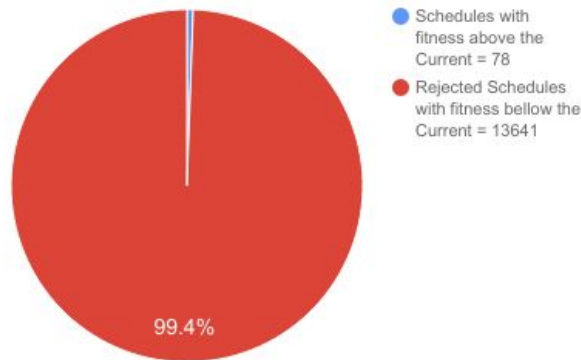
Fitness Values of the Simulated Annealing algorithm with the variation of the parameter acceptance ( $x$ ). InitialFitness = 36760, temperature = 40, temperature reduction = 0.5.



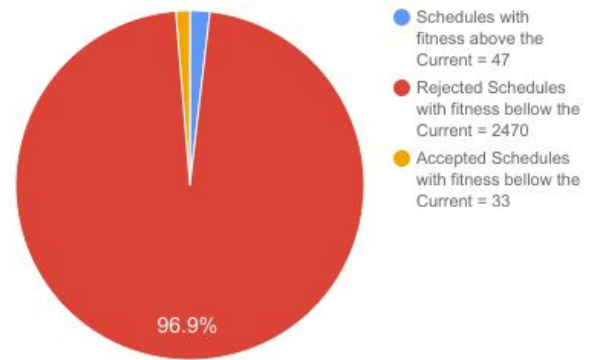
Total Time of execution of the Simulated Annealing algorithm with the variation of the parameter acceptance ( $x$ ). InitialFitness = 36760, temperature = 40, temperature reduction = 0.5.



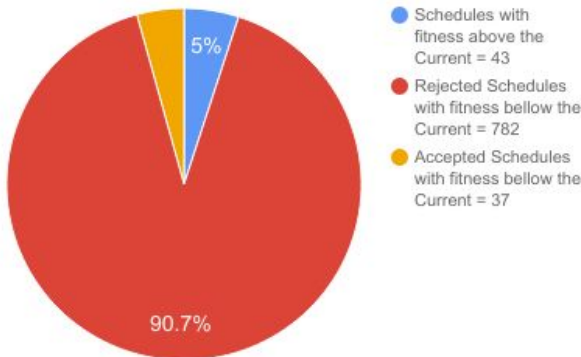
Generated Schedules in the Simulated Annealing algorithm.  
InitialFitness = 36760, temperature = 40, temperature reduction = 0.5,  
acceptance = 0. Total Schedules Generated = 13720.



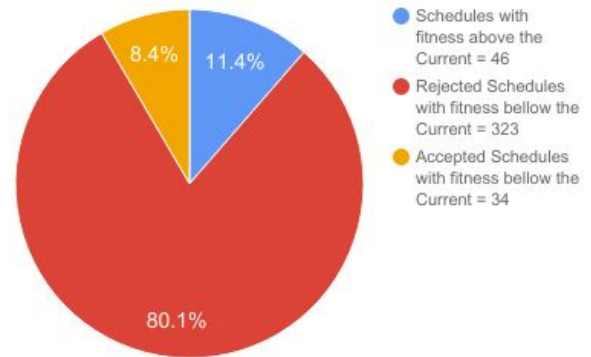
Generated Schedules in the Simulated Annealing algorithm.  
InitialFitness = 36760, temperature = 40, temperature reduction = 0.5,  
acceptance = 10. Total Schedules Generated = 2551.



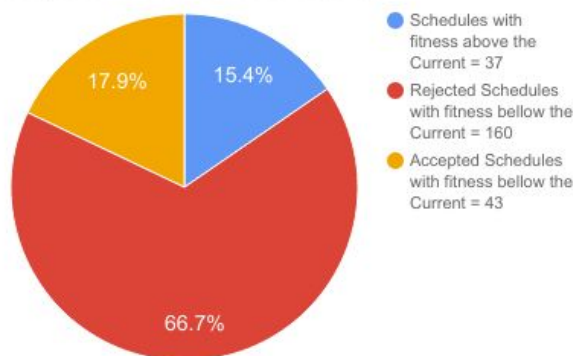
Generated Schedules in the Simulated Annealing algorithm.  
InitialFitness = 36760, temperature = 40, temperature reduction = 0.5,  
acceptance = 20. Total Schedules Generated = 863.



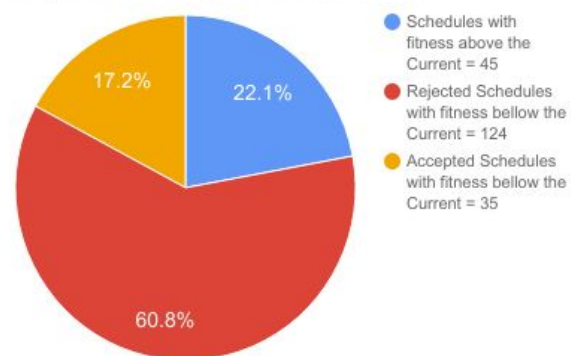
Generated Schedules in the Simulated Annealing algorithm.  
InitialFitness = 36760, temperature = 40, temperature reduction = 0.5,  
acceptance = 30. Total Schedules Generated = 404.



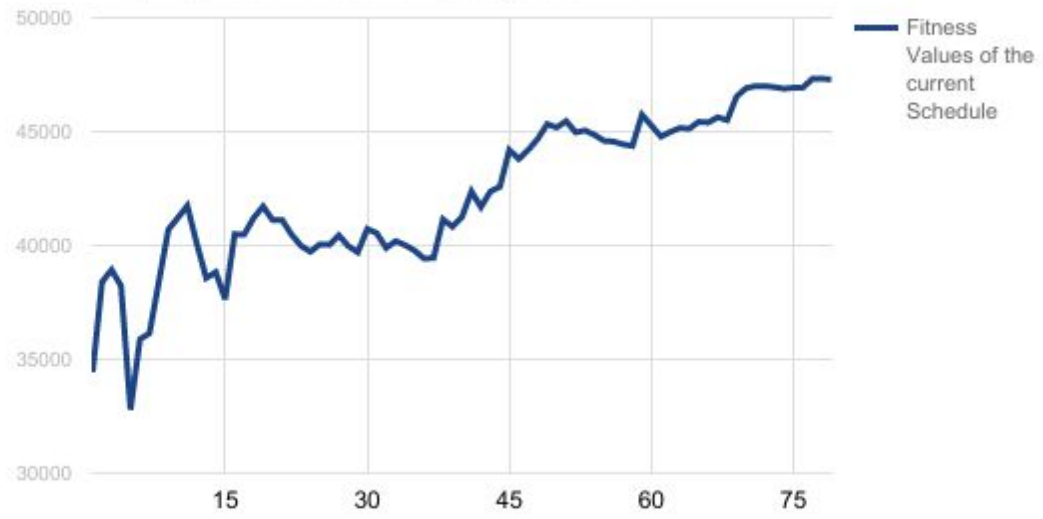
Generated Schedules in the Simulated Annealing algorithm.  
InitialFitness = 36760, temperature = 40, temperature reduction = 0.5,  
acceptance = 40. Total Schedules Generated = 241.



Generated Schedules in the Simulated Annealing algorithm.  
InitialFitness = 36760, temperature = 40, temperature reduction = 0.5,  
acceptance = 50. Total Schedules Generated = 205.



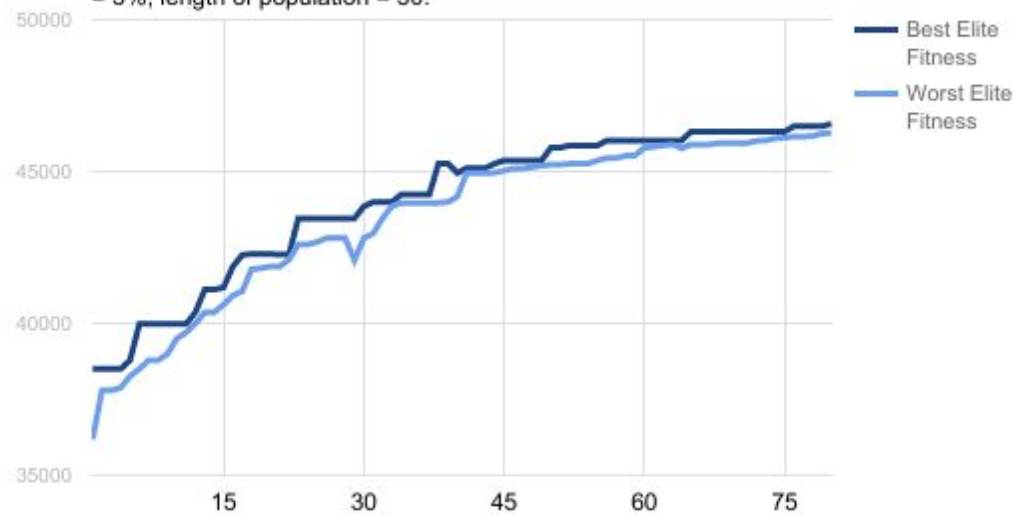
Evolution of the current schedule through the algorithm Simulated Annealing iterations. Hours per day = 0, repetitions = 80, temperature = 40, temperature reduction = 0.5, acceptance = 20.



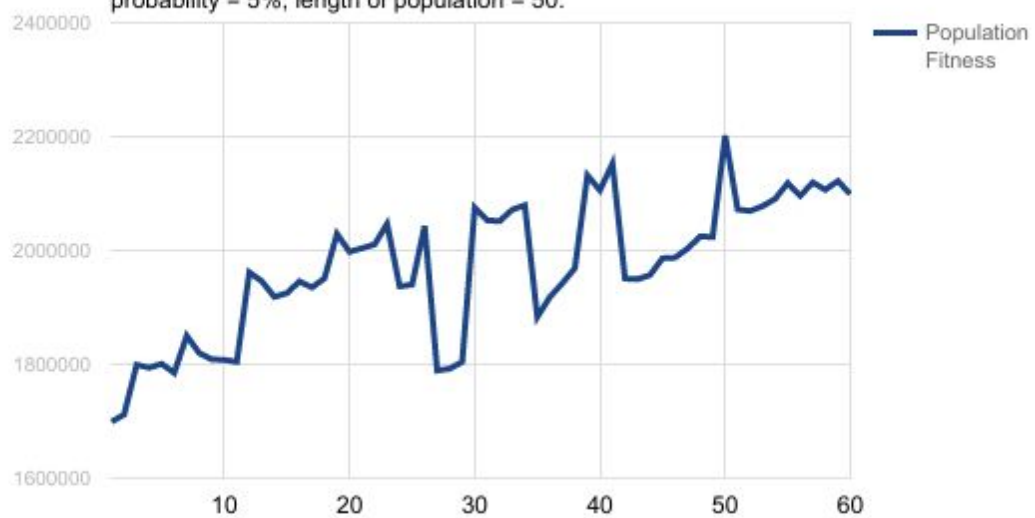


## Anexo B

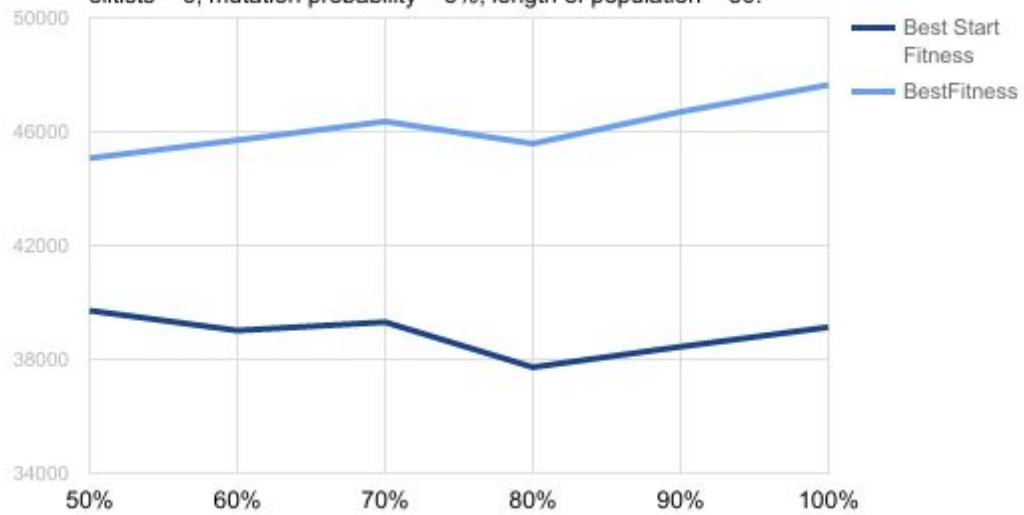
Fitness values of the elite in the Genetic algorithm with the different repetitions of the algorithm. Hour per day = 8, repetitions = 80, number of elitists = 6, crossing probability = 90%, mutation probability = 5%, length of population = 50.



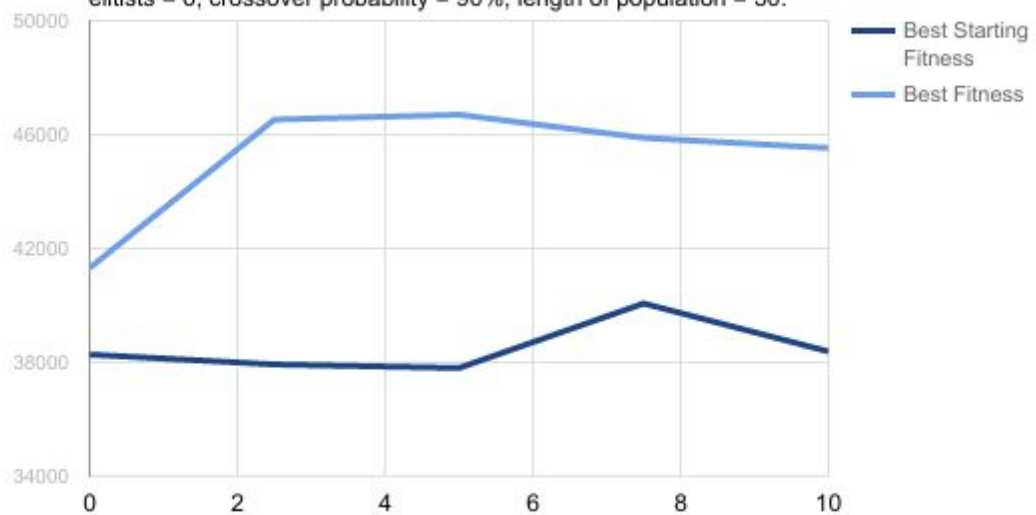
Fitness values of the population in the Genetic algorithm with the different repetitions of the algorithm. Hour per day = 8, repetitions = 80, number of elitists = 6, crossing probability = 90%, mutation probability = 5%, length of population = 50.



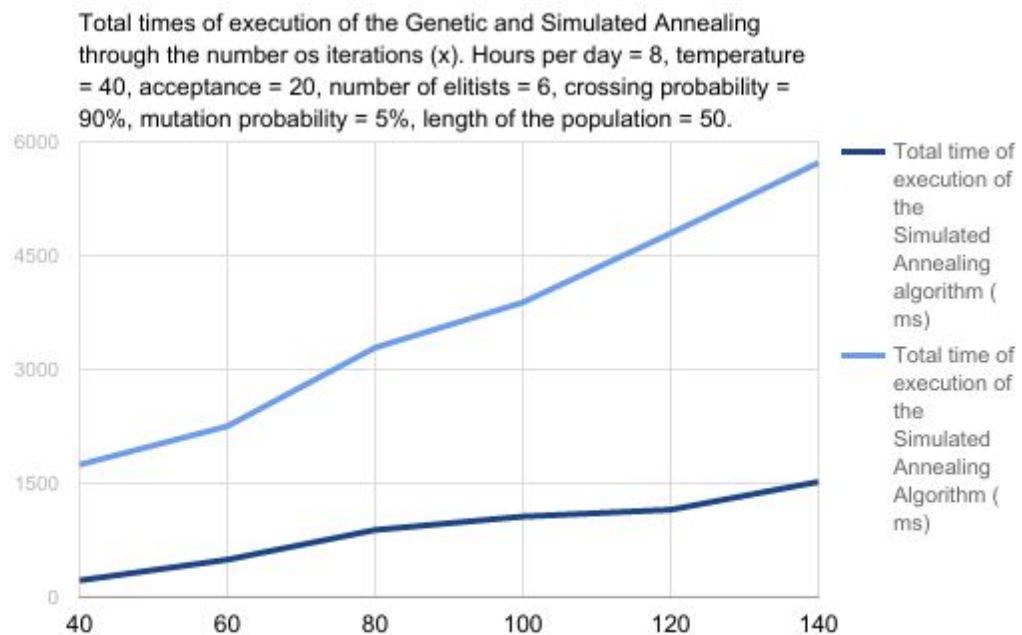
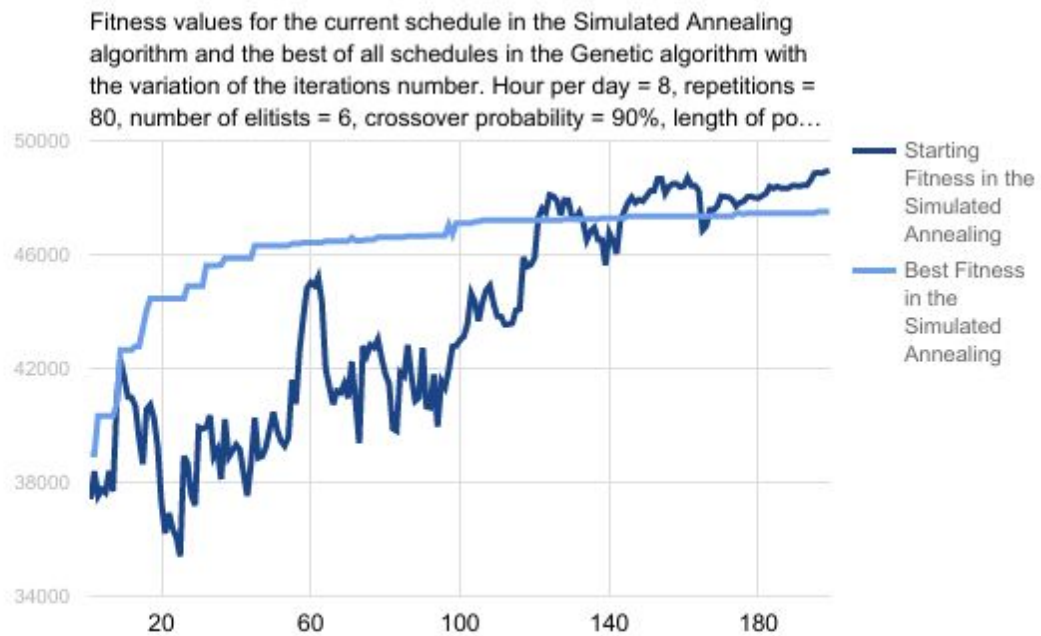
Fitness values of the best schedule in the start vs the best of all schedules in the Genetic algorithm with the variation of the parameter crossing probability. Hour per day = 8, repetitions = 80, number of elitists = 6, mutation probability = 5%, length of population = 50.



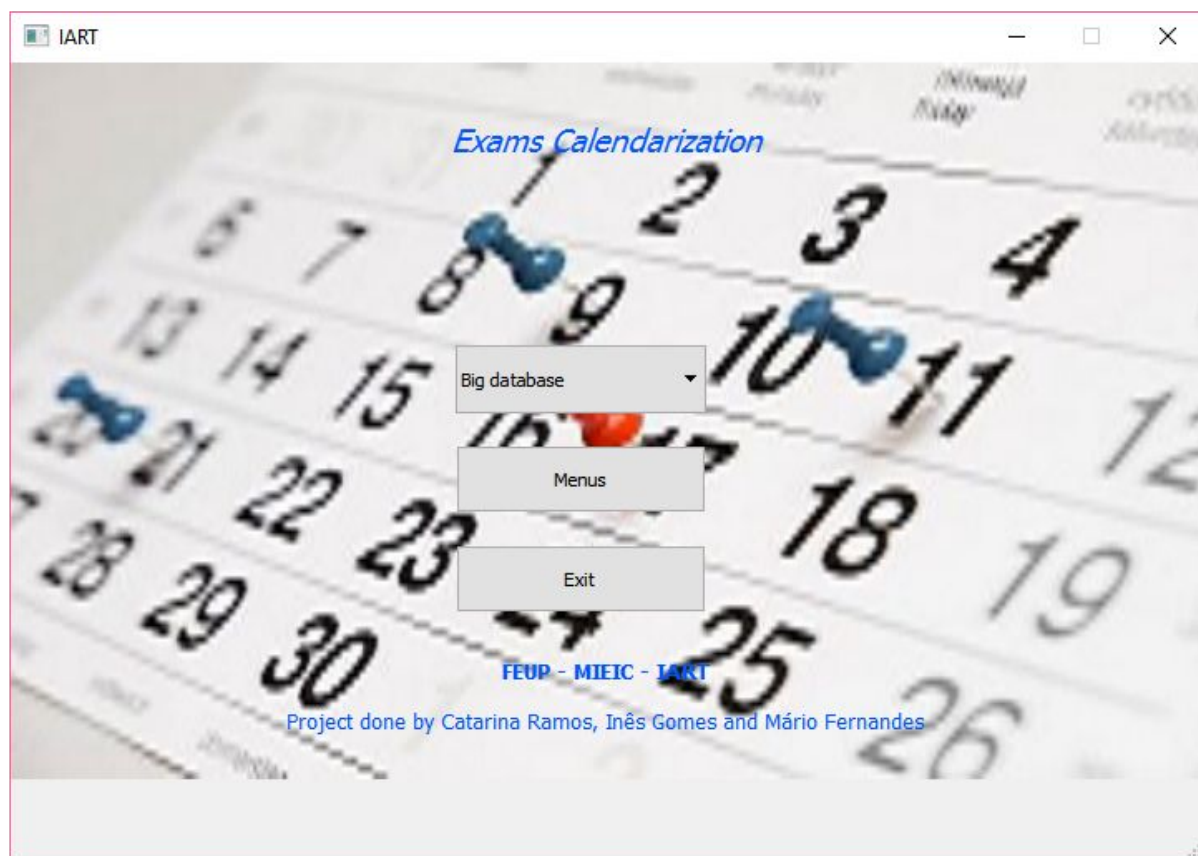
Fitness values of the best schedule in the start vs the best of all schedules in the Genetic algorithm with the variation of the parameter mutation probability. Hour per day = 8, repetitions = 80, number of elitists = 6, crossover probability = 90%, length of population = 50.



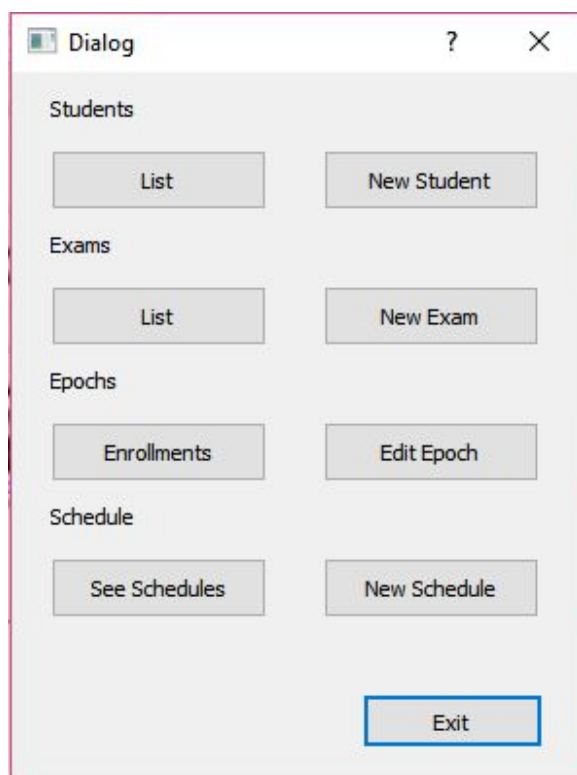
## Anexo C



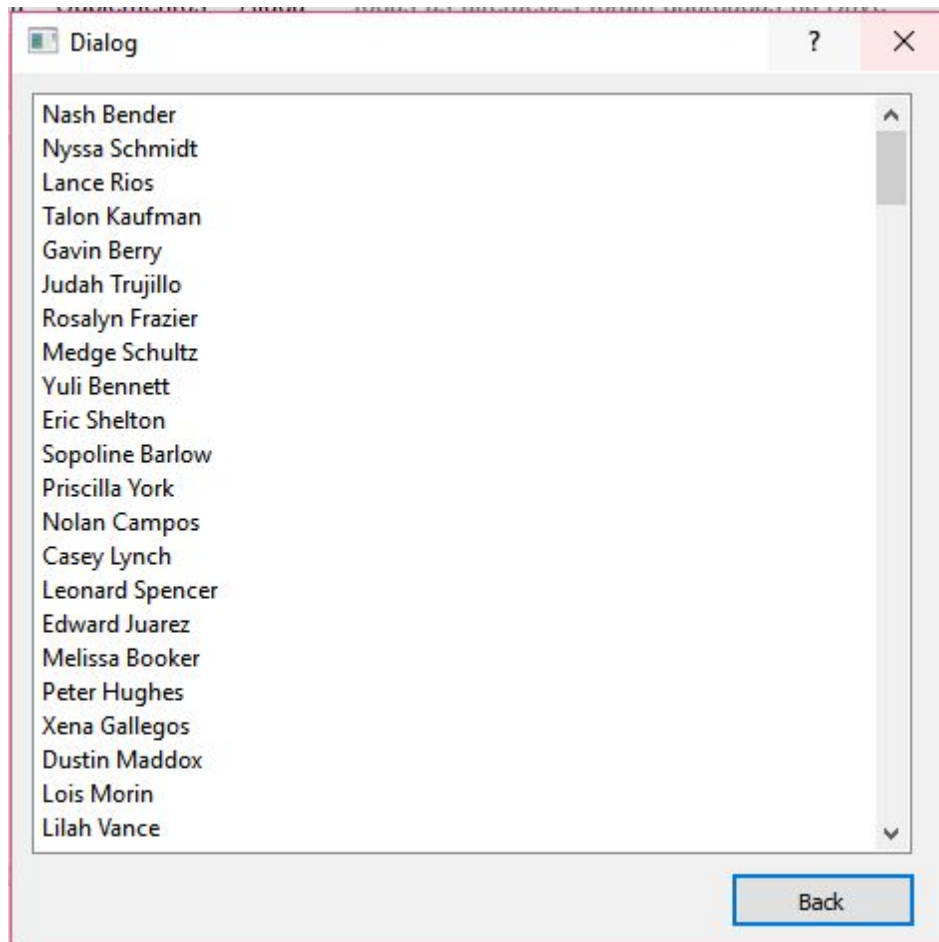
## Anexo D



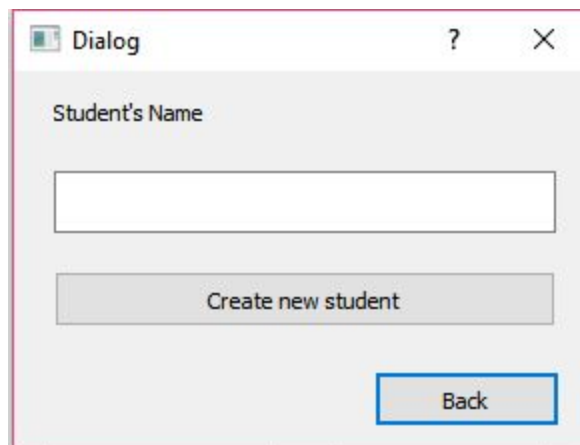
*Figura D1 - Menu inicial*



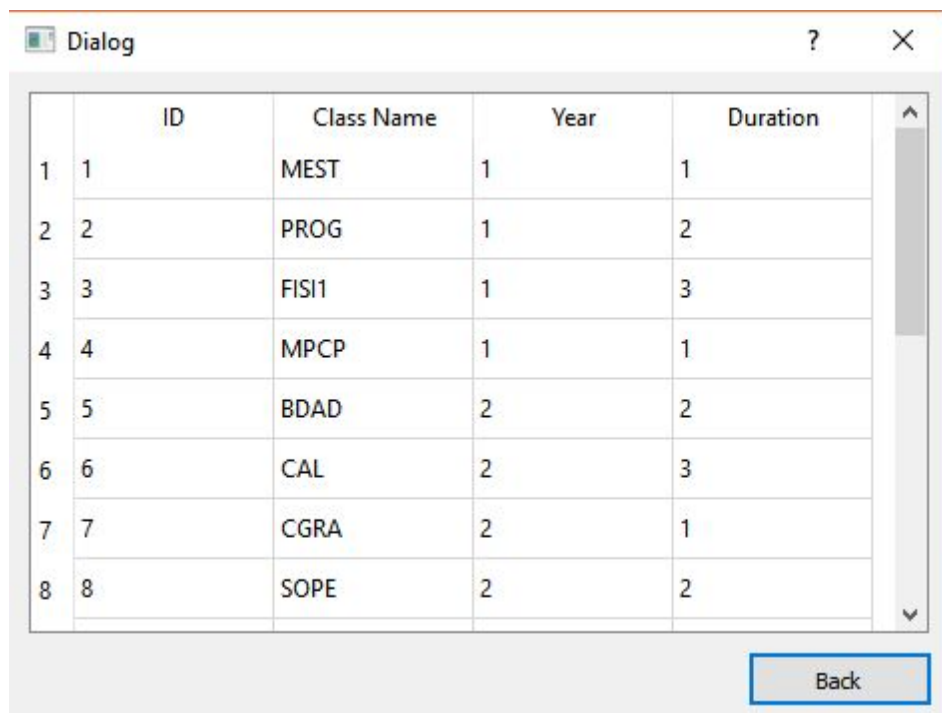
*Figura D2 - Menu principal*



*Figura D3 - Lista de todos os estudantes da universidade*



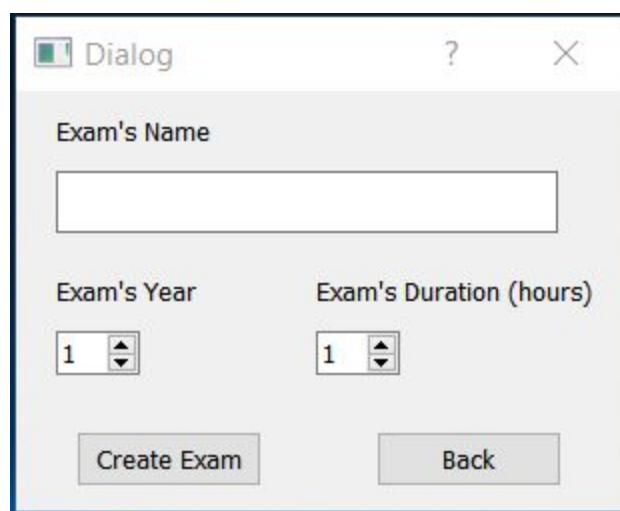
*Figura D4 - Criação de um novo estudante*



	ID	Class Name	Year	Duration
1	1	MEST	1	1
2	2	PROG	1	2
3	3	FISI1	1	3
4	4	MPCP	1	1
5	5	BDAD	2	2
6	6	CAL	2	3
7	7	CGRA	2	1
8	8	SOPE	2	2

Back

Figura D5 - Lista de todos os exames com o respectivo ano e duração



Dialog

Exam's Name

Exam's Year

Exam's Duration (hours)

Create Exam

Back

Figura D6 - Criação de um novo exame para uma nova cadeira

Dialog

MEST

- > Alexander Bruce
- > Sylvester Robertson
- > Jared Morrow
- > Justin Brock
- > Reagan Craft
- > Mechelle Gibson
- > Tyler Myers
- > Uriel Cain
- > Medge Schultz
- > Bo Baird
- > Mikayla Miranda
- > Cassady Anderson
- > Cassidy Vaughan
- > Jelani Beasley
- > Kirestin Wiley
- > Debra Paul
- > Joel Eaton
- > Ruby King
- > Talon Kaufman
- > Signe Stewart

PROG

- > Alexander Bruce
- > Sylvester Robertson
- > Jared Morrow
- > Justin Brock
- > Reagan Craft

Epoch's Name

Normal

List Epoch

Student's Name

Exam's Name

Add Remove

Back

*Figura D7 - Lista de subscrições por cadeira com possibilidade de adicionar/remover*



**Dialog**

Epoch's Name: Normal

Current Starting Date : 8-5-2017      Current Ending Date : 26-5-2017

New Starting Date :

	maio, 2017						
	dom	seg	ter	qua	qui	sex	sáb
18	30	1	2	3	4	5	6
19	7	8	9	10	11	12	13
20	14	15	16	17	18	19	20
21	21	22	23	24	25	26	27
22	28	29	30	31	1	2	3
23	4	5	6	7	8	9	10

New Ending Date :

	maio, 2017						
	dom	seg	ter	qua	qui	sex	sáb
18	30	1	2	3	4	5	6
19	7	8	9	10	11	12	13
20	14	15	16	17	18	19	20
21	21	22	23	24	25	26	27
22	28	29	30	31	1	2	3
23	4	5	6	7	8	9	10

Check Date
Edit
Back

**Statistics**

Number of Iterations	: 3
Average Iteration Time	: 169.33 ms
Algorithm Total Time	: 508.00 ms
Longest Time	: 143.00 ms
Starting Fitness	: 0
WorstFitness	: 0
Ending Fitness	: 66156
BestFitness	: 71408
Fitness Improvement	: 66156
Max Fitness Improvement	: 71408
Improvement/StartFitness : inf %	
MaxImprov/StartFitness : inf %	

**Genetic**

Population	: 20
Avg population fitness	: 870061.69
Avg selection fitness	: 489918.00
Avg crossover fitness	: 710022.69
Avg mutation fitness	: 825830.69
Improvement on selection	: -43.69 %
Improvement on crossover	: 44.93 %

**Back**



Dialog

Introduce epoch's name to create sche

Normal

Population Length

3

Repetitions

3

Temperature

0.00

Reduction

0.00

Acceptance

0.00

Algorithm to run :

Genetic

Simulated Annealing

Back

Figura D10 - Criação de um horário

**class Class Model**

```

classDiagram
    class University {
        + epochs: std::vector<Epoch*>
        + exams: std::vector<Exam*>
        + name: std::string
        + students: std::vector<Student*>
        + addEpoch(Epoch*) : void
        + addExam(Exam*) : void
        + addStudent(Student*) : void
        + addStudentExam(string, string) : Epoch*
        + addSubscription(int, int, int) : void
        + deleteStudent(string, string) : Epoch*
        + findById(vector<T*>, int) : T*
        + findExam(Exam*) : Exam*
        + findStudent(Student*) : Student*
        + getEpochByName(string) : Epoch*
        + getEpoch() : std::vector<Epoch*>
        + getExamId(string) : int
        + getExams() : std::vector<Exam*>
        + getStudentId(string) : int
        + getStudents() : std::vector<Student*>
        + removeExam(Exam*) : void
        + removeStudent(Student*) : void
        + University(std::vector<Student*>, std::vector<Exam*>)
        + ~University()
    }

    class Epoch {
        - currentId: int
        - endDate: tm
        - epochName: std::string
        - global: Schedule*
        - id: int
        - initDate: tm
        - numDays: int
        - subs: std::vector<Subscription*>
        + addStudentExam(Student*, Exam*) : bool
        + addSubscription(Subscription*) : void
        + commonStudents(Exam*, Exam*) : bool
        + deleteStudent(string, string) : bool
        + Epoch(std::string, int, int, int, int, int, int)
        + getEndDate() : struct tm {query}
        + getExams() : std::vector<Exam*> {query}
        + getId() : int {query}
        + getInitDate() : struct tm {query}
        + getName() : std::string {query}
        + getNumDays() : int {query}
        + getSchedule() : Schedule*
        + getStudentExam(string) : vector<string>
        + getStudents(Exam*) : vector<Student*> {query}
        + getSubscriptions() : std::vector<Subscription*> {query}
        + getWeekDay(int, int, int) : int
        + notFound(vector<T*>, T*) : bool {query}
        + setEndDate(int, int) : void
        + setInitDate(int, int) : void
        + setSchedule(Schedule*) : void
    }

    class Exam {
        - currentId: int
        - duration: int
        - id: int
        - myClass: Class
        + displayExam() : std::string {query}
        + displayInfo() : std::string {query}
        + Exam(string)
        + Exam(Class, int)
        + ~Exam()
        + getClass() : std::string {query}
        + getDuration() : int {query}
        + getId() : int {query}
        + getInfo() : std::string {query}
        + getYear() : long {query}
        + operator==(Exam*) : bool {query}
        + setClassName(std::string) : void
    }

    class Student {
        - currentId: int
        - id: int
        - name: std::string
        + getId() : int {query}
        + getInfo() : std::string {query}
        + getName() : std::string {query}
        + operator==(Student*) : bool {query}
        + setName(std::string) : void
        + Student()
        + Student(std::string)
        + Student(int, std::string)
        + ~Student()
    }

    class Subscription {
        - exam: Exam*
        - student: Student*
        + getExam() : Exam* {query}
        + getStudent() : Student* {query}
        + Subscription(Student*, Exam*)
    }

    class Schedule {
        - currentId: int
        - debug: bool
        - examSlot: vector<pair<Exam*, int>>
        - firstWeekDay: int
        - fitness: int
        - id: int
        - maxRouletteProb: double
        - schedule: vector<Exam*>
        - subs: vector<Subscription*>
        + addExams(std::vector<Exam*>, std::vector<pair<Exam*, int>>) : void
        + calculateFitness() : int
        + calculateMaxRouletteProb(double, double) : double
        + commonStudents(Exam*, Exam*) : bool
        + consecutiveDaysExams(int, int) : bool
        + createDisplay(int, Exam*, int) : std::string
        + createRandomSchedule(vector<Exam*>, int) : bool
        + getExamsAtDay(int, vector<string>, bool) : std::vector<string>
        + getExamSlot() : std::vector<pair<Exam*, int>> {query}
        + getFitness() : int {query}
        + getId() : int {query}
        + getMaxRouletteProb() : double {query}
        + getNumExams() : int {query}
        + getPossiblePositions(pair<Exam*, int>) : vector<int>
        + mutate(int) : void
        + optimize() : void
        + printExams() : void
        + removeFromVector(vector<int>, int, int) : vector<int>
        + Schedule()
        + ~Schedule()
        + setDebug(bool) : void
        + setFirstWeekDay(int) : void
        + setFitness(int) : void
        + setId(int) : void
        + setMaxRouletteProb(double) : void
        + setSubscriptions(vector<Subscription*>) : void
        + updateExamPosition(pair<Exam*, int>) : void
        + updateSchedule(std::vector<pair<Exam*, int>>, int) : void
        + friend operator<<(ostream&, Schedule&) : ostream&
    }

    class SimulatedAnnealing {
        - acceptance: float
        - bestSolutionEver: Schedule*
        - currentSolution: Schedule*
        - statistics: SAStatistics*
        - temperature: float
        - temperatureReduction: float
        + applyRandomChanges(Schedule*, int) : void
        + chooseNextSolution(float) : Schedule*
        + chooseWorstSolution(Schedule*, float) : bool {query}
        + getStatistic() : SAStatistics*
        + run() : void
        + SimulatedAnnealing(Epoch*, bool, float, float, float, Schedule*)
        + SimulatedAnnealing(Epoch*, bool, float, float, float)
    }

    class Algorithm {
        # debug: bool
        # epoch: Epoch*
        # maxSlots: int
        + Algorithm(Epoch*, bool)
        + createRandomSchedule(vector<Exam*>) : Schedule*
        + createRandomSchedule(bool, Epoch*) : Schedule*
        + randomExams(vector<Exam*>, vector<Exam*>)
        + run() : void
    }

    class Genetic {
        - elitistPop: vector<Schedule*>
        - numReps: int
        - population: vector<Schedule*>
        - populationLength: int
        - statistics: GStatistics*
        + calculateFitness() : void
        + calculateFitness(vector<Schedule*>) : void
        + calculatePopulationFitness(vector<Schedule*>) : int
        + createMap(vector<pair<Exam*, int>>, vector<pair<Exam*, int>>, int) : vector<pair<Exam*, int>>
        + createRandomProb(double, int) : void
        + crossover() : void
        + executeCrossover(vector<Schedule*>) : void
        + fitnessProbabilities(int) : void
        + Genetic(Epoch*, bool, int, vector<Schedule*>, int)
        + Genetic(Epoch*, bool, int, int)
        + getBestSchedule(vector<Schedule*>) : int
        + getPopulationFitness() : int
        + getStatistic() : GStatistics*
        + mutation() : void
        + populate(vector<Exam*>) : void
        + populate(Epoch*, int, bool) : vector<Schedule*>
        + run() : void
        + selectCrossoverPopulation() : vector<Schedule*>
        + selectElitistPopulation() : vector<Schedule*>
        + selectNextPopulation() : void
        + selectRemainingPopulation(double, vector<Schedule*>) : void
    }

    class Class {
        - name: std::string
        - year: int
        + Class(std::string, int)
        + Class()
        + getName() : string {query}
        + getYear() : int {query}
        + setName(string) : void
        + setYear(int) : void
    }

    University "1" -- "*" Epoch
    University "1" -- "*" Exam
    University "1" -- "*" Student
    Epoch "1" -- "*" Subscription
    Epoch "1" -- "*" Schedule
    Exam "1" -- "*" Class
    Student "1" -- "*" Subscription
    Subscription "1" -- "1" Exam
    Subscription "1" -- "1" Student
    Schedule "1" -- "*" Subscription
    Schedule "1" -- "*" Exam
    SimulatedAnnealing "1" -- "1" Epoch
    SimulatedAnnealing "1" -- "1" Schedule
    Algorithm "1" -- "1" Epoch
    Algorithm "1" -- "1" Schedule
    Genetic "1" -- "1" Epoch
    Genetic "1" -- "1" Schedule
    Class "1" -- "1" Exam
    Class "1" -- "1" Student
  
```

The diagram illustrates a Class Class Model for a university scheduling system. It includes the following classes and their attributes/operations:

- University**:
  - Attributes: `epochs: std::vector<Epoch*>`, `exams: std::vector<Exam*>`, `name: std::string`, `students: std::vector<Student*>`.
  - Operations: `addEpoch(Epoch*) : void`, `addExam(Exam*) : void`, `addStudent(Student*) : void`, `addStudentExam(string, string) : Epoch*`, `addSubscription(int, int, int) : void`, `deleteStudent(string, string) : Epoch*`, `findById(vector<T*>, int) : T*`, `findExam(Exam*) : Exam*`, `findStudent(Student*) : Student*`, `getEpochByName(string) : Epoch*`, `getEpoch() : std::vector<Epoch*>`, `getExamId(string) : int`, `getExams() : std::vector<Exam*>`, `getStudentId(string) : int`, `getStudents() : std::vector<Student*>`, `removeExam(Exam*) : void`, `removeStudent(Student*) : void`, `University(std::vector<Student*>, std::vector<Exam*>)`, `~University()`.
- Epoch**:
  - Attributes: `currentId: int`, `endDate: tm`, `epochName: std::string`, `global: Schedule*`, `id: int`, `initDate: tm`, `numDays: int`, `subs: std::vector<Subscription*>`.
  - Operations: `addStudentExam(Student*, Exam*) : bool`, `addSubscription(Subscription*) : void`, `commonStudents(Exam*, Exam*) : bool`, `deleteStudent(string, string) : bool`, `Epoch(std::string, int, int, int, int, int, int)`, `getEndDate() : struct tm {query}`, `getExams() : std::vector<Exam*> {query}`, `getId() : int {query}`, `getInitDate() : struct tm {query}`, `getName() : std::string {query}`, `getNumDays() : int {query}`, `getSchedule() : Schedule*`, `getStudentExam(string) : vector<string>`, `getStudents(Exam*) : vector<Student*> {query}`, `getSubscriptions() : std::vector<Subscription*> {query}`, `getWeekDay(int, int, int) : int`, `notFound(vector<T*>, T*) : bool {query}`, `setEndDate(int, int) : void`, `setInitDate(int, int) : void`, `setSchedule(Schedule*) : void`.
- Exam**:
  - Attributes: `currentId: int`, `duration: int`, `id: int`, `myClass: Class`.
  - Operations: `displayExam() : std::string {query}`, `displayInfo() : std::string {query}`, `Exam(string)`, `Exam(Class, int)`, `~Exam()`, `getClass() : std::string {query}`, `getDuration() : int {query}`, `getId() : int {query}`, `getInfo() : std::string {query}`, `getYear() : long {query}`, `operator==(Exam*) : bool {query}`, `setClassName(std::string) : void`.
- Student**:
  - Attributes: `currentId: int`, `id: int`, `name: std::string`.
  - Operations: `getId() : int {query}`, `getInfo() : std::string {query}`, `getName() : std::string {query}`, `operator==(Student*) : bool {query}`, `setName(std::string) : void`, `Student()`, `Student(std::string)`, `Student(int, std::string)`, `~Student()`.
- Subscription**:
  - Attributes: `exam: Exam*`, `student: Student*`.
  - Operations: `getExam() : Exam* {query}`, `getStudent() : Student* {query}`, `Subscription(Student*, Exam*)`.
- Schedule**:
  - Attributes: `currentId: int`, `debug: bool`, `examSlot: vector<pair<Exam*, int>>`, `firstWeekDay: int`, `fitness: int`, `id: int`, `maxRouletteProb: double`, `schedule: vector<Exam*>`, `subs: vector<Subscription*>`.
  - Operations: `addExams(std::vector<Exam*>, std::vector<pair<Exam*, int>>) : void`, `calculateFitness() : int`, `calculateMaxRouletteProb(double, double) : double`, `commonStudents(Exam*, Exam*) : bool`, `consecutiveDaysExams(int, int) : bool`, `createDisplay(int, Exam*, int) : std::string`, `createRandomSchedule(vector<Exam*>, int) : bool`, `getExamsAtDay(int, vector<string>, bool) : std::vector<string>`, `getExamSlot() : std::vector<pair<Exam*, int>> {query}`, `getFitness() : int {query}`, `getId() : int {query}`, `getMaxRoulette`