**4**

**Descrição do Programa**

**4.1. Introdução**

Neste capitulo pretende-se descrever os elementos chave do programa, a metodologia adotada no seu desenvolvimento assim como dar uma ideia clara da comunicação existente entre os vários componentes do software.

O programa é composto pelo algoritmo genético e pela interface de comunicação com o software FEM. Neste caso o software FEM utilizado é o Robot Structural Analysis da Autodesk.

A comunicação com o Robot foi feita com a API (Application Programming Interface) disponibilizada pela Autodesk e que permite controlo total do Robot de forma automática em qualquer linguagem com suporte de COM interface (C++, C#, VB, etc.)

Para desenvolver este programa utilizou-se a linguagem de programação C# pois oferecia um bom compromisso entre performance e tempo de desenvolvimento. Ao não utilizar C++ evitou-se o controlo manual de alocação de memória o que permitiu focar mais tempo no desenvolvimento do programa e não a resolver problemas de *memory leaks*. Por outro lado, ao não utilizar uma linguagem de mais alto nível como por exemplo pyhton obtemos um programa com melhor performance e onde a comunicação com o Robot fica facilitada.

Como software de teste na definição inicial do ADN de um individuo tipo foi utilizado o Rhinoceros + Grasshopper pois a sua capacidade de design paramétrico permitiu testar de forma rápida cada alteração feita no algoritmo de geração do ADN, esta metodologia sera descrita com mais detalhe mais à frente.

**4.2. Síntese de funcionamento**

Este subcapítulo pretende dar uma ideia geral dos vários componentes do programa e da comunicação entre eles, cada componente será detalhado nos subcapítulos seguintes.

Na primeira fase do programa é pedido ao utilizador para introduzir as características das secções que com que o programa deve procurar a estrutura mais eficiente. De seguida o utilizador introduz dados acerca da geometria da torre, parâmetros como largura entre montantes, altura dos cabos, e numero de cabos.

Com esta informação é gerada uma estrutura que é a representação do geométrica das informações contidas no código genético da estrutura base. Esta é uma estrutura em que cada nó está ligado aos nós imediatamente a cima e a baixo no seu próprio plano, ou seja a estrutura inicial é uma estrutura altamente redundante com todas as barras possíveis activas e é o ponto de partida do algoritmo genético.

Após criada esta primeira estrutura o utilizador pode no Robot criar os casos de carga adequados e que o programa vai aplicar a todas as estruturas analisadas durante a busca do opimo. É de seguida criada a população inicial com a dimensão desejada pelo utilizador e onde as secções das barras são escolhidas de forma aleatória (incluído a opção de descativar a barra) e as coordenadas dos nós também sofrem uma ligeira mutação nos pontos onde tal é permitido.

Em cada individuo da população inicial é feita a analise FEM do robot de onde se recebem esforços em cada barra, essa informação é introduzida como input na função de fitness.

De forma a determinar os comprimentos de encurvadura dos diferentes tipos de elementos da estrutura (Montantes, bracing, barras horizontais) a informação geométrica do modelo é processada por um algoritmo que percorre cada tipo de barra, analisa que outros elementos lhe estão conectados e daí cria uma lista de calculo com a informação de uma nova barra analítica, por que barras do model é ela constituída e o seu comprimento de encurvadura. Após finalizada esta rotina de processamento da geometria recebemos a lista de calculo a serem feitos para cada individuo de forma a obter o fitness de cada um.

Esse calculo é feito na função de verificação do EC3 presente na classe “Calc\_operations”, daí é recebido o fator de utilização (u/f) em cada barra. São elaboradas três listas: u/f baixo, aceitável, demasiado alto. Barras presentes na primeira ou ultima lista adicionam peso à estrutura como critério de penalização de barras sob/subdimensionadas.

Por fim à eventual pontuação resultante da penalização anterior é adicionado o peso real da estrutura que constitui a classificação de fitness da solução.

Após esta classificação da solução inicial a função de seleção é chama a escolher dois elementos para dar origem a um novo individuo da próxima geração. Esse individuo passa pelos operadores genéticos d crossover e mutação. Esse novo individuo substitui o pior individuo da população atual, é avaliado e o ciclo de evolução é repetido.

[IMAGEM DOS VARIOS ELEMENTOS E SETINHAS]

**4.3. Código genético base**

A descrição da estrutura base é critica para o algoritmo genético pois uma definição incorreta pode levar o algoritmo a procurar soluções desinteressantes, ou não adequadas ao problema, uma solução demasiado restritiva pode impedir o algoritmo de explorar caminhos que levariam a um melhor ótimo local.

Para não restringir o algoritmo à partida optou-se por gerar uma estrutura com todas as conexões entre nós possíveis (e aceitáveis numa solução ótima).

Para desenvolver o algoritmo que gera esta estrutura base foi utilizado o Rhinoceros em conjunto com o addon Grasshopper. Estes dois programas em conjunto aumentam a rapidez na definição da geometria pois qualquer alteração de código é imediatamente representada graficamente sem ser necessário compilar cada alteração feita. Apesar do grasshopper se comercialmente conhecido pela sua capacidade de programação visual esses componentes foram utilizados apenas para representar parâmetros de input como altura da estrutura e posição dos cabos. O algoritmo em si foi feito com o elemento “*C# scripting*” presente no programa pois desta forma a transferência do código para o programa final fica reduzida a uma operação de copiar e colar.

[Imagem do Grasshopper]

No programa final após transferido o algoritmo desenvolvido no grasshopper foi necessário pensar na melhor forma de introduzir as barras e nós numa estrutura adequada às operações de mutação, crossover e analise regulamentar a que cada estrutura vai estar sujeita.

Optou-se por utilizar como data type um array bidimensional com a seguinte estrutura:

O array dos nós para alem do numero de cada nó e as suas coordenadas existe um campo que recebe uma constante de mutação, esta constante permite definir pontos onde as operações de mutação têm de ser mais incrementais, exemplo dessa situação são os braços onde uma mutação à escala do permitido na torre gera uma distorção inaceitável do braço.

O array das barras define o numero da barra, os seus pontos de inicio e fim, de seguida atribui uma classificação de 1 ou 0 conforme essa barra possa ou não desaparecer, por exemplo uma secção de montante não pode desaparecer pelo que este campo recebe o valor de 0, por fim é definida a seção e o id que será útil para identificar tipos diferente de barras quando se calcular os comprimentos de encurvadura para a verificação do EC3.

De seguida mostram-se excertos de código da criação do array, e do algoritmo de geração da estrutura.

|  |
| --- |
| public Genome(double Largura,int Altura, double horiz\_div,double subdiv, int N\_cabos, int[] h\_cabos,double[] dist\_centro)  {  (…)  // criação da matriz dos nós e barras  pt\_cloud = new double[5, (int)(17\*N\_cabos + 4 + (4 \* subdiv) \* (horiz\_div - 1))];  bars = new double[6, (int)((4 \* horiz\_div - 8) \* (subdiv \* subdiv) + (12 \* horiz\_div - 12) \* subdiv - 8 \* horiz\_div + 36 \* N\_cabos + 20)];  // chamada para as funções que geram os nós e barras  pt\_add\_tower(ref pt\_cloud, Largura, Altura,horiz\_div,subdiv,ref pt\_cnt);  pt\_add\_arms(ref pt\_cloud, Largura, Altura, horiz\_div, subdiv, N\_cabos, h\_cabos, dist\_centro, ref pt\_cnt, ref connection\_rings);  bar\_cnt = connect\_bars(ref bars,(int)subdiv,(int)horiz\_div);  add\_arm\_bars(ref bars, ref bar\_cnt, connection\_rings, (int)subdiv, N\_cabos,(int)horiz\_div);  // armazenar qual a numero de barras da torre (subtrair ao total as barras dos braços)  towerBar\_cnt = bar\_cnt - 36 \* N\_cabos;  } |

|  |
| --- |
| private void pt\_add\_tower(ref double[,] pt, double Largura, int Altura, double horiz\_div, double subdiv,ref int \_pt\_cnt)  {  (…)  //##################//  //main pt cloud loop//  //##################//  for (h = 1; h <= horiz\_div - 1; h++)  {  double scale\_factor = (1 - (h / horiz\_div));  double step = h \* tilt;  if (!reverse)  { // x++ y++  for (x = 0; x <= subdiv; x++)  {  addPt(ref pt, pt\_num, step + x \* (Largura / subdiv) \* scale\_factor, step, h \* ring\_z\_step,(double)Altura);  pt\_num++;  if (x == subdiv)  {  for (y = 1; y <= subdiv; y++)  {  addPt(ref pt, pt\_num, Largura - step, step + y \* (Largura / subdiv) \* scale\_factor, h \* ring\_z\_step, (double)Altura);  pt\_num++;  if (x == subdiv && y == subdiv) { reverse = true; }  (…)  private int connect\_bars(ref double[,] bars,int subdiv, int horiz\_div)  {  int bar\_num = 0;  //Support pts  for (int i = 0; i <= 4; i++)  {  if (i == 0)  {  for (int j = 4; j <= 4 + subdiv; j++)  {  if (j == 4)  {  addBar(ref bars, bar\_num, 0, j, 0, 0,0);//cantos não são desativados: id=0  bar\_num++;  }  else {  addBar(ref bars, bar\_num, 0, j,1,0,1); // id =1 bracing  bar\_num++;  }  }  for (int j = 8 + 4 \* (subdiv - 1) - 1; j >= 8 + 4 \* (subdiv - 1) - subdiv; j--)  {  addBar(ref bars, bar\_num, 0, j, 1, 0,1); // id =1 bracing  bar\_num++;  }  }  (…) |

[Adicionar também a imagem do grasshopper -> robot]

**4.4. População inicial**

A população inicial é criada no momento em que o utilizador define a geometria inicial, o seu tamanho é também definido pelo utilizador.

O tamanho adequado da população inicial varia com o numero de secções a experimentar e o numero de barras da estrutura, isto é: para uma estrutura com poucas barras e poucas secções a experimentar um numero reduzido de indivíduos é suficiente para propagar o espaço de soluções de forma a chegar a uma solução aceitável, por outro lado para uma estrutura complexa o espaço de busca tem de ser alargado para permitir uma busca com o mesmo rigor do espaço de soluções.

Como cada individuo parte da definição genética base é necessário introduzir uma mutação inicial aos nós por alteração de coordenadas e às secções por alteração das secções de cada individuo. Para este efeito é utilizado um random seeder presente na biblioteca .Net cujo código é de seguida apresentado:

|  |
| --- |
| public Individual(Genome \_baseDNA, ref Random rndm)  {  this.fitness = 0.0;  \_DNA = new Genome(); // create new Genome  //if the following is not done the same matrix is always changing (need to create copies):  \_DNA.pt\_cloud = (double[,])\_baseDNA.pt\_cloud.Clone(); // copy by value the pt cloud[]  \_DNA.bars = (double[,])\_baseDNA.bars.Clone(); //copy by value the bars[]  for (int i = 4; i < Genome.pt\_cnt; i++) // start at 4 to fix supports  {  //mutate initial pt coord.  this.\_DNA.pt\_cloud[1, i] += rndm.Next(-1, 1) \* rndm.NextDouble() \* this.\_DNA.pt\_cloud[4, i];//X  this.\_DNA.pt\_cloud[2, i] += rndm.Next(-1, 1) \* rndm.NextDouble() \* this.\_DNA.pt\_cloud[4, i];//Y  this.\_DNA.pt\_cloud[3, i] += rndm.Next(-1, 1) \* rndm.NextDouble() \* this.\_DNA.pt\_cloud[4, i]\*0.2;//Z  }  //define init sections  for (int i = 0; i < Genome.bar\_cnt; i++)  {  if (this.\_DNA.bars[4,i] == -1)  {  this.\_DNA.bars[4, i] = 0; //a primeira secção a definir será a dos braços  //secção de braços  }else  {  this.\_DNA.bars[4, i] = Population.rand.Next(0,Sections.count-1); // começar com random section  } |

Com esta rotina finalizada para cada individuo a população inicial está pronta para ser analisada pela função “*Evaluate*” descrita no próximo subcapítulo. A rotina responsável pela avaliação de cada individuo é a seguinte:

|  |
| --- |
| for (int i=0; i<199; i++)  {  Generation.Text = i.ToString();  // CurrentPop = new Population(NextPop.ind);  if (i == 0)  {  for (int a = 0; a < Population.Pop\_size; a++)  {  CurrentPop.ind[a].Evaluate();  c++;  series.Points.AddXY(c, CurrentPop.ind[a].fitness);  //Robot\_call.Robot\_interactive(true);  }  }  (…) |

**4.5. Função “evaluate”**

A função Evaluate é responsável pela maior parte dos cálculos intensivos responsáveis pelo calculo de fitness de cada individuo e engloba uma serie de operações intermedias.

É responsável por representar cada estrutura no robot, adicionar apoios, correr a analise e receber os resultados.

De seguida corre para cada tipo de barra (montante, bracing ou horizontal bars) uma rotina responsável por analisar as conexões existentes e traduzir essa informação num modelo analítico interno (IAM) que servira de instruções de calculo para as verificações de acordo com os regulamentos. De seguida estão excertos das rotinas responsáveis pelo IAM.

|  |
| --- |
| List<Int32> temp = new List<Int32>();  for (int i = 0; i < Genome.horizd-1; i++)  {  if (i == 0) // barra inicial  {  temp.Add(i + 1);  //se tiver barras horiz ou bracing no sup e/ou mudar a secção  if (v\_braced(1, new List<Int32>() { 1, 2 })) {  this.Leg\_ops.Add(new Calc\_operations(1, temp, (int)this.\_DNA.bars[4, 0], (int)this.\_DNA.bars[5, 0])); // se a 1 barra esta braced add logo ao calculo  temp = new List<Int32>();  } else if (this.\_DNA.bars[4, 0] != this.\_DNA.bars[4, (8 \* (int)Genome.subd) + 4])  {  this.Leg\_ops.Add(new Calc\_operations(1, temp, (int)this.\_DNA.bars[4, 0], (int)this.\_DNA.bars[5, 0]));  temp = new List<Int32>();  }else{// passa para a outra barra )}  }else {  int bar\_ind = 8 \* (int)Genome.subd + 5 + (i - 1) \* (4 \* (int)Genome.subd \* (int)Genome.subd + 8 \* (int)Genome.subd - 8);  if (v\_braced(bar\_ind, new List<Int32>() { 1, 2 }))  {  if (temp != null) // se nao for null o start bar é o temp[0]  {  temp.Add(bar\_ind); // add esta barra  this.Leg\_ops.Add(new Calc\_operations(temp[0], temp, (int)this.\_DNA.bars[4, bar\_ind-1], (int)this.\_DNA.bars[5, bar\_ind-1]));  temp = new List<Int32>();  } else  (…) |

Cada um dos três algoritmos funciona com o mesmo principio, percorrer cada tipologia de barra, em cada extremo das barras verificar se estão conectadas a barras cuja tipologia assegure o contraventamento, caso essa condição se verifique adiciona imediatamente a barra à lista final. Caso isso não se verifique prossegue para a barra seguinte até encontrar um ponto de contraventamento, nesse momento adicionar outra entrada à lista final onde inclui uma barra ficticea composta por essa mesma barra contraventada e todas as outras anteriores que foi registando como não contraventadas.

[meter imagem exemplo de um elemento LEG]

A lista final que irá ser introduzida no calculo pelos regulamentos tem a seguinte forma:

Neste formato a lista está pronta para servir de input à função de verificações de acordo com os regulamentos. Os procedimentos de verificação já foram descritos no capitulo 2 em detalhe, a função aqui utilizada é apenas a transcrição do eurocódigo para algoritmo razão pela qual se dispensa uma descrição detalhada do seguinte excerto da função:

|  |
| --- |
| /// Leg calc excerpt  /// BS EN 1993-1-1:2005 ///  /// BS EN 1993-3-1.2006 ///  ///#######################///  for (int i = 0; i < calc\_ops.Count; i++)  {  double Nsd = 0;  double L = 0;  // get max N of every bar //  for (int b = 0; b < calc\_ops[i].next\_bars.Count; b++)  {  if (Math.Abs(results[2, calc\_ops[i].next\_bars[b] - 1]) > Nsd) { Nsd = results[2, calc\_ops[i].next\_bars[b] - 1]; }  }  // get total L for buckling //  for (int b = 0; b < calc\_ops[i].next\_bars.Count; b++)  {  L += results[1, calc\_ops[i].next\_bars[b] - 1];  }  if (Nsd > 0)//tension  {  double u\_f = 0;  double Nu\_rd = Sections.Area[calc\_ops[i].section\_id] \* 275000; //Aeff \* fy  u\_f = Nsd / Nu\_rd;  for (int y = 0; y < calc\_ops[i].next\_bars.Count; y++)  {  repair\_instructions.Add(new double[] { calc\_ops[i].next\_bars[y], Math.Abs(u\_f) }); // add to repair list (each individual bar)  }  }  else { //compression  //resistence check  double u\_f = 0;  double Nc\_rd = Sections.Area[calc\_ops[i].section\_id] \* 275000;  u\_f = Nsd / Nc\_rd;  //buckling check  double lambda = L / Sections.ivv[calc\_ops[i].section\_id]; // L/ivv  double \_lambda = lambda / (93.9 \* Math.Sqrt(235 / 275));  double k = 0.8 + (\_lambda / 10);  if (k > 1) { k = 1; }  if (k < 0.9) { k = 0.9; }  double \_lambda\_eff = k \* \_lambda;  double fi = 0.5 \* (1 + 0.34 \* (\_lambda - 0.2) + \_lambda \* \_lambda);  double xi = 1 / (fi + Math.Sqrt(fi \* fi - \_lambda \* \_lambda));  double Nb\_rd = xi \* Sections.Area[calc\_ops[i].section\_id] \* 275000;  double b\_uf = Nsd / Nb\_rd;  if (u\_f < b\_uf) { u\_f = b\_uf; }  for (int y = 0; y < calc\_ops[i].next\_bars.Count; y++)  {  repair\_instructions.Add(new double[] { calc\_ops[i].next\_bars[y], Math.Abs(u\_f) }); // add to repair list (each individual bar)  }  }  } |

Como será percetível do excerto, cada barra analisada é adicionada a uma lista que detalha a barra analisada e o facto de utilização obtido. Esta lista é introduzida numa rotina responsável por colocar cada barra em três possíveis listas, lista de barras subdimensionadas, sobredimensionadas e com u/f aceitável. Esta função recorre a um simples algoritmo de *sorting* pelo que não é aqui detalhada.

As três listas resultantes desta operação são utilizadas durante a avaliação do fitness da estrutura, o fitness de cada individuo é traduzido pelo peso em toneladas da estrutura somado de possíveis penalizações. Essas penalizações têm por base a quantidade de barras sub/sobredimensionadas e são um parâmetro que deve ser ajustado de estrutura para estrutura para a penalização ser da escala do peso próprio da estrutura. De seguida apresenta-se um excerto relativo à penalização para barras subdimensionadas:

|  |
| --- |
| ///Under Designed  ///  for (int i = 0; i < udr\_dsgn.Count; i++)  {  double[] temp = udr\_dsgn[i];  if (this.\_DNA.bars[4, (int)temp[0] - 1] != Section\_count - 1) // se ainda não estiver com a maior secção pode aumentar  {  Console.WriteLine("Poder aumentar Sec da barra" + temp[0]);  this.fitness += 5; // penalização de 5 ton  }  else  {  this.fitness += 10;  need\_bigger\_sect = true;  Console.WriteLine("---------------------------");  Console.WriteLine("!!!NEEDS BIGGER SECTIONS!!!");  Console.WriteLine("---------------------------");  }  } |

Após correr todas estas operações o individuo em analise recebe a sua avaliação de fitness, esse valor será utilizado para ordenar o individuo na população, por ordem crescente de fitness.

**4.6. Função seleção**

Para a implementação da função de seleção optou-se pelo algoritmo de Tournament Selection após testes de convergência entre este e o algoritmo roulette wheel. Ambos atingiram a convergência numa solução, mas dado ao problema em causa ser de minimização (do peso da estrutura) o algoritmo de Tournament Selection é mais simples de implementar sem introduzir problema de fitness scaling no momento de selecção.

Tal como descrito no Cap 3 este algoritmo funciona ao construir uma lista de uma dada dimensão onde coloca elementos selecionados aleatoriamente da população atual. Os elementos dessa lista são depois comparados entre si e o melhor individuo é o que tem maior probabilidade de ser selecionado.

No programa desenvolvido implementou-se este algoritmo apenas com um parâmetro ajustável, o numero de elementos que são selecionados para fazer parte da lista. O algoritmo com apenas este parâmetro ajustável demonstrou ser suficiente para facilmente ajustar a selection pressure e desta forma controlar a rapidez de convergência e o detalhe da busca pelo espaço de soluções.

Algoritmo de seleção:

|  |
| --- |
| public static Individual Tournament\_selection(Individual[] pop)  {  int selection\_pressure = (int)Pop\_size/6; // adjust selection pressure  int[] tournament = new int[selection\_pressure];  for (int i = 0; i < selection\_pressure; i++)  {  tournament[i] = Population.rand.Next(0, Pop\_size - 1);  }  Array.Sort(tournament);  return pop[tournament.Last<int>()];  } |

**4.7. Operadores genéticos**

4.7.1. Crossover

4.7.2. Mutação gaussiana