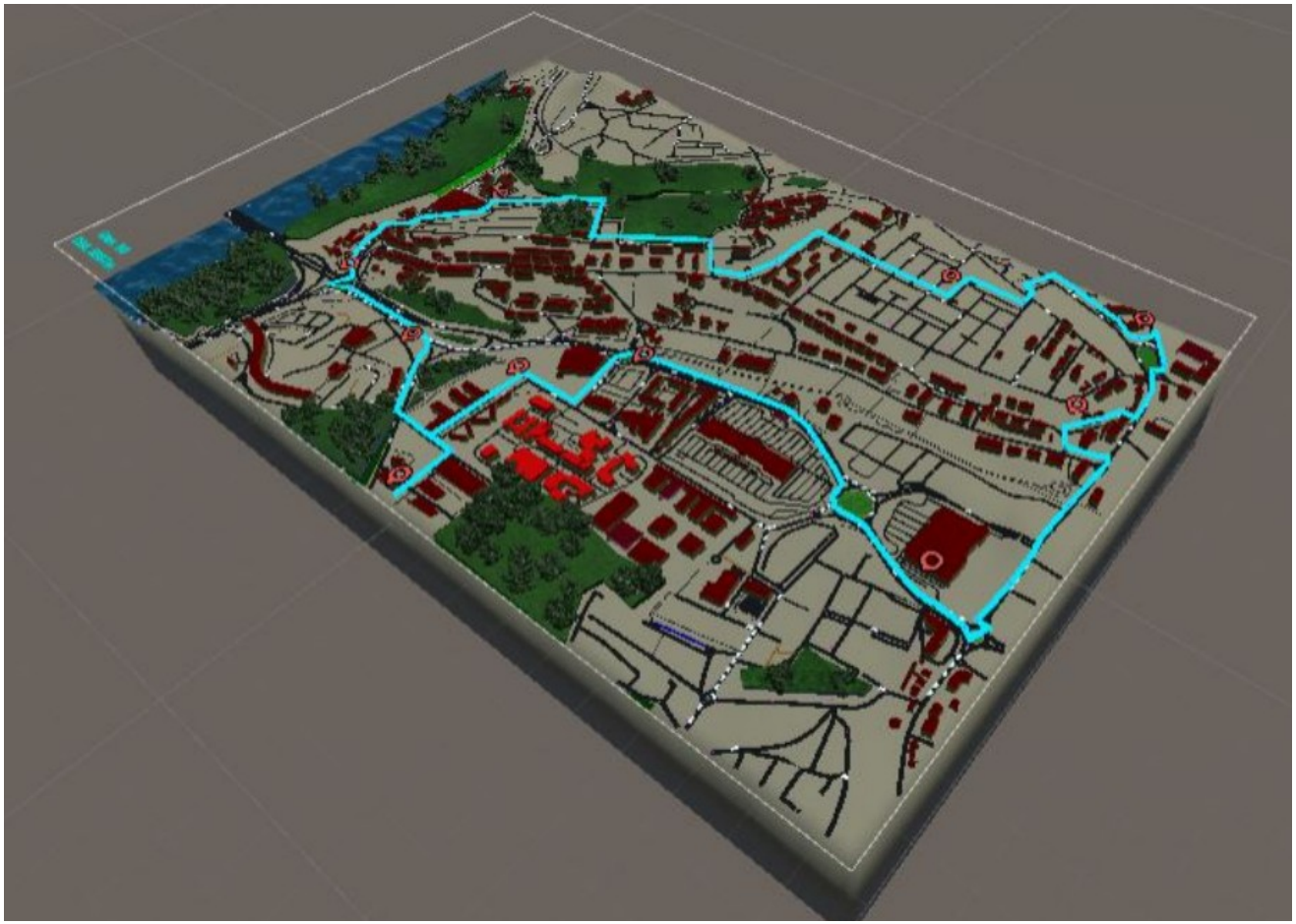
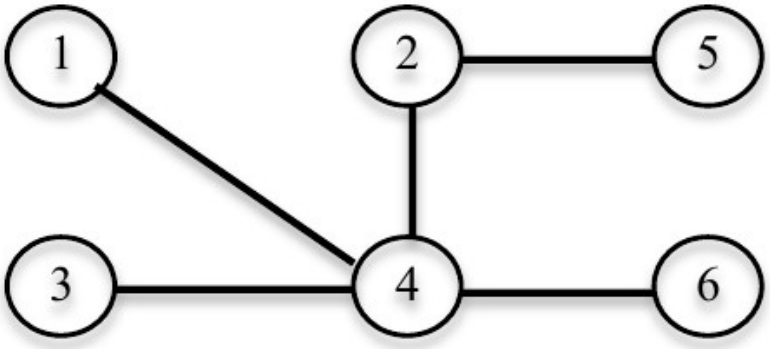
Computação Evolucionária

- Ficha 8



- Encontrar conjunto estável grupo de vértices - vértices sem ligação entre si




```
c Instancia de teste para
problema
c IIA 2021 22
p edge 6 5 Grafo de 6 vértices, 5 arestas
e 1 4 Aresta que liga vértices 1 e 4
e 2 4
e 2 5
e 3 4
e 4 6
```

$S1 = \{1, 3, 5\}$ → Solução de qualidade 3

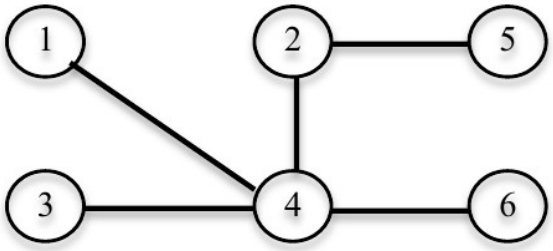
$S2 = \{1, 2, 3, 6\}$ → Solução de qualidade 4 (melhor)

$S3 = \{1, 2, 3, 4\}$ → Solução inválida, porque há uma aresta entre 3 e 4

Trabalho prático

- Gerar soluções, avaliar qualidade delas com base nos dados do ficheiro, usando
  - Pesquisa local
  - Algoritmo evolutivo
  - Métodos híbridos
- Lançar hipóteses, testar, analisar resultados

```
c Instancia de teste para
problema
c IIA 2021 22
p edge 6 5 Grafo de 6 vértices, 5 arestas
e 1 4 Aresta que liga vértices 1 e 4
e 2 4
e 2 5
e 3 4
e 4 6
```



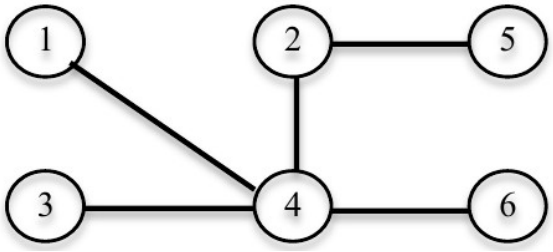
$S1 = \{1, 3, 5\}$ → Solução de qualidade 3

$S2 = \{1, 2, 3, 6\}$ → Solução de qualidade 4 (melhor)

$S3 = \{1, 2, 3, 4\}$ → Solução inválida, porque há uma aresta entre 3 e 4

Trabalho prático

- 0 – nodo não pertence ao conjunto estável
- 1 – nodo pertence ao conjunto estável




```
c Instancia de teste para
problema
c IIA 2021 22
p edge 6 5
e 1 4
e 2 4
e 2 5
e 3 4
e 4 6
```

Grafo de 6 vértices, 5 arestas

Aresta que liga vértices 1 e 4

$S1 = \{1, 3, 5\}$ → S1 = [1 0 1 0 1 0] → Solução de qualidade 3

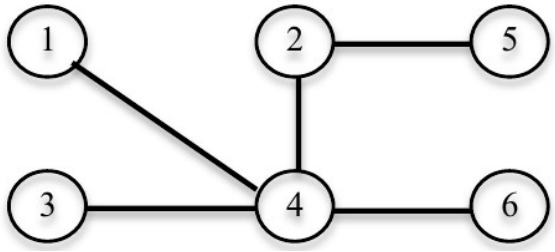
$S2 = \{1, 2, 3, 6\}$ → S2 = [1 1 1 0 0 1] → Solução de qualidade 4 (melhor)

$S3 = \{1, 2, 3, 4\}$ → S3 = [1 1 1 1 0 0] → Solução inválida, porque há uma aresta entre 3 e 4

Trabalho prático

- Calcular fitness:
  - Percorrer o vetor, verificar se algum dos vértices a 1 tem uma aresta que liga com outro vértice a 1

```
c Instancia de teste para
problema
c IIA 2021 22
p edge 6 5 Grafo de 6 vértices, 5 arestas
e 1 4 Aresta que liga vértices 1 e 4
e 2 4
e 2 5
e 3 4
e 4 6
```



$S1 = \{1, 3, 5\}$ → S1 = [1 0 1 0 1 0] → Solução de qualidade 3

$S2 = \{1, 2, 3, 6\}$ → S2 = [1 1 1 0 0 1] → Solução de qualidade 4 (melhor)

$S3 = \{1, 2, 3, 4\}$ → S3 = [1 1 1 1 0 0] → Solução inválida, porque há uma aresta entre 3 e 4

Trabalho prático

Implementar os 3 métodos seguintes e efetuar um estudo comparativo sobre o desempenho da otimização:

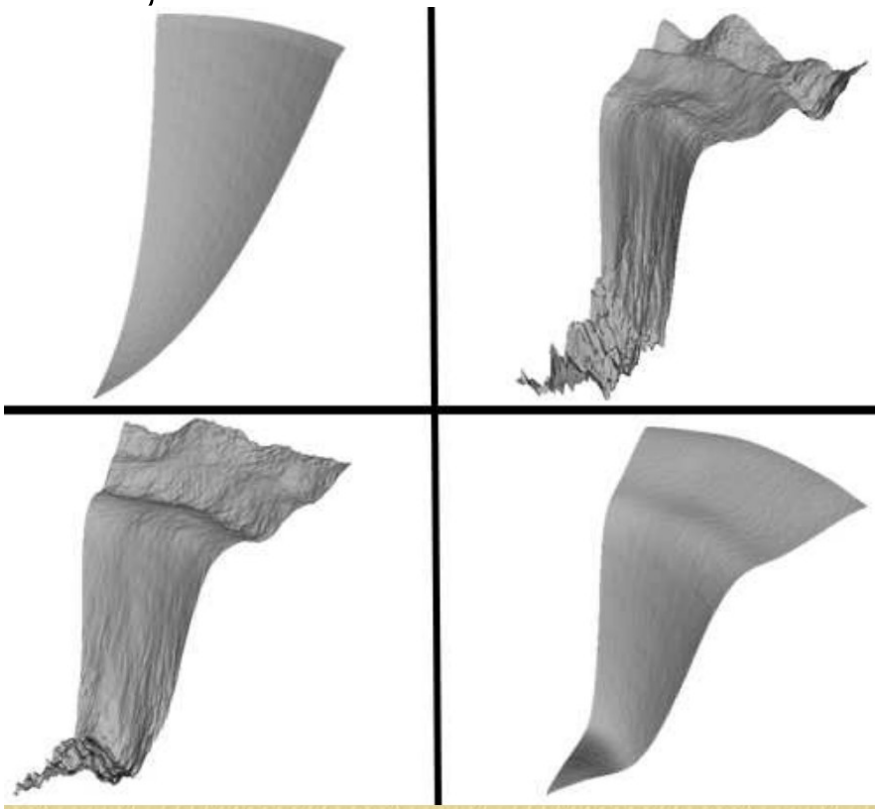
- Algoritmo de **pesquisa local** (trepa-colinas, recristalização simulada ou outro);
  - Proponha forma de lidar com soluções inválidas: evitando que surjam na população, ou se surgirem, comparar estratégias de penalização ou de reparação
- **Algoritmo evolutivo;**
  - Devem ser explorados diferentes operadores genéticos;
  - Proponha forma de lidar com soluções inválidas: evitando que surjam na população, ou se surgirem, comparar estratégias de penalização ou de reparação.
- **Método híbrido** combinando as duas abordagens anteriores.

Algoritmos genéticos

- Agente materializa a solução para um problema
- Exemplo: caminhos como soluções com diferentes custos:
  - Coimbra-Lisboa-Porto-Viseu (200+300+120 = 620)
  - Porto-Viseu-Coimbra-Lisboa (120+50+200 = 370)
  - Lisboa-Viseu-Porto-Coimbra (220+120+100 = 440)
  - ...
- A estrutura do agente **evolui**, de forma a melhorar a sua adaptação ao ambiente
- Metáfora biológica
  - Indivíduos mais adaptados são valorizados
  - Indivíduos menos adaptados são tendencialmente descartados

Exemplos de soluções encontradas por computação evolucionária

- Diogo Duarte, MSc Univ. Coimbra, 2014: otimização da forma de uma lente para distribuição dos raios de luz de um poste sobre a estrada
  - Cada desenho de lente é um indivíduo, mais ou menos adaptado conforme a sua capacidade de distribuir a luz



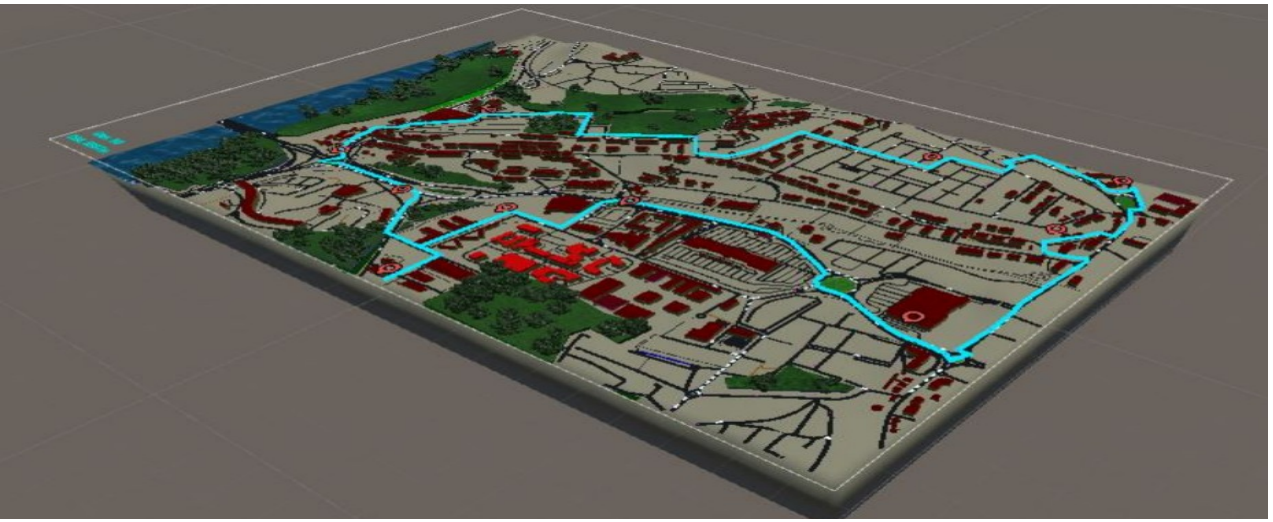
Exemplos de soluções encontradas por computação evolucionária

- Hajji Mohammed, ISEC 2018: resolução do problema do caixeiro viajante usando algoritmo genético e A*

Neste exemplo cada caminho possível é um "indivíduo" da população – os mais curtos terão melhor adaptação

Metáfora biológica na origem dos algoritmos genéticos

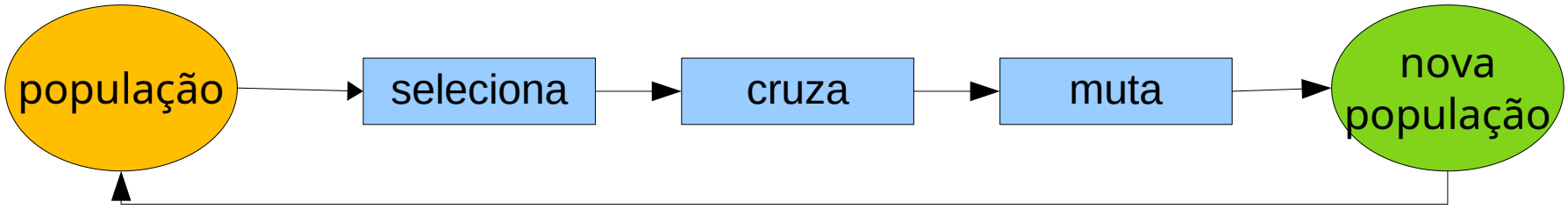
- **Cromossoma**: cadeias de ADN que caraterizam indivíduo
- Os cromossomas são constituídos por **genes**, que por sua vez caraterizam feições (ex, cor dos olhos, cor do cabelo, cor da pele, etc.)
- **Função de adaptação**: permite calcular valor do cromossoma (ex., distância que o caixeiro viajante tem de percorrer)



Neste exemplo cada caminho é um cromossoma.
A soma dos km é a adaptação.

Como funciona?

- Função genetico (pop_inicial, f_adaptação, cond_paragem)
  1) Calcula **adaptação** de cada indivíduo de pop_inicial
  2) **REPETE**
     1.1 **Seleciona** indivíduos para reproduzir (segundo método de seleção escolhido)
     1.2 **Cruza** indivíduos escolhidos segundo algoritmo de cruzamento adotado e gera população nova P
     1.3 **Muta** alguns genes de indivíduos de P segundo critérios de mutação escolhidos
     1.4 Calcula **adaptação** de cada indivíduo de P
     ATÉ encontrar indivíduo suficientemente apto ou atingir a condição de paragem



Parâmetros importantes de um algoritmo genético

- Tamanho da população
  - Uma população grande tem maior diversidade genética, mas o processamento de cada geração é também mais demorado
  - Populações de 30 a 100 indivíduos normalmente resolvem a maior parte dos problemas
- Elitismo
  - Consiste em copiar um ou mais indivíduos dos melhor adaptados para a geração seguinte inalterado
  - Os melhores nunca se perdem

Formas de cruzamento

- Cruzamento com corte num ponto
  - A=000000, B= 111111 => AB1 = 000**111**, AB2 = **111**000
- Cruzamento com corte em 2 pontos
  - A=000000, B= **111111** => AB1 = 00**11**00, AB2 = **11**00**11**
- Cruzamento uniforme, um gene de cada progenitor (normalmente escolhido aleatoriamente)
  - A=000000, B= 111111 => AB1 = 0**1**0**1**0**1**, AB2 = **1**0**1**0**1**0

Exemplo - problema

- Pesquisa de número binário de 4 bits em que o objetivo é encontrar o máximo número de dígitos 1.
  - Tamanho da população inicial: 5 indivíduos gerados aleatoriamente
  - **Função de adaptação**: número de dígitos a 1
  - **Seleção**: torneio binário
  - **Cruzamento**: 1 ponto
  - **Elitismo**: passa o indivíduo mais adaptado
  - **Mutação**: 1 bit num indivíduo
  - Condição de paragem: número máximo de gerações (2)

Indivíduo	***Adaptação***
0001	*1*

Exemplo – população inicial

- População inicial e respetiva adaptação de cada indivíduo

Indivíduo	***Adaptação***
0001	*1*
1010	*2*
0000	*0*
1011	*3*
1100	*2*

Exemplo – seleção por torneio binário

- Torneio binário: Agrupam-se cromossomas dois a dois, seleciona-se para crossover o melhor dos dois
- No exemplo o 0000 até vai 2 vezes a torneio e perde sempre

Torneios G1, G2

Indivíduo	*Adaptação*	*Grupo*
~~0001~~	~~*1*~~	~~*G1*~~
1010	***2***	***G1***
0000	*0*	
1011	***3***	***G2***
~~1100~~	~~*2*~~	~~*G2*~~

Torneios G3, G4

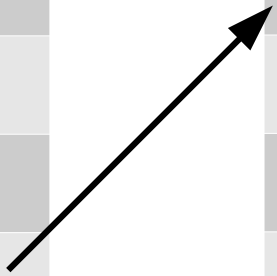
Indivíduo	*Adaptação*	*Grupo*
0001	***1***	***G3***
1010	***2***	*Selecionado G1*
~~0000~~	~~*0*~~	~~*G3 G4*~~
1011	***3***	*Selecionado G2*
1100	***2***	***G4***

Exemplo – elitismo

- Passagem automática de 1 indivíduo por elitismo

População inicial

Indivíduo	*Adaptação*	
0001	*1*	
1010	*2*	
~~0000~~	*~~0~~*	
1011	*3*	
1100	*2*	



População nova

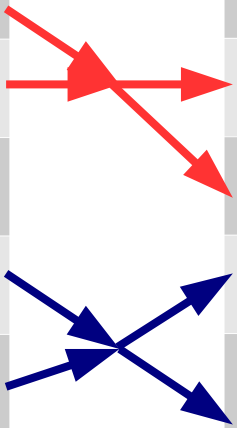
Indivíduo
1011

Exemplo – crossover

- Seleção para cruzamento. Primeiro cruza com segundo, quarto cruza com quinto
  - Poderia atribuir-se maior probabilidade de seleção aos elementos mais adaptados, permitindo que cruzassem mais vezes

População selecionada

Indivíduo	*Adaptação*	
0001	*1*	
1010	*2*	
1011	*3*	
1100	*2*	



População nova

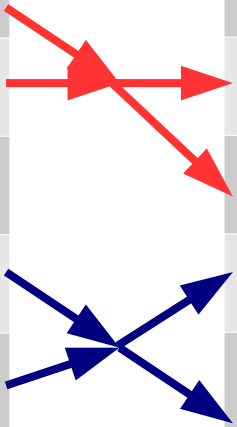
Indivíduo
1011

Exemplo – cruzamento 1 ponto

- Cruzamento 1 ponto, em que os descendentes recebem 50% dos genes de cada progenitor

População selecionada

Indivíduo	*Adaptação*	
0001	*1*	
1010	*2*	
1011	*3*	
1100	*2*	



População nova

Indivíduo
1011
1001
0010
1111
1000

Exemplo – mutação

- Mutação: altera aleatoriamente um bit de um dos novos indivíduos. Neste exemplo a alteração é prejudicial, - o indivíduo deixa de estar 100 % adaptado e passa a estar apenas 75 %

População inicial

Indivíduo	*Adaptação*	
0001	*1*	
1010	*2*	
0000	*0*	
1011	*3*	
1100	*2*	

População nova

Indivíduo
1011
1001
0010
11**1**0**0**1
1000

Troca-se aqui um 1 por um 0

Exemplo – resultado de 1 iteração

- A segunda geração já tem 2 indivíduos com adaptação 3, um com adaptação 2 e dois com adaptação 1. A adaptação média é superior.

Geração 1

Indivíduo	*Adaptação*
0001	*1*
1010	*2*
0000	*0*
1011	*3*
1100	*2*

Geração 2

Indivíduo	Adaptação
1011	3
1001	2
0010	1
1101	3
1000	1

Sugestões

- Verifique a geração da “Mona Lisa” usando a implementação adaptativa no site http://alteredqualia.com/visualization/evolve/
- Verifique a evolução de uma população artificial no site http://math.hws.edu/eck/js/genetic-algorithm/ga-info.html

Ficha 8 – Problema da mochila

- Existem **N** objetos, cada um com peso $\mathbf{W_i}$ e valor $\mathbf{V_i}$
- É preciso maximizar **V**, sem exceder a capacidade **C** da mochila
- Objetivo:
  - Maximizar **sum(V_i)**, garantindo **sum (W_i) < C**
- Cromossoma
  - 1 indica que objeto i está na mochila, 0 que não está:

0	1	2	3	4	...	N-1
0	**0**	**1**	**1**	**0**	**...**	**1**

Ficha 8 – Problema da mochila

- População inicial: gerar aleatoriamente
- Condição de paragem: máximo de gerações
- Função de adaptação:
  - Soma do valor dos objetos na mochila, sendo que:
    - Valor é zero se o peso exceder a capacidade da mochila

0	1	2	3	4	...	N-1
0	**0**	**1**	**1**	**0**	**...**	**1**

- Ficheiro de dados

```
pop:      100          população inicial
pm:       0.01         probabilidade de mutação
pr:   0.3              probabilidade de cruzamento
tsize:    2            tamanho do torneio
max_gen: 2500          máximo de gerações
obj:      100          número de objetos
cap:      250          capacidade da mochila (peso)

Weight      Profit     peso e lucro de cada objeto
2    8
5    1
10   5
```

- Ficheiros de dados knap_xxx.txt
  - São lidos no main() para variável struct info EA_param;

```
pop:      100        população inicial
pm:       0.01       probabilidade de mutação
pr:  0.3             probabilidade de cruzamento
tsize:    2          tamanho do torneio
max_gen: 2500        máximo de gerações
obj:      100        número de objetos
cap:      250        capacidade da mochila (peso)

Weight      Profit   peso e lucro de cada objeto
2    8
5    1
10   5
```

- População guardada na tabela *pop* (os selecionados para cruzamento na tabela *parents*)
- Tabela *pop* constituída por indivíduos com
  - Cromossoma, fitness, “valido”

```
typedef struct individual chrom, *pchrom;

struct individual {
    int     p[MAX_OBJ]; // Solução (cromossoma)
    float   fitness;          // Adaptação
    int     valido;           // 1 se for uma solução válida
};
```

Ficha 8 – 4.1

- Para ficheiro knap_100.txt, preencher folha de cálculo com resultados (média 10 - 30 repetições para maior confiança)
  - Probabilidade de recombinação pr de 0.3, 0.5, 0.7
  - Probabilidade de mutação pm de 0, 0.001, 0.01, 0.05
  - Tamanho da população de 10, 50 e 100 (máx. gerações 25000, 5000, 2500)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Ficheiro Knap_100.txt															
2			Algoritmo base		com penalização		com reparação 1		com reparação 2		com mutação por troca		com recombinação 2 pontos		com recombinação uniforme	
3	Parâmetros Fixos	Parâmetros a variar	Best	MBF	Best	MBF	Best	MBF	Best	MBF	Best	MBF	Best	MBF	Best	MBF
4	ger = 2500	pr = 0.3														
5	pop = 100	pr = 0.5														
6	pm = 0.01	pr = 0.7														
7	ger = 2500	pm = 0.0														
8	pop = 100	pm = 0.001														
9		pm = 0.01														
10	pr = 0.7	pm = 0.05														
11	pr = 0.7	pop = 10 (ger = 25K)														
12	pm = melhor valor obtido	pop = 50 (ger = 5K)														
13		pop = 100 (ger = 2.5K)														

Ficha 8 – 4.1

```
1   pop:»     100
2   pm:» 0.01
3   pr:» 0.3
4
5   tsize:»   2
6   max_gen: 2500
7   obj:»     1000
8   cap:»     12747
9
10  Weight» Profit
11  48» 27
12  12» 34
13  50» 5
14  42» 14
```

- Os parâmetros são alterados diretamente no ficheiro
  - Probabilidade de recombinação pr de 0.3, 0.5, 0.7
  - pm de 0, 0.001, 0.01, 0.05
  - máx. gerações 25000, 5000, 2500

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Ficheiro Knap_100.txt															
2			Algoritmo base		com penalização		com reparação 1		com reparação 2		com mutação por troca		com recombinação 2 pontos		com recombinação uniforme	
3	Parâmetros Fixos	Parâmetros a variar	Best	MBF	Best	MBF	Best	MBF	Best	MBF	Best	MBF	Best	MBF	Best	MBF
4	ger = 2500	pr = 0.3														
5	pop = 100	pr = 0.5														
6	pm = 0.01	pr = 0.7														
7	ger = 2500	pm = 0.0														
8	pop = 100	pm = 0.001														
9		pm = 0.01														
10	pr = 0.7	pm = 0.05														
11	pr = 0.7	pop = 10 (ger = 25K)														
12	pm = melhor valor obtido	pop = 50 (ger = 5K)														
13		pop = 100 (ger = 2.5K)														

Ficha 8 – 4.2 – penalização linear

- Alterar função de fitness – eval_individual(), ficheiro funcao.c
- Calcular ro = max(valor / peso)
- Para soluções em que o peso excede a capacidade da mochila, subtrair ao fitness a penalidade
  - ro * (sum_weight – capacidade)

```
for (i=0; i < d.numGenes; i++){
   if (sol[i] == 1) { // Verifica se objecto i esta na mochila
     sum_weight += mat[i][0];// Actualiza o peso total
     sum_profit += mat[i][1];// Actualiza o lucro total
     }
}
if (sum_weight > d.capacity){        // Solucao inválida
    *v = 0;
    return 0;
}
```

Ficha 8 – 4.2 – Reparação1 - aleatória

- Retirar cálculo do ro
- Quando uma solução é inválida, escolher aleatoriamente objetos e retirá-los até satisfazer peso máximo
- Alterar função de fitness – eval_individual(), ficheiro util.c:
  - Quando peso é maior do que a capacidade:
    - Alterar aleatoriamente um 1 para 0 na tabela sol[]
    - Recalcular peso e lucro
- Preencher coluna na folha de cálculo

Ficha 8 – 4.2 – Reparação2 com pesquisa sôfrega

- Quando uma solução é inválida, retirar objetos até satisfazer peso máximo, escolhendo primeiro os de valor mais baixo
- Alterar função de fitness – eval_individual(), ficheiro util.c:
  - Quando peso é maior do que a capacidade:
    - Procurar objeto a 1 na tabela sol[] que tem valor mais baixo e retirá-lo
    - Recalcular peso e lucro
- Preencher coluna na folha de cálculo

Ficha 8 – 4.3 – Mutação por troca

- Manter a Reparação 2
- Implementar Mutação por troca:
  - Ficheiro algoritmo.c
  - Alterar a função mutation() para selecionar aleatoriamente dois objetos e trocá-los, garantindo que um deles estava fora e outro dentro da mochila

```
void mutation(pchrom offspring, struct info d){
    int i, j;

    for (i=0; i<d.popsize; i++)
       for (j=0; j<d.numGenes; j++)
          if (rand_01() < d.pm)
             offspring[i].p[j] = !(offspring[i].p[j]);
}
```

Ficha 8 – 4.4 – Recombinação com dois pontos de corte

- Implementar recombinação com dois pontos de corte
- Esta alteração é na função crossover(), ficheiro algoritmo.c, e que faz apenas um corte no ponto **point**
- Gerar outro ponto **point2** e implementar outro ciclo for() para o novo ponto e corte

```
void crossover(pchrom parents, struct info d, pchrom offspring){
  int i, j, point;

  for (i=0; i<d.popsize; i+=2){
    if (rand_01() < d.pr){
        point = random_l_h(0, d.numGenes-1);
        for (j=0; j<point; j++)
……
```

Ficha 8 – 4.4 – Recombinação uniforme

- Implementar recombinação ponto a ponto
- Deixa de haver pontos de corte
- Na função crossover, para cada gene, gerar um número aleatório: se der 0, filho 1 recebe gene do pai 1 e filho 2 recebe gene do pai 2, se der 1 faz-se ao contrário

- Cruzamento com corte num ponto
  - A=000000, B= 111111 => AB1 = 000**111**, AB2 = **111**000
- Cruzamento com corte em 2 pontos
  - A=000000, B= **111111** => AB1 = 00**11**00, AB2 = **11**00**11**
- Cruzamento uniforme, um gene de cada progenitor (normalmente escolhido aleatoriamente)
  - A=000000, B= 111111 => AB1 = 0**1**0**1**0**1**, AB2 = **1**0**1**0**1**0

Ficha 8 – 4.5 – Seleção com diferente tamanho de torneio

- Alterar função tournament(), do ficheiro algoritmo.c, para o torneio em vez de ser binário ser entre t_size elementos
- Variável t_size é lida do ficheiro

```
void tournament(pchrom pop, struct info d, pchrom parents){
  int i, x1, x2;

  …

}
```

Ficha 8 – 4.6 – Algoritmo híbrido trepa-colinas + genético

- Criar função gera_vizinho(), recebe uma solução e troca elementos, retirando e/ou colocando objetos na mochila
- Criar a função trepa_colinas(), conforme algoritmo visto anteriormente: gera vizinho, aceita-o se tiver melhor fitness do que a solução atual

Ficha 8 – 4.6.i – Otimizar população inicial

- Gerar população inicial do algoritmo genético, otimizá-la usando o trepa_colinas antes de entrar no algoritmo genético
- No main(), chamar o trepa_colinas a seguir ao init_pop()

```
…
for (r=0; r<runs; r++){
   printf("Repeticao %d\n",r+1);

   // Geração da população inicial
   pop = init_pop(EA_param);
// Chamar   trepa_colinas () aqui, para cada elemento
   // Avalia a população inicial
   evaluate(pop, EA_param, mat);
   gen_actual = 1;
...
```

Ficha 8 – 4.6.ii – Refinar soluções geradas

- Usar o trepa colinas no fim, depois de ter a melhor solução encontrada pelo algoritmo genético
- No main(), quando pára o ciclo, realizar o trepa_colinas sobre o best_run

```
…
while (gen_actual <= EA_param.numGenerations){
        ...
   best_run = get_best(pop, EA_param, best_run);
   gen_actual++;
}
...
// Escreve resultados da repetição que terminou
printf("\nRepeticao %d:", r);
// Chamar trepa_colinas aqui
write_best(best_run, EA_param);
...
```

Ficha 8 – 4.6.iii – Refinar soluções geradas a cada geração

- Usar o trepa colinas no fim, depois de ter a melhor solução encontrada pelo algoritmo genético
- No main(), ciclo de otimização, realizar o trepa_colinas antes de ser determinado o best_run

```
…
while (gen_actual <= EA_param.numGenerations){
        ...
   // Chamar trepa_colinas aqui
   best_run = get_best(pop, EA_param, best_run);
   gen_actual++;
}
```