RELATÓRIO

Trabalho Prático 2

Introdução à Inteligência Artificial

Licenciatura em Engenharia Informática

O segundo trabalho prático consiste em desenvolver um programa em C capaz de resolver o Problema do Conjunto Estável Máximo.

Trabalho realizado por:

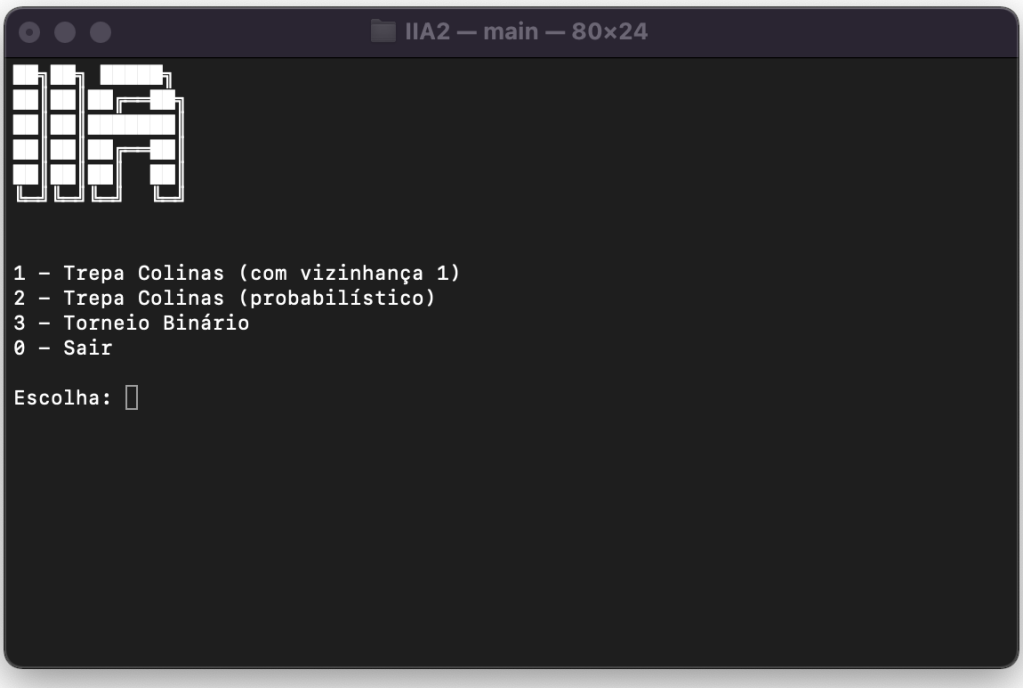
Tomás Gomes Silva - 2020143845

Tomás da Cunha Pinto - 2020144067

Índice

Introdução

O segundo trabalho prático de Introdução à Inteligência Artificial consiste no desenvolvimento de um programa na linguagem de programação C capaz de resolver o Problema do Conjunto Estável Máximo.
Foram-nos fornecidas várias instâncias de variados tamanhos para correrm os no programa e efetuarmos um estudo experimental.



Implementação

Trepa-Colinas (com vizinhança 1)

Começamos por ler o ficheiro contendo a informação das ligações entre os vértices com a função preenche_matriz e seguidamente alocamos memória para os vetores que vão guardar a solução atual e a melhor solução.

```
matriz = preenche_matriz(file, &vert, &iter);
solucao = malloc(sizeof(int) * vert);
best = malloc(sizeof(int) * vert);

if (solucao == NULL || best == NULL)
{
    printf("[ERRO] Ocorreu um problema ao alocar memória.");
    exit(1);
}
```

Para efetuarmos várias iterações do Trepa-Colinas corremos um ciclo o número de vezes definida anteriormente que consiste em gerar uma solução com o trepa-colinas utilizando a solução anterior e verificando se a solução devolvida pelo algoritmo é superior à anterior (problema de maximização). Caso isso se verifique a melhor solução é substituída com a solução calculada.

```
for (i = 0; i < runs; i++)
{

    gerar_solinicial(solucao, vert);
    custo = trepaColinas(solucao, matriz, vert, iter);

    // printf("\nRepeticao %d: ", i);
    // escrever_solucao(solucao, vert);
    // printf("Custo final: -\n", custo);

    mbf += custo;
    if (i == 0 || bestCusto < custo)
    {
        bestCusto = custo;
        substitui(best, solucao, vert);
    }
}
```

No final do ciclo, é mostrada a melhor solução e a média das soluções encontradas para o ficheiro que foi dado ao programa no início.

```
printf("\n\nMBF: %f\n", mbf / i);
printf("\nMelhor solucao encontrada");
escrever_solucao(best, vert);
printf("Custo final: -\n", bestCusto);
```

O trepa-colinas com vizinhança 1, avalia a solução, e aceita o vizinho apenas se este for maior do que o atual.

```
int trepaColinas(int sol[], int *mat, int vert, int num_iter)
{
    int *nova_sol, custo, custo_viz, i;

    nova_sol = malloc(sizeof(int) * vert);
    if (nova_sol == NULL)
    {
        printf("Erro na alocacao de memoria");
        exit(1);
    }
    // Avalia solucao inicial
    custo = calcula_fit(sol, mat, vert);
    for (i = 0; i < num_iter; i++)
    {

        // Gera vizinho
        gerar_vizinho(sol, nova_sol, vert);
        // Avalia vizinho
        custo_viz = calcula_fit(nova_sol, mat, vert);

        if (custo_viz > custo)
        {
            substitui(sol, nova_sol, vert);
            custo = custo_viz;
            break;
        }
    }
    free(nova_sol);

    return custo;
}
```

Trepa-Colinas (probabilístico)

O trepa-colinas probabilístico é bastante semelhante ao trepa-colinas só que para além de depender de um vizinho depende de uma probabilidade pré-determinada.

```c
int trepaColinasProb(int sol[], int *mat, int vert, int num_iter)
{
    int *nova_sol, custo, custo_viz, i;
    float probs = 0.05;
    nova_sol = malloc(sizeof(int) * vert);
    if (nova_sol == NULL)
    {
        printf("Erro na alocacao de memoria");
        exit(1);
    }
    // Avalia solucao inicial
    custo = calcula_fit(sol, mat, vert);
    for (i = 0; i < num_iter; i++)
    {

        // Gera vizinho
        gerar_vizinho(sol, nova_sol, vert);
        // Avalia vizinho
        custo_viz = calcula_fit(nova_sol, mat, vert);
        if (custo_viz > custo)
        {
            substitui(sol, nova_sol, vert);
            custo = custo_viz;
        }
        else if (probEvento(probs))
        {
            substitui(sol, nova_sol, vert);
            custo = custo_viz;
        }
    }
    free(nova_sol);

    return custo;
}
```

```c
void gerar_vizinho(int a[], int b[], int n)
{
    int i, p1, p2, temp;

    for (i = 0; i < n; i++)
        b[i] = a[i];

    p1 = random_int(0, n - 1);

    p2 = random_int(0, n - 1);

    // Troca
    temp = b[p1];
    b[p1] = b[p2];
    b[p2] = temp;
}
```

```c
int calcula_fit(int a[], int *mat, int vert)
{
    int total = 0;
    int i, j;

    for (i = 0; i < vert; i++)
        if (a[i] == 0)
        {
            for (j = 0; j < vert; j++)
                if (a[j] == 1 && *(mat + i * vert + j) == 1)
                    total++;
        }
    return total;
}
```

Torneio Binário

O algoritmo Torneio Binário, ou em inglês, Binary Tournament foi implementado da seguinte forma:

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

    for (i = 0; i < d.pop; i++)
    {
        x1 = random_int(0, d.pop - 1);
        do
            x2 = random_int(0, d.pop - 1);
        while (x1 == x2);
        if (pop[x1].fitness < pop[x2].fitness)
            parents[i] = pop[x1];
        else
            parents[i] = pop[x2];
    }
}
```

```c
void mutation(pchrom offspring, info d)
{
    int p1, p2, aux, i, j;
    for (i = 0; i < d.pop; i++)
    {
        if (random_float_01() < d.pm)
        {
            for (j = 0; j < d.vertices / 2; j++)
            {
                p1 = random_int(0, d.vertices - 1);
                do
                {
                    p2 = random_int(0, d.vertices - 1);
                } while (p1 == p2);
                aux = offspring[i].p[p1];
                offspring[i].p[p1] = offspring[i].p[p2];
                offspring[i].p[p2] = aux;
            }
        }
    }
}
```

```c
void genetic_operators(pchrom parents, info d, pchrom offspring)
{
    crossover(parents, d, offspring);
    mutation(offspring, d);
}
```

```c
void crossover(pchrom parents, info d, pchrom offspring)
{
    int point, i, j, m, n;
    int aux = 0;
    for (i = 0; i < d.pop; i += 2)
    {
        if (random_float_01() < d.pr)
        {
            point = random_int(0, d.vertices - 1);
            for (j = 0; j < point; j++)
            {
                offspring[i].p[j] = parents[i].p[j];
                offspring[i + 1].p[j] = parents[i + 1].p[j];
            }
            for (j = point; j < d.vertices; j++)
            {
                for (m = 0; m < d.vertices; m++)
                {
                    for (n = 0; n < j; n++)
                    {
                        if (parents[i + 1].p[m] == offspring[i].p[n])
                            aux = 1;
                    }
                    if (aux == 0)
                    {
                        offspring[i].p[j] = parents[i + 1].p[m];
                        break;
                    }
                    aux = 0;
                }
```

```
                for (m = 0; m < d.vertices; m++)
                {
                    for (n = 0; n < j; n++)
                    {
                        if (parents[i].p[m] == offspring[i + 1].p[n])
                            aux = 1;
                    }
                    if (aux == 0)
                    {
                        offspring[i + 1].p[j] = parents[i].p[m];
                        break;
                    }
                    aux = 0;
                }
            }
        }
        else
        {
            offspring[i] = parents[i];
            offspring[i + 1] = parents[i + 1];
        }
    }
}
```

Na função main, temos um ciclo para correr o algoritmo n vezes de modo a aprimorar a solução final.

```
for (i = 0; i < runs; i++)
{
    printf("Repetição %d\n", i);
    pop = init_pop(param);
    evaluate(pop, param, mat);
    best_run = pop[0];
    best_run = get_best(pop, param, best_run);
    parents = malloc(sizeof(chrom) * param.pop);
    if (parents == NULL)
    {
        printf("[ERRO] Ocorreu um problema ao alocar memória.");
        exit(1);
    }
    gen_atual = 1;
    while (gen_atual <= param.numGenerations)
    {
        tournament(pop, param, parents);
        genetic_operators(parents, param, pop);
        evaluate(pop, param, mat);
        best_run = get_best(pop, param, best_run);
        gen_atual++;
    }
    write_best(best_run, param);
    mbf += best_run.fitness;
    if (i == 0 || best_ever.fitness < best_run.fitness)
    {
        best_ever = best_run;
    }
    free(parents);
    for (int j = 0; j < param.pop; j++)
    {
        free(pop[j].p);
        free(parents[j].p);
    }
    free(pop);
    free(parents);
}
```

No final do ciclo é mostrada a solução final com maior cardinalidade (problema de maximização).

Análise de Resultados

O estudo experimental consistiu em correr o programa com vários ficheiros e com um número de iterações variável para observar a qualidade das soluções calculadas.
Como é de esperar, correr 10000 vezes o algoritmo leva-nos mais perto da solução.

Trepa-Colinas (com vizinhança 1)

Não aceitando soluções iguais

Para não aceitar soluções iguais basta verificar se o custa da solução calculada é maior do que o custo da melhor solução. Se isso se verificar basta substituir a melhor solução com a solução calculada.

```
if (i == 0 || bestCusto < custo)
{
    bestCusto = custo;
    substitui(best, solucao, vert);
}
```

Pesquisa Local - Trepa-Colinas (com vizinhança 1)				
	10 iterações	100 iterações	1000 iterações	10000 iterações
Melhor	9	10	10	10
MBF	6.3	7.36	7.089	7.0794

Aceitando soluções iguais

De forma a aceitar soluções iguais, quando verificamos se uma solução calculada é melhor do que a melhor atual, basta verificar se ela é maior ou **igual**. Desta forma, aceitamos soluções que são tão boas ou melhores do que a melhor global.

```
if (i == 0 || bestCusto <= custo)
{
    bestCusto = custo;
    substitui(best, solucao, vert);
}
```

Pesquisa Local - Trepa-Colinas (com vizinhança 1)				
	10 iterações	100 iterações	1000 iterações	10000 iterações
Melhor	9	10	10	10
MBF	6.3	7.36	7.089	7.0794

Trepa-Colinas (probabilístico)

Não aceitando soluções iguais

Para não aceitar soluções iguais basta verificar se o custa da solução calculada é maior do que o custo da melhor solução. Se isso se verificar basta substituir a melhor solução com a solução calculada.

```
if (i == 0 || bestCusto < custo)
{
    bestCusto = custo;
    substitui(best, solucao, vert);
}
```

Pesquisa Local - Trepa-Colinas (probabilístico)					
	10 iterações	100 iterações	1000 iterações	10000 iterações	
Melhor	10	10	10	10	Prob = 0.05
MBF	7.5	8.52	8.429	8.3369	
	10 iterações	100 iterações	1000 iterações	10000 iterações	
Melhor	10	10	10	10	Prob = 0.1
MBF	8.20	7.98	8.063	6.0956	
	10 iterações	100 iterações	1000 iterações	10000 iterações	
Melhor	9	10	10	10	Prob = 0.3
MBF	7	7.41	7.232	7.2552	

Foram testadas diferentes probabilidades.

Aceitando soluções iguais

De forma a aceitar soluções iguais, quando verificamos se uma solução calculada é melhor do que a melhor atual, basta verificar se ela é maior ou **igual**. Desta forma, aceitamos soluções que são tão boas ou melhores do que a melhor global.

```
if (i == 0 || bestCusto <= custo)
{
    bestCusto = custo;
    substitui(best, solucao, vert);
}
```

Pesquisa Local - Trepa-Colinas (probabilístico) aceitando soluções de custo igual					
	10 iterações	100 iterações	1000 iterações	10000 iterações	
Melhor	8	10	10	10	Prob = 0.05
MBF	9	8.21	8.327	8.3279	
	10 iterações	100 iterações	1000 iterações	10000 iterações	
Melhor	10	10	10	10	Prob = 0.1
MBF	8.6	8.14	8.051	8.0796	
	10 iterações	100 iterações	1000 iterações	10000 iterações	
Melhor	10	10	10	10	Prob = 0.3
MBF	7.3	7.16	7.318	7.2664	

Foram testadas diferentes probabilidades.

Conclusão

Este trabalho permitiu-nos observar vários algoritmos em funcionamento capazes de resolver um problema utilizando aprendizagem contínua. Para além disso permitiu-nos aprimorar o nosso nível de programação em C e tratamento de dados.
O estudo experimental permitiu-nos verificar os resultados obtidos e o quanto eles variavam com a mudança de certos parâmetros.