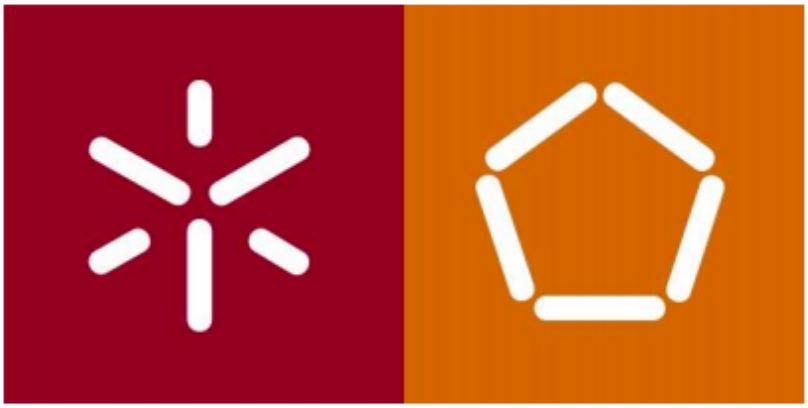

Universidade do Minho

Mestrado Integrado em Engenharia Informática

27/03/2020

Algoritmos Paralelos
Trabalho Prático 1

António Gonçalves (A85516)
Gonçalo Esteves (A85731)

Contents

1 Introdução

Neste relatório iremos expor e explicar as nossas resoluções e estratégias no que toca à resolução do trabalho proposto para a cadeira de Algoritmos Paralelos.

Neste primeiro trabalho foi-nos proposto paralelizar um algoritmo conhecido como *Room Assignment Problem*. Este é baseado na escolha de quartos por várias pessoas, que possuem preferências quanto aos companheiros de quarto, de forma a que haja um menor grau de descontentamento possível, cujos valores para cada variam entre 1 e 10.

Para isto teremos uma matriz D simétrica, onde temos o grau de descontentamento entre as várias pessoas. Assim, a posição *D[i][j]* irá representar quanto a pessoa i não gosta da pessoa j. De forma a encontrar uma boa solução, será necessário encontrar uma combinação de pessoas/quartos que minimize o somatório dos valores de descontentamento.

Uma vez que o número de possíveis distribuições é igual ao valor do produtório de todos os números ímpares entre 1 e N, onde N é o número de pessoas, torna-se fácil concluir que consoante o número de pessoas aumenta, mais difícil se torna encontrar a solução ótima do problema em tempo útil, e como tal, abordagens que nos permitem calcular uma boa solução, mesmo não sendo a melhor, em menores intervalos de tempo podem ser muito úteis.

2 Abordagens ao Problema

2.1 Método de Monte Carlo

Uma possível solução será recorrendo a Métodos de Monte Carlo. A partir de uma distribuição inicial criada aleatoriamente, serão aplicadas várias trocas com o objetivo de diminuir o valor total de descontentamento. Estas apenas serão aceites caso diminuam o custo (no nosso caso, o nível de desagrado) total da distribuição. O número de trocas é variável, uma vez que sempre que uma troca é efetuada, o contador que contabiliza o número de passos seguidos sem trocas, retorna a 0. Após várias trocas será de esperar que este valor tenda para o mínimo atingível, uma vez que o número de trocas possíveis que geram um valor de descontentamento menor do que o calculado inicialmente será gradualmente inferior.

Uma forma de otimizar a procura de uma boa distribuição (de pessoas pelos quartos), poderá passar por executar o programa múltiplas vezes, em paralelo, onde cada instância teria por base uma distribuição inicial diferente. Para isso iremos utilizar **OpenMp**, de forma a conseguir distribuir estas simulações por diferentes *threads*, utilizando o cluster SEARCH para efetuar testes, mais particularmente, um dos nodos 652.

2.2 *Simulated Annealing*

Outra solução poderá passar por uma abordagem com *Simulated Annealing*. Neste caso, a lógica seria semelhante à utilizada com o método de Monte Carlo, no entanto, para além das trocas que reduzem o custo total da distribuição, também as que aumentam poderão ser aceites. A aceitação daquelas que levam a um aumento do custo tem por base a seguinte função de probabilidade:

$$exp(-delta/T) >= random$$

Assim, *delta* representa a diferença entre o custo da distribuição atual e o da nova distribuição calculada; T é o valor da "temperatura", que diminui a cada iteração do ciclo; e *random* é um valor aleatório entre 0 e 1. Deste modo, podemos concluir que para valores de *delta* maiores a probabilidade da troca acontecer

torna-se exponencialmente menor, e que com o aumento do número de iterações a probabilidade de uma troca custosa ser aceite também vai diminuindo, já que o valor de T diminui.

Mais uma vez, por forma a determinar eficientemente qual poderá ser uma boa distribuição de pessoas pelos quartos poderemos realizar várias simulações em paralelo, onde cada uma parte de uma distribuição inicial diferente.

2.3 Algoritmo *Greedy*

Por fim, uma abordagem distinta às adotadas em cima passa pelo uso de um algoritmo *Greedy*. Desta feita, não haverá uma distribuição inicial aleatória, e, em vez disso, a cada iteração iremos procurar o parceiro ótimo, de entre os ainda disponíveis, para uma dada pessoa.

A primeira pessoa é escolhida aleatoriamente e a partir daí verificamos qual a compatibilidade desta com todas as outras, emparelhando-a num quarto com aquela com quem se sente melhor e marcando ambas como já alocadas. Depois, determinamos qual a pessoa a emparelhar no quarto seguinte e repetimos o processo, tendo em conta apenas as pessoas ainda disponíveis (por norma, a pessoa seguinte terá o identificador que se segue ao da previamente escolhida; no caso desta já se encontrar num quarto, passamos para a próxima e assim sucessivamente).

Tal como nas outras abordagens, uma possível forma de otimizar o cálculo da distribuição ótima passa por permitir que ocorram diversas simulações em simultâneo. Neste caso, cada simulação iria começar com uma pessoa diferente, por forma a que todas elas tomem o máximo de decisões diferentes, permitindo a obtenção de diferentes distribuições.

3 Análise dos Resultados

De seguida iremos analisar os resultados obtidos com a execução dos algoritmos anteriormente citados.

Por forma a analisar a performance dos mesmos em diferentes situações, fizemos variar diversos valores, de modo a verificar o comportamento dos programas perante tais alterações. Testamos variações de N (número de pessoas), *iter* (número máximo de iterações sem ocorrerem trocas), T (valor inicial da temperatura) e o *decreasing value* (o valor pelo qual se multiplica a temperatura em cada iteração, por forma a que diminua). Para garantir que as conclusões que tiravamos eram o mais corretas possível, apenas alteramos um destes valores de cada vez, e por isso foi necessário definir valores *default* para estas variáveis, sendo eles N=10000, *iter*=100, T=1 e *decreasing value*=0.999. Para além disto, decidimos também tentar perceber qual seria um bom número de *threads* por forma a efetuar os testes, sendo os resultados deste pequeno estudo abordados seguidamente.

3.1 Variação do número de *threads*

Tal como fora referido, a nossa primeira análise consistiu num pequeno estudo para ver qual poderia ser um potencial número de *threads* "ideal" para efetuar os testes. Tendo em conta os programas desenvolvidos, que fazem com que cada *thread* corra uma instância do algoritmo selecionado, sabíamos à partida que esta escolha seria importante não só pelo tempo final de execução mas também porque iria influenciar a solução final (quantas mais vezes o algoritmo é corrido para uma mesma matriz, maior é a probabilidade de encontrar uma distribuição que minimize o custo do problema).

Assim sendo, efetuamos testes para várias quantidades de *threads*, entre 1 e 20, sobre uma mesma matriz de descontentamento, sendo os resultados obtidos apresentados na página seguinte.

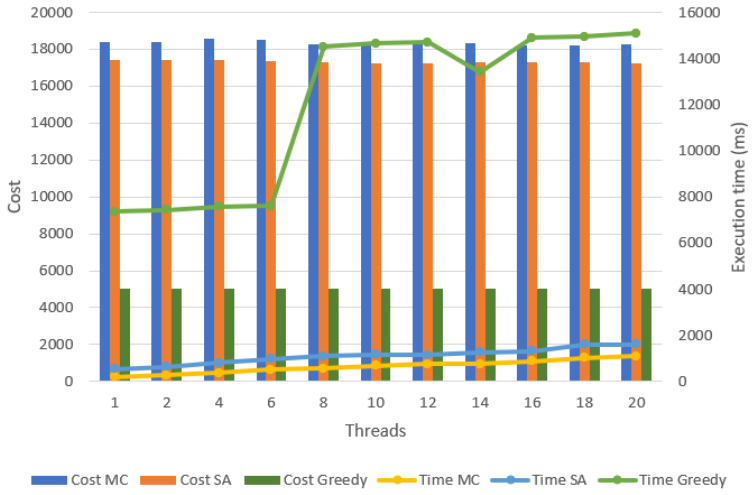

Figure 1: Resultados obtidos variando o número de *threads*

Threads	Implementation	Cost	Execution Time (ms)	Threads	Implementation	Cost	Execution Time (ms)
1	MC	18388	202,3574	12	MC	18355	761,2578
	SA	17438	537,0615		SA	17214	1135,313
	Greedy	5009	7356,937		Greedy	5005	14695,56
2	MC	18375	273,3467	14	MC	18302	769,1421
	SA	17431	633,7075		SA	17278	1266,161
	Greedy	5009	7430,963		Greedy	5004	13419,64
4	MC	18579	363,0273	16	MC	18228	884,8735
	SA	17414	814,5693		SA	17281	1316,845
	Greedy	5005	7590,103		Greedy	5005	14903,59
6	MC	18527	526,8818	18	MC	18237	1018,161
	SA	17344	989,4014		SA	17272	1611,925
	Greedy	5006	7603,509		Greedy	5004	14936,36
8	MC	18287	569,6099	20	MC	18244	1094,718
	SA	17318	1104,984		SA	17210	1617,447
	Greedy	5005	14515,71		Greedy	5005	15082,36
10	MC	18383	683,5386				
	SA	17243	1179,892				
	Greedy	5005	14656,7				

Table 1: Resultados obtidos variando o número de *threads*

Pela interpretação dos resultados obtidos, torna-se evidente concluir que o uso de mais *threads* leva a um aumento do tempo de execução, particularmente acentuado para o algoritmo *Greedy*. Embora, em teoria, isto não devesse acontecer, torna-se justificável de duas formas: primeiro, pelas naturais limitações do *hardware*, que pode responder de forma mais lenta à execução de tarefas em paralelo; segundo, porque o tempo de execução final do programa equivale ao tempo de execução da *thread* que mais demorou, e como tal, mais *threads* a correr em paralelo levam a uma maior probabilidade de que uma delas se atrase em relação às outras.

Por outro lado, também percebemos que mais *threads* a executar em paralelo nem sempre é sinónimo de obtenção de uma melhor distribuição final, uma vez que há casos em que, apesar do maior número de execuções do algoritmo obtêm-se piores custos finais.

Assim sendo, e tendo tudo isto em consideração, optamos por usar sempre 10 *threads* de agora em diante, uma vez que consideramos que este seja um valor que permite que os tempos de execução não aumentem muito quando comparados com a execução *single-thread*, e ao mesmo tempo já consegue assegurar que são corridas algumas instâncias do programa, de modo a potenciar a procurar pela solução de menor custo.

3.2 Variação do Número de Pessoas (N)

Começando depois a abordagem aos valores que faríamos variar nos próprios algoritmos, analisamos o que a variação do número total de pessoas provoca. Testamos assim para valores de N iguais a 100, 500, 1000, 5000 e 10000, usando sempre os valores *default* para as restantes variáveis. Os resultados obtidos são apresentados em seguida, e interpretando-os podemos tirar conclusões interessantes relativamente ao custo final esperado usando os diferentes algoritmos, bem como o seu tempo de execução.

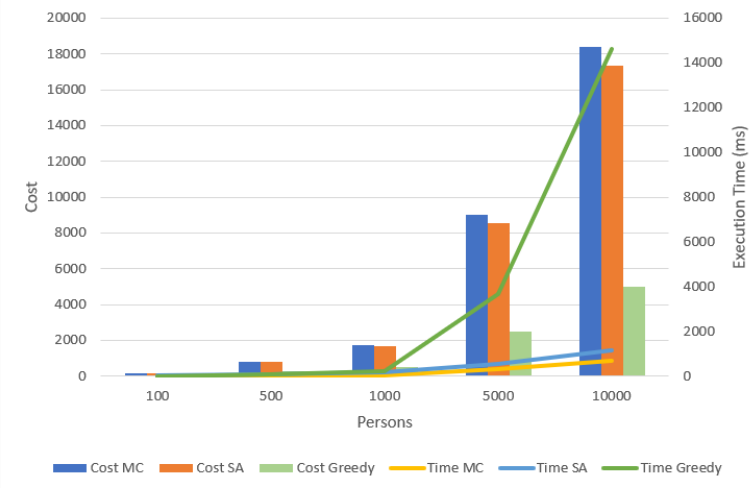

Figure 2: Resultados obtidos variando o número de pessoas

Persons	Implementation	Cost	Steps	Execution Time (ms)
100	MC	153	371	8,734375
	SA	145	2411	53,486328
	Greedy	54	-	9,560547
500	MC	831	1173	24,106445
	SA	804	1365	103,383301
	Greedy	252	-	68,891602
1000	MC	1716	1342	48,10791
	SA	1662	3001	189,362305
	Greedy	505	-	209,182129
5000	MC	9030	7888	300,70459
	SA	8573	18901	558,850586
	Greedy	2506	-	3691,000488
10000	MC	18372	14650	703,001465
	SA	17326	30981	1154,130371
	Greedy	5003	-	14619,490234

Table 2: Resultados obtidos variando o número de pessoas

Como seria de esperar, ao aumentar o número de pessoas estaremos consequentemente a aumentar não só o tempo necessário para encontrar uma solução, como também o custo das soluções possíveis.

No que toca aos custos finais, é relevante realçar que todas as implementações aparentam ter um crescimento constante, no entanto, os custos obtidos pelas que recorrem aos Métodos de *Monte Carlo* e a *Simulated Annealing* aparentam andar na ordem de $1.5N$ e $2N$, onde N é o número de pessoas, sendo que também se espera que a implementação com *SA* seja ligeiramente melhor; enquanto que a implementação com um algoritmo *Greedy* produz custos na ordem dos $0.5N$, muito próximo do que será o custo mínimo.

Por outro lado, o tempo de execução esperado com *Monte Carlo*, para qualquer N, é inferior ao esperado para as outras implementações, sendo que o aumento do número de pessoas leva a um aumento aproximadamente proporcional do tempo de execução. No caso do algoritmo com SA, o aumento do número de pessoas não leva a um aumento do tempo de execução na mesma proporção, no entanto, este torna-se naturalmente maior devido ao maior número de trocas realizado. Abordando os dados obtidos para a implementação *Greedy*, torna-se evidente que o crescimento do tempo de execução, com o aumento do número de pessoas, é exponencial.

3.3 Variação do número de iterações sem trocas

Desta vez, fizemos variar o número máximo de iterações seguidas sem ocorrência de trocas, por forma a verificar de que modo este influenciava a busca pela solução. Assim, executamos os algoritmos em que esta variável é necessária (as implementações com *Monte Carlo* e *Simulated Annealing*), com o valor máximo de *iter* a corresponder a 50, 75, 100, 125 e 150. Para além disto, decidimos que seria uma melhor métrica, de agora em diante, comparar o custo final obtido com o número de *steps* (em termos práticos, as iterações do ciclo) que são efetuados, já que estes são medidos tanto na implementação com MC como na de SA.

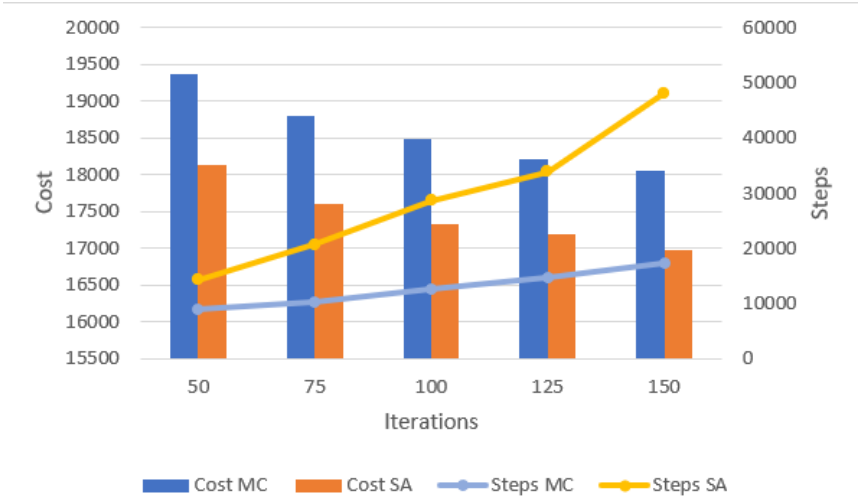

Figure 3: Resultados obtidos variando o número de iterações sem trocas

Iterations	Implementation	Cost	Steps	Execution Time
50	MC	19360	8958	554,347656
	SA	18141	14279	796,614746
75	MC	18795	10353	630,691406
	SA	17614	20699	1035,196289
100	MC	18481	12646	618,288086
	SA	17328	28611	1356,936523
125	MC	18222	14608	764,19043
	SA	17200	33865	1393,049316
150	MC	18061	17398	867,146973
	SA	16978	48126	2365,898438

Table 3: Resultados obtidos variando o número de iterações sem trocas

Torna-se fácil confirmar o esperado: quanto maior o limite máximo de iterações sem trocas, maior será o número de *steps* que serão realizados, o que leva a um menor custo final. Isto acontece uma vez que aumentar o número máximo de iterações sem trocas necessárias aumenta também a possibilidade de encontrar uma troca possível, que irá levar a um menor custo e a uma necessidade de haver mais *steps*.

Para além disto, é de realçar que a diminuição do custo final obtido bem como o aumento dos *steps* dados são mais acentuados na implementação com *SA*, algo que se justifica com o facto de que nesta implementação há mais trocas aceites.

3.4 Variação do valor da temperatura inicial

Posteriormente, efetuaram-se testes para diferentes valores da temperatura inicial T, no algoritmo com *Simulated Annealing*. Para isto, consideramos que T assumiria o valor de 1, 2, 3, 4 e 5, e as restantes variáveis os valores *default*. Os resultados obtidos são apresentados no gráfico que se segue.

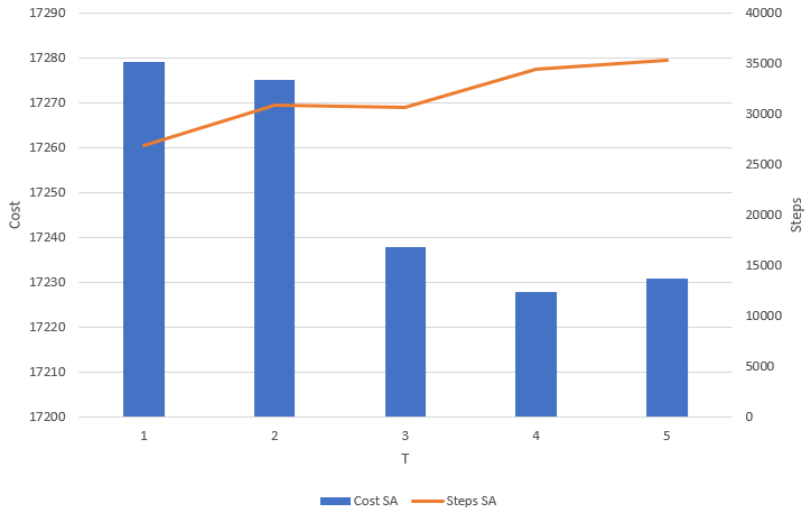

Figure 4: Resultados obtidos variando o valor da temperatura inicial

T	Cost	Steps	Execution Time (ms)
1	17279	26898	1165,461914
2	17275	30934	977,851074
3	17238	30660	1016,495605
4	17228	34417	1069,278809
5	17231	35311	1043,957031

Table 4: Resultados obtidos variando o valor da temperatura inicial

Através da análise dos dados apresentados, podemos perceber que um aumento da temperatura inicial leva a uma diminuição do custo final, bem como a um aumento do número de *steps*. Estes resultados derivam do facto de que o aumento da temperatura inicial aumenta também a probabilidade de que uma troca (que aumenta o custo da distribuição) seja aceite. Assim, serão realizados mais *steps*, e consequentemente, aumenta-se a probabilidade de encontrar trocas que levem a uma diminuição do custo da distribuição final.

3.5 Variação do *Decreasing Value*

Por fim, analisamos o desempenho do algoritmo com *SA* para diferentes valores da função que diminui T a cada iteração. A abordagem que adotamos, neste caso, foi de aumentar a precisão do multiplicador da função, aproximando-o cada vez mais a 1. Assim, testamos com *decreasing values* de 0.9, 0.99, 0.999, 0.9999 e 0.99999. Os resultados foram coletados e construiu-se o seguinte gráfico:

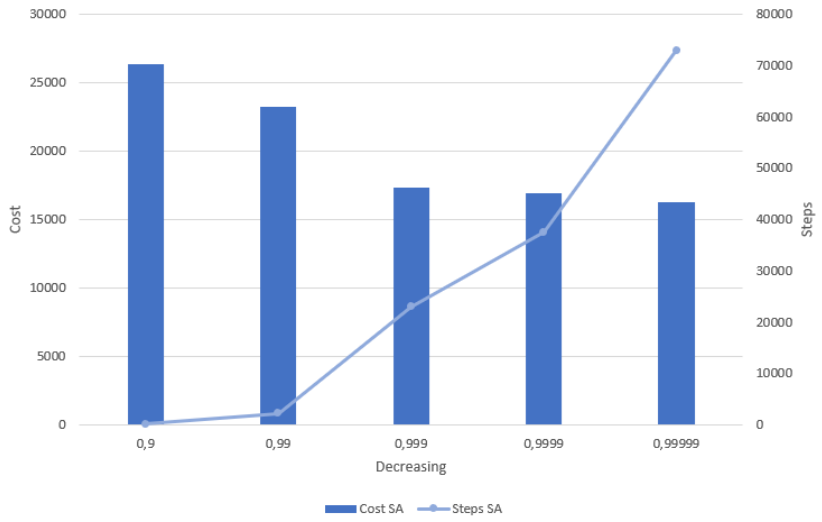

Figure 5: Resultados obtidos variando o *Decreasing Value*

Decreasing Value	Cost	Steps	Execution Time (ms)
0,9	26377	219	515,935425
0,99	23258	2292	501,746307
0,999	17350	23015	1034,116211
0,9999	16971	37589	1494,389313
0,99999	16306	73012	1882,936707

Table 5: Resultados obtidos variando o *Decreasing Value*

Interpretando os resultados obtidos, percebe-se que o aumento da precisão do *decreasing value* influencia bastante a procura pela distribuição ótima, já que leva uma diminuição do custo final obtido. Isto deve-se ao facto de que com o aumento do *decreasing value*, a diminuição da temperatura T torna-se cada vez mais gradual, o que leva a um aumento da probabilidade de uma troca ser aceite numa dada iteração (considerando uma iteração i e uma troca com um custo *delta*, esta é mais provável de acontecer em i quando o *decreasing value* é de 0.99 do que quando é 0.9, uma vez que se espera que o valor da temperatura T em i seja superior no primeiro caso do que no segundo). Deste modo, há um aumento do número de trocas e consequentemente obtém-se um custo final inferior.

4 Conclusão

Sentimos que fomos capazes de atingir os objetivos pretendidos com a elaboração deste trabalho, aprofundando os nossos conhecimentos na construção de programas paralelos e abordando mais concretamente a questão da realização de diversos testes em simultâneo, o que permitiu recorrer à ferramenta *OpenMP* de uma forma diferente do habitual.

Para além disto, fomos também confrontados com a necessidade de analisar os diferentes parâmetros que compõe um algoritmo, o que levou a um aumento da nossa capacidade de compreender os diferentes impactos que estes podem ter nos resultados finais.

Anexos

4.1 Ficheiro roomsMAIN.c

```
int main(int argc, char* argv[]){

    if(argc < 4){
        printf("<program> #PERSONS #PROCESSES #MODE\n");
        printf("Values of #MODE:\n");
        printf("    a        - All different implementations, with the standard values\n");
        printf("    i #ITER  - Monte Carlo and SA implementation, using ITER iterations\n");
        printf("    t #TEMP  - SA implementation, with TEMP temperature\n");
        printf("    f #VALUE - SA implementation, with the decreasing function being T =
    VALUE*T\n");
        return 1;
    }

    int N = atoi(argv[1]);

    if(N%2 != 0){
        printf("The number of persons must not be an odd number.\n");
        return 1;
    }

    int processes = atoi(argv[2]);
    int iter = 100;
    double temp = 1, value = 0.999, exectime;
    int* ret = malloc(sizeof(int) * 2);
    int** D;

    srand(time(NULL));

    initD(N, &D);

    switch(argv[3][0]){
        case 'a':
            for(int i = 0; i < 10; i++){
                clearCache();
                startTimer();
                rooms(N, &D, iter, &ret, processes);
                exectime = stopTimer();
                printf("MC,%d,%d,%lf\n", ret[0], ret[1], exectime);
            }

            for(int i = 0; i < 10; i++){
                clearCache();
                startTimer();
                roomsSA(N, &D, iter, temp, value, &ret, processes);
                exectime = stopTimer();
                printf("SA,%d,%d,%lf\n", ret[0], ret[1], exectime);
            }

            for(int i = 0; i < 10; i++){
                clearCache();
                startTimer();
                roomsGreedy(N, &D, &ret, processes);
                exectime = stopTimer();
                printf("Greedy,%d,,%lf\n", ret[0], exectime);
            }
```

```
55
            break;
57
        case 'i':
59          if (argc == 5){
                iter = atoi(argv[4]);
61
                for(int i = 0; i < 10; i++){
63                  clearCache();
                    startTimer();
65                  rooms(N, &D, iter, &ret, processes);
                    exectime = stopTimer();
67                  printf("MC,%d,%d,%lf\n", ret[0], ret[1], exectime);
                }
69
                for(int i = 0; i < 10; i++){
71                  clearCache();
                    startTimer();
73                  roomsSA(N, &D, iter, temp, value, &ret, processes);
                    exectime = stopTimer();
75                  printf("SA,%d,%d,%lf\n", ret[0], ret[1], exectime);
                }
77          }
            break;
79
        case 't':
81          if (argc == 5){
                temp = strtod(argv[4], NULL);
83
                for(int i = 0; i < 10; i++){
85                  clearCache();
                    startTimer();
87                  roomsSA(N, &D, iter, temp, value, &ret, processes);
                    exectime = stopTimer();
89                  printf("SA,%d,%d,%lf\n", ret[0], ret[1], exectime);
                }
91          }
            break;
93
        case 'f':
95          if (argc == 5){
                value = strtod(argv[4], NULL);
97
                for(int i = 0; i < 10; i++){
99                  clearCache();
                    startTimer();
101                 roomsSA(N, &D, iter, temp, value, &ret, processes);
                    exectime = stopTimer();
103                 printf("SA,%d,%d,%lf\n", ret[0], ret[1], exectime);
                }
105         }
            break;
107
        case 'p':
109         for(int p = 1; p <= processes; p++)
            {
111             printf("\n%d Threads\n", p);
                for(int i = 0; i < 10; i++)
113             {
                    clearCache();
115                 startTimer();
```

```
                    rooms(N, &D, iter, &ret, p);
                    exectime = stopTimer();
                    printf("MC,%d,%d,%lf\n", ret[0], ret[1], exectime);
                }
                for(int i = 0; i < 10; i++)
                {
                    clearCache();
                    startTimer();
                    roomsSA(N, &D, iter, temp, value, &ret, p);
                    exectime = stopTimer();
                    printf("SA,%d,%d,%lf\n", ret[0], ret[1], exectime);
                }
                for(int i = 0; i < 10; i++)
                {
                    clearCache();
                    startTimer();
                    roomsGreedy(N, &D, &ret, p);
                    exectime = stopTimer();
                    printf("Greedy,%d,,%lf\n", ret[0], exectime);
                }
            }

            break;

        default:
            printf("Not a valid parameter.\n");
            break;
    }

    return 0;
}
```

4.2 Ficheiro rooms.c

```
#include "../include/rooms.h"

// Implementation in which a Monte Carlo method is used.
void rooms(int N, int*** D, int iter, int** ret, int processes){
    int N2 = N/2;
    int** retAux = malloc(sizeof(int*) * processes);
    for(int i = 0; i < processes; i++)
        retAux[i] = malloc(sizeof(int) * 2);

    #pragma omp parallel num_threads(processes)
    {
        int** room;
        int id, i, c, d, delta, aux;
        unsigned int myseed = rand();
        id = omp_get_thread_num();
        retAux[id][0] = 0;
        retAux[id][1] = 0;

        //First we initialize the rooms with random occupants
        randPerm(N, &room, myseed);

        //Next, we calculate the cost of this distribution
        for(i = 0; i < N2; i++)
            retAux[id][0] += (*D)[room[i][0]][room[i][1]];

        i = 0;

        /*After this, we'll try to see if it is possible to obtain a better distribution,
          by switching the occupants of different random places*/
        while(i < iter){
            (retAux[id][1])++;

            //Selecting the random occupant
            c = (rand_r(&myseed))%N2;

            //The occupant selected will switch with one in the next room
            if(c == N2-1)
                d = 0;
            else
                d = c+1;

            //Determining the difference between the old cost and the new one
            delta = ((*D)[room[c][0]][room[d][1]]+(*D)[room[d][0]][room[c][1]]) - ((*D)[room
    [c][0]][room[c][1]]+(*D)[room[d][0]][room[d][1]]);

            //If the difference is negative, then the switch will improve the cost, so it is
     accepted
            if(delta < 0){
                aux = room[c][1];
                room[c][1] = room[d][1];
                room[d][1] = aux;

                retAux[id][0] += delta;

                i = 0;
            }
            else
                i++;
        }
```

```
58      }
60      (*ret)[0] = retAux[0][0];
        (*ret)[1] = retAux[0][1];
62
        for(int i = 1; i < processes; i++){
64          if(retAux[i][0] < (*ret)[0]){
                (*ret)[0] = retAux[i][0];
66              (*ret)[1] = retAux[i][1];
            }
68      }
    }
```

4.3 Ficheiro roomsSA.c

```
#include "../include/rooms.h"

// Implementation in which Simulated Annealing is used.
void roomsSA(int N, int*** D, int iter, double temp, double value, int** ret, int processes)
    {
    int N2 = N/2;
    int** retAux = malloc(sizeof(int*) * processes);
    for(int i = 0; i < processes; i++)
        retAux[i] = malloc(sizeof(int) * 2);

    #pragma omp parallel num_threads(processes)
    {
        int** room;
        int id, i, c, d, delta, aux;
        double random, T = temp;
        unsigned int myseed = rand();
        id = omp_get_thread_num();
        retAux[id][0] = 0;
        retAux[id][1] = 0;

        //First we initialize the rooms with random occupants
        randPerm(N, &room, myseed);

        //Next, we calculate the cost of this distribution
        for(i = 0; i < N2; i++)
            retAux[id][0] += (*D)[room[i][0]][room[i][1]];

        i = 0;

        /*After this, we'll try to see if it is possible to obtain a better distribution,
          by switching the occupants of different random places
          (we had to add a restriction to the value of T because, for a small N,
           in some cases, T would get to 0, which would make the cycle infinite)*/
        while(i < iter && T > 0){
            (retAux[id][1])++;

            //Selecting the random occupant
            c = (rand_r(&myseed))%N2;

            //The occupant selected will switch with one in the next room
            if(c == N2-1)
                d = 0;
            else
                d = c+1;

            //Determining the difference between the old cost and the new one
            delta = ((*D)[room[c][0]][room[d][1]]+(*D)[room[d][0]][room[c][1]]) - ((*D)[room
    [c][0]][room[c][1]]+(*D)[room[d][0]][room[d][1]]);

            random = (double)rand_r(&myseed)/(double)RAND_MAX;
            aux = 0 - delta;

            //Accept or discard new arrangement
            if((delta < 0) || (exp(((double) aux)/T) >= random)){
                aux = room[c][1];
                room[c][1] = room[d][1];
                room[d][1] = aux;

                retAux[id][0] += delta;
```

```
59                  i = 0;
                }
61              else
                    i++;
63
                T = value*T;
65          }
        }
67
        (*ret)[0] = retAux[0][0];
69      (*ret)[1] = retAux[0][1];

71      for(int i = 1; i < processes; i++){
            if(retAux[i][0] < (*ret)[0]){
73              (*ret)[0] = retAux[i][0];
                (*ret)[1] = retAux[i][1];
75          }
        }
77  }
```

4.4 Ficheiro roomsGreedy.c

```
#include "../include/rooms.h"

// Implementation in which a Greedy method is used.
void roomsGreedy(int N, int*** D, int** ret, int processes){
    int* retAux = malloc(sizeof(int) * processes);

    #pragma omp parallel num_threads(processes)
    {
        int** room = malloc(sizeof(int*) * (N/2));
        int* allocated = malloc(sizeof(int) * N);
        int id, i, j, toAllocate, min, aux;
        id = omp_get_thread_num();
        retAux[id] = 0;

        //Initialization of the rooms, making all of them empty
        for(i = 0; i < N/2; i++){
            room[i] = malloc(sizeof(int) * 2);

            room[i][0] = -1;
            room[i][1] = -1;
        }

        //No one is allocated to a room initially
        for(i = 0; i < N; i++)
            allocated[i] = 0;

        min = 11;
        i = 0;

        //Deciding which will be the first person to allocate
        toAllocate = rand()%N;

        //This cycle will run until all of the rooms are fill
        while(i < (N/2)){
            aux = -1;

            //Determining which is the best available option to put in a room with the
    chosen person (defined in "toAllocate")
            for(j = 0; j < N; j++){
                if(toAllocate != j && !(allocated[j]) && (*D)[toAllocate][j] < min){
                    min = (*D)[toAllocate][j];
                    aux = j;
                }
            }

            if(aux != -1){
                //Putting both persons into the next room available
                room[i][0] = toAllocate;
                room[i][1] = aux;

                //Updating the cost
                retAux[id] += min;

                //Marking both of them as allocated
                allocated[toAllocate] = 1;
                allocated[aux] = 1;

                j = 0;
                //Determining which will be the next person to get allocated
```

```
59                  while(j < N){
                        toAllocate = (toAllocate+1)%N;
61
                        if(!(allocated[toAllocate]))
63                          j = N;
                        else
65                          j++;
                    }
67
                    min = 11;
69                  i++;
                }
71
                else
73                  break;
            }
75      }

77      (*ret)[0] = retAux[0];
        (*ret)[1] = 0;
79
        for(int i = 1; i < processes; i++){
81          if(retAux[i] < (*ret)[0]){
                (*ret)[0] = retAux[i];
83          }
        }
85  }
```

4.5 Ficheiro auxiliar.c

```c
#include "../include/auxiliar.h"

double clearcache[30000000];
double initime;

void initD(int N, int*** D){
    int i, j, aux;
    (*D) = malloc(sizeof(int*) * N);

    for(i = 0; i < N; i++)
        (*D)[i] = malloc(sizeof(int) * N);

    for(i = 0; i < N; i++){
        (*D)[i][i] = 0;

        for(j = i+1; j < N; j++){
            aux = ((rand())%10) + 1;
            (*D)[i][j] = aux;
            (*D)[j][i] = aux;
        }
    }
}

void randPerm(int N, int*** room, unsigned int myseed){
    int i, N2, a , b , c, d, aux;
    N2 = N/2;
    (*room) = malloc(sizeof(int*) * N2);

    for(i = 0; i < N2; i++){
        (*room)[i] = malloc(sizeof(int) * 2);

        (*room)[i][0] = i*2;
        (*room)[i][1] = (i*2)+1;
    }

    for(i = 0; i < N2; i++){
        a = (rand_r(&myseed))%N2;
        b = (rand_r(&myseed))%N2;
        c = (rand_r(&myseed))%2;
        d = (rand_r(&myseed))%2;

        aux = (*room)[i][0];
        (*room)[i][0] = (*room)[a][c];
        (*room)[a][c] = aux;

        aux = (*room)[i][1];
        (*room)[i][1] = (*room)[b][d];
        (*room)[b][d] = aux;
    }
}

void clearCache(){
    for(int i = 0; i < 30000000; i++)
        clearcache[i] = i;
}

void startTimer(){
    double aux = omp_get_wtime();
    initime = aux * 100000;
```

```
}

double stopTimer(){
    double aux = omp_get_wtime();
    double aux2 = aux * 100000;
    return (aux2 - initime);
}
```