UNIVERSITY OF MINHO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

PARALLEL ALGORITHMS

Monte-Carlo Travelling Salesman Problem

Adriana Meireles (A82582) Shahzod Yusupov (A82617)

23 de Abril de 2020

Conteúdo

1 Introduction

The travelling salesman problem consists in the following problem: given n points by its (x, y) coordinates, D(i, j) holds the distance between each two points (x(i),y(i)) and (x(j),y(j)). The aim is to find the shortest route that starts at one of the points ("towns"), randomly chosen, goes once through every other town and finally returns to the starting point.

It is an **NP-hard** problem in combinatorial optimization, important in operations research and theoretical computer science, however due to its complexity it is in most cases impossible to calculate the overall optimal solution of the problem in a reasonable amount of time.

Given a number **R (N)** of routes to **N** cities, considering the first fixed city ,we have the possibility of travel to any of the others on the first trip, repeating the process until all cities have been visited only once, resulting in the formula N * (N - 1) * (N - 2) * ... * 2 * 1 and can be expressed as a factorial notation **N!**.

2 Problem Description

In order to find the best solution for this problem there were tested some different algorithms and then compared with each other.

2.1 Greedy Algorithm

This kind of algorithms are very often used in optimization problems and consist in a paradigm of making the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. In this case this means that for each journey (but the last one, when the salesman returns to the starting point), the closest town not yet visited, is always chosen.

However, despite its simplicity sometimes this kind of algorithms fails because it makes decisions based only on the information it has at any one step, without regard to the overall problem.

2.2 Monte-Carlo Method

Monte Carlo methods are a class of methods that depend on the generation of random numbers to obtain results to problems that could be deterministic.

In the particular case of the traveling salesman problem, the complexity of the problem makes it impossible to put the solution in time and the application of this method is one of the effective ways for obtaining approximate solutions to the optimal solution. To do so, it is necessary that the values generated randomly have a uniform distribution, otherwise we would be focusing our search for the best solution in specific area, twisting the results . The base algorithm is explained below:

1. It starts with an initial energy - random route (a random permutation of the first n integer numbers).
2. The new route is chosen and it is calculated the difference between the current energy(route) and the energy before. This difference will be designated by delta.
3. If $delta < 0$, means that the new total distance is lower than the earlier so it is accepted and the iterations are restarted.
4. If $delta > 0$,just iterations are increased.
5. Stopping criteria if number of iterations is reached.

Iterations represents the maximum number of iterations without changes in the path(the new total distance is always bigger than the old one).

2.3 Simulated Annealing

Simulated annealing is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem. “Annealing” refers to an analogy with thermodynamics, specifically with the way that metals cool and anneal. Simulated annealing uses the objective function of an optimization problem instead of the energy of a material.[0]

Implementation of SA is very simple. The algorithm is basically hill-climbing except instead of picking the best move, it picks a random move. If the selected move improves the solution, then it is always accepted. Otherwise, the algorithm makes the move anyway with some probability less than 1. The probability decreases exponentially with the “badness” of the move, which is the amount deltaE by which the solution is worsened.[0][0]. This alghoritm is very similar to the previous one, except the fact than when $delta > 0$ the new configuration is accepted with probability of:

$$Prob(acceptinguphillmove) = exp(-deltaE/T)) \geq r. \tag{1}$$

A parameter T is also used to determine this probability. It is analogous to temperature in an annealing system. At higher values of T, uphill moves are more likely to occur. As T tends to zero, they become more and more unlikely, until the algorithm behaves more or less like hill-climbing. The variable r is some some generated random number between 0 and 1. [0]

3 Measurements

In order to test the previous algorithms, we used 662 node from the SeARCH cluster at University of Minho.We took several important parameters for performance such as execution time and distance between algorithms.

We can observe bellow the execution time(in microseconds)for the three algorithms increasing the number of cities.As we can see this growth is not factorial in terms of the number of cities as the calculation of all combinations for an optimal global solution required.The greedy algorithm has the worst performance because of the decisions based on the information it has in each step, searching always for the local minimum, in this case the nearest city.The Simulated Annealing Algorithm has worse performance than Monte Carlo Algorithm due to the increase in the number of operations and iterations proven by the acceptance of previous results that restart the iterations. For this algorithm we used 200 iterations and a variable number of cities. The simulated annealing algorithm uses 1.5 for initial temperature.

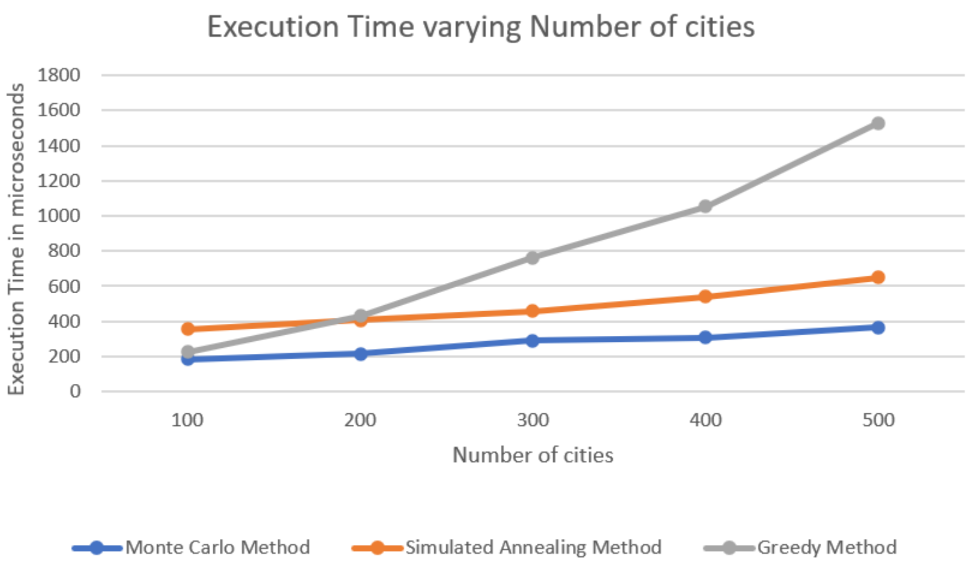

Figura 1: Execution Time varying Number of cities

The graph bellow shows the calculated distance for each algorithm increasing the number of cities. The Greedy algorithm, despite having a longer execution time has the best performance when it comes to overall distance due to his simplicity and because of the choice at each step that attempts to find the overall optimal way to solve the entire problem. The Simulated Annealing Algorithm has a better performance than Monte Carlo Algorithm. For this algorithm we used 200 iterations and a variable number of cities. The simulated annealing algorithm uses 1.5 for initial temperature.

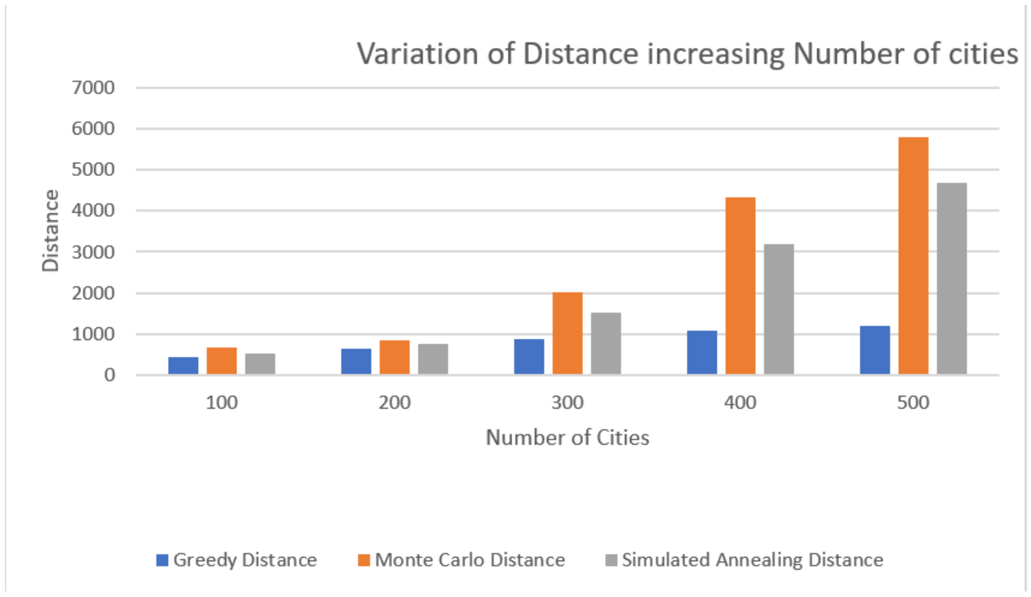

Figura 2: Variation of Distance increasing number of cities

We now present the graph that shows the calculated distance for each algorithm increasing the number of iterations. The values obtained for the distances do not introduce significant improvements with the increase of the number of iterations. The results indicate that a balanced number of iterations is enough to get stable results.We can see an improvement of the values obtained with Simulated Annealing method compared to the Monte Carlo method. For this algorithm we used 100 cities and variable number of iterations.

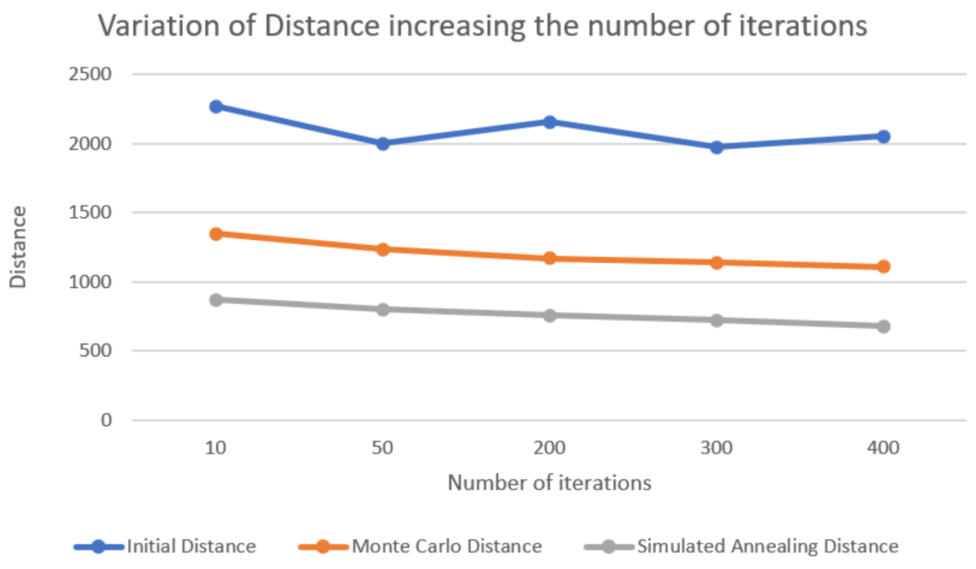

Figura 3: Variation of Distance increasing number of iterations

Lastly, we present the graph that shows the calculated distance for Simulated Annealing algorithm increasing the initial temperature(defines the probability of the worst order values to be accepted).We can see that the higher the initial temperature,the shorter the distance will be which leads to better performance. However, if this value is too high,the paths may deteriorate in their distance. For this algorithm we used 200 iterations, 200 cities and variable initial temperatures.

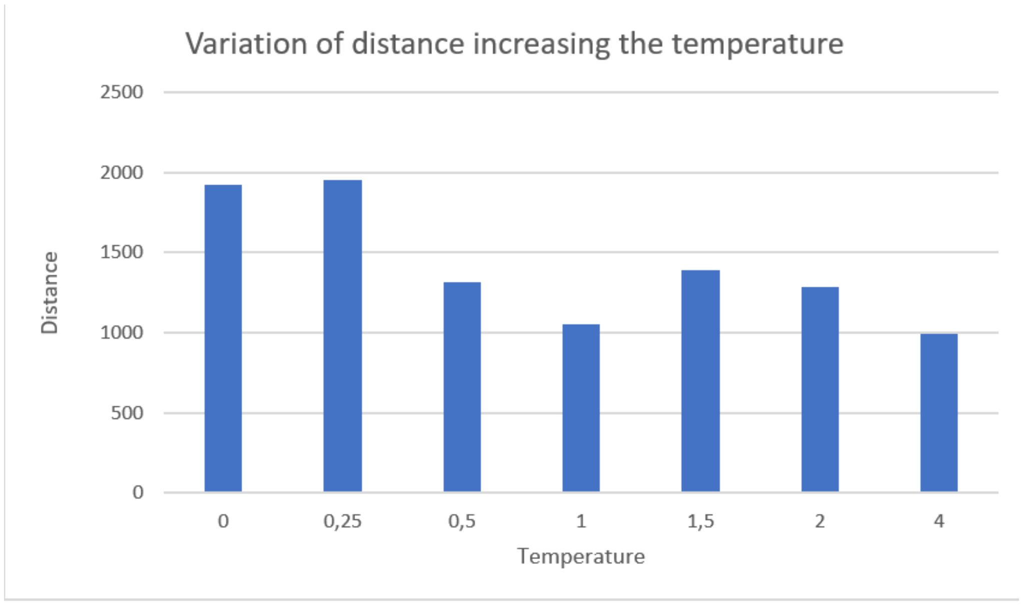

Figura 4: Variation of Distance increasing initial temperature

4 Conclusion

The accomplishment of this work allowed us to get more knowledge about *Matlab* as well as to analyze several algorithms of the traveling salesman problem that is already known from another subjects.

We conclude that the best algorithm to solve this problem is the *Greedy Algorithm*, because as we could see at previous section despite of having longer execution times it always finds the shorter overall distance which is the main goal of this study. About the *Simulated Annealing Method* we can say that when it comes to execution time it is slower than the *Monte Carlo Method* but in terms of finding the shorter distance it has better performance.

In short, we consider that we acquire the basic concepts which may help us in further projects.

5 Bibliography

2020. [online] Available at: ¡https://en.wikipedia.org/wiki/Simulatedannealing¿[Accessed 10 April 2020].

leemon@cs.cmu.edu, L., 2020. What Is Simulated Annealing?. [online] Cs.cmu.edu. Available at:¡http://www.cs.cmu.edu/afs/cs.cmu.edu/project/learn-43/lib/photoz/.g/web/glossary/anneal.html¿[Accessed 10 April 2020].

Myweb.ntut.edu.tw. 2020. [online] Available at:¡https://myweb.ntut.edu.tw/ gyen/handout/4-%20Simulated%20Annealing.pdf?fbclid=IwAR2rPEyYgqy0paNPC7QX6eT_ufbDw9Qtlg3RNvQB3FLE852TQjeFM
[Accessed 10 April 2020].

6 Attachments

The code used is showed bellow. We used 3 files to develop our work in order to facilitate the job between the elements of the group.

Greedy Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <omp.h>
#include <unistd.h>
#include <limits.h>

#define TIME_RES 1000000
#define FLT_MAX 3.402823466e+38F /* max value */

double initial_time;
double clear_cache [30000000];
float** matrix_distances;
int * destino;
int cities;
int nodo;// calculado com a funcao rand

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

void start(){
  double time= omp_get_wtime();
  initial_time= time* TIME_RES;
}

double stop(){
   double time = omp_get_wtime();
   double final = time * TIME_RES;
   return final - initial_time;
}

// funcao para alocar e inicializar a matriz de distancias e o array com as cidades
void initialize_matrix(){
  int i,j;
    matrix_distances = (float**) malloc(sizeof(float*) * cities );
  destino = (int*) malloc(sizeof(int) * cities);

  for (i =0;i<cities;i++){
    destino[i] = -1;
      matrix_distances[i] =malloc(sizeof(float) * cities);
  }
  for(i=0;i<cities;i++){
    for(j=0;j<cities;j++){
    matrix_distances[i][j] = 0;
    }
  }
}

// funcao que consoante as posicoes(x,y)  calcula as distancias entre as varias
    cidades
void find_distances(int* x, int* y, int n){
  int i,j;

  for(i=0;i<n;i++){
    for(j=0;j<n;j++){
        if(i!=j){
```

```
      matrix_distances[i][j] = sqrt(pow(x[i]-x[j],2) + pow(y[i]-y[j],2));
        }
    }
  }
}

// funcao principal que calcula,sucessivamente  o caminho mais curto de uma cidade
    inicial ate a seguinte
// anda nao visitada
void greedy(int num_proc){

  int i,j,pos;
  float  min= FLT_MAX;
  float count=0;
  int iter= 0;

  i = nodo; // cidade inicial calculado pelo rand

      #pragma omp parallel num_threads(num_proc)
      {
  while(iter<(cities-1)){

    for(j=0;j<cities;j++){
      if(matrix_distances[nodo][j]<min && (matrix_distances[nodo][j]!=0) && (
    destino[j]==-1)){
        min = matrix_distances[nodo][j];
        pos = j;
      }
    }
    destino[nodo] = pos;
    nodo = pos;
    iter++;
    count+=min ;
    min = FLT_MAX;
  }
  for(j=0;j<cities;j++){
    if(destino[j]<0){
      destino[j] = i;
      #pragma omp atomic
      count+= matrix_distances[j][i];
    }
  }
   }
  printf("A distancia total e: %f\n", count);
}

int main(int argc, char** argv){

      if ( argc != 3 ){
           printf ("Usage : ./ tsp <nr cidades> <nr de processos >\n");
     return 0;
  }
  srand((unsigned) time(NULL));
  cities=atoi(argv[1]);
  int num_proc = atoi(argv[2]);
  int* posX;
  int* posY;
  int j,i;
  nodo = rand()% cities;

  posX = (int*) malloc(sizeof(int) * cities);
  posY = (int*) malloc(sizeof(int)  *cities);

  for(i = 0; i<cities;i++){
    posX[i] = rand() % 50;
    posY[i] = rand() % 50;
    }

  initialize_matrix();
  start();
  find_distances(posX,posY,cities);
  greedy(num_proc);
  double tempo = stop();
  printf("Demorou %f  segundos\n  ",tempo);
}
```

Monte Carlo Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <omp.h>
#include <unistd.h>
#include <math.h>

double initial_time;
double clearcache [30000000];
float** matrix_distances;
int * path;
int cities;

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

void start (void) {
    double time = omp_get_wtime();
    initial_time = time * 1000000;
}

double stop() {
    double time = omp_get_wtime();
    double final_time = time * 1000000;

    return final_time - initial_time;
}

//calcula as permutações aleatórias
void randperm(int* perm,int N){
  int i,j,tmp;
    for(i=0;i<N;i++)
      perm[i] = i;

    for(i=0;i<N;i++){
      j=rand()% ( N-i) + i;
      tmp = perm[j];
      perm[j] = perm [i];
      perm[i] = tmp;
    }
}

//calcula custos do caminho calculado
float calculate_cost(int* path,int N){
  int i,cost;
  cost = matrix_distances[path[N-1]][path[0]];

  for(i=0;i<N;i++)
    cost += matrix_distances[path[i]][path[i+1]];
  return cost;
}

```

```
//calcula distancia entre cidades
void find_distances(int* x, int* y,int N){
  int i,j;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      matrix_distances[i][j] = sqrt(pow(x[i]-x[j],2) + pow(y[i]-y[j],2));
}

void initialize_matrix(){
  int i,j;

    matrix_distances = (float**) malloc(sizeof(float*) * cities );
  path = (int*) malloc(sizeof(int) * cities);


  for (i =0;i<cities;i++){

    path[i] = -1;
    matrix_distances[i] =malloc(sizeof(float) * cities);
  }
  for(i=0;i<cities;i++){
    for(j=0;j<cities;j++){
    matrix_distances[i][j] = 0;
    }
  }
}

float tsp(float** dist, int* path, int N, int iter, int num_proc){

  int i,c,previous,next1,next2,priv_path[N],tmp;

  float cost,delta,newCost;
  delta = 0.0;

  cost = calculate_cost(path,N);

  #pragma omp parallel num_threads(num_proc)
  {
  newCost = cost;

  for (i = 0; i < N; i++)
  {
    priv_path[i] = path[i];
  }
  i=0;
  while(i<iter){
    c= (rand() % N);
    if(c==0){
      previous = N-1;
      next1=1;
      next2=2;
    }
    else{
      previous = c-1;
      next1=(c+1) % N;
      next2 = (c+2) % N;
    }
    delta = dist[priv_path[previous]][next1] + dist[priv_path[c]][priv_path[next2]]
     - dist[priv_path[previous]][priv_path[c]] - dist[priv_path[next1]][priv_path[
    next2]];

    if(delta < 0){
      tmp = priv_path[c];
      priv_path[c] = priv_path[next1];
      priv_path[next1] = tmp;
      newCost += delta;
      i=0;
    }
    else i++;
  }
  #pragma omp critical
    {
  if(newCost < cost){
    cost = newCost;

    for(i=0;i<iter;i++)
      path [i] = priv_path [i];
  }
  }
}
  return cost ;
}

int main(int argc, char const *argv[])
{
  int i;
  float cost, tspCost;

  srand((unsigned) time(NULL));

  if ( argc != 4 ){
           printf ("Usage : ./ tsp <nr cidades> <nr de processos ><nr de iteracoes >\n");
     return 0;
  }
  cities = atoi(argv[1]);
  int num_proc = atoi(argv[2]);
  int iter = atoi(argv[3]);

    int* x_pos = malloc (cities * sizeof (int)) ;
    int* y_pos = malloc (cities * sizeof (int)) ;

    //gera coordenadas para cada cidade num quadrado de lado 50
     for ( i = 0; i < cities; i++ ) {
    x_pos [i] = rand () % 50;
    y_pos [i] = rand () % 50;
  }
    initialize_matrix();

    find_distances(x_pos,y_pos,cities);

    randperm(path,cities);

    cost = calculate_cost(path,cities);
    printf("Custo Inicial :%f\n",cost);

    clearCache();

    start();

    tspCost = tsp(matrix_distances,path,cities,iter,num_proc);
    printf("Tempo Monte Carlo:%f\n",stop());

    printf("Custo TSP : %f\n", tspCost);

    return 0;
}
```

Simulated Annealing Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
```

```
#include <omp.h>
#include <unistd.h>
#include <math.h>

double initial_time;
double clearcache [30000000];
float** matrix_distances;
int * path;
int cities;

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

void start (void) {
    double time = omp_get_wtime();
    initial_time = time * 1000000;
}

double stop() {
    double time = omp_get_wtime();
    double final_time = time * 1000000;

    return final_time - initial_time;
}
//calcula as permutações aleatórias
void randperm(int* perm,int N){
  int i,j,tmp;
    for(i=0;i<N;i++)
      perm[i] = i;

    for(i=0;i<N;i++){
      j=rand()% ( N-i) + i;
      tmp = perm[j];
      perm[j] = perm [i];
      perm[i] = tmp;
    }
}

//calcula custos do caminho calculado
float calculate_cost(int* path,int N){
  int i,cost;
  cost = matrix_distances[path[N-1]][path[0]];

  for(i=0;i<N;i++)
    cost += matrix_distances[path[i]][path[i+1]];
  return cost;
}

//calcula distancia entre cidades
void find_distances(int* x, int* y,int N){
  int i,j;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      matrix_distances[i][j] = sqrt(pow(x[i]-x[j],2) + pow(y[i]-y[j],2));
}

void initialize_matrix(){
  int i,j;
    matrix_distances = (float**) malloc(sizeof(float*) * cities );
  path = (int*) malloc(sizeof(int) * cities);

  for (i =0;i<cities;i++){

    path[i] = -1;
    matrix_distances[i] =malloc(sizeof(float) * cities);
  }
  for(i=0;i<cities;i++){
    for(j=0;j<cities;j++){
    matrix_distances[i][j] = 0;
    }
  }
}

float tspAnnealing(float** dist, int* path, int N, int iter, int num_proc,float
    temperatura){

  int i,c,previous,next1,next2,priv_path[N],tmp;

  float cost,delta,newCost,random;
  delta = 0.0;

  cost = calculate_cost(path,N);

  #pragma omp parallel num_threads(num_proc)
  {
  newCost = cost;

  for (i = 0; i < N; i++)
  {
    priv_path[i] = path[i];
  }
  i=0;
  while(i<iter){
    c= (rand() % N);
    if(c==0){
      previous = N-1;
      next1=1;
      next2=2;
    }
    else{
      previous = c-1;
      next1=(c+1) % N;
      next2 = (c+2) % N;
    }
    delta = dist[priv_path[previous]][next1] + dist[priv_path[c]][priv_path[next2]]
     - dist[priv_path[previous]][priv_path[c]] - dist[priv_path[next1]][priv_path[
    next2]];

        random = (float)((double)rand()/(double)(RAND_MAX)) * 1; // nr random entre
     0 e 1 para ver se aceita o caminho ou não
    if(delta < 0 || (exp(-delta/temperatura))>random){
      tmp = priv_path[c];
      priv_path[c] = priv_path[next1];
      priv_path[next1] = tmp;
      newCost += delta;
    }
      if (delta<0) i=0;
    else i++;
    temperatura = 0.999*temperatura;
  }
  #pragma omp critical
    {
  if(newCost < cost){
    cost = newCost;

    for(i=0;i<iter;i++)
      path [i] = priv_path [i];
  }
   }
 }
  return cost ;
```

```
}

int main(int argc, char const *argv[])
{
  int i;
  float cost, tspCostAnn;

  srand((unsigned) time(NULL));

  if ( argc != 5 ){
           printf ("Usage : ./ tsp <nr cidades> <nr de processos ><nr de iteracoes
    ><temperatura inicial>\n");
     return 0;
  }
  cities = atoi(argv[1]);
  int num_proc = atoi(argv[2]);
  int iter = atoi(argv[3]);
  int temp = atof(argv[4]);

    int* x_pos = malloc (cities * sizeof (int)) ;
    int* y_pos = malloc (cities * sizeof (int)) ;

    //gera coordenadas para cada cidade num quadrado de lado 50
     for ( i = 0; i < cities; i++ ) {
    x_pos [i] = rand () % 50;
    y_pos [i] = rand () % 50;
  }
    initialize_matrix();

    find_distances(x_pos,y_pos,cities);

    randperm(path,cities);

    cost = calculate_cost(path,cities);
    printf("Custo Inicial :%f\n",cost);

    clearCache();

    start();

    tspCostAnn = tspAnnealing(matrix_distances,path,cities,iter,num_proc,temp);
    printf("Tempo Simulated Annealing:%f\n",stop());

    printf("Custo Ann  : %f\n", tspCostAnn);

    return 0;
}
```