O Problema do Caixeiro Viajante - PCV
Travelling Salesman Problem - TSP

Aluna:
Raquel Lopes
GB 106 Introdução à Computação Evolucionista
Professor: Helio Barbosa
Laboratório Nacional de Computação Científica - LNCC
25651-075, Petrópolis, RJ

November, 2010

1 Introdução

Metaheurística é um procedimento destinados a encontrar uma boa solução, consistindo na aplicação em cada passo, de uma heurística subordinada, a qual deve ser modelada para cada problema específico. Metaheurísticas possuem um carater geral e com condições de escapar de ótimos locais. Dentre as técnicas utilizadas, encontram-se a Computação Evolutiva (CE) que incorpora princípios de evolução biológica dentro de algoritmos usados para solucionar problemas grandes e com complicadas otimizações (Foster, 2001). As linhas de pesquisa que constituem a CE são os Algoritmos Geneticos, Programação Evolutiva, Estrategia Evolutiva, Sistemas Classificadores e Programação Genética.

1.1 Algoritmos Genéticos AGs

Na busca de soluções em espaços complexos e numerosos, inspirados nos conceitos de seleção natural, reprodução diferencial, fitness, crossover mutação, surgiu o algoritmo genético proposto inicialmente por Holland e colaboradores por volta de 1976. Os algoritmos genéticos empregam uma terminologia originada da teoria da evolução natural e da genética. Um indivíduo da população é representado por um único cromossomo, o qual contém a codificação (genótipo) de uma possível solução do problema (fenótipo). Cromossomos são usualmente implementados na forma de listas de atributos ou vetores, onde cada atributo é conhecido como gene. Os possíveis valores que um determinado gene pode assumir são denominados de alelos. O processo de evolução executado por um algoritmo genético corresponde a um procedimento de busca em um espaço de soluções potenciais para o problema. Dentre as vantagens no uso de AGs podemos citar: seu paralelismo inerente, simplicidade de implementação e a possibilidade de utilização para qualquer estrutura.

Um tipico AG funciona de acordo com o pseudo-algoritmo descrito abaixo:

Algorithm 1 Algoritmo Genetico tipico

Seja S(t) a população de cromossomos na geração t.
t$\longleftarrow 0$
inicializar S(t)
avaliar S(t)
while o critério de parada não for satisfeito **do**
t $\longleftarrow t+1 selecionar S(t) a partir de S(t-1) aplicar crossover sobre S(t) aplicar mutação o sobre S(t) avaliar S(t)$
end while

2 O Problema do Caixeiro Viajante (PCV), Travelling Salesman Problem

2.1 Descrição

Um determinado vendedor precisa encontrar o caminho mais curto dentre uma lista de cidades a serem visitadas, devendo passar por todas apenas uma única vez e retornar a cidade de origem. Cada cidade está ligada a uma outra por meio de uma estrada.

2.2 Classificação

O problema do caixeiro viajante é considerado um problema do tipo NP-difícil, pois não tem solução deteminística polinomial. Para um número pequeno de cidades, pode-se fazer uma busca exaustiva, entretanto a medida que aumenta o número de cidades esta solução torna-se computacionalmente impraticável.

3 Representações em AGs para o PCV

3.1 Cromossomo e População Inicial

Os cromossomos foram representados por sequências de números inteiros referentes a cada cidade, que, tomadas em conjunto, referem-se a um percurso hipotético (solução candidata), e um valor de aptidão associado.

3.2 Função Objetivo

A função objetivo natural para este problema é o comprimento total do caminho, que começa em uma cidade origem e retorna a mesma. A função objetivo é dada por:

$$f(x,y) = \sum_{i=1}^{n} = \sqrt{(x_i - x_i + 1)^2 + (y_i - y_i + 1)^2}$$

3.3 Função Aptidão

A aptidão de cada solucao candidata é dada pela soma total do percurso entre as cidades a partir de suas coordenadas cartesianas.

3.4 Operadores Genéticos

SELECÃO Foi utilizado o criterio de selecao por Ranking. Dessa forma, os 50%melhores (50m) sao integralmente passados para a geracao seguinte. Os demais 50%da geracao seguintesao formados a partir da aplicacao de operadores geneticos em 50m. **MUTACÃO** Optou-se pelo operador de mutação baseada em ordem, na qual dois elementos do cromossomo (cidades) são escolhidos aleatoriamente e suas posições são trocadas. A taxa de mutação utilizada foi de 10%, de forma a compensar o forte elitismo do operador de seleção.

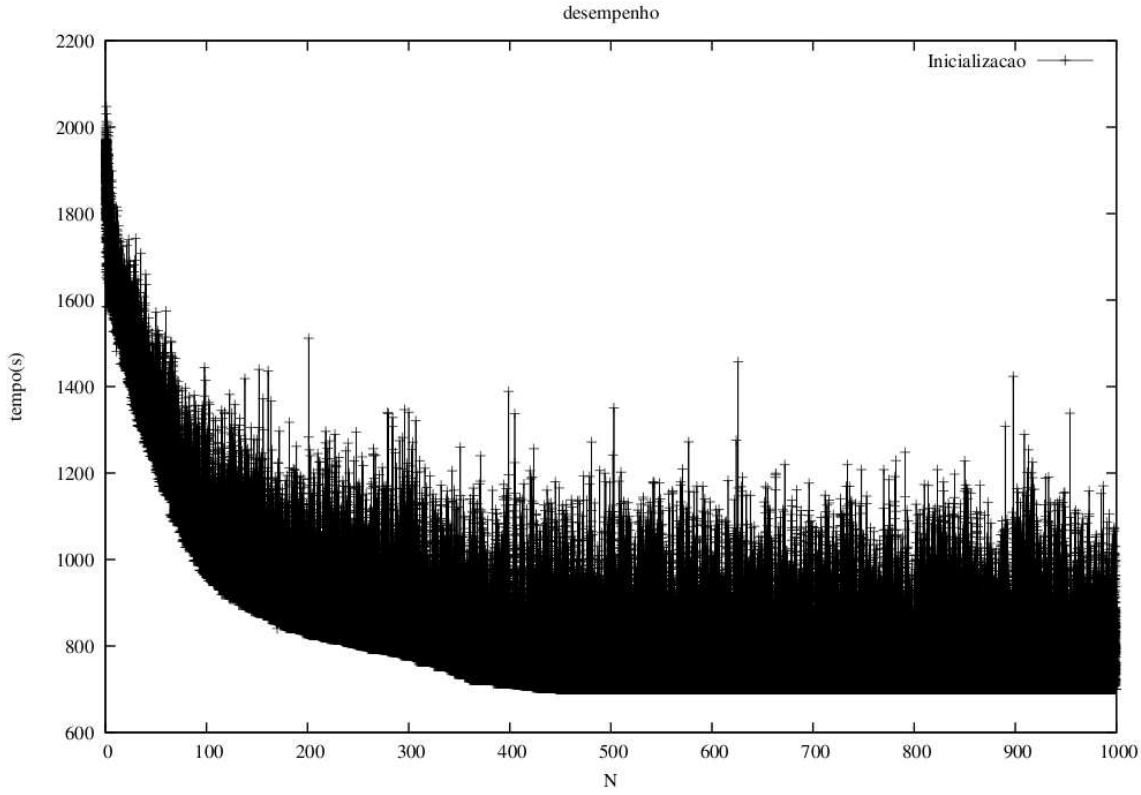
3.5 Critério de Parada

O critério de parada utilizado foi o número de gerações.

4 Resultados obtidos

A Figura 1 indica que existe convergência para a minimização da função-custo, permitindo que seja encontrada uma solução ótima ou aproximada. Adicionalmente, o gráfico reflete a tendência de toda a população a ter o valor percurso minimizado. Tais resultados permitem concluir o bom desempenho do algoritmo, com as seguintes ressalvas: o algoritmo apresenta convergência relativamente prematura (cerca de 400 gerações), provavelmente associada com o alto elitismo no critério de seleção. Valores

distantes do ótimo podem ser mesmo encontrados nas últimas gerações, em função da alta taxa de mutação empregada.



1000700

Figura 1: Resultado do desempenho do processo evolutivo.

5 Anexo 1 - Codigo Fonte em C

```c
 /*============================================
Problema do Caixeiro Viajante resolvido com Algoritmo genetico
Autores:
 * Raquel lopes
 * Raphael Trevizani
versao 0.8
data: 04/02/09
LNCC Laboratorio Nacional de Computacao Cientifica
============================================*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <limits.h>
#include <time.h>

#include "caixeiroViajante_TAD.h"
#define MAXBREED 1001 /*numero maximo de geracoes*/

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

    int *vet; /*vetor*/
    float **dist;  /* matriz de distancias entre as cidades*/
    int j, nBreed=1;
    FILE* fp = fopen("data.txt","wt");
    if(!fp){
  printf("Erro ao abrir arquivo de saida\n");
      exit(1);
    }
```

```
31      vet = vetorCreate (NUMPOP); /*retorna o vetor alocado dinamicamente de acordo com o
            numero de cidades*/
32      dist = matrixDist(NUMCITY, NUMCITY); /*retorna uma matriz alocada para calculo das
            distancias entre as cidades*/
33      matrixArq("city50.txt", dist); /*le o arquivo com as cidades e cordenada (xy), calcula
            pares de distancias entre as cidades*/
34      printMatrizDist(dist); /*imprime valores da matriz de distancia calculados*/
35
36      /*AGs*/
37      CaixeiroViajante populacao[NUMPOP]; /*cria a populacao inicial (vazia) com um conjunto
            de solucoes candidatas*/
38      birthPop(populacao); /*nasce a populacao inicial com individuos*/
39      fitnessPop(populacao, dist); /*avalia a aptidao de todos os individuos na populacao*/
40      CaixeiroViajante *filho, *ptPop = populacao; /*ponteiro para vetor populacao*/
41      ptPop = popSort(ptPop); /*ordena populacao por aptidao (ordem decrescente)*/
42      /*testes com fitness pop*/
43      printPop(populacao); /*imprime a populacao gerada com aptidao*/
44
45      while (nBreed < MAXBREED){
46        filho = selecao(ptPop);
47    filho = mutacao(filho,rand()%(NUMPOP/2));
48    fitnessPop(filho, dist); /*avalia a aptidao de todos os individuos na populacao*/
49    fp= printFile(fp,nBreed, filho);//saida em arquivo
50    if(nBreed%250 == 0)
51      melhor(fp, nBreed, filho);
52        ptPop = popSort(filho); /*ordena populacao por aptidao (ordem decrescente)*/
53        printf("GeraÃ§Ã£o: %d\n",nBreed);
54        nBreed++;
55      }
56
57      printf("\nFIM DA SIMULACAO\nMelhor solucao encontrada...");
58      printf("\nAptidao do percurso: %f \n", populacao[0].aptidao);
59      printf("Percurso: ");
60         for (j=0; j<NUMCITY; j++){
61           printf("%d ", populacao[0].percurso[j]);
62          }
63      printf("\n\n");
64
65  return (EXIT_SUCCESS);
66  }
67  /*********************************************************************/
68  /*********************************************************************/
69  #ifndef _CAIXEIROVIAJANTE_TAD_H
70  #define _CAIXEIROVIAJANTE_TAD_H
71
72  #define NUMCITY 50 /*numero de cidades para visitar*/   /*VERIFICAR NO ARQUIVO DE ENTRADA*/
73  #define NUMPOP 500 /*tamanho da populacao*/
74
75  struct caixeiroViajante{
76     int   percurso[NUMCITY]; /*Uma rota possivel (solucao candidata)*/
77     float   aptidao; /*Aptidao da solucao (Valor total do percusso somado)*/
78  };
79  typedef struct caixeiroViajante CaixeiroViajante;
80
81
82
83  /***FUNCOES RELACIONADAS A CONSTRUCAO DA MATRIZ***/
84  /* Cria memoria para um vetor de acordo com o numero de cidades */
85  int *vetorCreate(int tamVet);
86  /* Cria matriz de ponteiros para float de acordo com o num de linhas e colunas */
87  float **matrixDist(int nlinhas, int ncolunas);
88  /*imprime matriz*/
89  void printMatrizDist(float **distancia);
90  /*fornece a */
91  float getMatrix(float** distance, int i, int j);
92  /*============================================*/
93  /***FUNCOES RELACIONADAS COM LEITURA DE ARQUIVOS***/
94  /* Le arquivo e calcula os pares de distancias das cidades e insere na matriz*/
95  void matrixArq(char nomeArq[], float **distance);
96  /*============================================*/
97  /***FUNCOES RELACIONADAS COM HEURISTICAS AGs***/
98  /* Gera a populacao inicial*/
99  void birthPop(CaixeiroViajante *populacao);
100 /*Imprime a populacao gerada*/
```

```
101 void printPop(CaixeiroViajante *populacao);
102 /*Avalia cada uma das rotas (individuos) na populacao*/
103 void fitnessPop(CaixeiroViajante *populacao, float **distance);
104 /*Organiza a populacao de acordo com o fitness*/
105 CaixeiroViajante* popSort(CaixeiroViajante* pPop);
106 /*funcao auxiliar da ordenacao. Compara o fitness de dois individuos*/
107 static int compFit(const void* a,const void* b);
108 CaixeiroViajante* selecao(CaixeiroViajante* pai);
109 CaixeiroViajante* mutacao(CaixeiroViajante* filho,int nmut);
110 FILE* printFile(FILE* f, int breed, CaixeiroViajante* p);
111 int getCity(FILE* fcity, int n, int* x, int* y);
112 void melhor(FILE* fcity, int breed, CaixeiroViajante* p);
113 #endif  /* _CAIXEIROVIAJANTE_TAD_H */
114 /*****************************************************************/
115 /*****************************************************************/
116 #include <stdio.h>
117 #include <stdlib.h>
118 #include <string.h>
119 #include <math.h>
120 #include <limits.h>
121 #include <time.h>
122
123 #include "caixeiroViajante_TAD.h"
124
125 #define NUMCITY 50
126
127 /*=============================================*/
128 /*** Cria memoria para um vetor de acordo com o numero de cidades */
129 int *vetorCreate(int tamVet){
130
131     int *vetor;
132     vetor = (int*)malloc(tamVet*sizeof(int));
133     if (!vetor){
134     printf("Falta memoria para alocar o vetor de ponteiros");
135         exit(1);
136     }
137   return vetor;
138 }
139 /*=============================================*/
140 /*** Cria matriz de ponteiros para float de acordo com o num de linhas e colunas */
141 float **matrixDist(int nlinhas, int ncolunas){
142
143     int i;
144     float **matrix;
145     matrix = (float**)malloc(nlinhas*sizeof(float*));
146     if (!matrix) {
147        printf("Falta memoria para alocar a matriz de ponteiros\n");
148        exit(1);
149     }
150     for (i=0; i< nlinhas; i++) {
151        matrix[i] = (float*)malloc(ncolunas*sizeof(float));
152        if (!matrix[i]){
153        printf("Falta memoria para alocar a matriz de ponteiros.\n");
154        exit(1);
155        }
156     }
157   return matrix;
158 }
159 /*=============================================*/
160 /*** Le arquivo e calcula os pares de distancias d_ij das cidades e insere na matriz*/
161 void matrixArq(char nomeArq[], float **distance){
162
163   int *vetorX, *vetorY;
164   int x, y, i, j;
165   float xi, yi;
166   FILE *arq;
167
168   vetorX = vetorCreate(NUMCITY);/*cria vetor para inserir coordenadas X*/
169   vetorY = vetorCreate(NUMCITY);/*cria vetor para inserir coordenadas Y*/
170
171   arq = fopen(nomeArq,"r");
172
173   if (!arq) {
174      printf("O Arquivo %s nao pode ser aberto.\n", nomeArq);
```

```
175 //       getchar();
176       exit(1);
177    }
178    while (!feof(arq)){
179       fscanf(arq, "%d %d %d", &i, &x, &y);
180       vetorX[i] = x;
181       vetorY[i] = y;
182    }
183    /*matriz de distancias calculado a partir das distancias euclidianas dos pontos xy*/
184    for(i=0; i <NUMCITY-1; i++) {
185       distance[i][i] = 0;
186       for(j=i+1; j <NUMCITY; j++){
187          xi= (float)pow((vetorX[i]- vetorX[j]),2);
188          yi= (float)pow((vetorY[i]- vetorY[j]),2);
189          distance[i][j]= sqrt(xi+ yi);
190          distance[j][i] = distance[i][j]; /*pq a matriz e simetrica*/
191        }
192     }
193    fclose(arq);
194 
195    free(vetorX); /*libera a memoria dos vetores */
196    free(vetorY);
197 }
198 /*============================================*/
199 void printMatrizDist(float **distancia){
200 
201 int i, j;
202 float valor;
203    for(i=0; i <NUMCITY- 1; i++) {
204      for(j=i+1; j <NUMCITY; j++){
205        valor= distancia[i][j]; //conteudo da matriz valor da distancia
206        printf("\nvalor da coordenada x: %d Y: %d -> %f\n", i, j, valor);
207      }
208    }
209 }
210 /*============================================*/
211 float getMatrix(float** distance, int i, int j){
212 
213    float valor;
214    return (valor= distance[i][j]);
215 }
216 /*********************************************************************/
217 /*********************************************************************/
218 #include <stdio.h>
219 #include <stdlib.h>
220 #include <string.h>
221 #include <math.h>
222 #include <limits.h>
223 #include <time.h>
224 
225 #include "caixeiroViajante_TAD.h"
226 
227 #define NUMCITY 50 /*numero de cidades para visitar*/    /*VERIFICAR NO ARQUIVO DE ENTRADA*/
228 #define NUMPOP 500 /*tamanho da populacao*/
229 
230 /* Gera a populacao inicial*/
231 void birthPop(CaixeiroViajante *populacao){
232 
233    int i; //contador para a populacao;
234    int j,k;
235    int sort, quantos;
236    int check=1; //parametro para ajudar a verificar se determinada cid ja foi sorteada;
237    srand((unsigned) time(NULL)/2); //sorteia um numero aleatorio
238 
239    for(i=0; i<NUMPOP; i++){
240      j=0;
241      quantos=0;
242      while (j < NUMCITY){
243        sort = rand()%NUMCITY;
244        for (k=0; k<quantos; k++)
245          if (populacao[i].percurso[k] == sort){ /*cidade ja sorteada*/
246            check = 0; //pq isto??
247    }
248          if(check){
```

```
249             populacao[i].percurso[j] = sort;
250             quantos++;
251             j++;
252           }
253           check= 1;
254       }
255     // fitness (populacao, i, numPop);
256     }
257 }
258 /*============================================*/
259 /*Imprime a populacao gerada*/
260 void printPop(CaixeiroViajante *populacao){
261
262     int i, j;
263     for (i=0; i<NUMPOP; i++){
264        j=0; //vou retirar
265        printf("\n\nNumero do individuo da solucao candidata: %d\n", i);
266        printf("Aptidao do percurso: %f \n", populacao[i].aptidao);
267        printf("Percurso: ");
268        for (j=0; j<NUMCITY; j++){
269          printf("%d ", populacao[i].percurso[j]);
270         }
271     }
272     printf("\n");
273 }
274 /*============================================*/
275 /*Avalia a aptidao de toda a populacao*/
276 void fitnessPop(CaixeiroViajante *populacao, float **distance){
277
278     int i, j;
279     float apt; /*valor da aptidao*/
280     for (i=0; i<NUMPOP; i++){
281        apt= 0;
282        j=0;//vou retirar
283       for (j=0; j<NUMCITY; j++){
284           if(j==NUMCITY -1){ /*ultimo elemento no vetor*/
285             apt= apt + getMatrix(distance, populacao[i].percurso[j], populacao[i].percurso
                      [0]);/*retorno a cidade de origem*/
286       populacao[i].aptidao = apt;
287     }
288     else
289             apt = apt + getMatrix(distance, populacao[i].percurso[j], populacao[i].percurso
                      [j+1]);
290       }
291     }
292 }
293 /*============================================*/
294 /*Funcao para ordenar populacao*/
295 CaixeiroViajante* popSort(CaixeiroViajante* pPop){
296
297   printf("Organizando o vetor em ordem decrescente de acordo com o fitness...");
298   qsort(pPop,NUMPOP,sizeof(CaixeiroViajante),compFit);
299   //printf("done\n");
300   return pPop;
301 }
302 /*============================================*/
303 /*funcao auxiliar da ordenacao. Compara o fitness de dois individuos*/
304 static int compFit(const void* a,const void* b){
305
306   CaixeiroViajante *pt1= (CaixeiroViajante*)a;
307   CaixeiroViajante *pt2= (CaixeiroViajante*)b;
308   if(pt1->aptidao < pt2->aptidao)
309      return -1;
310   else if(pt1->aptidao > pt2->aptidao)
311      return 1;
312   else
313      return 0;
314 }
315 /*============================================*/
316 CaixeiroViajante* selecao(CaixeiroViajante* pai){//recebe ponteiro para pai e retorna para
      filho
317   CaixeiroViajante* filho = (CaixeiroViajante*) malloc(sizeof(CaixeiroViajante)*NUMPOP);
318   printf("Aplicando selecao...");
319
```

```
320   for(int i=0; i<NUMPOP/2; i++)
321     filho[i] = pai[i];
322   for(int i=0; i<NUMPOP/2; i++){
323     for(int j=NUMPOP/2; j<NUMPOP; j++){
324       filho[j] = pai[i];
325       i++;
326     }
327   }
328   printf("feito\n");
329   return filho;
330 }
331 /*============================================*/
332 CaixeiroViajante* mutacao(CaixeiroViajante* filho,int n){//novo operador de mutacao
333   int temp;
334   int a, b1,b2;
335   printf("Aplicando mutacao...");
336   for(int i=0; i<n; i++){
337     a = rand()%(NUMPOP);//mutacao atinge somente a primeira metade do vetor 'filho'
338     b1 = rand()%NUMCITY;
339     b2 = rand()%NUMCITY;
340     temp = filho[a].percurso[b2];
341     filho[a].percurso[b2] = filho[a].percurso[b1];
342     filho[a].percurso[b1] = temp;
343   }
344   printf("feito\n");
345   return filho;
346 }
347 /*============================================*/
348 FILE* printFile(FILE* f, int breed, CaixeiroViajante* p){
349   for(int i=0; i<NUMPOP; i++)
350     fprintf(f,"%d\t%f\n",breed,p[i].aptidao);
351   return f;
352 }
353
354 /*============================================*/
355 int getCity(FILE* fcity, int n, int* x, int* y){
356   int n2;
357   char line[10];
358   while(fgets(line,10,fcity)){
359     sscanf(line,"%d",&n2);//compara primeira coluna: n da cidade
360     if(n==n2){//se for mesma cidade
361       sscanf(line,"%*d %d %*d",&x);
362       sscanf(line,"%*d %*d %d",&y);
363       rewind(fcity);
364       return 1;
365     }
366   }
367 }
368
369 /*============================================*/
370 /*saida em arquivo do melhor da pop. fcity = txt de cidades*/
371 void melhor(FILE* fcity, int breed, CaixeiroViajante* p){
372   int ok;
373   char filename[20], str[20];
374   sprintf(filename,"path%d.txt",breed);
375   FILE* fout = fopen(filename,"wt");
376   if(!fout)
377     printf("Erro ao abrir arquivo %s\n",filename);
378
379   int vx[NUMCITY], x;
380   int vy[NUMCITY], y;
381   for(int i=0; i<NUMCITY; i++){
382     ok = getCity(fcity,p[0].percurso[i], &x, &y);
383 //    vx[i] = x;
384 //    vy[i] = y;
385     if(ok){
386       sprintf(str,"%d %d %d\n",i,x,y);
387       fputs(str,fout);
388     }
389   }
390
391   return;
392 }
```