IN 101 - TD 13
17 décembre 2010

Corrigé

ENSTA ParisTech

Solution 1 : Fermeture transitive d'un graphe

1.a] Voici le code de la fonction `roy_warshall` :

```
1  graph* roy_warshal (graph* G) {
2    int i,j,k;
3    graph* H = init_graph(G->n);
4    for (i=0; i<G->n; i++) {
5      for (j=0; j<G->n; j++) {
6        if (i == j) {
7          H->mat[i][j] = 1;
8        } else {
9          H->mat[i][j] = G->mat[i][j];
10       }
11     }
12   }
13   for (k=0; k<G->n; k++) {
14     for (i=0; i<G->n; i++) {
15       for (j=0; j<G->n; j++) {
16         H->mat[i][j] = H->mat[i][j] || (H->mat[i][k] && H->mat[k][j]);
17       }
18     }
19   }
20   return H;
21 }
```

1.b] Voici le reste du code :

```
1  typedef struct {
2    int n;
3    int** mat;
4  } graph;
5
6  void print_graph (graph* G) {
7    int i,j;
8    for (i=0; i<G->n; i++) {
9      for (j=0; j<G->n; j++) {
10       printf("%d ", G->mat[i][j]);
11     }
12     printf("\n");
13   }
14 }
15
16 int main() {
17   graph* G = init_graph(7);
```

```
18   /* attention, les indices sont décalés de 1 */
19   G->mat[0][1] = 1; G->mat[0][3] = 1;
20   G->mat[1][4] = 1;
21   G->mat[2][4] = 1; G->mat[2][5] = 1;
22   G->mat[3][1] = 1;
23   G->mat[4][3] = 1;
24   G->mat[5][5] = 1;
25
26   graph* H = roy_warshall(G);
27   print_graph(H);
28 }
```

1.c] Pour lire la matrice depuis un fichier on fait comme cela :

```
1  #include <stdio.h>
2  graph* read_graph (char* filename) {
3    int i,j;
4    FILE* input = fopen(filename, "r");
5    if (input == NULL) {
6      fprintf(stderr, "Impossible de lire le fichier %s.\n", filename);
7      return NULL;
8    }
9    int nb;
10   if (fscanf(input, "%d", &nb) != 1) {
11     fprintf(stderr,"Le fichier %s n'est pas au bon format.\n",filename);
12     fclose(input);
13     return NULL;
14   }
15
16   graph* G = (graph*) malloc(sizeof(graph));
17   int** matrix = (int**) malloc(nb * sizeof(int*));
18   for (i=0; i<nb; i++) {
19     matrix[i] = (int*) malloc(nb * sizeof(int));
20     for (j=0; j<nb; j++) {
21       if (fscanf(input, "%d", &(matrix[i][j])) != 1) {
22         fprintf(stderr,"Le fichier %s n'est pas au bon format.\n",filename);
23         fclose(input);
24         return NULL;
25       }
26     }
27   }
28   fclose(input);
29   G->n = nb;
30   G->mat = matrix;
31   return G;
32 }
```

1.d] Ensuite, le `main` est encore plus simple :

```
1  int main() {
2    graph* G = read_graph("~finiasz/graph");
3    graph* H = roy_warshall(G);
```

```
  print_graph(H);
}
```

1.e] Cette fonction affiche pour chaque sommet l'ensemble de ses successeurs. Avec une fonction très similaire on pourrait aussi convertir le graphe vers une véritable structure de liste de successeurs.

```
void print_successors (graph* G) {
  int i,j;
  for (i=0; i<G->n; i++) {
    printf("successeurs du sommet %d: ",i);
    for (j=0; j<G->n; j++) {
      if (G->mat[i][j] == 1) {
        printf("%d ", j);
      }
    }
    printf("\n");
  }
}
```

Solution 2 : Recherche de plus courts chemins dans un graphe

2.a] Voici l'algorithme de Aho-Hopcroft-Ullman et l'initialisation associée :

```
#define infinity -1

int min(int a, int b) {
  if (a == infinity) {
    return b;
  } else if (b == infinity) {
    return a;
  } else if (a < b) {
    return a;
  }
  return b;
}

int add(int a, int b) {
  if ((a == infinity) || (b == infinity)) {
    return infinity;
  } else {
    return a+b;
  }
}

void init_graph2 (int nb, graph* G) {
  int i,j;
  int** matrix = (int**) malloc(nb * sizeof(int*));
  for (i=0; i<nb; i++) {
    matrix[i] = (int*) malloc(nb * sizeof(int));
    for (j=0; j<nb; j++) {
      matrix[i][j] = infinity;
```

```
    }
  }
  G->n = nb;
  G->mat = matrix;
}

graph* AHU(graph* G) {
  int i,j,k;
  graph* H = (graph*) malloc(sizeof(graph));
  init_graph2(G->n, H);
  for(i=0; i<G->n; i++) {
    for(j=0; j<G->n; j++) {
      if (i == j) {
        H->mat[i][j] = 0;
      } else {
        H->mat[i][j] = G->mat[i][j];
      }
    }
  }
  for(k=0; k<G->n; k++) {
    for(i=0; i<G->n; i++) {
      for(j=0; j<G->n; j++) {
        H->mat[i][j] = min(H->mat[i][j], add(H->mat[i][k], H->mat[k][j]));
      }
    }
  }
  return H;
}
```

2.b] Il suffit de lire le fichier, symétriser la matrice, puis lancer `AHU`.

```
int main() {
  int i,j;
  graph* G = read_graph(""~finiasz/metro");

  // on symétrise le graphe
  for(i=0; i<G->n; i++) {
    G->mat[i][i] = 0;      // la diagonale
    for(j=0; j<i; j++) {
      G->mat[i][j] = G->mat[j][i];
    }
  }
  graph* H = AHU(G);
  print_graph(H);
}
```

2.c] La réponse est dans le poly...

2.d] Voici la structure de liste de successeurs et l'algorithme de Dijkstra (il y a aussi un peu de travail pour implémenter la structure de tas) :

```
typedef struct {
  int n;
  node* tab;
} graph_L;

typedef struct node_st {
  int end;   // le sommet d'arrivé
  int dist;  // la distance du sommet
  struct node_st* next; // successeur suivant, ou NULL
} node;

// On suppose que l'on a déjà une structure de tas implémentée.
// Les sommets sont triés en fonction des distances contenues dans
// le tableau dists, le plus proche à la racine.
// heap_insert déplace le sommet dans le tas si il est déjà dedans
// et seul sa distance a changée, il ne l'insère pas une deuxième fois.
void heap_insert(heap* H, int sommet, int* dists);
int heap_extract(heap* H, int* dists);

void dijkstra(graph_L* G, int start) {
  int i, cur;
  node* tmp;
  int* fathers = (int*) malloc(G->n*sizeof(int));
  int* dists = (int*) malloc(G->n*sizeof(int));
  int* clors = (int*) malloc(G->n*sizeof(int));
  for (i=0; i<G->n; i++) {
    fathers[i] = -1;
    dists[i] = 100000; // 100000 = l'infini, pour simplifier
    colors[i] = 0;
  }

  dists[start] = 0;

  heap* H = heap_init(G->n);
  heap_insert(H, start, dists);

  while ((cur = heap_extract(H, dists)) != -1) {
    // on suppose que heap_extract retourne -1 quand le tas est vide
    color[cur] = 1;
    tmp = G->tab[cur];
    while (tmp != NULL) {
      if ((colors[tmp->end]==0) && (dists[tmp->end] > dists[cur]+tmp->dist)) {
        fathers[tmp->end] = cur;
        dists[tmp->end] = dists[cur] + tmp->dist;
        heap_insert(H, tmp->end, dists); // insère ou met à jour
      }
      tmp = tmp->next;
    }
  }
  // les tableaux des pères et des distances sont remplis.
}
```

Solution 3 : Spéculation monétaire

3.a] Voici le code qui va bien :

```
1  #include <stdio.h>
2  #define infinity -1
3
4  typedef struct {
5    int n;
6    double** mat;
7  } graph;
8
9  void init_graph (int nb, graph* G) {
10   int i,j;
11   double** matrix =  (double**) malloc(nb*sizeof(double*));
12   for (i=0; i<nb; i++) {
13     matrix[i] = (double*) malloc(nb*sizeof(double));
14     for (j=0; j<nb; j++) {
15       matrix[i][j] = 0;
16     }
17   }
18   G->n = nb;
19   G->mat = matrix;
20 }
21 double max(double a, double b) {
22   if ((a == infinity) || (b == infinity)) {
23     return infinity;
24   }
25   if (a < b) {
26     return b;
27   }
28   return a;
29 }
30 double mult(double a, double b) {
31   if ((a == infinity) || (b == infinity)) {
32     return infinity;
33   } else {
34     return a*b;
35   }
36 }
37 double star(double a) {
38   if (a > 1) {
39     return infinity;
40   } else {
41     return 1;
42   }
43 }
44 void print_graph (graph* G) {
45   int i,j;
46   for (i=0; i<G->n; i++) {
47     for (j=0; j<G->n; j++) {
48       if (G->mat[i][j] == infinity) {
49         printf("infini ");
50       } else {
51         /* imprime un flottant avec 4 décimales */
```

```
52          printf("%.4f ", G->mat[i][j]);
53        }
54      }
55      printf("\n");
56    }
57  }
58  graph* AHU(graph* G) {
59    int i,j,k;
60    graph* H = (graph*) malloc(sizeof(graph));
61    init_graph(G->n, H);
62    for(i=0; i<G->n; i++) {
63      for(j=0; j<G->n; j++) {
64        if (i == j) {
65          H->mat[i][j] = max(1, H->mat[i][j]);
66        } else {
67          H->mat[i][j] = G->mat[i][j];
68        }
69      }
70    }
71    for(k=0; k<G->n; k++) {
72      for(i=0; i<G->n; i++) {
73        for(j=0; j<G->n; j++) {
74          H->mat[i][j] = max(H->mat[i][j], mult(H->mat[i][k],
75                                  mult(star(H->mat[k][k]),H->mat[k][j])));
76        }
77      }
78    }
79    return H;
80  }
81  #define euro 0
82  #define dollar 1
83  #define livre 2
84  #define suisse 3
85
86  int main() {
87    graph* G = (graph*) malloc(sizeof(graph));
88    init_graph(4,G);
89    G->mat[euro][euro] = 1.0;      G->mat[dollar][dollar] = 1.0;
90    G->mat[livre][livre] = 1.0;    G->mat[suisse][suisse] = 1.0;
91
92    G->mat[euro][dollar] = 1.28;   G->mat[euro][livre] = 0.68;
93    G->mat[euro][suisse] = 1.55;   G->mat[dollar][euro] = 0.78;
94    G->mat[dollar][livre] = 0.53;  G->mat[dollar][suisse] = 1.21;
95    G->mat[livre][euro] = 1.46;    G->mat[livre][dollar] = 1.87;
96    G->mat[livre][suisse] = 2.27;  G->mat[suisse][euro] = 0.64;
97    G->mat[suisse][dollar] = 0.82; G->mat[suisse][livre] = 0.43;
98
99    graph* H = AHU(G);
100   print_graph(H);
101
102   /* on introduit une erreur volontairement */
103   G->mat[livre][suisse] = 2.30;
104   H = AHU(G);
105   print_graph(H);
106 }
```