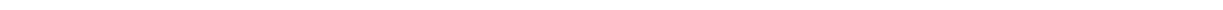


TIPE

# Fichier Code

Olivier Caffier



```

1  #include "structure3.h"
2
3  #include <time.h>
4
5
6  typedef int** cell; // De taille 4, 0 pour top, on poursuit dans le sens des aiguilles d'une montre
7
8
9
10 struct plateau_s{
11     int len; // Donne le nombre de slots, pour un plateau de n*n cases, len = n
12     cell* components;
13     bool completed;
14 };
15 typedef struct plateau_s plateau;
16
17 int nb_liaisons(cell* compo, int num_card){
18     int compteur = 0;
19     for (int i=0; i<4; i++){
20         compteur += (*compo)[num_card][i] ;
21     }
22     return compteur % 2;
23 }
24
25 void trouve_slots_prio(int** conditions, int taille_plateau, MaxPriorityQueue* file_prio){
26     int* compteurs_concern = malloc(taille_plateau * sizeof(int));
27     for (int i=0; i< taille_plateau; i++){
28         compteurs_concern[i] = 0;
29     }
30
31     int nb_conditions = conditions[0][0];
32
33     for (int i=1; i<nb_conditions+1; i++){
34         int slot_A = conditions[i][0];
35         int slot_B = conditions[i][1];
36         compteurs_concern[slot_A] +=1;
37         compteurs_concern[slot_B] +=1;
38     }
39
40     for (int id_case=0 ; id_case<taille_plateau; id_case++){
41         insert(file_prio,compteurs_concern[id_case],id_case);
42     }
43 }
44
45
46
47 int* compte_nb_slots_cards(cell* cards, int nb_cases){
48     int* res = malloc(nb_cases * sizeof(int));
49     for (int id_card=0; id_card<nb_cases; id_card++){
50         int tmp =0;
51         for (int id_side=0; id_side<4; id_side++){
52             if (cards[id_side][0] >0){
53                 tmp+=1;
54             }
55         }
56         res[id_card]=tmp;
57     }
58     return res ;
59 }
60
61
62
63
64 void trouve_cards_useful(int** conditions, int taille_plateau, bool** cards_usefuls, cell* cards){
65
66     int* compteurs_concern = malloc(taille_plateau * sizeof(int));

```

```

67
68     for (int i=0; i< taille_plateau; i++){
69         compteurs_concern[i] = 0;
70     }
71
72     int nb_conditions = conditions[0][0];
73
74     for (int i=1; i<nb_conditions+1; i++){
75         int slot_A = conditions[i][0];
76         int slot_B = conditions[i][1];
77         compteurs_concern[slot_A] +=1;
78         compteurs_concern[slot_B] +=1;
79     }
80
81
82
83     int* nb_slots_cards = compte_nb_slots_cards(cards,taille_plateau);
84
85
86     for (int id_case=0; id_case<taille_plateau; id_case++){
87         for (int id_card=0; id_card<taille_plateau; id_card++){
88             // La carte id_card pour l'emplacement id_case
89             if (compteurs_concern[id_case] <= nb_slots_cards[id_card]){
90                 // La carte a assez de slots pour convenir aux contraintes imposees a id_case
91                 cards_usefuls[id_case][id_card] = true;
92             }
93             else{
94                 cards_usefuls[id_case][id_case] = false;
95             }
96         }
97     }
98
99
100
101 }
102
103
104 void rotate_right(cell card){
105     // 1ere etape
106     for (int i=0; i<4; i++){
107         int nb_neighbors = card[i][0];
108         for (int j=1; j< nb_neighbors +1; j++){
109             card[i][j] = (card[i][j] + 1)%4;
110         }
111     }
112
113     int* tmp = malloc(4*sizeof(int));
114     for (int i=0; i<card[0][0]+1; i++){
115         tmp[i]=card[0][i];
116     }
117
118     for (int i=0; i<card[3][0]+1; i++){
119         card[0][i]=card[3][i];
120     }
121     for (int i=0; i<card[2][0]+1; i++){
122         card[3][i]=card[2][i];
123     }
124     for (int i=0; i<card[1][0]+1; i++){
125         card[2][i]=card[1][i];
126     }
127     for (int i=0; i<tmp[0]+1; i++){
128         card[1][i]=tmp[i];
129     }
130     free(tmp);
131
132

```

```

133 }
134
135 bool in_bornes(int n,int nb){
136     return (nb >= 0 && nb < n);
137 }
138
139 bool accessible(plateau* board, int case_start, int case_end, int side_start, int side_end, bool* vus){
140     printf("appel accessible \n");
141     if (case_start == case_end){
142         if (side_start == side_end){
143             return true;
144         }
145         else{
146             int nb_neighbor = (board->components)[case_start][side_start][0];
147             for (int i=1; i<nb_neighbor+1; i++){
148                 if ((board->components)[case_start][side_start][i]==side_end){
149                     return true;
150                 }
151             }
152             return false;
153         }
154     }
155     else{
156         if (vus[case_start]){
157             // Si la case est deja parcourue, on ne la parcourt pas une deuxieme fois
158             return false;
159         }
160         else{
161             int nb_neighbor = (board->components)[case_start][side_start][0];
162             int len = board->len;
163             vus[case_start]=true;
164             for (int i=1; i<nb_neighbor+1;i++){
165                 // On parcourt chaque voisin
166                 if ((board->components)[case_start][side_start][i]==0 && in_bornes(board->len*board->len,
167                     case_start-len)){
168                     if ( accessible(board,case_start-len,case_end,2,side_end,vus)){
169                         return true;
170                     }
171                 }
172                 else if ((board->components)[case_start][side_start][i]==1 && in_bornes(board->len*board->
173                     len,case_start+1) && (case_start %len!=len-1)){
174                     // A CORRIGER
175                     if (accessible(board,case_start+1,case_end,3,side_end,vus)){
176                         return true;
177                     }
178                 }
179                 else if ((board->components)[case_start][side_start][i]==2 && in_bornes(board->len*board->
180                     len,case_start+len)){
181                     if (accessible(board,case_start+len,case_end,0,side_end,vus)){
182                         return true;
183                     }
184                 }
185                 else if ((board->components)[case_start][side_start][i]==3 && in_bornes(board->len*board->
186                     len,case_start-1) && (case_start%len !=0)){
187                     // A CORRIGER
188                     if (accessible(board,case_start-1,case_end,1,side_end,vus)){
189                         return true;
190                     }
191                 }
192             }
193             return false;
194         }
195     }
196 }

```

```

195 }
196
197
198 bool is_border(plateau* board, int ind_case, int side_case){
199     if (!in_bornes(board->len*board->len, ind_case-board->len)){
200         // Partie haute du plateau
201         if (ind_case % board->len == 0){
202             // Position 2 : tester sur le cote gauche
203             return side_case == 3;
204         }
205         else if (ind_case % board->len == board->len-1){
206             // Position 4 : tester en bas et a gauche
207             return side_case == 1;
208         }
209         else{
210             // Position 3 : tester en bas, a droite et a gauche
211             return side_case == 0;
212         }
213     }
214     else if (!in_bornes(board->len*board->len, ind_case + board->len)){
215         // Partie basse du plateau
216
217
218         if (ind_case % board->len == 0){
219             // Position 8 : tester la gauche et en bas
220
221             return side_case == 3 || side_case == 2;
222
223         }
224         else if (ind_case % board->len == board->len-1){
225             // Position 6 : tester en bas et a droite
226
227             return side_case == 1 || side_case == 2;
228         }
229         else{
230             // Position 7 : tester en bas
231             return side_case == 2;
232         }
233     }
234 }
235 else{
236     // Pos 9,1,5
237     if (ind_case % board->len == 0){
238         // Position 9 : tester a droite, en haut et en bas
239         return side_case == 3;
240     }
241     else if (ind_case % board->len < board->len-1){
242         // Case du milieu
243         return false;
244     }
245     else{
246         // Position 5 : tester a gauche, en bas et en haut
247         return side_case == 1;
248     }
249 }
250 }
251
252
253
254 void affiche_case(cell card){
255     for (int i=0; i<4; i++){
256         printf("Sommet %d : ", i);
257         for (int j=1; j< card[i][0]+1; j++){
258             printf("%d ", card[i][j]);
259         }
260         printf("\n");

```

```

261     }
262     printf("\n");
263 }
264
265 bool est_valide_board(plateau* board, int** conditions){
266
267     int len = board->len;
268     const int NB_COND = conditions[0][0];
269     bool* vus = malloc(len*len*sizeof(bool));
270
271     for (int j=0;j<len*len; j++){
272         vus[j]=false;
273     }
274
275
276     for (int i=1; i<NB_COND+1; i++){
277
278         int case_start = conditions[i][0];
279         int case_end = conditions[i][1];
280         int side_start = conditions[i][2];
281         int side_end = conditions[i][3];
282
283
284
285         bool res = accessible(board,case_start,case_end,side_start,side_end,vus);
286
287
288         if (!accessible(board,case_start,case_end,side_start,side_end,vus)){
289
290             return false;
291         }
292     }
293
294     for (int j=0;j<len*len; j++){
295         vus[j]=false;
296     }
297
298     return true;
299 }
300
301 void reset_board(plateau* board){
302     int len = board->len;
303     for (int i=0; i<len*len; i++){
304         board->components[i]=NULL;
305     }
306     board->completed = false;
307 }
308 void reset_u_cards(bool* tab, int len){
309     for (int i=0; i<len; i++){
310         tab[i] = false;
311     }
312 }
313 void reset_rotations_and_ordre(int* tab, int len){
314     for (int i=0;i<len;i++){
315         tab[i]=-1;
316     }
317 }
318 void affiche_case_fichier(int* ordre_cards, int* rotations_cards, int len, FILE* fichier){
319     // On suppose que le fichier est deja open en ecriture
320     fprintf(fichier,"%d \n",len);
321     for (int i=0; i<len;i++){
322         fprintf(fichier,"%d \n",ordre_cards[i]);
323     }
324     for (int i=0; i<len;i++){
325         fprintf(fichier,"%d \n",rotations_cards[i]);
326     }

```

```

327 }
328
329 void transfer_files(MaxPriorityQueue* f1, MaxPriorityQueue* f2){
330     // On transfere tout le contenu de f2 dans f1
331
332     while (f2->size != 0){
333         Element elt = extractMax(f2);
334         insert(f1,elt.weight,elt.identifiant);
335     }
336 }
337
338 bool bruteforce(int** conditions, plateau* board, cell* cards, bool* used_cards, int indice_case_start, int
    current_slot, int* ordre_cards, int* rotations_cards, FILE* fichier){
339     if (board->completed || current_slot >= board->len* board->len){
340         if (est_valide_board(board,conditions)){
341             printf("SUCCESS ! \n");
342
343             affiche_case_fichier(ordre_cards,rotations_cards,board->len*board->len,fichier);
344             // Afficher le board dans le fichier de retour
345             return true;
346         }
347         return false;
348     }
349     else if (current_slot == 0){
350         if (indice_case_start== board->len * board->len){
351             return false;
352         }
353         else{
354             for (int i = 0; i < 4; i++){
355                 rotate_right(cards[indice_case_start]);
356                 board->components[0]= cards[indice_case_start];
357                 used_cards[indice_case_start] = true;
358                 ordre_cards[0]=indice_case_start+1;
359                 rotations_cards[0]=i+1;
360                 if (bruteforce(conditions,board,cards,used_cards,indice_case_start,current_slot+1,
361                     ordre_cards,rotations_cards,fichier)){
362                     return true;
363                 }
364                 else{
365                     reset_board(board);
366                     int LEN = board->len * board->len;
367                     reset_u_cards(used_cards,LEN);
368                     reset_rotations_and_ordre(ordre_cards,LEN);
369                     reset_rotations_and_ordre(rotations_cards,LEN);
370                 }
371             }
372             // Si on a teste sur toutes les positions de la carte au depart, il faut changer de carte de
373             depart
374
375             // On teste sur la compo suivante
376             return bruteforce(conditions,board,cards,used_cards,indice_case_start+1,0,ordre_cards,
377                 rotations_cards,fichier);
378         }
379     }
380     else{
381         // Le board n'est ni completed ni en position de depart
382         for (int i=0; i<board->len*board->len; i++){
383             if (!used_cards[i]){
384                 // Pour toutes les cartes non-utilisees
385                 for (int j=0; j<4; j++){
386                     // On teste toutes les orientations possibles
387                     bool* copy_tmp = malloc( board->len * board->len * sizeof(bool));
388                     int* copy_ordre = malloc(board->len * board->len * sizeof(int));
389                     int* copy_rotation = malloc(board->len * board->len * sizeof(int));
390                     for (int w = 0; w< board->len * board->len; w++){

```

```

389         copy_tmp[w] = used_cards[w];
390         copy_ordre[w]=ordre_cards[w];
391         copy_rotation[w]=rotations_cards[w];
392     }
393
394
395     rotate_right(cards[i]);
396     board->components[current_slot]=cards[i];
397     used_cards[i]=true;
398     ordre_cards[current_slot]=i+1;
399     rotations_cards[current_slot]=j+1;
400
401     if (bruteforce(conditions,board,cards,used_cards,indice_case_start,current_slot+1,
402         ordre_cards,rotations_cards,fichier)){
403         free(copy_ordre);
404         free(copy_rotation);
405         free(copy_tmp);
406         return true;
407     }
408     else{
409         // Reset la partie modifiee sur le board
410         for (int z =current_slot; z < board->len * board->len ; z++){
411             board->components[z]=NULL;
412         }
413         // Reset le tableau used_cards
414         for (int w = 0; w< board->len * board->len; w++){
415             used_cards[w] = copy_tmp[w] ;
416             rotations_cards[w]=copy_rotation[w];
417             ordre_cards[w]=copy_ordre[w];
418         }
419         free(copy_ordre);
420         free(copy_rotation);
421         free(copy_tmp);
422     }
423 }
424 }
425 return false;
426 }
427 }
428
429
430 bool bruteforce_opti(int** conditions, plateau* board, cell* cards,int taille_plateau, bool start_situation,
431     bool* used_cards, int* ordre_cards, int* rotations_cards, MaxPriorityQueue* file_prio, MaxPriorityQueue
432     * file_preservation,bool** cards_useful, FILE* fichier, int* nb_appels){
433
434     *nb_appels +=1;
435     printf("taille file_prio : %d | ", file_prio->size);
436     if (board->completed || file_prio->size <= 0){
437         // bool res = est_valide_board(board,conditions);
438         printf("\n test final \n");
439
440         if (est_valide_board(board,conditions)){
441             printf("SUCCESS %d ! \n", *nb_appels);
442
443             // Renvoyer le plateau dans un fichier de retour
444             affiche_case_fichier(ordre_cards,rotations_cards,board->len * board->len, fichier);
445
446             //Renvoyer un boolean pour le signaler aux appels d'au dessus
447             return true;
448         }
449         printf("echec");
450         return false;
451     }
452     else if (start_situation){

```



```

452     printf("start");
453     // Le plateau est vide
454
455     // On identifie la case du plateau avec le plus de priorites
456     Element case_start = extractMax(file_prio);
457     int indice_case_start = case_start.identifieur;
458     int prio_case_start = case_start.weight;
459
460     // On enfile la case et sa valeur de prio dans la file de preservation
461     // insert(file_preservation, prio_case_start, indice_case_start);
462
463
464     // On va essayer les cases utiles pour sa position
465
466     for (int id_card=0; id_card < taille_plateau; id_card++){
467         if (cards_useful[indice_case_start][id_card]){
468             // On a trouve un carte qui etait jugee comme utile pour cet emplacement et qui n'a pas ete
469             // utilisee avant sur celui-ci vu qu'on procede iterativement
470
471             for (int rot=0; rot<4; rot++){
472                 rotate_right(cards[id_card]);
473
474                 board->components[id_card] = cards[id_card];
475                 used_cards[id_card] = true;
476                 ordre_cards[indice_case_start]= id_card;
477                 rotations_cards[indice_case_start]=rot+1;
478                 if (bruteforce_opti(conditions,board,cards,taille_plateau,false,used_cards,ordre_cards,
479                     rotations_cards, file_prio, file_preservation, cards_useful,fichier,nb_appels)){
480                     return true;
481                 }
482             }
483             else{
484                 // On reinitialise toutes les donnees modifiees
485                 reset_board(board);
486                 reset_u_cards(used_cards,taille_plateau);
487                 reset_rotations_and_ordre(ordre_cards,taille_plateau);
488                 reset_rotations_and_ordre(rotations_cards, taille_plateau);
489
490                 // IL FAUT REMETTRE EN PLACE LA FILE BON DIEU
491                 transfer_files(file_prio,file_preservation);
492             }
493         }
494         // La carte a ete testee dans toutes les rotations
495     }
496     // On teste donc la carte utile suivante en bouclant
497 }
498
499 // On a teste toutes les cartes utiles pour cet emplacement et rien de concluant, cette situation
500 // est impossible mais je renvoie false au cas ou
501 return false;
502 }
503 else {
504     // Le plateau n'est ni en position de depart, ni complete
505     // On identifie la case a traiter en premier
506
507     Element current_case = extractMax(file_prio);
508     int indice_current_case= current_case.identifieur;
509     int prio_current_case= current_case.weight;
510     printf("current_case : %d \n", indice_current_case);
511
512
513     // On va essayer les cartes utiles pour cet emplacement
514

```

```

515     for (int id_card = 0; id_card < taille_plateau; id_card++){
516         if (cards_useful[indice_current_case][id_card] && !used_cards[id_card]){
517
518             // On a trouve une carte qui etait jugee comme utile pour cet emplacement et qui n'a pas
                    ete utilisee avant sur celui-ci vu qu'on procede iterativement et qui n'est pas
                    activement utilisee
519
520             for (int rot=0; rot<4; rot++){
521                 // On sauvegarde notre progression
522                 bool* copy_tmp = malloc( taille_plateau * sizeof(bool));
523                 int* copy_ordre = malloc(taille_plateau * sizeof(int));
524                 int* copy_rotation = malloc(taille_plateau * sizeof(int));
525                 for (int w = 0; w< taille_plateau; w++){
526                     copy_tmp[w] = used_cards[w];
527                     copy_ordre[w]=ordre_cards[w];
528                     copy_rotation[w]=rotations_cards[w];
529
530                 }
531                 rotate_right(cards[id_card]);
532                 board->components[id_card] = cards[id_card];
533
534                 used_cards[id_card] = true;
535
536                 ordre_cards[indice_current_case]= id_card;
537                 rotations_cards[indice_current_case] = rot+1;
538                 if (bruteforce_opti(conditions,board,cards,taille_plateau,false,used_cards,ordre_cards,
539                     rotations_cards, file_prio, file_preservation, cards_useful,fichier,nb_appels)){
540                     free(copy_ordre);
541                     free(copy_rotation);
542                     free(copy_tmp);
543                     return true;
544                 }
545                 else{
546                     // On remet en place la sauvegarde
547
548                     // ATTENTION : a MODIFIER -> il faut remettre en place la sauvegarde des cartes (i.e mettre a NULL les
                    modifs des appels d'en dessous)
549
550                     // Reset le tableau used_cards
551                     for (int w = 0; w< taille_plateau; w++){
552                         used_cards[w] = copy_tmp[w] ;
553                         rotations_cards[w]=copy_rotation[w];
554                         ordre_cards[w]=copy_ordre[w];
555                     }
556                     free(copy_ordre);
557                     free(copy_rotation);
558                     free(copy_tmp);
559
560                     for (int w = 0; w<taille_plateau; w++){
561                         if (!used_cards[w]){
562                             board->components[w]=NULL;
563                         }
564                     }
565
566                     // Il faut remettre la file en place
567                     // IL FAUT REMETTRE EN PLACE LA FILE BON DIEU
568                     transfer_files(file_prio,file_preservation);
569                 }
570             }
571             // La carte a ete testee dans toutes les rotations
572
573         }
574     }
575     // On enfile la case et sa valeur de prio dans la file de preservation
576     insert(file_preservation, prio_current_case, indice_current_case);

```

```

577
578     // On a teste toutes les cartes dispos sur cet emplacement et ... rien de concluant
579     return false;
580 }
581 }
582
583 void detruire_file_prio(MaxPriorityQueue* pq){
584     while (pq->size != 0){
585         Element tmp = extractMax(pq);
586         printf("element detruit : %d \n", tmp.identifieur);
587     }
588     free(pq);
589 }
590 /*
591     MAIN
592 */
593
594 int main(){
595
596     /*
597         INITIALISATION DU PLATEAU
598     */
599
600
601     // Initialisation des cases
602     cell p1 = malloc(4*sizeof(int*));
603     p1[0]=malloc(5*sizeof(int));
604     p1[1]=malloc(5*sizeof(int));
605     p1[2]=malloc(5*sizeof(int));
606     p1[3]=malloc(5*sizeof(int));
607     p1[0][0] = 2;
608     p1[0][1] = 1;
609     p1[0][2] = 2;
610     p1[1][0] = 2;
611     p1[1][1] = 0;
612     p1[1][2] = 2;
613     p1[2][0] = 2;
614     p1[2][1] = 0;
615     p1[2][2] = 1;
616     p1[3][0] = 0;
617
618
619
620     cell p2 = malloc(4*sizeof(int*));
621     p2[0]=malloc(5*sizeof(int));
622     p2[1]=malloc(5*sizeof(int));
623     p2[2]=malloc(5*sizeof(int));
624     p2[3]=malloc(5*sizeof(int));
625     for (int i=0;i<4;i++){
626         p2[i][0]=1;
627     }
628     p2[0][1] = 3;
629     p2[1][1] = 2;
630     p2[2][1] = 1;
631     p2[3][1] = 0;
632
633     cell p3 = malloc(4*sizeof(int*));
634     p3[0]=malloc(5*sizeof(int));
635     p3[1]=malloc(5*sizeof(int));
636     p3[2]=malloc(5*sizeof(int));
637     p3[3]=malloc(5*sizeof(int));
638     for (int i=0;i<4;i++){
639         p3[i][0]=1;
640     }
641     p3[0][1]=2;
642     p3[1][1]=3;

```

```

643 p3[2][1]=0;
644 p3[3][1]=1;
645
646
647 cell p4 = malloc(4*sizeof(int*));
648 p4[0]=malloc(5*sizeof(int));
649 p4[1]=malloc(5*sizeof(int));
650 p4[2]=malloc(5*sizeof(int));
651 p4[3]=malloc(5*sizeof(int));
652 p4[0][0] = 2;
653 p4[0][1] = 2;
654 p4[0][2] = 3;
655 p4[3][0] = 2;
656 p4[3][1] = 0;
657 p4[3][2] = 2;
658 p4[2][0] = 2;
659 p4[2][1] = 0;
660 p4[2][2] = 3;
661 p4[1][0] = 0;
662
663
664 cell p5 = malloc(4* sizeof(int*));
665 p5[0]=malloc(5*sizeof(int));
666 p5[1]=malloc(5*sizeof(int));
667 p5[2]=malloc(5*sizeof(int));
668 p5[3]=malloc(5*sizeof(int));
669 p5[0][0]=0;
670 p5[1][0]=1;
671 p5[1][1]=2;
672 p5[2][0]=1;
673 p5[2][1]=1;
674 p5[3][0]=0;
675
676 cell p6 = malloc(4* sizeof(int*));
677 p6[0]=malloc(5*sizeof(int));
678 p6[1]=malloc(5*sizeof(int));
679 p6[2]=malloc(5*sizeof(int));
680 p6[3]=malloc(5*sizeof(int));
681 p6[0][0]=3;
682 p6[0][1]=1;
683 p6[0][2]=2;
684 p6[0][3]=3;
685 p6[1][0]=3;
686 p6[1][1]=0;
687 p6[1][2]=2;
688 p6[1][3]=3;
689 p6[2][0]=3;
690 p6[2][1]=0;
691 p6[2][2]=1;
692 p6[2][3]=3;
693 p6[3][0]=3;
694 p6[3][1]=0;
695 p6[3][2]=1;
696 p6[3][3]=2;
697
698
699 cell p7 = malloc(4* sizeof(int*));
700 p7[0]=malloc(5*sizeof(int));
701 p7[1]=malloc(5*sizeof(int));
702 p7[2]=malloc(5*sizeof(int));
703 p7[3]=malloc(5*sizeof(int));
704 p7[0][0]=0;
705 p7[1][0]=1;
706 p7[1][1]=2;
707 p7[2][0]=1;
708 p7[2][1]=1;

```

```

709 p7[3][0]=0;
710
711 cell p8 = malloc(4* sizeof(int*));
712 p8[0]=malloc(5*sizeof(int));
713 p8[1]=malloc(5*sizeof(int));
714 p8[2]=malloc(5*sizeof(int));
715 p8[3]=malloc(5*sizeof(int));
716 p8[0][0]=1;
717 p8[0][1]=3;
718 p8[1][0]=0;
719 p8[2][0]=0;
720 p8[3][0]=1;
721 p8[3][1]=0;
722
723 cell p9 = malloc(4* sizeof(int*));
724 p9[0]=malloc(5*sizeof(int));
725 p9[1]=malloc(5*sizeof(int));
726 p9[2]=malloc(5*sizeof(int));
727 p9[3]=malloc(5*sizeof(int));
728 for (int i=0;i<4;i++){
729     p9[i][0]=1;
730 }
731 p9[0][1]=1;
732 p9[1][1]=0;
733 p9[2][1]=3;
734 p9[3][1]=2;
735
736
737
738
739
740 plateau* main_board = malloc(sizeof(plateau));
741 const int LEN = 3;
742 const int NB_CASES = 9;
743 main_board->len=LEN;
744 main_board->completed=false;
745 main_board->components= malloc(NB_CASES * sizeof(cell));
746 cell* compo = malloc(NB_CASES * sizeof(cell));
747 compo[0]=p1;
748 compo[1]=p2;
749 compo[2]=p3;
750 compo[3]=p4;
751 compo[4]=p5;
752 compo[5]=p6;
753 compo[6]=p7;
754 compo[7]=p8;
755 compo[8]=p9;
756
757 bool* used_cards = malloc(NB_CASES * sizeof(bool));
758
759 int* ordre = malloc(NB_CASES * sizeof(int));
760 int* rotation = malloc(NB_CASES * sizeof(int));
761 for (int i=0; i<NB_CASES;i++){
762     used_cards[i]=false;
763     ordre[i]=-1;
764     rotation[i]=-1;
765 }
766
767 // File de priorites
768
769 MaxPriorityQueue file_prio;
770 file_prio.size=0;
771 MaxPriorityQueue file_preservation;
772 file_preservation.size=0;
773 MaxPriorityQueue* file_prio_main = malloc(sizeof(MaxPriorityQueue));
774 MaxPriorityQueue* file_preservation_main = malloc(sizeof(MaxPriorityQueue));

```

```

775 *file_prio_main = file_prio;
776 *file_preservation_main= file_preservation;
777
778
779 // Cards useful
780
781
782
783 /*
784     Initialisation des conditions
785 */
786 int** conditions = malloc(6*sizeof(int*));
787 conditions[0]=malloc(sizeof(int));
788 conditions[1]=malloc(4*sizeof(int));
789 conditions[2]=malloc(4*sizeof(int));
790 conditions[3]=malloc(4*sizeof(int));
791
792
793
794
795 conditions[0][0]=3;
796
797 conditions[1][0]=5;
798 conditions[1][1]=1;
799 conditions[1][2]=1;
800 conditions[1][3]=0;
801
802 conditions[2][0]=1;
803 conditions[2][1]=0;
804 conditions[2][2]=0;
805 conditions[2][3]=3;
806
807 conditions[3][0]=0;
808 conditions[3][1]=8;
809 conditions[3][2]=3;
810 conditions[3][3]=1;
811
812 bool** cards_useful = malloc(NB_CASES * sizeof(bool*));
813 for (int i=0; i<NB_CASES; i++){
814     cards_useful[i]=malloc(NB_CASES * sizeof(bool));
815 }
816 trouve_cards_useful(conditions,NB_CASES,cards_useful,compo);
817
818 trouve_slots_prio(conditions,NB_CASES,file_prio_main);
819
820 /*
821     MISC.
822 */
823
824 //FILE* fichier = fopen("test.txt","w+");
825
826 clock_t t;
827
828 /*
829 if (bruteforce(conditions,main_board,compo,used_cards,0,0,ordre,rotation,fichier)){
830     printf("BRUTEFORCE -> OK \n");
831 }
832
833 bool* vus = malloc(NB_CASES *sizeof(bool));
834 for (int i=0;i<NB_CASES;i++){
835     vus[i]=false;
836 }
837
838 if (accessible(main_board,6,8,2,1,vus)){
839     printf("accessible");
840 }*/

```

```

841
842
843
844 printf(" \n Bruteforce opti : \n");
845 FILE* fichier2 = fopen("bruteforce_opti.txt","w+");
846 t = clock();
847 int* nb_appels = malloc(sizeof(int));
848 *nb_appels=1;
849
850 if (bruteforce_opti(conditions,main_board,compo,NB_CASES,true,used_cards,ordre,rotation,file_prio_main,
851     file_preservation_main,cards_useful,fichier2,nb_appels)){
852     printf("BRUTEFORCE OPTI -> OK \n");
853     t = clock() - t;
854     printf(" \n ==> Perf = %f seconds", ((float)t) / CLOCKS_PER_SEC);
855 }
856
857
858
859
860
861
862
863 /*
864     LIBERATION DE MEMOIRE
865 */
866
867 for (int i=0;i<4;i++){
868     free(p1[i]);
869     free(p2[i]);
870     free(p3[i]);
871     free(p4[i]);
872     free(p5[i]);
873     free(p6[i]);
874     free(p7[i]);
875     free(p8[i]);
876     free(p9[i]);
877 }
878
879 free(p1);
880 free(p2);
881 free(p3);
882 free(p4);
883 free(p5);
884 free(p6);
885 free(p7);
886 free(p8);
887 free(p9);
888
889 free(main_board->components);
890 free(main_board);
891
892 for (int i=0; i<NB_CASES; i++){
893     free(cards_useful[i]);
894 }
895 free(cards_useful);
896 detruire_file_prio(file_prio_main);
897 detruire_file_prio(file_preservation_main);
898
899
900
901
902
903 //fclose(fichier);
904 //fclose(fichier2);
905 }

```