05/2015

Project's report : Shannon game

HAVARD
Pierrick

RYMER
Julie

Contents

I - Instructions and first approach

A) Instructions

The instructions we received for the Shannon game were as follow :

" Soit un graphe, dont deux noeuds (D et A) sont particuliers et deux joueurs, le lieur et le casseur. À chaque tour de jeu, le casseur casse une arête entre deux noeuds. À chaque tour de jeu, le lieur rend incassable une arête (non cassée) entre deux noeuds. Si, à un moment du jeu, il n'y a plus aucun moyen d'aller de D à A, le casseur a gagné. Si, à un moment donné, il existe un chemin incassable reliant D à A, le lieur a gagné.

- Faites un programme qui permet à deux joueurs de jouer.
- Faites un programme qui permet de jouer contre la machine.

L'implémentation du ou des graphes devra être faite sous la forme soit de listes chaînées soit de tableaux d'arêtes et de brins (voir votre enseignant). "

B) The choice of the structure and the first try

At first when we where wondering what structure to use, we thought that the best to do would be to use linked lists. Each element, representing a point, would contain a number to identify it and an array containing the addresses of the other elements it could go to. Then we thought we would try to do that with dynamics arrays. We started by defining the structure, then by defining a function and a main that used it to try to see if we would manage to use it, all these that you can see below.

```
struct point{
  int num;
  struct tab * tab;
};
typedef struct point point;
struct tab{
  int n;
  point**t;
};
typedef struct tab * tab;

int effacer (point a, point b){
  int i,j;
  tab temp=NULL;
  for (i=0;i<a.tab->n;i++){
    if(*(a.tab->t+i)==&b){
      temp->n=(a.tab->n)-1;
      temp->t=(point**)malloc((size_t) (temp->n * sizeof(point*)));
      for(j=0;j<i;j++)
        *(temp->t+j)=*(a.tab->t+j);
      for(j=i+1;j<a.tab->n;j++){
        *(temp->t+i)=*(a.tab->t+j);
        i++;
      }
```

```
      a.tab=temp;
      return 1;
    }
  }
  return 0;
}
// Erase the connection from point A to point B (unilaterally), and re-
turn 1 if it worked.
int erase (point a, point b){
  return effacer(a,b) & effacer(b,a);
}
// same thing but bilaterally

int main(){
  point* a=(point*)malloc((size_t) sizeof(point));
  point* b=(point*)malloc((size_t) sizeof(point));
  point* c=(point*)malloc((size_t) sizeof(point));
  point* d=(point*)malloc((size_t) sizeof(point));
  a->num=1;
  b->num=2;
  c->num=3;
  d->num=4;
  a->tab->n=3;
  *(a->tab->t)=b;
  *(a->tab->t+1)=c;
  *(a->tab->t+2)=d;
  b->tab->n=2;
  *(b->tab->t)=a;
  *(b->tab->t+1)=c;
  c->tab->n=2;
  *(c->tab->t)=a;
  *(c->tab->t+1)=b;
  d->tab->n=1;
  *(d->tab->t)=a;
  printf("%d\n",erase(*a,*b));
  printf("%d\n",erase(*a,*b));
}
```

First structure tested

Even after several corrections, we never managed to obtain anything else than a "segmentation fault 11" so we finally abandonned the idea, thinking we still didn't have the knowledge necessary to use correctly pointers of pointers, never having used them before, much less write an entire game with them.

We changed the structures so that the arrays would be static ones, much more easy for us. After reflecting about how to represent the fact that a connection between two points would be unbreakable, we finally settled for a final structure "point" which would contain, beside its identifiying number, two lists of other points' addresses : one with the vulnerable links, and another with the unbreakable ones, this way they easily wouldn't be trifled with by the cutting move. You can see the final structure on the next page.

C) Planning

Anticipating that the entire program would be really long and need lots of tests, we opted to use separate files for each part and a Makefile to regroup them all. Then, for a better organization, we decided to make a planning of all the main functions we would need to write and in what file they would be. This text file was updated regularly when an advancement was made, either by adding a file or function, or by refining the description of the functions we had already planned to make. Here is the final entire plan of the Shannon program :

```
typedef struct point* array[100];
struct tab{
  int n;
  array t;
};
typedef struct tab tab;
struct point{
  int num;
  tab link;
  tab locked;
};
typedef struct point point;
```

Final structure used in the program

I- shannon.h central library.

II- point.c : regroup all basic operations on a point.

- **display1:** display a point,
- **display:** display all points in a level(=array of points),
- **arelinked:** check to see if two points are linked together,
- **erase:** erase links between two points,
- **arelocked:** check to see if two points are locked together,
- **lock:** lock links between two points.

III- level.c : functions to create a level,

- **create:** we give the number of points, it returns an array of points (the first will be the starting point to be linked to the last in the game),
- **link:** create possible links between points,
- **level(1,2,3):** three functions to create three possible levels (return an array of points).

IV- player.c : moves of a player (work with an array of points, need scanf to interact).

- **join:** make the join move, need to check if the link isn't already locked.
- **cut:** make the cut move, need to check if link exist.

V- ai.c : moves of the artificial intelligence (work with an array of points).

- **aijoin**: computer's join move, always win if computer is in advantageous position,
- **aicut**: computer's cut move, always win if computer is in advantageous position,
- **aijoin2**: computer's alternative join move, making the best possible move if in a disadvantageous position,
- **aicut2**: computer's alternative cut move, making the best possible move if in disadvantageous position.

VI- aitest.c : all the tests used for the ai moves.

- **advantagejoin**: test if level is advantageous for player Join,
- **neutral**: test if level is advantageous for the first player,
- **advantagecut**: test if level is advantageous for player Cut,

VII- ending.c :

- **winjoin**: check to see if the game is finished, return false if it isn't, true if Join won,
- **wincut**: check to see if the game is finished, return false if it isn't, true if Cut won,
- **announce**: with one argument, announce winner (Join or Cut if pvp, Player or Computer if pve).

VIII-start.c : use scanf, start the game with player choices.

- **choice**: ask the player to choose between the levels, or reading the rules or quitting, and load the level if choosen,
- **choice2**: ask the player what mode they want to play (pvp or pvai), if pvai ask player to choose between being join or cut,
- **pvp**: launch join and cut intermittently until end is true, and announcement is made.
- **pvai**: launch join and aicut or aijoin and cut until end is true, and announcement is made.
- **rules**: display the rules

IX- main.c : launch start.

Final planning of the entire program

II - All basic operations on a point

The functions talked about in this section are all in the file “point.c”. For references, please see their code detailed on pages 28-29.

A) Display

The first function we did was **display**. It was an easy one that permitted us to familiarise ourselves with the structure we had chosen. The function **display** is used to visualize the graph we would be playing on. For every point in it, it tells the number of that point, the numbers of each point it is linked to and the numbers of each point it has a locked link with. For commodity, we first did a function **display1** that acted on a single point. It simply finds the point’s number and display it, then navigate throught the two lists of the point and indicate the number of every point present, and to what list they belong.

We then had to reflect on what form would take a graph that would represent a level in the game, to be able to display it and keep it in a single variable. We realised that our stucture tab that we already use inside the stucture point to list its links, could also be reused for that purpose ! It is after all an array of addresses of points, so it was perfect to stock all the points inside a level. So we simply finished by making the function **display** that called **display1** on every point in a level.

• *Traces of the program execution :*

```
point a, b, c, d, e, f;
tab level;
a.num=1; b.num=2; c.num=3; d.num=4;

a.link.n=2; a.link.t[0]=&b; a.link.t[1]=&c;
a.locked.n=1; a.locked.t[0]=&d;

b.link.n=2; b.link.t[0]=&a; b.link.t[1]=&c;
b.locked.n=0;

c.link.n=2; c.link.t[0]=&a; c.link.t[1]=&b;
c.locked.n=0;

d.link.n=0; d.locked.n=1; d.locked.t[0]=&a;

level.n=4; level.t[0]=&a; level.t[1]=&b;
level.t[2]=&c; level.t[3]=&d;

printf(“\n The points of Level Test are :\n”);
display(level);
```

```
 The points of Level Test are :
Point 1 is linked to :   2   3
and has a locked link with :   4
Point 2 is linked to :   1   3
and has a locked link with :
Point 3 is linked to :   1   2
and has a locked link with :
Point 4 is linked to :
and has a locked link with :   1
```

An extract of a main function to test display and its result

B) Erase and lock

After that we needed a function that could delete the connection between two points, so that the player Cut could make their move. We started with the function **erase1** which take two points as arguments and delete the connexion only unilaterally. Since each point has their own specific number, we could use that to identify it with a simple "==" instead of creating a more complex function to compare two points like we have thought to do first. This function find the place of the second point's address in the list of those linked to the first. It then shift back all the addresses that are after it and reduce of one the lenght of the list so that the second point is no longer accessible from the first. At first we used a temporary point to do that, but we later found out it wasn't necessary.

The **erase** function takes three arguments. At first those where a tab and two points. But later we discovered that it wasn't practical to have to call it with points, mainly because the player would enter numbers, and to convert them in the appropriate points would mean to navigate throught the level, and we already need to navigate throught it in **erase**. So to check the level only once we modifyed the function to take a tab and two int as arguments instead. It simply goes throught the level to find where are the two points we need to modify and then call **erase1** on both and return the resulting level.

Similarly, to enable the player Join to make their move, we needed a function that lock the connection between two points, which meant erasing their basic connection and putting the address of each in the other's locked links list. At first the **lock** function called upon **erase** before navigating the level to make its own operation on the two points. We later changed that so that it navigated first, then called twice upose **erase1**, similarly to **erase**, simply, once again, to avoid looking throught the level twice. It then add one to the number of locked links of each points and add the address of the other at the end of it. We had some problem with this function when we still used points as arguments, having a memory address instead of the point's numbers displayed in the new locked lists. The problem has been solved with the use of int and working directly on the level.

Finally, we made **arelinked** and **arelocked**, to check the level to see if two points are already, correspondingly, linked or lock-linked. Since every connection is two-sided, we only need to check it for one point.

• *Traces of the program execution :*

```
printf("\n The points of Level Test are
:\n");
display(level);
printf("\n After deleting the connexion
between point 1 and 2, and after locking
the connexion between point 3 and 2, the
level become :\n");
level=erase(level,1,2);
level=lock(level,3,2);
display(level);
printf("\n Some tests :");
if(arelinked(level,3,1)) printf("\nPoint
1 and 3 are linked together");
if(!arelinked(level,2,1)) printf("\nPoint
1 and 2 aren't linked together");
if(arelocked(level,4,1)) printf("\nPoint
1 and 4 are locked together");
if(!arelocked(level,2,1)) printf("\nPoint
and 2 aren't locked together\n");
```

```
 After deleting the connexion between
point 1 and 2, and after locking the
connexion between point 3 and 2, the
level become :
Point 1 is linked to :   3
and has a locked link with :   4
Point 2 is linked to :
and has a locked link with :   3
Point 3 is linked to :   1
and has a locked link with :   2
Point 4 is linked to :
and has a locked link with :   1

 Some tests :
Point 1 and 3 are linked together
Point 1 and 2 aren't linked together
Point 1 and 4 are locked together
Point 1 and 2 aren't locked together
```

An extract of a main function to test erase and lock, and its result

III - The construction of a level

The functions talked about in this section are all in the file “level.c”. For references, please see their code detailed on pages 30-32.

A) Create

To be able to easily create a level, the first function we needed was one which we gave a number to and that would return a level of that number of points, each point getting an identifying number in increasing order and without a link between any point yet. We ran into trouble with this one, the first few functions we created to that end making us get either a “bus error 10” or a “segmentation fault: 11”. You can see one of those first tries on the right. We resolved this once we had the idea to work with a point’s pointer first instead of directly filling the tab. Doing that, **create** worked without further problem. It declares a point’s pointer, attribute a memory space to it, give it its caracteristics (number and empty lists) end only then fill the level we want created with that pointer. And it does that of course as many time as the number of points we want in the level.

```
tab create(int n){
     int i;
     tab level;
     level.n=n;
     for(i=0;i<n;i++){
          level.t[i]->num=i+1;
          level.t[i]->link.n=0;
          level.t[i]->locked.n=0;
     }
     return level;
}
// Create a level (aka an array of
points) of n points
```

One of the incorrect function we tried

• *Traces of the program execution :*

```
printf(“\n A Level
with 5 points is
:\n”);
level0=create(5);
display(level0);
```

```
 A Level with 5 points is :
Point 1 is linked to :          and has a locked link with :
Point 2 is linked to :          and has a locked link with :
Point 3 is linked to :          and has a locked link with :
Point 4 is linked to :          and has a locked link with :
Point 5 is linked to :          and has a locked link with :
```

An extract of a main function to test create, and its result

B) Levels

Another function we needed to easily create a level was **link**, that would, as its name indicate, add a link between two points. This function work exactly the same as **lock**, except it has no need to call on **erase**. We later tweaked it a little so that the link would be added at the beginning of the list instead of simply at the end. It has no incidence on the creation of a level, except that for the player reading comfort the links would need to be added by decreasing order. The reason we did that will be explained in the chapter on the Arificial Intelligence, since it was needed for it. After that the only thing left to do was to write the functions that would create the levels we would play on themselves. It was only a matter of, for each level we wanted, indicate the number of point and use **create** with it, then use **link** one by one for each connection we wanted to make. The first level is simply a square of four points with another point in the middle. For the level two and three we took our inspiration from the internet to create them.[1]

1 http://www.physics.ox.ac.uk/users/iontrap/ams/graph_theory/graph.htm (consulted on 04/28/15).

• *Traces of the program execution :*

```
levelone=level1();
printf("\n The
points of Level 1
are :\n");
display(levelone);
```

```
 The points of Level 1 are :
Point 1 is linked to :   2   3   4              and has a locked link with :
Point 2 is linked to :   1   3   5              and has a locked link with :
Point 3 is linked to :   1   2   4   5          and has a locked link with :
Point 4 is linked to :   1   3   5              and has a locked link with :
Point 5 is linked to :   2   3   4              and has a locked link with :
```

An extract of a main function to test level1, and its result

```
leveltwo=level2();
printf("\n The
points of Level 2
are :\n");
display(leveltwo);
```

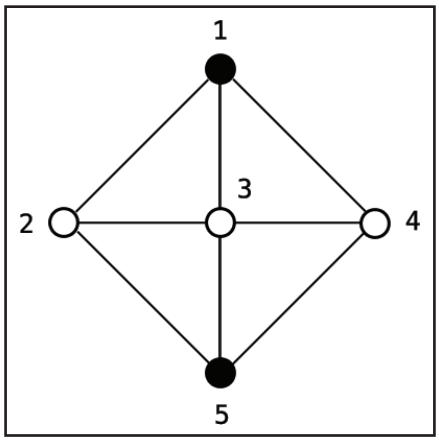
```
 The points of Level 2 are :
Point 1 is linked to :   2   3          and has a locked link with :
Point 2 is linked to :   1   4   5              and has a locked link with :
Point 3 is linked to :   1   7   8              and has a locked link with :
Point 4 is linked to :   2   6   9              and has a locked link with :
Point 5 is linked to :   2   6   9              and has a locked link with :
Point 6 is linked to :   4   5   7   8          and has a locked link with :
Point 7 is linked to :   3   6  10              and has a locked link with :
Point 8 is linked to :   3   6  10              and has a locked link with :
Point 9 is linked to :   4   5  11              and has a locked link with :
Point 10 is linked to :   7   8  11             and has a locked link with :
Point 11 is linked to :   9  10          and has a locked link with :
```

An extract of a main function to test level2, and its result

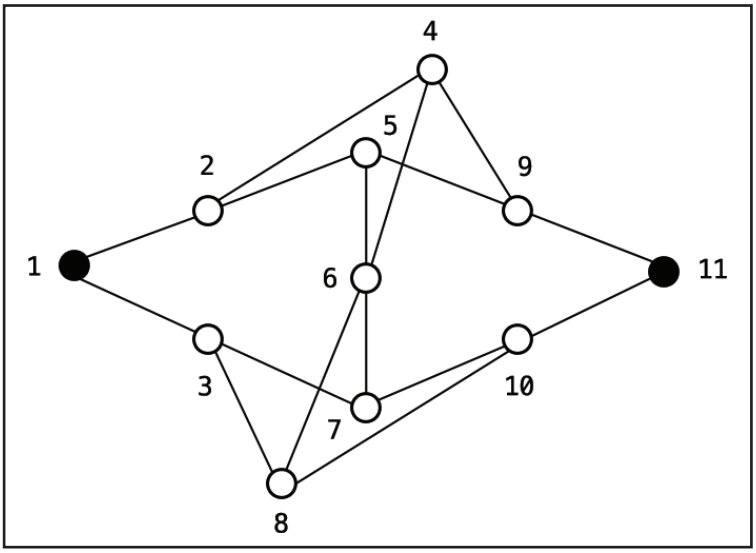
```
levelthree=level3();
printf("\n The
points of Level 3
are :\n");
display(levelthree);
```

```
 The points of Level 3 are :
Point 1 is linked to :   3   6          and has a locked link with :
Point 2 is linked to :   4   5   8              and has a locked link with :
Point 3 is linked to :   1   4   7              and has a locked link with :
Point 4 is linked to :   2   3  10              and has a locked link with :
Point 5 is linked to :   2   6  10              and has a locked link with :
Point 6 is linked to :   1   5   9              and has a locked link with :
Point 7 is linked to :   3  11          and has a locked link with :
Point 8 is linked to :   2  10  11              and has a locked link with :
Point 9 is linked to :   6  11          and has a locked link with :
Point 10 is linked to :   4   5   8             and has a locked link with :
Point 11 is linked to :   7   8   9             and has a locked link with :
```

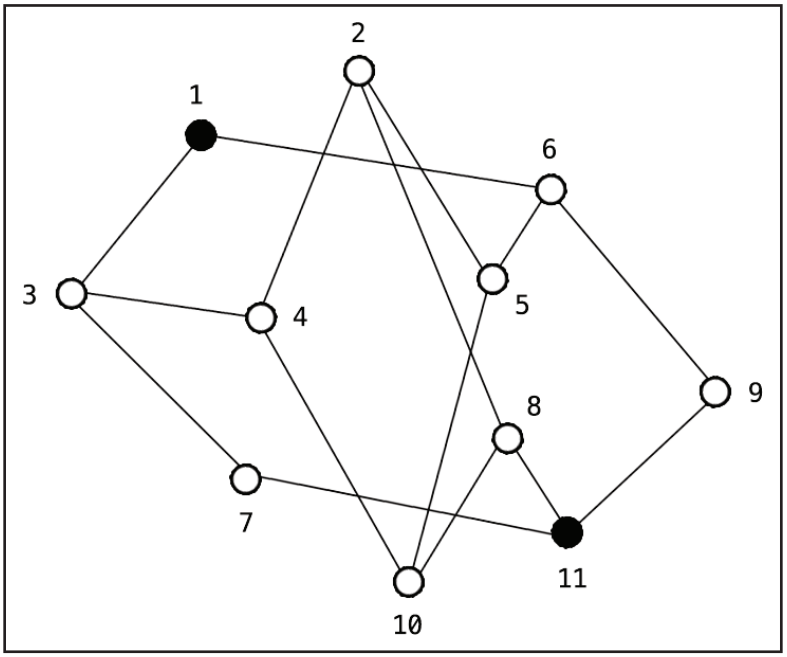
An extract of a main function to test level3, and its result



A visual of the Level 1



A visual of the Level 2



A visual of the Level 3

C) Random level generator

As a final touch for the levels, we wrote a random level generator. After educating ourselves abour the **rand** function in C[1], we started. We wrote the function **levelrandom** that create a level of a random number of points between 5 and 15. To be sure that each point of the level would have at least one link, in a for loop, **levelrandom** adds one link for each point. Then for each point, there is a one chance out of four for it to have another link added, to be able to have a random number of links. The last thing being done by **levelrandom** is to add a link to the first point and to the last point, if one of those has only one link, to prevent the player Cut to win on their very first move in case they start. The number of links a point has is found by another simple function we made called **numberlinks**.

All of these links would be added with **randomlink**, a function that add the link between a point and another chosen at random only if there is no previous link between those and if they're not the same. If one of this condition is met, the function calls itself recursively to try another linking with the number in the original arguments. This function also tests to see if the point we are trying to add a link to isn't already linked to all the other points, which should be rare but could happen. In this case the recurssive call is done with a new random number instead of the original argument.

For a reason we didn't have time to find an explanation to, our random level generator sometimes provoke a "segmentation fault: 11". It only happens very occasionnaly but if we are to continue working on this game we will have to find and repair the error causing this. Furthermore the resulting level can sometimes be very desequilibred so additionnal tweaking to the adding of links would need to be done for a better performing random level generator.

• *Traces of the program execution :*

```
levelthree=levelrandom();
printf("\n The points of a ran-
domly generated level are :\n");
display(levelthree);
```

```
 The points of a randomly generated level are :
Point 1 is linked to :   8   7   5   4          and has a locked link with :
Point 2 is linked to :   6   4   7         and has a locked link with :
Point 3 is linked to :   8   4   7         and has a locked link with :
Point 4 is linked to :   5   2   3   1          and has a locked link with :
Point 5 is linked to :   1   6   4         and has a locked link with :
Point 6 is linked to :   2   5        and has a locked link with :
Point 7 is linked to :   1   3   2         and has a locked link with :
Point 8 is linked to :   1   3        and has a locked link with :
```

An extract of a main function to test levelrandom, and one of its result

```
Process 15007 stopped
* thread #1: tid = 0xf75bb, 0x00000001000047b5 shannon`randomlink + 21, queue = 'com.
apple.main-thread', stop reason = EXC_BAD_ACCESS (code=2, address=0x7fff5f3ffe58)
    frame #0: 0x00000001000047b5 shannon`randomlink + 21
shannon`randomlink + 21:
-> 0x1000047b5:  movq   %rax, -0x338(%rbp)
   0x1000047bc:  movq   %rcx, -0x340(%rbp)
   0x1000047c3:  movq   %rdi, -0x348(%rbp)
   0x1000047ca:  callq  0x1000071ea               ; symbol stub for: rand
```

The lldb report on a try where there was a SegFault

1 http://openclassrooms.com/courses/l-aleatoire-en-c-et-c-se-servir-de-rand-1 (consulted on 05/06/2015).

IV - The player's moves

The functions talked about in this section are all in the file "player.c". For references, please see their code detailed on page 33.

A) Join and cut

Like we anticipated, these two interactive functions **join** and **cut** needed the use of **scanf**. Therefore, the first thing we did was to do some research about how to use it. After browsing through some tutorials, notably those of the Open Classroom website[1], we were nearly ready. To prevent errors we also needed a way to empty the buffer after a call to scanf. After a quick search we found one of the way to do that[2] and for a quick access we put a define in our library file : "#define emptybuffer while (x != '\n' && x != EOF) x = getchar()". This line suppose that x is already defined as an "int x=0" in the function where we use **emptybuffer**, so to avoid any risk because of that, we later replaced it by "#define emptybuffer scanf ("%*[^\n]"); getchar ();" which has the same end result.

After that **join** and **cut** themselves weren't hard to write, we just needed not to forget about using some tests to restart the function for each situation where the player input would be unwanted (invalid entry, points already locked together or not having any link between them). For that we used **arelinked** and **arelocked** which we already wrote. The only thing left to do was to write the prompt specific to each function and to call **lock** and **erase** respectively with the player input.

• *Traces of the program execution :*

```
display(levelone);
join(levelone);
cut(levelone);
display(levelone);
```

```
Point 1 is linked to :   2   3   4             and has a locked link with :
Point 2 is linked to :   1   3   5             and has a locked link with :
Point 3 is linked to :   1   2   4   5         and has a locked link with :
Point 4 is linked to :   1   3   5             and has a locked link with :
Point 5 is linked to :   2   3   4             and has a locked link with :


Please enter the numbers of the two points you wish to lock together : 1 3


Please enter the numbers of the two points between which you want to erase
the connection : 3 5
Point 1 is linked to :   2   4          and has a locked link with :   3
Point 2 is linked to :   1   3   5              and has a locked link with :
Point 3 is linked to :   2   4          and has a locked link with :   1
Point 4 is linked to :   1   3   5              and has a locked link with :
Point 5 is linked to :   2   4          and has a locked link with :
```

An extract of a main function to test join and cut, and its result

1 http://openclassrooms.com/courses/la-saisie-securisee-avec-scanf (consulted on 05/02/2015).
2 http://openclassrooms.com/forum/sujet/vider-le-buffer-14371 (consulted on 05/02/2015).

V - The ending of a game

The functions talked about in this section are all in the file “ending.c”. For references, please see their code detailed on pages 34-35.

A) Winjoin and wincut

The function **winjoin1** is one that take two points A and B as arguments (which will be the first and last point of a level) and try to see if there is a way to go from A to B while using only the locked links. For that it calls itself recursively on each point that is lock-linked to A until B is reached (it then return true) or until all attainable points have been reached (return false). Since the links are two-sided, we needed a way to know if a point had already been verified or not.

In this, the two of us weren’t in agreement over what we wanted to use. One of us wanted to add a boolean “checked” to the point structure so that we would change its value once a point had been verified. The other wanted to create a function-local list of the numbers of the points already checked, and to create a **belong** function to easily run through that list and see if the number we want is in it. Each method had its own pros and cons. While the boolean in the structure would take a little more constant memory for each point and constrain us to have another thing to set when creating a custom level with **create**, the list would take more time to be checked and take more temporary memory while used. We finally settled to use the list if only not to have to uncheck as well as check the points. Therefore we wrote that **belong** function and also a function **winjoin** to be able to call **winjoin1** with an empty list. But we still did a try with the booleans that work and that you can see right below.

```
bool victoireBetonneur(point* depart, point* arrivee){
     int i;
     depart->checked=true;
     if(depart==arrivee){
          depart->checked=false;
          return true;
     }
     for(i=0;i<depart->locked.n;i++){
          if(depart->locked.t[i]->checked==false)
               if(victoireBetonneur(depart->locked.t[i], arrivee)){
                    depart->checked=false;
                    return true;
               };
     }
     depart->checked=false;
     return false ;
}
```

The function winjoin written to use a check boolean instead of a list

The function **wincut** works exactly in the same way as **winjoin**. the only differences are that it also run throught the links that are not locked, and, of course, that its conclusions are reversed. Since we want to know if it is impossible to go from point A to point B, only if all attainable points have been reached without ever crossing B can the function return true. But except for that, it was witten exactly in the same way as the previous function.

After that we just needed to write the function **announce** that would be called depending on the results obtained by **wincut** or **winjoin** and that would simply announce the winner of a game, weither a player or the computer. We could have added a line “exit(0);” at the end but we choose to add a “return;” to the void function that would call it instead.

• *Traces of the program execution :*

```
display(level);
if(!winjoin(a,c)) printf(“\nPlayer Join hasn’t yet
won if point 1 and 3 are the start and finish”);
if(!wincut(a,b)) printf(“\nPlayer Cut hasn’t yet
won if point 1 and 2 are the start and finish”);
if(winjoin(a,d)) printf(“\nPlayer Join has won if
point 1 and 4 are the start and finish\n\n”);
erase(level,4,2);
erase(level,3,1);
display(level);
if(wincut(a,b)) printf(“\nPlayer Cut has won if
point 1 and 2 are the start and finish\n”);
```

```
Point 1 is linked to :   3      and has a locked link with :   4
Point 2 is linked to :          and has a locked link with :   3
Point 3 is linked to :   1      and has a locked link with :   2
Point 4 is linked to :          and has a locked link with :   1

Player Join hasn’t yet won if point 1 and 3 are the start and finish
Player Cut hasn’t yet won if point 1 and 2 are the start and finish
Player Join has won if point 1 and 4 are the start and finish

Point 1 is linked to :          and has a locked link with :   4
Point 2 is linked to :          and has a locked link with :   3
Point 3 is linked to :          and has a locked link with :   2
Point 4 is linked to :          and has a locked link with :   1

Player Cut has won if point 1 and 2 are the start and finish
```

An extract of a main function to test winjoin and wincut, and its result

VI - The first AI

The functions talked about in this section are all in the files “aitest.c” and “ai.c”. For references, please see their code detailed on pages 36-38.

A) The tests to know the type of a game

We wanted to do an AI that would win absolutely each time it could. for that we did a little research. Unfortunately, most of the articles we found were hightly mathematical ones using edge disjoint spanning trees and were really difficult to understand, and even more difficult to make a function out of[1][2]. Then in *Switching the Shannon Switching Game* by Kimberly Wood[3] we found a text that was much more accessible to us. It describe in a simple way the “Disjoint Spanning Tree Theorem”, the same solution we saw before for player Join to play perfectly, but we still found it too difficult to make a function with that algorithm. But there were other interesting informations about the game. For one thing she describes the three possible tyme of game we can have in a Shannon’s game as follow : a positive game is a game where player Join can win even if player Cut has the first move ; a negative game is a game where player Cut can win even if player Join has the first move ; finally, a neutral game (which is non-negative and non-positive at the same time) is a game where the player that start should win.

But the description that interested us the most is a recursive definition of positive and non-negative games. In this definition, the game used is a little different from our own because the player Cut can play a n number of times, but by changing this n by 1 we can apply the definition to us. A game is represented by the notation (G,u,v) with G being the graph, u the first point and v the last and if a point is equal to another it means that they have locked links (here called edges) connecting them. The notation (G-{e_1,e_2…e_n}) means that the elements between the braces are edges deleted from the graph. Finally, the notation G/e means that the edge e has been locked. Knowing that we are able to understand the following definition :

“Let (G,u,v) be a game. If u=v then the game is always positive and
non-negative.
If u≠v then we define positive and non-negative games recursively as follows:

• (G,u,v) is a positive game if G has at least n edges and (G-{e1,e2…en},u,v) is a non-negative game for all distinct edges e1,e2…en belonging to E(G).

• (G,u,v) is a non-negative game if there exists an edge e belonging to E(G) such that the graph(G=e,u,v) is a positive game

Note that if G has zero edges and u≠v then the game is neither positive nor non-negative.”[4]

---

1 http://www.cs.yale.edu/homes/srivaths-arvind/networks/tarjan.pdf (consulted on 05/07/2015).

2 http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15859-f01/www/notes/shannon.pdf (consulted on 07/02/2015).

3 http://math.bard.edu/student/pdfs/Kimberly-Wood.pdf (consulted on 05/07/2015).

4 *Switching the Shannon Switching Game* by Kimberly Wood (page 22).

This definition gave us a solution to make a function to test to what type a game belong, weither positive, neutral or negative. We first did the function **advantagejoin** to test if a game is positive. It returns true only if **winjoin** is also true or if, no matter the link deleted, the resulting level is neutral. Which can be translated as, no matter what move the player Cut make, the player Join will take the advantage by playing right after. We then did the function **neutral** to test if a game is neutral, like its name indicate. It return true if there exists a link that, if locked, make the game become positive. We could also have tested to see if there exists a link that, if deleted, make the game become negative, the end result would have been the same, the point being to see if the first player take the advantage.

Finally we did the function **advantagecut** to test if a game is negative. At first we did so it tests if the level isn't a positive one then tests if there is no link that, after being locked, make the game become positive. But then we found out that it was equivalent to test if the game isn't positive then if it isn't neutral and we could just as simply have wrote "return !(advantagejoin(level) || neutral(level));", but that it wasn't the most efficient way to do it. We rewrote **advantagecut** so it worked the same as **advantagejoin**, only testing to see if no matter the player Join move, the result would be a neutral level. This method, while still needing a significant number of recursive call, would still take less of them than the previous. But for a reason we didn't find, that function, while being nearly exactly the same as **advantagejoin** which work, didn't get us the result we wanted. So we simply used the double call to **advantagejoin** then to **neutral**, because of a lack of time to correct it. You can see below our try of a better solution.

We also wrote the **unlock** function to be able to undo the locking the **neutral** function in its tests, not to have our level modified by it. It simply erase the locked link and replace it by a normal link.

```
bool advantagecut1(tab level, int n, int list[100]){
     int i,j,x,y;
     point a,b;
     tab leveltest;
     a=*level.t[0];
     b=*level.t[level.n-1];
     if(wincut(a,b)) return true;
     if(winjoin(a,b)) return false;
     for(i=0;i<level.n;i++){
          for(j=0;j<level.t[i]->link.n;j++){
               x=level.t[i]->num;
               y=level.t[i]->link.t[j]->num;
               if(!belong(y,n,list)){
                    leveltest=lock(level,x,y);
                    if(!neutral(leveltest)){
                         leveltest=unlock(level,x,y);
                         return false;
                    }
                    leveltest=unlock(level,x,y);
               }
          }
          n++;
          list[n-1]=x;
     }
     return true;
}
```

The fonction advantagecut we tried to do which take less recursive call but which doesn't work for a yet unknown reason

• *Traces of the program execution :*

```
display(level0);
if(advantagejoin(level0)) printf("The Level 0 is advantageous to player Join.");
else advantagecut(level0) ? printf("The Level 0 is advantageous to player Cut.") :
printf("The Level 0 is advantageous to the first player to play.");
level0.n+=1;
level0.t[level0.n-1]=&e;
e.num=6;
e.link.n=0;
e.locked.n=0;
level0=link(level0,5,6);
printf("\n Level 0 :\n");
display(level0);
if(advantagejoin(level0)) printf("The Level 0 is advantageous to player Join.");
else advantagecut(level0) ? printf("The Level 0 is advantageous to player Cut.") :
printf("The Level 0 is advantageous to the first player to play.");
level0.n+=1;
level0.t[level0.n-1]=&f;
f.num=7;
f.link.n=0;
f.locked.n=0;
level0=link(level0,6,7);
printf("\n Level 0 :\n");
display(level0);
if(advantagejoin(level0)) printf("The Level 0 is advantageous to player Join.\n");
else advantagecut(level0) ? printf("The Level 0 is advantageous to player Cut.\n")
: printf("The Level 0 is advantageous to the first player to play.\n");
```

```
 Level 0 :
Point 1 is linked to :   4   3   2             and has a locked link with :
Point 2 is linked to :   5   3   1             and has a locked link with :
Point 3 is linked to :   5   4   2   1         and has a locked link with :
Point 4 is linked to :   5   3   1             and has a locked link with :
Point 5 is linked to :   4   3   2             and has a locked link with :
The Level 0 is advantageous to player Join.
 Level 0 :
Point 1 is linked to :   2   3   4             and has a locked link with :
Point 2 is linked to :   1   5   3             and has a locked link with :
Point 3 is linked to :   4   1   5   2         and has a locked link with :
Point 4 is linked to :   5   3   1             and has a locked link with :
Point 5 is linked to :   6   4   3   2         and has a locked link with :
Point 6 is linked to :   5              and has a locked link with :
The Level 0 is advantageous to player Cut.
 Level 0 :
Point 1 is linked to :   3   4   2             and has a locked link with :
Point 2 is linked to :   1   5   3             and has a locked link with :
Point 3 is linked to :   1   4   5   2         and has a locked link with :
Point 4 is linked to :   5   1   3             and has a locked link with :
Point 5 is linked to :   6   4   3   2         and has a locked link with :
Point 6 is linked to :   7   5          and has a locked link with :
Point 7 is linked to :   6              and has a locked link with :
The Level 0 is advantageous to the first player to play.
```

An extract of a main function to test aitest.c, and its result

B) AIjoin and AIcut

To have an AI that would be able to play perfectly, we used the previous functions. **aijoin** tries to lock an existing link and only if the level is positive after locking it does **aijoin** validate the move and do it. **aicut** works similarly, trying to delete a link, and only if deleting it make the level be negative does it returns the modified level.

The really big problem with these two functions is that, while it enable the computer to win every time it find itself in a situation where it can, it multiplies the number of recursive calls, already really significant in **advantagecut** and **advantagejoin**. Since these two last functions take time to execute and you have to call them several times, **aijoin** and **aicut** take a really long time to take effect. If the link having to be modified to be in a good position is one of the first, there is no problem, but if it is one of the last, depending on the number of links there are in the level, **aicut** or **aijoin** can takes several minutes to play, in some tests we did it never finished even after half an hour. So of course it isn't usable in these cases and the first problem to correct in this program would be to find a way to make it quicker. One thing to do would be to modify **advantagecut** so as not to have to call both **neutral** and **advantagejoin** like we wanted to do. Another thing could be to finally use a boolean "checked" in the point stucture instead of a checklist of point since it takes longer.

A second problem is that **aicut** and **aijoin** are only able to work if the computer is in a good position. If it isn't they make no move. So we did a second AI to be able to play the most cleverly possible when the computer isn't at its advantage. And we added a line in **aijoin** and **aicut** to call its corresponding function in the second AI if that was the case.

• *Traces of the program execution :*

```
level=level1();
display(level);
printf("AI is Joining :\n");
aijoin(level);
display(level);
```

```
Point 1 is linked to :   2   3   4              and has a locked link with :
Point 2 is linked to :   1   3   5              and has a locked link with :
Point 3 is linked to :   1   2   4   5          and has a locked link with :
Point 4 is linked to :   1   3   5              and has a locked link with :
Point 5 is linked to :   2   3   4              and has a locked link with :
AI is Joining :
Point 1 is linked to :   3   2          and has a locked link with :   4
Point 2 is linked to :   1   5   3              and has a locked link with :
Point 3 is linked to :   1   4   5   2          and has a locked link with :
Point 4 is linked to :   5   3          and has a locked link with :   1
Point 5 is linked to :   4   3   2              and has a locked link with :
```

An extract of a main function to test aijoin, and its result

```
level=level3();
display(level);
printf("AI is cutting :\n");
aicut(level);
display(level);
```

```
Point 1 is linked to :   3   6          and has a locked link with :
Point 2 is linked to :   4   5   8              and has a locked link with :
Point 3 is linked to :   1   4   7              and has a locked link with :
Point 4 is linked to :   2   3  10              and has a locked link with :
Point 5 is linked to :   2   6  10              and has a locked link with :
Point 6 is linked to :   1   5   9              and has a locked link with :
Point 7 is linked to :   3  11          and has a locked link with :
Point 8 is linked to :   2  10  11              and has a locked link with :
Point 9 is linked to :   6  11          and has a locked link with :
Point 10 is linked to :   4   5   8             and has a locked link with :
Point 11 is linked to :   7   8   9             and has a locked link with :

AI is cutting :
Point 1 is linked to :   6              and has a locked link with :
Point 2 is linked to :   5   8   4              and has a locked link with :
Point 3 is linked to :   4   7          and has a locked link with :
Point 4 is linked to :  10   3   2              and has a locked link with :
Point 5 is linked to :  10   6   2              and has a locked link with :
Point 6 is linked to :   1   9   5              and has a locked link with :
Point 7 is linked to :  11   3          and has a locked link with :
Point 8 is linked to :  11  10   2              and has a locked link with :
Point 9 is linked to :  11   6          and has a locked link with :
Point 10 is linked to :   8   5   4             and has a locked link with :
Point 11 is linked to :   9   8   7             and has a locked link with :
```

An extract of a main function to test aicut, and its result

VII - The second AI

The functions talked about in this section are all in the files “ai.c” and “ai2.c”. For references, please see their code detailed on pages 39-41.

A) Bestpath

When playing the Shannon game, we can notice an interresting fact. No matter if we are playing as Join or Cut, our way of playing will be similar : we find the link with the best chance to create a securised path from the beginning to the end of the level, then we either delete it or lock it, according to our role in the game. So, the principle of this AI is quite simple : by calling the function **bestlink** with the level in parameter, it will return the more significant link of said level. Then this link will either be cut or locked, according to the role of the computer. A link is represented by a new structure **lien** which contains two pointers to a point, and an integer called importance which will be useful later.

So we need to find a way to find the “best link” of the graph. But what is the “best link” of a graph ? First of all, this link need to be in one of the quickest path from the beginning to the end of the graph. For that, the link will be chosen in one of the “best paths” of the graph. And that’s when the function **bestpath** make its first appearance.

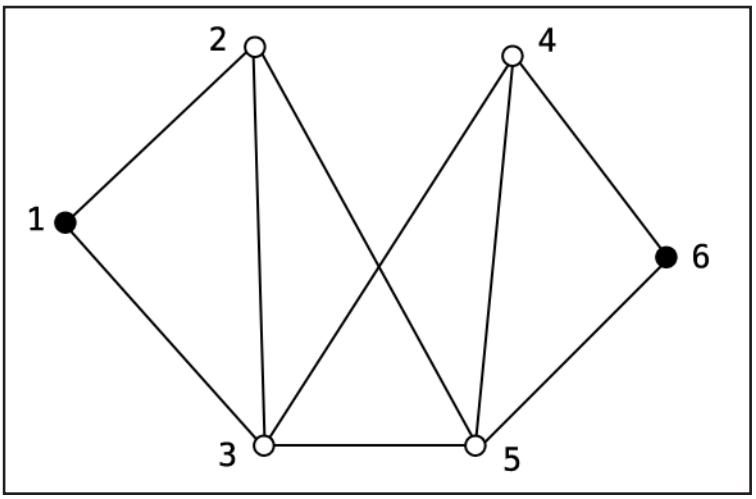


Diagram of a level

This function takes two arguments. The first is the level where the game takes place, and the second is an integer which will represent a size. The function will return an array of all the paths of that size which links the beginning with the end of the level. As an exemple, you can see on the image on the left that if we use **bestpath** with this level and the number three, it returns a list of three paths : (1,2,5,6), (1,3,4,6) and (1,3,5,6). To do that, the function covers all paths of the level by moving recursively from point to point. When it moves from a point to another using a locked link, it will not count that travel in the size of the path. For example, if on that same image, the link from point 3 to point 5 is locked, the path (1,3,5,6) will only be returned by **bestpath** called with 2 but not with 3. When the function finally ends its course, it returns an array of only the paths which were exactly of the good size. Isn’t it beautiful ?

The main function of the AI, **bestlink**, will call **bestpath** multiple times while each time increasing the size in the arguments, until it find at least one path (with a limit of fifteen times since we don’t have any larger level). All the links in these paths will be candidate to be the “best link” returned by the function.

B) Importance

But we need a way to decide between these links, and this is where the integer **importance** in the structure lien comes to play. The role of this integer is to quantify the worth of a link. It quantify the number of important paths going through this link. The first function to use this principle is the function **addlien**. It takes two arguments, a link, and an array of links. If the link doesn’t already belong to the array, the function place

it in ; if it does, the function increase its **importance** by one.

Once **bestlink** has found the candidates to be returned by the function, it use the function **addlien** on each of these links to determine their importance, then it use another function called **veriflien** to obtain an array containing only the links with the best importance. Sometimes, only one link will be left. In that case, the function will just return it, seeing that this link is obviously the more significant of the graph. But in most cases, there will be much more left, and the function will call its sister : the function **bestlink2**. This function takes four arguments : the level, an array of paths which will be of use later, an array containing the potential links that could be returned, and the size of the paths to which these links are belonging. To be able to determine the best link to be returned, this function will count the winning paths with a certain size passing by these links, then, keeping only the best links, it will restart by increasing the size of the winning paths by one until only one link is left, then returns it. To do that, we use the function **incrementlien**, quite similar to **addlien**, which take as arguments one link and one array of links, increasing the importance of the link if it belong to the array, and doing nothing otherwise. In the image on the left, you can see the importance of each link if we test them with a path having a lenght of three. This final link will be the one returned by **bestlink**. As a last thing, we made two functions, **aijoin2** and **aicut2** that respectively lock or delete that link after calling **bestlink** on the level.

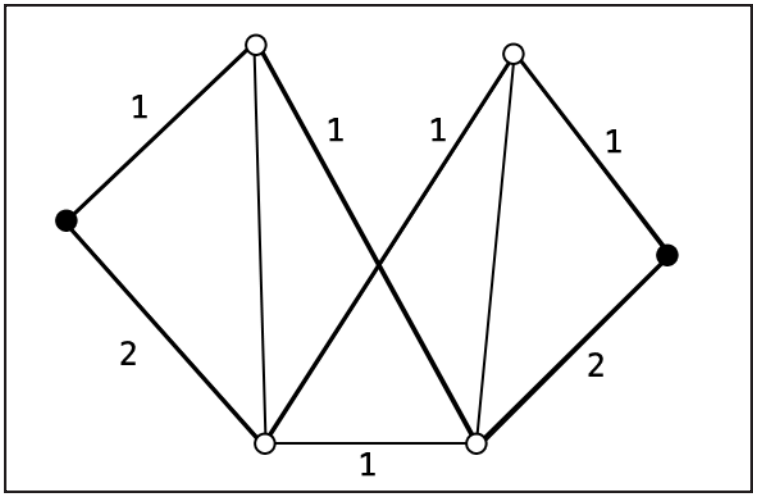


Diagram of a level with the importance of the links

This AI, while not perfect, works well and makes for a good adversary when called upon.

• *Traces of the program execution :*

```
level=level1();
display(level);
printf("AI is Joining :\n");
aijoin2(level);
display(level);
printf("AI is cutting :\n");
aicut2(level);
display(level);
```

```
Point 1 is linked to :   2   3   4         and has a locked link with :
Point 2 is linked to :   1   3   5              and has a locked link with :
Point 3 is linked to :   1   2   4   5          and has a locked link with :
Point 4 is linked to :   1   3   5              and has a locked link with :
Point 5 is linked to :   2   3   4              and has a locked link with :
AI is Joining :
Point 1 is linked to :   2   4          and has a locked link with :   3
Point 2 is linked to :   1   3   5              and has a locked link with :
Point 3 is linked to :   2   4   5              and has a locked link with :   1
Point 4 is linked to :   1   3   5              and has a locked link with :
Point 5 is linked to :   2   3   4              and has a locked link with :
AI is cutting :
Point 1 is linked to :   2   4          and has a locked link with :   3
Point 2 is linked to :   1   3   5              and has a locked link with :
Point 3 is linked to :   2   4          and has a locked link with :   1
Point 4 is linked to :   1   3   5              and has a locked link with :
Point 5 is linked to :   2   4          and has a locked link with :
```

An extract of a main function to test aijoin2 and aicut2, and its result

VIII - Initialization

The functions talked about in this section are all in the files “start.c” and “main.c”. For references, please see their code detailed on pages 42-47.

A) Playing

The only thing left to do was to make a way to start all of this and to play. For that we made the functions **choices**. There are three of them and they’re here to start a game depending on the player demands, all the while taking care of eventual **scanf** error. The first function ask for the choice of the level, while also offering the possibility to quit or to view the rules of the game. Then the player get the choice to play against another player or against th machine. If the PvP is selected, then the game starts by lauching **pvp**, displaying the level at each turn and calling **join** and **cut** one by one. It will stop and announce the winner when the game is finished.

If PvAi is selected, then the player gets further choices. They are asked to chose weither they want to play as Join or Cut and what AI they want to play against, wether the first or the second one. At the end of the choices, the function **pvai** or **pvai2** is launched, taking turn between asking the player to play or having one of the AI makes a move. The first player to start is always chosen at random.

Finally, we wrote the main function to all this. First it clear the terminal window for a better playing confort. The way to do that was found once again on the Open Classroom website[1]. Then we set the random function to start with an int based on the clock, to have the randomier result possible. Finally we launch the function **choice** which will start the game, and we return 0 not to have any warning.

• *Traces of the program execution :*

```
imac-de-julie-rymer:Shannon julierymer$ make
gcc -Wall -g -c ai.c
gcc -Wall -g -c ai2.c
gcc -Wall -g -c aitest.c
gcc -Wall -g -c ending.c
gcc -Wall -g -c level.c
gcc -Wall -g -c main.c
gcc -Wall -g -c player.c
gcc -Wall -g -c point.c
gcc -Wall -g -c start.c
gcc -Wall -g ai.o ai2.o aitest.o ending.o level.o main.o player.o point.o
start.o -o shannon
imac-de-julie-rymer:Shannon julierymer$ make clean
imac-de-julie-rymer:Shannon julierymer$ ./shannon
```

The terminal window when launching our program

1 http://openclassrooms.com/courses/des-couleurs-dans-la-console-linux (consulted on 05/11/15).

```
   In the Shannon Switching Game, you play with a board containing several points,
                    each point is linked to several others.
                There are two players, player Join and player Cut.

The goal of player Join is to have an undestructible path that goes from the first
point to the last.
For that they will, at each turn, lock a connection between two points, so it can't
be destroyed.

The goal of player Cut is that there is no longer any path that link the first point
to the last.
For that they will, at each turn, destroy a connection between two points, except if
the connection is locked.

    If you understand better the principle of the game, you can now try to play !


                        Welcome to Shannon switching game !
                  To try one of the four levels, please type 1, 2, 3 or 4.
                       The fourth level is randomly generated.
                              To view the rules, type 0.
                    To quit the game, type anything else you want.
yolo
Goodbye
```

```
1
Would you like to play against another player or against the machine ?
For PvsP enter 1, for PvsAI enter 2 : 2
Would you like to play as playet Join or Cut ?
For Join enter 1, for Cut enter 2 : 2
Would you like to play against the basic AI ? (type 0)
Or would you like to play against the cleverer one (warning: much more slow) (type 1)
: 0


Point 1 is linked to :   2   3   4              and has a locked link with :
Point 2 is linked to :   1   3   5              and has a locked link with :
Point 3 is linked to :   1   2   4   5          and has a locked link with :
Point 4 is linked to :   1   3   5              and has a locked link with :
Point 5 is linked to :   2   3   4              and has a locked link with :


It's your turn to cut !


Please enter the numbers of the two points between which you want to erase the con-
nection : 4 3
Point 1 is linked to :   2   3   4              and has a locked link with :
Point 2 is linked to :   1   3   5              and has a locked link with :
Point 3 is linked to :   1   2   5              and has a locked link with :
Point 4 is linked to :   1   5          and has a locked link with :
Point 5 is linked to :   2   3   4              and has a locked link with :


It's the machine's turn to join !
Point 1 is linked to :   3   4          and has a locked link with :   2
Point 2 is linked to :   3   5          and has a locked link with :   1
Point 3 is linked to :   1   2   5              and has a locked link with :
Point 4 is linked to :   1   5          and has a locked link with :
Point 5 is linked to :   2   3   4              and has a locked link with :
```

Examples of our program running

Conclusion

A lot of things are left to do and this project is far from finished. But I think we did a good start in the time imposed for a complete game that maybe one day we will finish. We had a lot of trouble with void functions that work directly with the adresses since for a reason we still ignore, it caused “segmentation fault: 11” on the computer of one of us, while perfectly working on the other’s or those of the bocal. For that reason we made nearly only non-void function so that we could both work efficiently. But the second AI still has some issues (segFault if being called upon more than two or three times) on that same computer so it is also one of the issue needing to be fixed. This project gave us a lot of training in Makefile, and initiated us on the use of **rand** and **scanf**. It also made us want to be able to work with the graphic library, so that we could play with a graphical display, since the way we did is impractical to be able visualize the level. But since we will do that next year, maybe it will be the occasion to further improve our program !

The whole program

Makefile

```
OBJ=ai.o ai2.o aitest.o ending.o level.o main.o player.o point.o start.o
CC=gcc -Wall -g

shannon:   $(OBJ) shannon.h
           $(CC) $(OBJ) -o shannon

%.o:       %.c
           $(CC) -c $<

clean:
      @rm -f *.o
      @rm -f core
      @rm -f *~
```

shannon.h

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#define emptybuffer scanf (“%*[^\n]”); getchar ();

typedef struct point* array[100];
struct tab{
  int n;
  array t;
};
typedef struct tab tab;
struct point{
  int num;
  tab link;
  tab locked;
};
typedef struct point point;

struct lien{
     point* p1;
     point* p2;
     int importance;
};
typedef struct lien lien;
//lien is composed of two point’s adress and an int being used by the AI, repre-
sent a link between two point and its worth
typedef lien* tablien[100];

struct arrayLien{
     int n;
     tablien t;
};
typedef struct arrayLien arrayLien;
//array of links = a path
typedef arrayLien* tabarrayLien[100];

struct arrayParcours{
     int n;
     tabarrayLien t;
};
typedef struct arrayParcours arrayParcours;
//array of paths = a list of the different paths we can take


//ai.c
tab aijoin(tab);
tab aicut(tab);
tab aijoin2(tab);
tab aicut2(tab);

//ai2.c
arrayLien* nouveaupal(arrayLien*);
arrayParcours* bestpath1(tab, point*, int, arrayParcours*, arrayLien*, int,
```

```
int[100]);
arrayParcours* bestpath(tab,int);
arrayLien* addlien(lien*, arrayLien*);
arrayLien* incrementlien(lien*, arrayLien*);
arrayLien* veriflien(arrayLien*);
lien* bestlink(tab);
lien* bestlink2(tab, arrayParcours*, arrayLien*, int);

//aitest.c
bool advantagejoin1(tab, int, int[100]);
bool advantagejoin(tab);
bool neutral1(tab, int, int[100]);
bool neutral(tab);
bool advantagecut(tab);
point unlock1(point a, point b);
tab unlock(tab, int, int);

//ending.c
bool belong(int, int, int[100]);
bool winjoin1(point, point, int, int[100]);
bool winjoin(point, point);
bool wincut1(point, point, int, int[100]);
bool wincut(point, point);
void announce(int);

//level.c
tab create(int);
tab link(tab, int, int);
tab level1();
tab level2();
tab level3();
tab levelrandom();
tab randomlink(tab, int);
int numberlinks(tab, int);

//player.c
tab join(tab);
tab cut(tab);

//point.c
void display1(point);
void display(tab);
bool arelinked (tab,int, int);
point erase1(point, point);
tab erase(tab,int, int);
bool arelocked (tab,int, int);
tab lock(tab,int, int);

//start.c
void choice();
void choice2(tab);
void choice3(tab, point*, point*, int,int);
void pvp(tab, point*, point*, int);
void pvai(tab, point*, point*, int,int);
void pvai2(tab, point*, point*, int,int);
void rules();
```

point.c

```
#include “shannon.h”


void display1(point a){
      int i;
      printf(“Point %d is linked to :”,a.num);
      for (i=0;i<a.link.n;i++)
            printf(“%4d”,a.link.t[i]->num);
      printf(“        and has a locked link with :”);
      for (i=0;i<a.locked.n;i++)
            printf(“%4d”,a.locked.t[i]->num);
      printf(“\n”);
}
// Display, for a singular point, with which other points it is linked and with
which is has an undestructible link.

void display(tab level){
      int i;
      for (i=0;i<level.n;i++)
            display1(*level.t[i]);
}
// Display all the points in an array of points


bool arelinked(tab level,int a, int b){
      int i,j;
      for (i=0;i<level.n;i++)
            if (level.t[i]->num==a){
                  for(j=0;j<level.t[i]->link.n;j++){
                        if(level.t[i]->link.t[j]->num==b) return true;
                  }
            };
      return false;
}
// Boolean function that tell us if two points have a link together.

point erase1(point a, point b){
  int i,n;
  n=a.link.n;
  for (i=0;i<n;i++){
      if (a.link.t[i]->num==b.num){
            for(;i<n-1;i++)
                  a.link.t[i]=a.link.t[i+1];
            a.link.n=n-1;
            return a;
      }
  }
  return a;
}
// Enable to erase the connection from point A to point B (one-sided), and return
the resulting point.
```

```
tab erase(tab level,int a, int b){
      int i,j;
      for (i=0;i<level.n;i++){
            if (level.t[i]->num==a)
                  for (j=0;j<level.n;j++){
                        if (level.t[j]->num==b){
                              *level.t[i]=erase1(*level.t[i],*level.t[j]);
                              *level.t[j]=erase1(*level.t[j],*level.t[i]);
                        }
                  };
      }
      return level;
}
// Erase bilaterally the connexion between point number a and point number b in
an array of points.


bool arelocked(tab level,int a, int b){
      int i,j;
      for (i=0;i<level.n;i++)
            if (level.t[i]->num==a){
                  for(j=0;j<level.t[i]->locked.n;j++){
                        if(level.t[i]->locked.t[j]->num==b) return true;
                  }
            };
      return false;
}
// Boolean function that tell us if two points have an undestructible link to-
gether.

tab lock(tab level,int a, int b){
      int i,j,n;
      for (i=0;i<level.n;i++){
            if (level.t[i]->num==a)
                  for (j=0;j<level.n;j++){
                        if (level.t[j]->num==b){
                              *level.t[i]=erase1(*level.t[i],*level.t[j]);
                              *level.t[j]=erase1(*level.t[j],*level.t[i]);
                              n=++level.t[i]->locked.n;
                              level.t[i]->locked.t[n-1]=level.t[j];
                              n=++level.t[j]->locked.n;
                              level.t[j]->locked.t[n-1]=level.t[i];
                        }
                  };
      }
      return level;
}
// lock the connexion between point number a and point number b in an array of
points.
```

level.c

```
#include “shannon.h”


tab create(int n){
      int i;
      tab level;
      point* y;
      level.n=n;
      for(i=0;i<n;i++){
            y=(point*) malloc((size_t) sizeof(point));
            y->num=i+1;
            y->link.n=0;
            y->locked.n=0;
            level.t[i]=y;
      }
      return level;
}
// Create a level (aka an array of points) of n points (without any links between
points yet)


tab link(tab level, int a, int b){
      int i,j,n,x;
      point* temp;
      point* temp2;
      for (i=0;i<level.n;i++){
            if (level.t[i]->num==a)
                  for (j=0;j<level.n;j++){
                        if (level.t[j]->num==b){
                              n=++level.t[i]->link.n;
                              temp=level.t[j];
                              for(x=0;x<n;x++){
                                    temp2=level.t[i]->link.t[x];
                                    level.t[i]->link.t[x]=temp;
                                    temp=temp2;
                              }
                              n=++level.t[j]->link.n;
                              temp=level.t[i];
                              for(x=0;x<n;x++){
                                    temp2=level.t[j]->link.t[x];
                                    level.t[j]->link.t[x]=temp;
                                    temp=temp2;
                              }
                        }
                  };
      }
      return level;
}
// Add a possible link betxeen two points
// Add the link in the first position instead of simply at the end to prevent a
loop in aitest
```

```
tab level1(){
      tab level;
      level=create(5);
      level=link(level,4,5);
      level=link(level,3,5);
      level=link(level,3,4);
      level=link(level,2,5);
      level=link(level,2,3);
      level=link(level,1,4);
      level=link(level,1,3);
      level=link(level,1,2);
      return level;
}
// Creation of Level 1

tab level2(){
      tab level;
      level=create(11);
      level=link(level,10,11);
      level=link(level,9,11);
      level=link(level,8,10);
      level=link(level,7,10);
      level=link(level,6,8);
      level=link(level,6,7);
      level=link(level,5,9);
      level=link(level,5,6);
      level=link(level,4,9);
      level=link(level,4,6);
      level=link(level,3,8);
      level=link(level,3,7);
      level=link(level,2,5);
      level=link(level,2,4);
      level=link(level,1,3);
      level=link(level,1,2);
      return level;
}
// Creation of Level 2

tab level3(){
      tab level;
      level=create(11);
      level=link(level,9,11);
      level=link(level,8,11);
      level=link(level,8,10);
      level=link(level,7,11);
      level=link(level,6,9);
      level=link(level,5,10);
      level=link(level,5,6);
      level=link(level,4,10);
      level=link(level,3,7);
      level=link(level,3,4);
      level=link(level,2,8);
      level=link(level,2,5);
      level=link(level,2,4);
      level=link(level,1,6);
      level=link(level,1,3);
      return level;
}
```

```
// Creation of Level 3

tab levelrandom(){
      tab level;
      int n,i,x;
      n=rand()%10+5;
      level=create(n);
      for(i=0;i<n;i++){
            x=rand()%4;
            level=randomlink(level,i+1);
            if(x==1)
                  level=randomlink(level,i+1);
      }
      if(numberlinks(level,1)<2)level=randomlink(level,1);
      if(numberlinks(level,n)<2)level=randomlink(level,n);
      return level;
}
// Random level generator, between 5 and 15 points, a point always has at least
one link.

tab randomlink(tab level, int a){
      int b;
      b=(rand()%(level.n-1))+1;
      if(numberlinks(level,a)==level.n-1)
            return randomlink(level,b);
      if(a==b || arelinked(level,a,b))
            return randomlink(level,a);
      level=link(level,a,b);
      return level;
}
// Make a link in the level between point a and another point choosen at random
to which point a isn't already linked.

int numberlinks(tab level, int a){
      int i;
      for(i=0;i<level.n;i++){
            if(level.t[i]->num==a)
                  return level.t[i]->link.n;
      }
      return -1;
}
```

player.c

```
#include “shannon.h”

tab join(tab level){
      int a,b,n;
      printf(“\n\nPlease enter the numbers of the two points you wish to lock
together : “);
      n=scanf(“%2d %2d”,&a,&b);
      if(n!=2 || a>level.n || b>level.n){
            printf(“Invalid entry, please try again.\n”);
      emptybuffer;
            return join(level);
      }
      if(arelocked(level,a,b)){
            printf(“These points are already locked together, please try again.\n”);
      emptybuffer;
            return join(level);
      }
      if(!arelinked(level,a,b)){
            printf(“These points don’t have any connection, you can’t lock them,
please try again.\n”);
      emptybuffer;
            return join(level);
      }
      level=lock(level,a,b);
    emptybuffer;
      return level;
}// Do the join move with the player’s input from the keyboard, function restart
if there are errors in the input.
tab cut(tab level){
      int a,b,n;
      printf(“\n\nPlease enter the numbers of the two points between which you
want to erase the connection : “);
      n=scanf(“%2d %2d”,&a,&b);
      if(n!=2){
            printf(“Invalid entry, please try again.\n”);
      emptybuffer;
            return cut(level);
      }
      if(arelocked(level,a,b)){
            printf(“These points are already locked together, you can’t erase
their connection, please try again.\n”);
      emptybuffer;
            return cut(level);
      }
      if(!arelinked(level,a,b)){
            printf(“These points already don’t have any connection, please try
again.\n”);
      emptybuffer;
            return cut(level);
      }
      level=erase(level,a,b);
    emptybuffer;
      return level;
}// Same thing with cut move
```

ending.c

```
#include “shannon.h”


bool belong(int x, int n, int list[100]){
      int i;
      for(i=0;i<n;i++){
            if (x==list[i]) return true;
      }
      return false;
}
// Test if int x belong to the list.


bool winjoin1(point a, point b, int n, int list[100]){
      int i;
      if (a.num==b.num) return true;
      if(belong(a.num,n,list)) return false;
      n++;
      list[n-1]=a.num;
      for (i=0;i<a.locked.n;i++){
                  if(winjoin1(*a.locked.t[i], b,n,list))
                        return true;
      }
      return false;
}
// Run through all the locked path from point a to see if we can reach point b

bool winjoin(point a,point b){
      int list[100];
      return winjoin1(a,b,0,list);
}
// Return true if player Join has won


bool wincut1(point a, point b, int n, int list[100]){
      int i;
      if (a.num==b.num) return false;
      if(belong(a.num,n,list)) return true;
      n++;
      list[n-1]=a.num;
      for (i=0;i<a.link.n;i++){
                  if(!wincut1(*a.link.t[i], b,n,list))
                        return false;
      }
      for (i=0;i<a.locked.n;i++){
                  if(!wincut1(*a.locked.t[i], b,n,list))
                        return false;
      }
      return true;
}
// Go through all the possible paths (linked and locked) of point a to see if
there is a way to go to b. Return true if there is none.
```

```
bool wincut(point a,point b){
      int list[100];
      return wincut1(a,b,0,list);
}
// Return true if player Cut has won


void announce(int x){
      printf("\n\n    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
      switch(x){
            case 0:
                  printf("\n\n    Player Join has won ! Congratulations !\n\n\n");
                  break;
            case 1:
                  printf("\n\n    Player Cut has won ! Congratulations !\n\n\n");
                  break;
            case 2:
                  printf("\n\n    You have won against the computer ! You're too
strong, congratulations !\n\n\n");
                  break;
            case 3:
                  printf("\n\n    The computer beat you ! That's too bad, maybe
you'll win next time !\n\n\n");
                  break;
      }
}
// Depending on the argument, announce the winner
```

```
#include “shannon.h”

bool advantagejoin1(tab level, int n, int list[100]){
      int i,j,x,y;
      point a,b;
      tab leveltest;
      a=*level.t[0];
      b=*level.t[level.n-1];
      if(winjoin(a,b)) return true;
      if(wincut(a,b)) return false;
      for(i=0;i<level.n;i++){
            for(j=0;j<level.t[i]->link.n;j++){
                  x=level.t[i]->num;
                  y=level.t[i]->link.t[j]->num;
                  if(!belong(y,n,list)){
                        leveltest=erase(level,x,y);
                        if(!neutral(leveltest)){
                              leveltest=link(level,x,y);
                              return false;
                        }
                        leveltest=link(level,x,y);
                  }
            }
            n++;
            list[n-1]=x;
      }
      return true;
}
bool advantagejoin(tab level){
      int list[100];
      return advantagejoin1(level,0,list);
}// If true it means the player Join can win weither or not they start first.

bool neutral1(tab level, int n, int list[100]){
      int i,j,x,y;
      tab leveltest;
      for(i=0;i<level.n;i++){
            for(j=0;j<level.t[i]->link.n;j++){
                  x=level.t[i]->num;
                  y=level.t[i]->link.t[j]->num;
                  if(!belong(y,n,list)){
                        leveltest=lock(level,x,y);
                        if(advantagejoin(leveltest)){
                              leveltest=unlock(level,x,y);
                              return true;
                        }
                        leveltest=unlock(level,x,y);
                  }
            }
            n++;
            list[n-1]=x;
      }
      return false;
}
```

```
bool neutral(tab level){
      int list[100];
      return neutral1(level,0,list);
}
// If true it means the starting player can win. Needed for advantagejoin

bool advantagecut(tab level){
      return !(advantagejoin(level) || neutral(level));
}
// If true it means the player Cut can win weither or not they start first.


point unlock1(point a, point b){
  int i,j,n;
  point temp=a;
  n=a.locked.n;
  for (i=0;i<n;i++){
      if (a.locked.t[i]->num==b.num){
            temp.locked.n=n-1;
            for(j=0;j<i;j++)
                  temp.locked.t[j]=a.locked.t[j];
            for(;i<n-1;i++)
                  temp.locked.t[i]=a.locked.t[i+1];
            return temp;
      }
  }
  return a;
}
// Enable to erase the locked connection from point A to point B (one-sided), and
return the resulting point.

tab unlock(tab level,int a, int b){
      int i,j;
      for (i=0;i<level.n;i++){
            if (level.t[i]->num==a)
                  for (j=0;j<level.n;j++){
                        if (level.t[j]->num==b){
                              *level.t[i]=unlock1(*level.t[i],*level.t[j]);
                              *level.t[j]=unlock1(*level.t[j],*level.t[i]);
                        }
                  };
      }
      level=link(level,a,b);
      return level;
}
// Erase bilaterally the connexion between point number a and point number b in
an array of points.
// Useful for the AI because it changes the level when doing its tests, so enable
to put back things together.
```

ai.c

```
#include “shannon.h”

tab aijoin(tab level){
      int i,j,x,y;
      tab leveltest;
      if(advantagecut(level)) return aijoin2(level);
      for(i=0;i<level.n;i++){
            for(j=0;j<level.t[i]->link.n;j++){
                  x=level.t[i]->num;
                  y=level.t[i]->link.t[j]->num;
                  leveltest=lock(level,x,y);
                  if(advantagejoin(leveltest)){
                        level=leveltest;
                        return level;
                  }
                  leveltest=unlock(level,x,y);
            }
      }
      return aijoin2(level);
}
// Test to see if there is a move that either put the AI at an advantage or make
it win, if there is not, play at random.
tab aicut(tab level){
      int i,j,x,y;
      tab leveltest;
      if(advantagejoin(level)) return aicut2(level);
      for(i=0;i<level.n;i++){
            for(j=0;j<level.t[i]->link.n;j++){
                  x=level.t[i]->num;
                  y=level.t[i]->link.t[j]->num;
                  leveltest=erase(level,x,y);
                  if(advantagecut(leveltest)){
                        level=leveltest;
                        return level;
                  }
                  leveltest=link(level,x,y);
            }
      }
      return aijoin2(level);
}
// Same thing but for the Cut move.

tab aijoin2(tab level){
      lien* r;
      r=bestlink(level);
      level=lock(level,r->p1->num,r->p2->num);
      return level;
}
tab aicut2(tab level){
      lien* r;
      r=bestlink(level);
      level=erase(level,r->p1->num,r->p2->num);
      return level;
}
```

```
#include “shannon.h”


arrayLien* nouveaupal(arrayLien* pal){
      int i;
      arrayLien* newpal =(arrayLien *) malloc(sizeof(arrayLien));
      newpal->n=pal->n ;
      for(i=0;i<newpal->n;i++){
            newpal->t[i]=pal->t[i] ;
      }
      return newpal;
}
//creation of new pal from pal containing the same elements

arrayParcours * bestpath1(tab level, point* ps, int tailleParcours, arrayPar-
cours * pap, arrayLien* pal, int n, int parcours[100]){
            int i;
            arrayLien* newpal;
            lien* pl;
            parcours[n]=ps->num;
            n++;
            if(tailleParcours==0 && ps==level.t[level.n-1]){
                  newpal = nouveaupal(pal);
                  pap->t[pap->n]=newpal;
                  pap->n++;
            }
            else if(tailleParcours !=0 && ps!=level.t[level.n-1]){
                  for(i=0; i<ps->link.n;i++){
                        if(!belong(ps->link.t[i]->num,n,parcours)){
                              pl = (lien *) malloc (sizeof(lien));
                              pl->p1=ps; pl->p2=ps->link.t[i]; pl->importance=0;
                              pal->t[pal->n] = pl ;
                              pal->n++;
                              pap=bestpath1(level,    ps->link.t[i],    taillePar-
cours-1, pap, pal, n, parcours);
                              pal->n--;
                        }
                  }
                  for(i=0; i<ps->locked.n;i++){
                        if(!belong(ps->locked.t[i]->num,n,parcours)){
                              pap=bestpath1(level,ps->locked.t[i], tailleParcours,
pap, pal,n,parcours);
                        }
                  }
            }
            return pap;
}
arrayParcours * bestpath(tab level,int tailleParcours){
      int parcours[100];
      int n=0;
      arrayParcours * pap;
      arrayLien* pal;
      pap=(arrayParcours *) malloc(sizeof(arrayParcours));
      pap->n=0;
```

```
	pal=(arrayLien *) malloc(sizeof(arrayLien));
	pal->n=0;
	return bestpath1(level, level.t[0], tailleParcours, pap, pal,n, parcours);
}
//return an array of all the paths of size tailleparcours

arrayLien* addlien(lien* pl, arrayLien* pal){
	int i;
	for(i=0;i<pal->n;i++){
		if(pl->p1==pal->t[i]->p1){
			if(pl->p2==pal->t[i]->p2){
				pal->t[i]->importance++;
				return pal;
			}
		}
		else if(pl->p1==pal->t[i]->p2){
			if(pl->p2==pal->t[i]->p1){
				pal->t[i]->importance++;
				return pal;
			}
		}
	}
	pal->t[pal->n]=pl;
	pal->n++;
	return pal;
}
// ajoute le lien pointé par lien* au tableau de de lien pal, ou incrémente
l'importance du lien si il y est déjà


arrayLien* incrementlien(lien* pl, arrayLien* pal){
	int i;
	for(i=0;i<pal->n;i++){
		if(pl->p1==pal->t[i]->p1){
			if(pl->p2==pal->t[i]->p2){
				pal->t[i]->importance++;
				return pal;
			}
		}
		else if(pl->p1==pal->t[i]->p2){
			if(pl->p2==pal->t[i]->p1){
				pal->t[i]->importance++;
				return pal;
			}
		}
	}
	return pal;
}
//incrémente l'importance du lien pointé par lien* si celui-ci est dans pal

arrayLien* veriflien(arrayLien* pal){
	int max=0;
	int i;
	for(i=0;i<pal->n;i++){
		if(pal->t[i]->importance>max){
			max=pal->t[i]->importance;
		}
	}
```

```
	arrayLien* resultat = (arrayLien *) malloc(sizeof(arrayLien));
	resultat->n=0;
	for(i=0;i<pal->n;i++){
		if(pal->t[i]->importance==max){
			resultat->t[resultat->n]=pal->t[i];
			resultat->n++;
		}
	}
	return resultat;
}
//cherche dans le tableau de lien pal le(s) lien(s) avec l'importance la plus grande et le(s) renvoie

lien* bestlink2(tab level, arrayParcours * pap, arrayLien* pal, int tailleParcours){
	int i,j;
	while(tailleParcours<15){
		tailleParcours++;
		pap=bestpath(level, tailleParcours);
		for(i=0;i<pap->n;i++){
			for(j=0;j<pap->t[i]->n;j++){
				pal=incrementlien(pap->t[i]->t[j],pal);
			}
		}
		pal=veriflien(pal);
		if(pal->n==1){
			return pal->t[0];
		}
	}
	return pal->t[0];
}

lien* bestlink(tab level){
	int i,j,k;
	arrayParcours* pap;
	arrayLien* resultat;
	pap=(arrayParcours*) malloc ((size_t) sizeof(arrayParcours));
	pap->n =0;
	resultat=(arrayLien*) malloc ((size_t) sizeof(arrayLien));
	resultat->n=0;
	for(i=1;i<15;i++){
		pap=bestpath(level,i);
		if(pap->n>0){
			for(j=0;j<pap->n;j++){
				for(k=0;k<pap->t[j]->n;k++){
					resultat=addlien(pap->t[j]->t[k],resultat);
				}
			}
			resultat=veriflien(resultat);
			if(resultat->n==1){
				return resultat->t[0];
			}
			return bestlink2(level, pap, resultat,i);
		}
	}
	return resultat->t[0];
}
//renvoie le pointeur du meilleur lien possible du graphe donné en entrée
```

start.c

```
#include “shannon.h”


void choice(){
      int a,n;
      tab level;
      printf(“\n                          Welcome to Shannon switching game !”);
      printf(“\n                 To try one of the four levels, please type 1,
2, 3 or 4.”);
      printf(“\n                               The fourth level is randomly gener-
ated.”);
      printf(“\n                               To view the rules, type 0.”);
      printf(“\n                          To quit the game, type anything else you
want.\n”);
      n=scanf(“%1d”,&a);
      if(n==1){
            if (a==0){
                  emptybuffer;
                  rules();
                  choice();
                  return;
            }
            if(a==1){
                  emptybuffer;
                  level=level1();
                  choice2(level);
                  return;
            }
            if(a==2){
                  emptybuffer;
                  level=level2();
                  choice2(level);
                  return;
            }
            if(a==3){
                  emptybuffer;
                  level=level3();
                  choice2(level);
                  return;
            }
            if(a==4){
                  emptybuffer;
                  level=levelrandom();
                  choice2(level);
                  return;
            }
      }
      emptybuffer;
      printf(“Goodbye !\n”);
}
// Ask the player to choose between the levels, or reading the rules or quitting,
```

```
void choice2(tab level){
      int x,a,n;
      x=rand()%2;
      printf(“Would you like to play against another player or against the ma-
chine ?\nFor PvsP enter 1, for PvsAI enter 2 : “);
      n=scanf(“%1d”,&a);
      if(n==1){
            if (a==1){
                  emptybuffer;
                  printf(“\n\n”);
                  display(level);
                  pvp(level,level.t[0],level.t[level.n-1],x);
                  return;
            }
            if (a==2){
                  emptybuffer;
                  printf(“Would you like to play as playet Join or Cut ?\nFor Join
enter 1, for Cut enter 2 : “);
                  n=scanf(“%1d”,&a);
                  emptybuffer;
                  choice3(level,level.t[0],level.t[level.n-1],x,a);
                  return;
            }
      }
      printf(“Invalid entry, please try again.\n”);
     emptybuffer;
      choice2(level);
}
// Ask the player to choose between PlayerversusPlayer or PlayerversusAI. If sec-
ond option, ask if player want to be Join or Cut. Random number 0 or 1 to decide
who start

void choice3(tab level,point* start, point* finish, int x,int y){
      int a,n;
      printf(“Would you like to play against the basic AI ? (type 0)\nOr would
you like to play against the cleverer one (warning: really slow) (type 1) : “);
      n=scanf(“%1d”,&a);
      if(n==1){
            if (a==1){
                  emptybuffer;
                  printf(“\n\n”);
                  display(level);
                  pvai(level,start,finish,x,y);
                  return;
            }
            if (a==0){
                  emptybuffer;
                  printf(“\n\n”);
                  display(level);
                  pvai2(level,start,finish,x,y);
                  return;
            }
      }
      printf(“Invalid entry, please try again.\n”);
     emptybuffer;
      choice3(level,start,finish,x,y);
}
// Ask the player to choose between the first or the second AI
```

```
void pvp(tab level,point* start, point* finish,int x){
     if (x==0){
          printf("\n\nIt's player Join's turn !\n");
          level=join(level);
          display(level);
          if(winjoin(*start,*finish)){
               announce(0);
               return;
          }
          pvp(level,start,finish,1);
     }
     if (x==1){
          printf("\n\nIt's player Cut's turn !\n");
          level=cut(level);
          display(level);
          if(wincut(*start,*finish)){
               announce(1);
               return;
          }
          pvp(level,start,finish,0);
     }
}
// Launch one by one the join move and the cut move, with first start decided by x.

void pvai(tab level,point* start, point* finish, int x,int y){
     if(y!=1 && y!=2){
          printf("\nInvalid entry, please try again.\n\n\n");
          choice2(level);
     }
     if(!x){
          if(y==1){
               printf("\n\nIt's your turn to join !\n");
               level=join(level);
               display(level);
               if(winjoin(*start,*finish)){
                    announce(2);
                    return;
               }
               pvai(level,start,finish,1,y);
          }
          if(y==2){
               printf("\n\nIt's your turn to cut !\n");
               level=cut(level);
               display(level);
               if(wincut(*start,*finish)){
                    announce(2);
                    return;
               }
               pvai(level,start,finish,1,y);
          }
     }
     if(x){
          if(y==1){
               printf("\n\nIt's the machine's turn to cut ! Please be patient
while it's thinking.\n");
               level=aicut(level);
               display(level);
               if(wincut(*start,*finish)){
```

```
                    announce(3);
                    return;
               }
               pvai(level,start,finish,0,y);
          }
          if(y==2){
               printf("\n\nIt's the machine's turn to join ! Please be patient
while it's thinking.\n");
               level=aijoin(level);
               display(level);
               if(winjoin(*start,*finish)){
                    announce(3);
                    return;
               }
               pvai(level,start,finish,0,y);
          }
     }
}
// Launch one by one the ai move and the player move, with first start decided by
x and roles (Join or Cut) decided by y.

void pvai2(tab level,point* start, point* finish, int x,int y){
     if(y!=1 && y!=2){
          printf("\nInvalid entry, please try again.\n\n\n");
          choice2(level);
     }
     if(!x){
          if(y==1){
               printf("\n\nIt's your turn to join !\n");
               level=join(level);
               display(level);
               if(winjoin(*start,*finish)){
                    announce(2);
                    return;
               }
               pvai2(level,start,finish,1,y);
          }
          if(y==2){
               printf("\n\nIt's your turn to cut !\n");
               level=cut(level);
               display(level);
               if(wincut(*start,*finish)){
                    announce(2);
                    return;
               }
               pvai2(level,start,finish,1,y);
          }
     }
     if(x){
          if(y==1){
               printf("\n\nIt's the machine's turn to cut !\n");
               level=aicut2(level);
               display(level);
               if(wincut(*start,*finish)){
                    announce(3);
                    return;
               }
               pvai2(level,start,finish,0,y);
```

```
            }
            if(y==2){
                  printf("\n\nIt's the machine's turn to join !\n");
                  level=aijoin2(level);
                  display(level);
                  if(winjoin(*start,*finish)){
                        announce(3);
                        return;
                  }
                  pvai2(level,start,finish,0,y);
            }
      }
}

void rules(){
      printf("\033[H\033[2J");
      printf("\n\n                In the Shannon Switching Game, you play with a
board containing several points,\n");
      printf("                          each point is linked to several others.\n");
      printf("                    There are two players, player Join and player
Cut.\n\n");
      printf("   The goal of player Join is to have an undestructible path that
goes from the first point to the last.\n");
      printf("      For that they will, at each turn, lock a connection between
two points, so it can't be destroyed.\n\n");
      printf("       The goal of player Cut is that there is no longer any path
that link the first point to the last.\n");
      printf("    For that they will, at each turn, destroy a connection between
two points, except if the connection is locked.\n\n");
      printf("              If you understand better the principle of the game,
you can now try to play !\n\n");
}
// Display the rules of the game
```

main.c

```
#include “shannon.h”


int main(){
      printf(“\033[H\033[2J”);
      srand(time(NULL));
      choice();
      return 0;
}
```