RAPPORT DE PROGRAMMATION : PROJET WHITEHALL

Florian SAMAT, Fabien BESSIERES

EISE4 – 2017/2018

Sommaire

I) Prérequis

Les librairies SDL2_ttf et Freetype6 sont nécessaires à la compilation du programme. Celles-ci peuvent être installées en exécutant les commandes suivantes dans un terminal :

```
sudo apt-get install libfreetype6-dev
sudo apt-get install libsdl-ttf2.0-dev
```

Il convient également de vérifier que les versions des bibliothèsques SDL image et ttf utilisées sont bien celles de SDL2. Un makefile est fourni afin de générer les exécutables du client, du serveur, et des applications de génération automatique du plateau que nous présenterons plus en détail dans la partie III.

II) Présentation de l'application

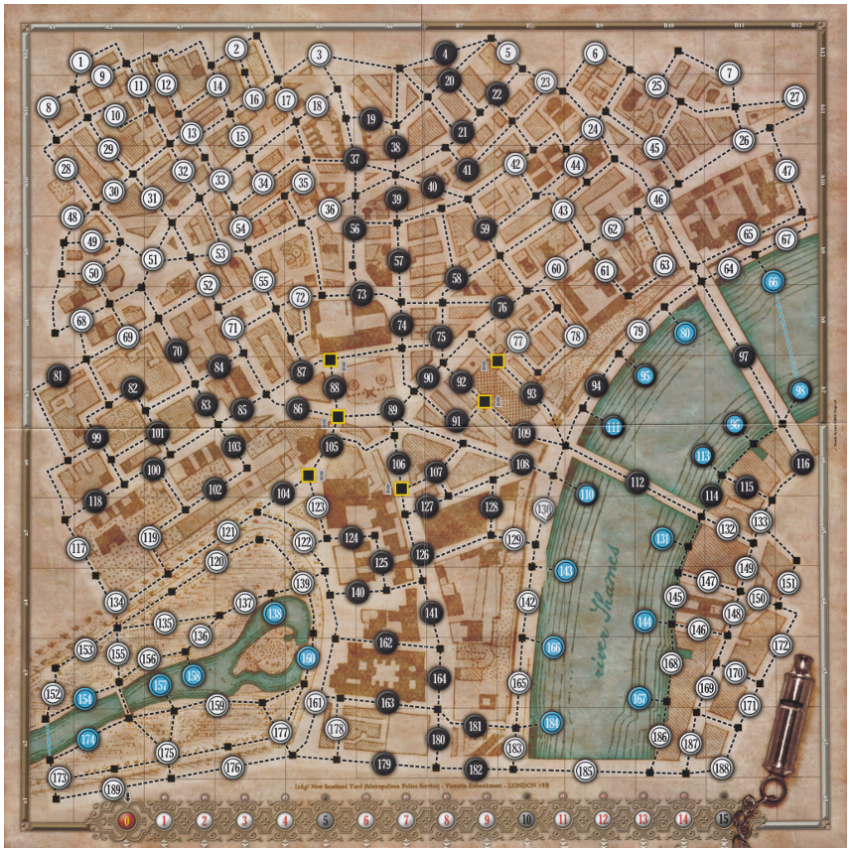
L'application serveur doit être exécutée avec la commande :

```
./server_white #portServeur
```

De la même manière, l'application client se lance grâce à la commande :

```
./client_white @IPServer #portServeur @IPClient #portClient nom
```

Whitehall est un jeu de société dans lequel un joueur incarne Jack l'Eventreur, face à trois autres joueurs incarnant des policiers devant l'arrêter, dont voici le plateau de jeu :



En début de partie, Jack l'éventreur désigne quatre cibles, représentées par des ronds blancs, dans quatre quartiers différents. Il commence alors sur l'une de ces cases, assassinant sa cible et révélant sa position aux policiers. De même, les policiers se placent chacun sur un carré jaune différent. Jack l'Eventreur dispose à chaque fois de quinze tours pour atteindre une prochaine cibles, sinon il perd la partie. Le but des policiers est donc soit de procéder à une arrestation, soit l'empêcher d'atteindre sa cible à temps. En effet, Jack se déplace d'un rond à chaque tour en passant par lignes pointillées et carrés, tandis que les policiers peuvent se déplacer de zéro à deux carrés par tour, et l'assassin ne peut évidemment pas passer au dessus d'un policier. Chaque tour se se divise en trois phases : Escape in the Night, Hunting the Monster, et Clues and Suspisions. Jack se déplace lors de la première phase, puis c'est au tour des policiers de se déplacer pendant la deuxième phase, puis de chercher des indices du passage de l'Eventreur ou de procéder à son arrestation. Ils peuvent ainsi soit interroger les ronds voisins pour déceler des traces jusqu'à ce qu'un indice soit révélé, soit procéder à une arrestation au niveau du rond sélectionné. Afin d'équilibrer le jeu, les meurtres ne sont révélés qu'à la fin d'un tour, et tous les indices sont supprimés, puisque les polciers connaissent une position plus récente de Jack l'éventreur.

Les joueurs, après avoir lancé l'application client_white comme spécifié ci-dessus, doivent cliquer dans la bande à droite du plateau de jeu afin de se connecter au serveur, et reçoivent leur rôle en fonction de leur ordre de connection (le premier jouera Jack, les autres le policier Jaune, Vert, puis Bleu), qui s'affiche en haut à droite de l'écran dans la couleur correspondante. Au cours de la partie, ils indiquent au serveur leur action en cliquant sur un indice sur le plateau. Le serveur, qui agit comme un maître du jeu, évalue la possibilité d'effectuer effectivement cette action. Au cours de la partie, les joueurs reçoivent des instructions du serveur (qui correspondent à la phase de jeu en cours) en vert au dessus du plateau. Un joueur peut également recevoir un message d'erreur, qui s'affiche en rouge et disparaît au prochain clic, lorsque c'est son tour de jouer et que l'action qu'il choisit enfreint les règles du jeu. Enfin, les policiers peuvent presser les touches l et a du clavier pour indiquer au serveur qu'ils choisissent respectivement de chercher des indices, ou procéder à une arrestation. Nous aurions voulu réaliser un bouton dans le volet latéral de l'écran afin de réaliser ces actions, mais n'avons pu par manque de temps. Lorsque la partie est terminée, les clients et le serveur doivent être relancés.

III) Implémentation du plateau de jeu

Afin de générer le plateau de jeu qui est extrêmement complexe puisqu'il compte 189 ronds, environ 300 carrés et près de 500 liaisons, nous avons été particulièrement astucieux.

Tout d'abord, nous avons créé une première version de l'application client, dans laquelle seule l'interface graphique est fonctionnelle et limitée, puisqu'elle nous permet de récupérer les coordonnées des points en cliquant dessus dans l'ordre. Ces informations sont ensuite stockées dans un fichier, et nous avons programmé un code simple qui génère les lignes de codes permettant de les initialiser.

Ensuite, nous avons modifié le code qui affichait les coordonées afin qu'il affiche cette fois l'indice lorsque l'on clique sur un point d'intérêt. Il fallait ensuite décider d'une façon de générer les déplacements. Il s'agit d'un problème assez complexe, puisque chaque liaison est bidirectionnelle, et les extrémités peuvent être soit des ronds, soit des carrés. Au lieu de cliquer sur chaque rond, puis sur chacun de ces voisins ainsi que les carrés empruntés pour passer de l'un à l'autre, ou même de cliquer sur chaque carré, puis sur chacun des extrémités des liaisons adjacentes, nous avons pu simplement cliquer une fois sur les extremités de chaque liaison et générer ce que nous voulions automatiquement. Cela diminue considérablement le nombre de clics à effectuer, et diminue

fortement les chances d'en oublier. En modifiant l'interface graphique pour qu'elle affiche un retour à la ligne lorsque nous pressons la touche Entrée, nous obtenons donc un fichier "LIAISONS" se présentant de cette façon :

1 201
201 9
201 8
201 221
…

Nous avons ensuite écrit un algorithme de génération automatique nommé gen.c. Nous avons remarqué que chaque liaison a au plus six voisins, et puisque les chaînes de caractères doivent doivent être formatées afin d'utiliser des fonctions comme sscanf(), nous avons décidé de représenter les liaisons partant d'un indice de la manière suivante :

13 219 213 214 218 -1 -1
14 204 213 214 -1 -1 -1
15 214 215 -1 -1 -1 -1
...

Le premier nombre correspond à l'indice concerné, puis les nombres suivants correspondent aux liaisons issues de cet indice, et nous complétons avec des -1 afin d'obtenir six laisons potentielles. L'algorithme fonctionne de la façon suivante : nous créons un tableau deux dimensions contenant deux indices par ligne, puis nous lisons le fichier LIAISONS. Ensuite, pour chaque indice, nous affichons cet indice, puis parcourons le tableau précédemment créé : si l'un des deux nombre correspond à l'indice, alors nous affichons le second nombre et augmentons un compteur, et affichons un certain nombre de -1 en fonction en fonction du nombre de liaisons trouvées jusqu'à obtenir sept indices par ligne. Le résultat de l'exécution est ensuite stocké dans un fichier nommé "carrés".

Puis le le code situé dans le fichier genLiaisons.c permet de générer le code d'un tableau liaisons[533][7] contenant ces nombres. Il sera en effet important pour le serveur de posséder ces informations, puisqu'il peut vouloir accéder aux indices voisins d'un carré pour les arrestations par exemple. Il est important de préciser qu'un numéro de ligne du tableau correspond aux liaisons partant de l'indice égal à ce numéro. Cela permet d'accéder immédiatement à l'information souhaitée sans parcourir l'ensemble du tableau à chaque fois en comparant les indices, et participe à une bonne optimisation du code et donc à une meilleure réactivité du serveur qui exécutera des fonctions de recherche de chemin pour les policiers.

Dans cette même optique, nous avons décidé de générer un algorithme de recherche de chemin genMap.c dont nous sommes assez fiers, permettant d'obtenir toutes les possibilités de mouvement pour Jack afin de les stocker directement en mémoire. De même, nous avons remarqué qu'au plus trois carrés séparent deux ronds, nous cherchions à produire un tableau à deux dimensions contenant 5 nombres par ligne (le premier étant le rond de départ, le second le rond d'arrivée, et enfin les carrés par lesquels il faut passer, en complétant avec des -1). Cet algorithme est robuste, puisqu'il ne génère pas de boucle infinie lorsqu'il entre dans un puits, comme on peut en trouver entre les ronds 3, 4, 19 et 29 par exemple, et optimisé, puisqu'on accède directement aux indices par leur position dans le tableau. Pour cela, après avoir lu le fichier carré et stocké les nombres dans un tableau, pour chaque indice, nous parcourons le tableau de ses voisins en lançant pour chacun d'eux un algorithme de recherche de chemin. Nous gérons pour cela différents tableau : le tableau pass

contient 533 entrées, dont la valeur à l'indice I correspond au passage de l'algorithme de pathfinding (bonne optimisation), un tableau res qui contient les résultats, et un tableau chemin de trois entrées contenant le chemin parcouru. Le fonctionnement de l'algorithme de pathfinding est le suivant : si l'indice actuel est compris entre 1 et 189, il s'agit d'un rond, il ne convient pas d'aller plus loin : on stocke alors dans le tableau résultat à la bonne ligne grâce à un compteur, l'indice de départ, l'indice d'arrivée, puis l'ensemble des carrés parcourus, stockés dans le tableau chemin jusqu'à un certain indice connu puisque passé en paramètre, l'on complète avec des -1 et on incrémente de 1 la ligne dans laquelle on écrira le prochain résultat, sinon, on indique qu'on est passé par là, on stocke la valeur de l'indice actuel dans le tableau chemin au bon indice, et on relance l'alogirthme sur tous les voisins de l'indice actuel, avec pour indice actuel le prochain indice, et comme indice d'écriture dans chemin celui passé en paramètre plus un, puis une fois que tous les recherches de voisins ont retourné, on supprime le passage à l'indice actuel et on retourne. Enfin, nous affichons le contenu du tableau résultat sous forme de code à inclure dans liaisons.h, de même que le tableau généré par genLiaisons.c. Cela permet d'éviter de surcharger le code du serveur qui est déjà très long.

IV) Fonctionnement du serveur :

Le fonctionnement des échanges entre les clients et le serveur et le client est représenté par un schéma ci-dessous. Afin de simplifier la lecture nous ne représentons que les échanges entre le serveur et Jack, et entre le serveur et le policier jaune puisque tous les policiers reçoivent les mêmes méssages et envoient des messages de structure identique. Les formats de messages reçus par chaque extrémité sont les suivants :

SERVEUR	CLIENT
• "C @IP #port nom" : tentative de connexion d'un client • "K ind" : Jack l'Eventreur a cliqué sur la case ind • "J ind" : le policier Jaune a cliqué sur l'indice ind • "V ind" : le policier Vert a cliqué sur l'indice ind • "B ind" : le policier Bleu a cliqué sur l'indice ind • "L [J,V,B]" : Jaune, Vert ou Bleu souhaite chercher des indices • "A [J,V,B]" : Jaune, Vert ou Bleu décide d'effectuer une arrestation	• "T message" : instructions reçues à afficher (message) • "E erreur" : message d'erreur à afficher • "I [0,1,2,3]" : rôle désigné par le serveur, dans l'ordre, Jack, Jaune, Vert, Bleu. • "[K,J,V,B] ind" : le joueur correspondant s'est déplacé à l'indice ind (seul Jack reçoit le premier) • "N nb" : début du tour nb • "M ind" : meurtre à l'indice ind • "S ind" : trace de Jack à l'indice ind • "X ind" : cible validée (seul Jack reçoit ce format).

Le serveur est géré par une machine à 19 états (fsmServer), dont voici la liste et l'ordre :

fsmServer = 0 : Attente de connexion de 4 clients.
fsmServer = 1 : Sélection des 4 points de meurtres pour Jack.
fsmServer = 2 : Placement policier Jaune.
fsmServer = 3 : Placement policier Vert.
fsmServer = 4 : Placement policier Bleu.
fsmServer = 5 : Placement Jack.
fsmServer = 6 : Déplacement Jack.
fsmServer = 7 : Déplacement policier Jaune.
fsmServer = 8 : Déplacement policier Vert.
fsmServer = 9 : Déplacement policier Bleu.
fsmServer = 10 : Choix d'actions du policier Jaune :
- **fsmServer = 11 :** Recherche d'indices
- Ou **fsmServer = 12 :** Arrestation de Jack. Si Jack est sur l'indice proposé par le policier alors fsmServer = 19.

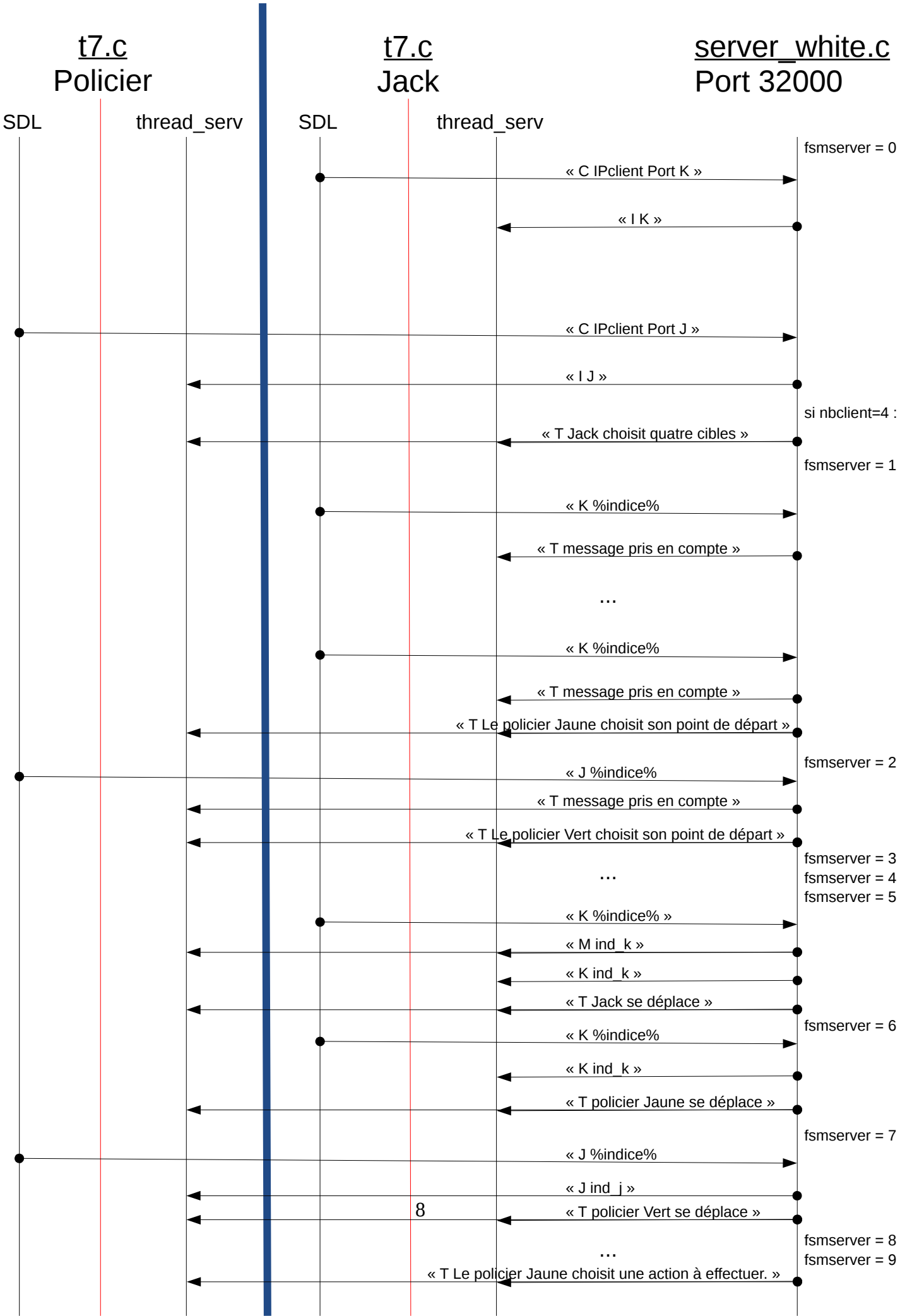
fsmServer = 13 : Choix d'actions du policier Vert :
- **fsmServer = 14 :** Recherches d'indices
- Ou **fsmServer = 15 :** Arrestation de Jack. Si Jack est sur l'indice proposé par le policier alors fsmServer = 19.

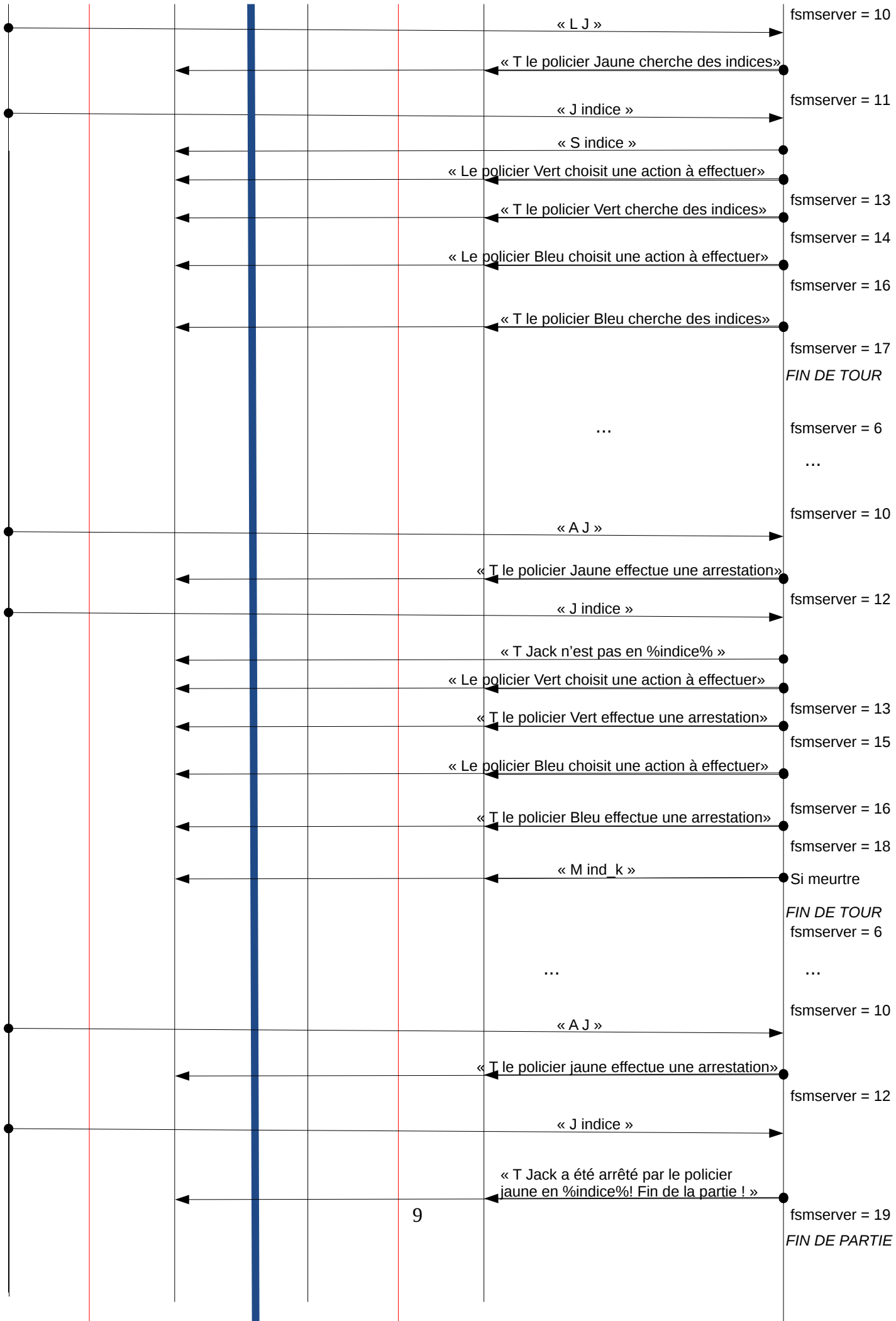
fsmServer = 16 : Choix d'actions du policier Bleu :
- **fsmServer = 17 :** Demande si Jack est passé par ici. Puis, fin du round on retourne à fsmServer = 6.
- Ou **fsmServer = 18 :** Arrestation de Jack. Puis, fin du tour, on retourne à fsmServer = 6. Si Jack est sur l'indice proposé par le policier alors fsmServer = 19.

Si le nombre de tour est 15, ou Jack est arrêté, ou toutes les cibles sont atteintes alors fsmServer passe à 19 : Fin de la partie.

Il serait long et fastidieux d'expliquer le fonctionnement du code ici, alors les parties intéressantes seront présentées dans la prochaine partie, tandis que le code source est accessible et commenté.

Lorsque le client reçoit le format X, alors il place une croix rouge à l'indice correspondant afin de permettre à Jack de se rappeler la position de ces cibles (cette fonctionnalité n'ayant été ajoutée qu'après rédaction de l'échange suivant, elle n'y est donc pas représentée, mais est reçue lorsque Jack a sélectionné une cible valide). Lorsque le client reçoit son rôle, il est stocké en mémoire afin d'être envoyé lorsque le joueur clique sur une case. Un message indiquant son rôle au joueur s'affiche alors dans la couleur correspondante jusqu'à la fin de la partie. En outre, lorsque le client reçoit un format N, alors il déplace le pion des tours sur le numéro correspondant (nous n'avons cependant pas pu nous assurer de son bon placement à cause la résolution de nos écrans). De même, lorsqu'il reçoit un format de déplacement, il déplace le pion correspondant à la case reçue. Enfin, lorsqu'il reçoit un format S, il place un rond jaune à l'indice correspondant, et s'il reçoit un format M, alors il place un rond rouge à cet indice et supprime les ronds jaunes.


t7.c
Policier
t7.c
Jack
server_white.c
Port 32000
SDL
thread_serv
SDL
thread_serv
fsmserver = 0
« C IPclient Port K »
« I K »
« C IPclient Port J »
« I J »
si nbclient=4 :
« T Jack choisit quatre cibles »
fsmserver = 1
« K %indice%
« T message pris en compte »
...
« K %indice%
« T message pris en compte »
« T Le policier Jaune choisit son point de départ »
fsmserver = 2
« J %indice%
« T message pris en compte »
« T Le policier Vert choisit son point de départ »
fsmserver = 3
...
fsmserver = 4
fsmserver = 5
« K %indice% »
« M ind_k »
« K ind_k »
« T Jack se déplace »
fsmserver = 6
« K %indice%
« K ind_k »
« T policier Jaune se déplace »
fsmserver = 7
« J %indice%
« J ind_j »
« T policier Vert se déplace »
fsmserver = 8
...
fsmserver = 9
« T Le policier Jaune choisit une action à effectuer. »


fsmserver = 10
« L J »
« T le policier Jaune cherche des indices»
fsmserver = 11
« J indice »
« S indice »
« Le policier Vert choisit une action à effectuer»
fsmserver = 13
« T le policier Vert cherche des indices»
fsmserver = 14
« Le policier Bleu choisit une action à effectuer»
fsmserver = 16
« T le policier Bleu cherche des indices»
fsmserver = 17
FIN DE TOUR
...
fsmserver = 6
...
fsmserver = 10
« A J »
« T le policier Jaune effectue une arrestation»
fsmserver = 12
« J indice »
« T Jack n'est pas en %indice% »
« Le policier Vert choisit une action à effectuer»
fsmserver = 13
« T le policier Vert effectue une arrestation»
fsmserver = 15
« Le policier Bleu choisit une action à effectuer»
fsmserver = 16
« T le policier Bleu effectue une arrestation»
fsmserver = 18
« M ind_k »
Si meurtre
FIN DE TOUR
fsmserver = 6
...
...
fsmserver = 10
« A J »
« T le policier jaune effectue une arrestation»
fsmserver = 12
« J indice »
« T Jack a été arrêté par le policier jaune en %indice%! Fin de la partie ! »
fsmserver = 19
FIN DE PARTIE

IV) Codes notables et difficultés rencontrées

Comme expliqué ci-dessus, nous sommes fiers de l'implémentation du plateau de jeu. De même, certaines parties des programmes sont intéressantes à présenter.

Fonctions de déplacements

```
int mouvementAutorise_Police(int destination, int joueur){
  if (destination <= 200)
    return 0;
  if (joueur == 1)
    return mouvementAutorise_Police_rec_j(ind_j,destination, 2);
  if (joueur == 2)
    return mouvementAutorise_Police_rec_v(ind_v,destination, 2);
  else
    return mouvementAutorise_Police_rec_b(ind_b,destination, 2);
}
```

```
int mouvementAutorise_Police_rec_b(int actuel, int destination, int mov){
  int i, next;
  if ((actuel == destination) && (actuel != ind_v) && (actuel != ind_j))
    return 1;
  else if (mov >= 1){
    pass[actuel] = 1;
    for (i = 1; i < 7; i++){
      next = liaisons[actuel][i];
      if ((next > 0) && (pass[next] != 1)){
        if (next <= 189){
          if (mouvementAutorise_Police_rec_b(next, destination, mov)){
            pass[actuel] = 0;
            return 1;
          }
        }
        else {
          if (mouvementAutorise_Police_rec_b(next, destination, mov - 1)){
            pass[actuel] = 0;
            return 1;
          }
        }
      }
    }
    pass[actuel] = 0;
  }
  return 0;
}
```

Ci-dessus les fonctions de déplacement des policiers. Pour des raisons d'optimisation de la vitesse de calcul, nous avons décidé de créer une fonction pour chaque policier afin d'éviter des comparaisons à chaque appel récursif. Avec du recul, un switch aurait également accéléré le code en n'effectuant pas de comparaison et en "branchant" alors simplement sur le bon 'if'. Cette fonction se base sur les mêmes principes d'optimisation évoqués dans la partie III : accès direct à la bonne case par son indice dans un tableau, tableau de passage, etc. Ici cependant, puisqu'un policier peut ne pas se déplacer, on vérifie d'abord si l'indice sur lequel on se trouve est égal à celle de la destination (donnée par le clic du joueur), sinon, on indique qu'on est passé par ici et parcourt l'ensemble des voisins par lesquels on n'est pas déjà passés en consommant 0 ou 1 mouvement selon la nature de la case voisine (puisque les policiers passent au dessus des ronds). Si l'un des appels est couronné de succès, on retourne 1.

```
int mouvementAutorise_Jack(int indice){
  int i,j,b;
  int test = 0;
  if (indice == ind_k)
    return 0;
  for (i = 0; i < 1582; i++){
    if (map[i][0] > ind_k) // map est classée par indice, donc optimisation
      return 0;
    if ((map[i][0] == ind_k) && (map[i][1] == indice)){
      b = 1;
      test = 1;
      for (j = 2; j < 5; j++){
        if ((map[i][j] == ind_j) || (map[i][j] == ind_v) || (map[i][j] == ind_b))
          b = 0;
      }
      if (b == 1)
        return 1;
    }
  }
  if (test == 0) // le tableau a ete parcouru sans succès, ne se passe que si ind_k vaut 189
    return 0;
  else // l'indice est valide mais des policiers bloquent le chemin
    return -1;
}
```

Ci-dessus la fonction pour valider les déplacements de Jack. On parcourt l'ensemble de map[][], en vérifiant le premier et second entier de chaque ligne, s'ils correspondent à la position actuelle de Jack et la case qu'il a choisi, on vérifie qu'il n'y a pas un policier sur un des carrés. Puisque les indices sont classés par ordre croissant, si le premier nombre dépasse la position actuelle de Jack

sans succès, on retourne un échec, afin d'optimiser le temps de calcul. Avec du recul, cette fonction aurait pu être optimisée encore davantage en ajoutant une table qui indiquerait à partir de quelle ligne commencent les indices concernés et donc limiter les comparaisons inutiles.

Gestion des traces (clues)

Nous sommes également assez fiers d'avoir réussi à implémenter le traitement des recherches d'indices, qui est un problème complexe. En effet, le nombre de ronds voisins (auxquels on a un accès immédiat grâce à l'implémentation du plateau) diffère selon la position, et il faut également savoir combien ont été vérifiés, et lesquels. Pour cela, à chaque déplacement, on a recours à l'allocation dynamique : on compte d'abord le nombre de ronds voisins, puis l'on crée deux tableaux de taille ce nombre, l'un correspond à l'indice d'un rond voisin, le second, initialisé à 0, indique si la case de même indice dans le premier tableau a déjà été visitée (le ième "booléen" du second tableau indique si le ième indice du premier tableau a été visité. Si ce n'est pas le cas, alors on passe a l'état suivant si il y a trace de Jack, sinon on augmente le nombre d'indices visités sans succès et on passe à l'état suivant s'il atteint le nombre de voisins.

```
    //fsmServer = 7 : Deplacement du policier Jaune : Hunting the Monster
    if (fsmServer == 7){
      sscanf(buffer, "%c %d", &tmp, &indice);
      if (mouvementAutorise_Police(indice, 1)){
        ind_j = indice;
        sprintf(reply,"J %d",ind_j);
        broadcastMessage(reply);
        lj = 0;
        nbtj = 0;
        for (i = 1; i <= 7; i++) // on regarde combien de cercles sont voisins
          if ((liaisons[ind_j][i] <= 189) && (liaisons[ind_j][i] >= 1))
            lj++;
        tmp_alloc = realloc(vj, lj);
        vj = tmp_alloc;
        tmp_alloc = realloc(vjt,lj);
        vjt = tmp_alloc;
        lj = 0;
        for (i = 1; i <= 7; i++) // on crée les tableaux associés
          if ((liaisons[ind_j][i] <= 189) && (liaisons[ind_j][i] >= 1)){
            vj[lj] = liaisons[ind_j][i];
            vjt[lj] = 0;
            lj++;
          }
        broadcastMessage("T Le policier Vert se deplace.");
        fsmServer = 8;
      }
      else
        sendMessageToClient(tcpClients[1].ipAddress, tcpClients[1].port,"E Veuillez selectionner une case valide.");
    }
  //fsmServer = 11 : Jaune recherche des indices
  if (fsmServer == 11){
    sscanf(buffer, "%c %d", &tmp, &indice);
    if((indice <= 189) && (est_present(liaisons[ind_j], indice, 7))){
      trace = traceJack(indice);
      if (trace){
        sprintf(reply, "S %d", trace);
        broadcastMessage(reply);
        broadcastMessage("T Le policier Vert choisit une action a effectuer");
        fsmServer = 13;
      }
      // Si pas de trace alors...
      else {
        for (i = 0; i < lj; i++) // on cherche la position de indice dans le tableau des voisins
          if (vj[i] == indice){
            break;
          }
        if (vjt[i] == 1) // Si cet indice a déja ete testé, on envoie une erreur
          sendMessageToClient(tcpClients[1].ipAddress, tcpClients[1].port,"E Cette case a deja ete vérifiee.");
        else{ // sinon, on marque qu'il a ete testé et on vérifie si tous les voisins ont ete verifiés sans succès pour passer au tour suivant
          vjt[i] = 1;
          nbtj++;
          if (nbtj == lj){
            broadcastMessage("T Le policier Vert choisit une action a effectuer");
            fsmServer = 13;
          }
        }
      }
    }
    else
      sendMessageToClient(tcpClients[1].ipAddress, tcpClients[1].port,"E Veuillez selectionner une case valide.");
  }
  //fsmServer = 12 : Jaune effectue une arrestation
```

Gestion des cibles

Lorsque Jack choisit quatre cibles, il faut s'assurer qu'il n'indique pas deux cibles dans le même quartier, il faut donc savoir quelles cases sont situées dans quel quartier, quels quartiers ont déjà une cible, etc. On a donc créé un tableau de 189 entrées (généré automatiquement à partir de quatre tableaux contenant les cases situées dans le quartier correspondant, visible dans genQuartiers.c), dont les valeurs valent entre 1 et 5 (1 correspond, au Nord-Ouest, 2 au Nord-Est, etc, et 5 correspond à une case noire), ainsi qu'un second tableau de quatre entrées initialisées à 0. Lorsque Jack sélectionne une cible, le serveur cherche à cet indice dans le premier tableau le numéro de quartier correspondant, s'il s'agit par exemple du quartier Nord Ouest, alors le serveur regarde le second tableau à l'indice 1 (pour Nord-Ouest) – 1 (offset) = 0. On vérifie auparavant qu'il ne s'agit pas de 5 (case noire) afin d'éviter les erreurs de segmentations (si cette condition est fausse, alors la suivante n'est pas évaluée grâce à &&). Si la case d'indice 0 dans le second tableau a pour valeur 0, alors on la passe à 1 et on ajoute au tableau des cibles l'indice envoyé, et on augmente le nombre de cibles sélectionnées (et on passe à l'état suivant si ce nombre atteint 4), sinon on envoie un message d'erreur à Jack.

```
//fsmServer = 1 : Jack choisit les 4 points
if(fsmServer == 1)
  {
    sscanf(buffer, "%c %d", &tmp, &indice);
    //printf("%d %d\n", quartiers[indice], cibles_quartier[quartiers[indice] - 1]);
    if ((indice <= 189) && (quartiers[indice] != 5) && (cibles_quartier[quartiers[indice] - 1] == 0)){
        ptKill[nbPts++] = indice;
        sprintf(reply, "X %d", indice);
        sendMessageToClient(tcpClients[0].ipAddress, tcpClients[0].port, reply);
        cibles_quartier[quartiers[indice] - 1] = 1;
        //printf("%d %d %d %d\n", ptKill[0], ptKill[1], ptKill[2], ptKill[3]);
      }
    else
      sendMessageToClient(tcpClients[0].ipAddress, tcpClients[0].port,"E Veuillez selectionner une clible valide.");
    //sendMessageToClient(tcpClients[0].ipAddress, tcpClients[0].port,"T messageprisencompte\n");
    if(nbPts == 4) {
      broadcastMessage("T Le policier Jaune choisit son point de depart");
      fsmServer = 2;
    }
  }
```

Difficultés rencontrées et retour d'experience

Nous sommes également fiers de l'implémentation de la fonction gérant les fins de tour, assez complexe à cause du nombre de cas à traiter et leur enchevêtrement, mais que nous ne détaillerons pas ici pour cette raison. De même, nous sommes particulièrement fiers que le code du serveur, qui a été rédigé en une fois sans pouvoir le tester au fur-et-à-mesure faute d'interface graphique fonctionnelle à cause du changement de résolution (les points étant décalés), ait fonctionné immédiatement et correctement à l'exception de quelques erreurs causées par des erreurs de frappe ou d'inattention (comme if(fsmServer = 12) qui faisait passer étrangement l'état du serveur à cette valeur sans envoyer le message d'instructions correspondant, nous laissant perplexes avant de remarquer cette erreur). Cependant, nous avons rencontré quelques difficultés en implémentant l'interface graphique puisqu'il s'agissait de notre premier travail sur la librairie SDL, et que TTF n'est pas particulièrement simple à utiliser. Nous avons par exemple appris qu'il fallait absolument libérer les surfaces et détruire les textures crées au fur-et-à mesure par les fonctions de TTF, et qui saturaient la mémoire à la manière d'une boucle infinie de fork() jusqu'à blocage complet de l'ordinateur.

Ainsi nous avons beaucoup appris grâce à ce projet, du fonctionnement des échanges réseaux et d'un serveur, la pertinence des machines à état qui permettent de simplifier le code en le rendant robuste, l'utilisation d'une librairie graphique (fonctionnant simultanément avec des messages reçus par un serveur), les optimisations et possibilités offertes par les tableaux, et perfectionné notre utilisation de fonctions récursives et de recherche de chemins.