Université de Montpellier

L2 Informatique

Recherche d'un trajet donné sur une carte.

Rapport de Projet T.E.R.

Projet Informatique ---HLIN405

Etudiants :

Soufyani Amine

Ouail Abed

Bouchengour Mohamed

Année : 2017-2018

Encadrant :

Mr Hervé DICKY

FACULTÉ DES SCIENCES

Nous tenons à remercier notre encadrant Mr Hervé Dicky pour son accompagnement, et ses conseils précieux tout au long de ce projet qui nous ont permis de finir le projet dans son intégralité.

Table des matières

Introduction

Insérer le sujet de TER (celui ci ayant été retirer de Moodle nous n'avons pas pu le récupérer)

Automate déterministe

a) Définition de la structure de données à utiliser

Le Sujet défini une carte comme tel :

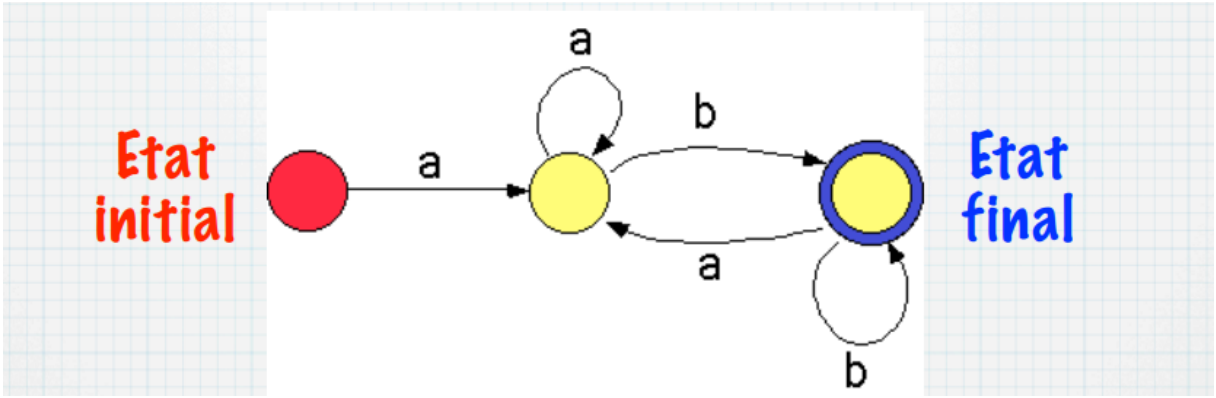
> « Une carte est composée d'un ensemble de salles et d'un ensemble de couloirs
>
> - chaque couloir est étiqueté par une lettre d'un alphabet A donné
>
> - un couloir ne peut être parcouru que dans un seul sens
> Deux couloirs qui partent d'une même salle ne peuvent être étiquetés de la même façon.
>
> Une carte contient en particulier une salle de départ et une salle d'arrivée. »

Cette structure nous fait fortement penser aux automates déterministes vu en Modèles de Calcul l'année passée.

En effet, de façon intuitive on a :

- Un Automate contient en particulier un Etat Initial et un Etat Final pouvant respectivement s'apparenter à la salle de départ et la salle d'arrivée.
- Une flèche a un seul sens (pas de <->) et est étiqueté par une lettre d'un alphabet
  Les transitions peuvent alors s'apparenter à un couloir

N. B : Un mot w de longueur n est reconnu ou accepté par un automate fini s'il existe un chemin menant de q0 à un état final de F étiqueté par la suite de lettres du mot w.



b) Définition formelle d'un automate

Un automate fini est un quintuplet A = (E, Q, R, I, F), où :

- E est un alphabet **(étiquètes du couloir)**
- Q est un ensemble fini, appelé ensemble des états. **(Ensemble de toutes les salles)**
- I est une partie de Q appelé ensemble des états initiaux **(salle de départ unique car automate déterministe)**
- F est une partie de Q appelé ensemble des états finaux **(salle d'arrivée)**
- R est une partie de Q x E x Q appelé ensemble des transitions

Un automate est **déterministe** si, d'une part, il a un et un seul état initial et si, d'autre part, la relation R est fonctionnelle au sens suivant :

Si (p, a, q) appartient à R et (p, a, q') appartient à R alors q = q'

S'il en est ainsi, on définit une fonction, appelée **fonction de transition**, notée traditionnellement δ. Cette fonction de transition prend en paramètre un état et un symbole (appartenant à l'alphabet) et qui renvoi un état.

– Notation : δ (qi, a) = qj (« signature » δ : Q × Σ → Q)

c) Définition de la Structure de donnée en C++

Notre première structure de donnée utilisait des vectors afin de représenter les ensembles ainsi que, l'état initial et final (ensemble réduit à un élément).

Les remarques faites par notre encadrant nous ont permis de se rendre compte que l'utilisation des vectors est inapproprié de trouver une autre structure de donnée plus ingénieuse :

```
19  class Automate{
20  private:
21    //quintuplet
22    std::map< char, int> alphabet ;
23    //char* alphabet; /*A*/
24    Etat* ensembleGlobal; //A
25    Etat* etatFinal; //A
26    Etat* etatInitial; //A
27    Etat*** ensembleTransitions; //A
28
```

En effet, cette structure de donné utilise un simple tableau a la place d'un vecteur pour l'ensemble global d'Etats car le nombre d'états étant connu et fixe, nous n'avions pas besoin de tableaux dynamiques (vectors).

Un point crucial de la représentation de l'automate a été la modélisation de la fonction de Transition un point sur lequel nous avons dû revenir très souvent avec notre encadrant.

Plusieurs représentations de cette fonction ont été menée avant d'aboutir à une représentation efficace au niveau de sa complexité :

La **première représentation** consistait à modéliser cette fonction autour du triplet transition construit comme tel :

```
 9  class Transition{
10  private:
11    Etat* depart;
12    char etiquette;
13    Etat* arrivee;
14
```

La fonction de transition aurait alors cherché dans l'ensemble de transitions (vector) l'état commençant par l'état de départ et ayant l'étiquette correspondante a la transition.
Nous aurions obtenu alors eu une complexité O(n).

Cependant, une des indications de notre chargé de Projet était que la fonction de transition doit avoir une complexité en O(1).

La **seconde représentation** trouvée fut alors la suivante :

Utiliser un tableau a 2 dimensions afin de représenter la fonction de transition sous forme matricielle.

Exemple :

Soit l'ensemble des états {0, 1, 2}, et l'alphabet {a, b}

Nous avons :

/	0	1	2
0	a	a	null
1	a	b	b
2	null	null	b

Nous obtenons alors une fonction de transition en $O(n^2)$.

Néanmoins en reprenant cette seconde représentation sous somme parvenus à trouver une représentation nous permettant de faire la transition en O(1), cette **troisième représentation** est la suivante :

Nous avons gardé la fonction de transition sous forme matricielle mais nous avons changé les indices.

Exemple :

Soit l'ensemble des états {0, 1, 2}, et l'alphabet {a, b}

Nous avons :

/	a	b
0	0	1
1	1	null
2	null	2

Cette forme matricielle nous permet en effet d'avoir une fonction de transition sous la forme O(1)

L'accès aux états d'arrivée au travers de cette fonction se fait de la façon suivante :

T[indiceEtat][indicelettre] -> retourne l'état d'arrivée.

Remarque : cette représentation est appelé la table de transition de l'automate.

c) Algorithmes et complexités

Dans ce projet nous devons principalement répondre à deux questions :

- La **première,** existe-t -il sur la carte un chemin étiqueté par le mot de l'Etat initial à l'état final ?
- La **seconde** quelle est ce chemin ?

Afin de répondre à ces deux questions de nombreux algorithmes vont nous être utiles, Dans cette prochaine partie nous allons vous les présenter ainsi que donner leur complexité.

a) Les Algorithmes de génération aléatoire.

Entrée : Nombre Salle : Int, Densité : flottant, Alphabet : tableau de char ,taille Alphabet :Int
Sortie : Un automate déterministe

Les Trois algorithmes de génération aléatoires suivants ont la même signature Celle-ci est :

```
Automate generationAleatoire(int nbSalles , float densite, char* n_alphabet,int nbLettresAlphabet);
```

Note : Dans ces algorithmes de génération les transitions réflexives, les transitions entrantes dans l'état de départ et les transitions sortantes de l'Etat Final ne sont pas autorisées. Chaque transition est assignée seulement si la transition[etat1][lettre] n'existe pas.

Notre encadrant de TD nous a posé comme contrainte que la génération aléatoire doit se faire en fonction de la densité qui est définie comme telle : densité : nb salles/nb transitions.

Algorithme 1 :
Le premier algorithme de génération aléatoire attribue les transitions sans aucune spécificité.
C'est-à-dire que tant que le nombre de Transition est inferieure au nombre de salle * la densité.

Algorithme 2 :
Ce deuxième algorithme de génération aléatoire attribue les transitions de façon à ce que le nombre de transition entrante et sortante soit pratiquement égal, pour cela nous avons donc créer deux conditions selon que cette dite transition soit entrante ou sortante.
Nb : une transition est dite entrante si Etat a < Etat b || sortante si Etat a > Etat b.

Algorithme 3 :
Ce troisième algorithme est une variation du premier algorithme, ce dernier effectue sa transition finale d'un état aléatoire vers l'état final. Afin de s'assurer que l'état final ai une transition entrante et donc de potentiellement augmenter ses chances d'avoir un chemin.

b) Génération d'un mot aléatoirement
Entrée : longeurdumot : int
Sortie : un mot de taille longeurdumot généré aléatoirement à partir de l'alphabet.

L'objectif de cet algorithme est de générer un mot à partir de l'alphabet de l'automate et de la longueur du mot souhaitée.
Afin de choisir ce mot de manière aléatoire nous avons utilisé la fonction srand() et rand() qui nous permettent de générer un int entre 0 et le nombre de lettre de l'alphabet .

Cet Algorithme de génération de mot aléatoire est trivial, il nous suffit de concaténer chaque lettre obtenue aléatoirement tant la longueur de cette dite concaténation est inférieure à celle passée en paramètre.

La signature de cette fonction sera donc :
string genaleat(int longueur)
La complexité de la génération d'un mot de manière aléatoire est en O(longeurMot).

c) Fonction de Transition
Entrée : un état a et une lettre b
Sortie : Un pointeur sur l'état d'arrivée si il existe null sinon.

La fonction de transition a grandement été détaillée précédemment cependant quelques précisions sont à faire :
La fonction de Transition, nous a mené à adapter l'ensemble des transitions et l'alphabet de l'automate afin de la faire fonctionner de manière optimale.

Tout d'abord, notons le changement apporté à l'alphabet :
A la place d'avoir un tableau de caractères, nous avons mis en place une table de hachage :
std::map< char, int> alphabet;
Cette dernière est initialisée dans les constructeurs aléatoires de manière à indicer chaque lettre par un int.

Exemple :
Soit l'alphabet {a,b,c,d,e} nous obtenons :

a	b	c	d	e
0	1	2	3	4

Ce choix a été fait car la fonction de transition a la signature suivante :
Etat* fonctionDeTransition(Etat a,char lettre);

En effet, la fonction de transition prend en paramètre un état et une lettre, or l'accès a un élément d'un tableau se fait au travers d'Int.

La fonction de transition renvoyant un (pointeur d') état contenu dans l'ensemble de transition, l'option d'une hashmap nous a semblé judicieux.

L'ensemble de transition est représenté comme expliquer plus haut, comme un tableau a deux dimensions (état/lettres_alphabet).

La complexité de la fonction de transition est donc en O(1)

d) Mot reconnu
Entrée : mot : String
Sortie : Vrai si le mot est reconnu par l'automate, Faux sinon

L'algorithme mot reconnu est l'un des algorithmes principaux de ce Projet de T.E.R, en effet cet algorithme répond à la **question** : existe-t-il sur la carte un chemin tel que le mot mène de l'état initial a l'état final.

Pour répondre à cette question il suffit de vérifier s'il existe un chemin menant de l'état initial vers l'état final tel que la concaténation des étiques des transitions parcourues est égale au mot passé en paramètre.

La signature de cette fonction est :
bool motReconnu(string mot);

Cet algorithme a pour point de départ l'état initial (stocké dans une variable temp). A partir ce cette variable temp nous effectuons ***itérativement*** les actions suivantes :
Nous enregistrons l'état suivant dans temp à l'aide de la fonction de transition avec comme paramètre la variable temp et la première lettre du mot, si cette transition n'existe pas nous renvoyons faux sinon nous répétons cette opération a l'aide d'une boucle (deuxième lettre … i -ème lettre ...).
Arrivé à la dernière lettre nous vérifions si l'état obtenu est celui d'arrivée. Si oui, nous retournons vrai, sinon faux.

La complexité de la fonction motReconnu est O(longeurMot).

e) Existe Chemin
Entrée : vide
Sortie : Vrai s'il existe un chemin, faux sinon.

La fonction de *recherche de chemin* consiste à trouver *s'il existe une suite de transition liant un état de départ et un état d'arrivée.*

Cette fonction répond directement à la première question du projet : existe-t-il sur la carte un chemin étiqueté par le mot de l'état initial à l'état final.

C'est un algorithme de fermeture transitive.

Nous avons décider d'implémenter cet algorithme de façon récursive :

Pour faire cela, nous avons utilisé 2 fonctions. La première a comme paramètre un état, et cherche s'il existe un chemin entre l'État passé en paramètre et l'état final en exécutant l'algorithme suivant :
- On crée un tableau de pointeurs avec comme taille le nombre de lettres de l'alphabet (État tableau[NbrLettresAlphabet]).

- On le remplit avec les transitions (s'il en existe) de l'État passé en paramètres.
- On parcourt le tableau et pour chaque case non vide, on vérifie si elle contient l'état final, si oui on fait une retourne true.
- Sinon, si on n'a pas encore traverse cet état, on le marque avec (setsalletraverse), et on fait une appeler récursive sur cet état.

La seconde fonction récupère ce résultat et le renvoi.
Cette dernière est en effet celle qu'on va utiliser dans la main, elle ne contient pas de paramètres, pour éviter tout mauvaise manipulation du part de l'utilisateur.

f) Plus petit chemin

Entrée : vide

Sortie : plus petit mot reconnu par l'automate si il existe un chemin, le mot vide sinon.

Cet algorithme vise à répondre à la deuxième question principale de ce projet de TER.

En effet, si l'automate admet un chemin alors cet algorithme renvoi le plus petit chemin menant de l'état de départ à l'état final. Sinon ce dernier renvoi la chaine de caractère vide.

Trouver le plus petit chemin revient à effectuer une traversée en largeur de l'automate (plus de détails en annexes).
La complexité de ce parcours en largeur est O (nbSalles + taille alphabet).

g) Afficher automate

Entrée : vide

Sortie : vide, mais génère un fichier. tex ainsi qu'un fichier .pdf

Cette méthode génère un fichier. tex en utilisant la librairie PGF/Tikz (plus de détails dans le fichier annexe mtd afficher Automate) a l'aide des flux sortie et de system afin de compiler le fichier en pdf (system("pdflatex AfficherAutomate.tex");)

Complexité en O(nbLettres*nbSalles) car parcours de toutes les transitions.

Automate indéterministe

1) Automate indéterministe définition formelle

La structure d'un automate indéterministe se rapproche de celle d'un automate déterministe a quelques exceptions près :

- Un automate non déterministe peut avoir plusieurs transitions partant d'un même état portant la même étiquette

Ceci se traduira lors de l'implémentation par une structure de donnée différente.

2) Possible Implémentation en C++

En effet, en nous basant sur la représentation de l'automate faite précédemment, les modifications suivantes seront à effectuer :

- La fonction de transition quant a elle devra retourner un ensemble d'états en fonction d'un état et d'une lettre sa structure sera donc un tableau a deux dimensions d'ensembles d'états.

  Exemple :

  Soit l'ensemble des états {0, 1, 2}, et l'alphabet {a, b}

  Nous avons :

/	a	b
0	{0,1,2}	{1}
1	{0,2}	null
2	null	{2,1}

Garder cette structure de tableau a deux dimensions nous permettre de conserver une fonction de transition en O(1) ;

3) Algorithmes et complexité

Pour l'automate non déterministe nous devons répondre aux deux mêmes questions :

- La **première,** existe-t-il sur la carte un chemin étiqueté par le mot de l'Etat initial à l'état final ?
- La **seconde** quelle est ce chemin ?

a) Les Constructeurs aléatoires.

Les Trois algorithmes de génération aléatoires suivants ont la même signature que pour l'automate déterministe.

Ici, un point important a été de gérer l'unicité des états dans la transition afin de ne pas avoir d'états en double dans la même transition.

Les trois algorithmes ont été inchangées mis à part les conditions menant a l'ajout d'un état a la transition.

Automate déterministe	Automate indéterministe
La transition mène t'elle a un état ?	L'état existe t'il dans la transition ?

b) Mot reconnu

L'algorithme de reconnaissance de mot est le même que précédemment a une exception près :

Nous devons gérer l'union des ensembles de transitions afin d'avoir un algorithme plus performant.

En effet, dans l'automate indéterministe nous partons d'un ensemble de départ et nous voulons arriver à un espace d'arrivée.

Nous cherchons donc à faire la fusion de tout les espace d'arrivées d'un ensemble d'état de départ par une lettre. Cette procédure nous permet de gagner en complexité car l'étude des chemins possibles est effectué simultanément.

Dans ce but nous avons créé la fonction Union états qui prend en paramètre un état et une lettre et retourne l'union des ensembles d'arrivée par cette lettre.

c) Existe Chemin

L'algorithme existe chemin est le même que précédemment : a l'exception de la gestion d'union d'états pour les raisons expliquées précédemment.

d) Plus petit chemin

L'algorithme plus petit chemin est le même que précédemment : a l'exception de la gestion d'union d'états pour les raisons expliquées précédemment.

Conclusion

Ce projet a été notre vrai premier projet de programmation depuis notre première année et nous a beaucoup appris, en parallèle des cours et Tps de ce semestre et de l'an dernier.
En effet ce projet nous a appris à penser et réfléchir a la structure de donnée que l'on va utiliser et a son implémentation car avant de se mettre à programmer des lignes et des lignes de codes ainsi qu'implémenter des algorithmes inefficients. Nous devons nous poser la question : Quelle structure de donnée présentera les meilleures performances pour ce que l'in veut faire ?
L'un des challenges que nous avons rencontré des le début du projet et celui de trouver une structure de donnée efficiente pour cette opération car en effet cette dernière est la plus utilisée dans notre programme et avoir une fonction de transition en $O(n^2)$ aurait rendu les algorithmes très inefficients et inutilisables car trop lents pour des automates contenant de nombreux états (ex : automate a 50000 états).

Nous avons réussi à respecter le cahier des charges et donc à créer une application qui cherche un un trajet sur une carte (automate), que ce soit pour un automate déterministe ou indéterministe, les fonctionnalités proposées pour chacun sont : générer aléatoirement un automate, vérifier s'il existe une trajectoire, chercher la plus courte trajectoire et vérifier si un mot est reconnu par l'automate.

Nous avons permis une visualisation de l'automate (carte) crée a l'aide du langage TikZ pour les petits automates (inferieur ou égal a 10 états).

Nous avons dû régulièrement tester nos fonctions (surtout "existe chemin" et "plus court chemin") avec des nouveaux cas afin d'assurer le bon fonctionnement de nos fonctions sur tous les automates.

Perspectives :
Un aspect que nous aurions aimé aborder aurait été l'utilisation réelle de notre programme pour rechercher un trajet sur une carte telle que Google maps ou autre. Prenons Google maps par exemple, l'environnement est un automate qui représenterai la carte, comportant un nombre d'adresses (états), les routes entre eux forment l'ensemble des transitions de l'automate. Une trajectoire dans une carte serait une séquence de k routes entre 2 adresses, pour sauver du temps nous pourrions très bien utiliser notre algorithme de recherche du plus court chemin qui pour un état de départ et d'arrive nous renvoie la plus courte trajectoire entre ces deux, ou bien utiliser notre algorithme existe chemin, pour voir si on peut passer par une certaine route pour arriver a notre destination.

ANNEXES : EXEMPLES

Automate déterministe > existe chemin :

Soit l'automate, ayant la table de transition suivante :

	a	b
0	0	1
1	0	2
2	3	*
3	*	*

On fait un appel de la fonction avec comme paramètre l'état 0, l'algorithme construira alors le tableau suivant :

Pointeur→ Etat 0	Pointeur → Etat 1

Puis on parcourt le tableau, on commence par la première case, elle ne contient pas l'état final 3, on n'a pas déjà traverse l'état 0 alors on le marque et on fait un appel récursif avec comme paramètre l'état 0, on construira ensuite le tableau suivant :

Pointeur→ Etat 0	Pointeur → Etat 2

On commence par la première case, elle ne contient pas l'état final, mais on a déjà traversé l'état 0 alors on passe à la prochaine case, elle ne contient pas l'état 3, et on n'a pas déjà traverse l'état 2, alors on fait un appel récursif de l'état 2 :

Pointeur→ Etat 3	NULL

La première case contient bien l'état final, on retourne alors true , le chemin existe. La deuxième fonction récupèrera alors ce résultat, et renvoi true.

Automate déterministe > plus petit chemin :
en marquant chaque fils par son père tant que l'état final n'a pas de père, ensuite remonter la chaine des pères à partir de l'état Final, jusqu'à obtenir l'état Initial et reverse la chaine de caractères afin de l'obtenir dans le bon ordre.

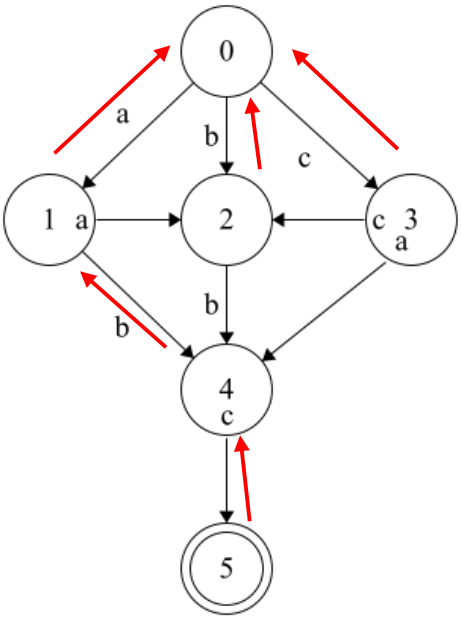
Expliquons cet algorithme au travers d'un exemple :

Soit l'automate :

- Les flèches rouges indiquent le marquage des pères
- Note : les états sont marqués uniquement au premier passage.

Nous obtenons alors la trace d'exécution suivante :

- 0 n'a pas de père
- 1 a pour père 0
- 2 a pour père 0
- 3 a pour père 0
- 4 a pour père 1
- 5 a pour père 4



Ensuite il nous reste seulement à remonter les états à l'aide de la fonction de transition. Nous partons de l'état final testons quelle transition ayant 4 pour état de départ et 5 pour état d'arrivée existe, une fois trouvé nous concaténons son étiquette au mot (initialement vide) et nous procédons de même avec le père de l'Etat final ainsi de suite.

Nous obtenons à la fin de cette étape le mot suivant : cba. Il nous suffit alors de reverse ce mot pour obtenir le plus petit chemin (plus exactement l'un des plus petit chemins) qui est abc.

Automate déterministe > afficher automate:

PGF/TikZ :

C’est une combinaison de deux langages informatiques pour la création de graphiques vectoriels. PGF est un langage de bas niveau, tandis que TikZ est un ensemble de macros qui fournit une syntaxe plus simple comparée à celle de PGF. Il a été créé en 2005 par Till Tantau qui est aussi le développeur principal de l'interpréteur de PGF et TikZ qui est écrit en TeX. PGF est le sigle de Portable Graphics Format, tandis que TikZ est un acronyme récursif de TikZ ist kein Zeichenprogramm. L'interpréteur de PGF/TikZ peut être utilisé depuis les paquets LaTeX.

ANNEXES : CODE SOURCE

Automate déterministe :

Automate.h

```
1.  #ifndef AUTOMATE_H
2.  #define AUTOMATE_H
3.
4.  #include "Etat.h"
5.  #include <map>
6.  #include <stdlib.h>
7.  #include <iostream>
8.  #include <string>
9.  #include <algorithm>
10. #include <vector>
11. #include <ctime>
12. #include <cmath>
13.
14. /*
15.   Automate représenté sous forme de quintuplet
16.   A = alphabet
17.   Q = enseble des etats
18.   D = ensemble etats de dédepart
19.   F = ensemble des etats etatsFinaux
20.   T = ensembleDesTransitions
21. */
22.
23. class Automate{
24. private:
25.   //quintuplet
26.   std::map< char, int> alphabet ;
27.   //char* alphabet; /*A*/
28.   char* alphabet2;
29.   Etat* ensembleGlobal; //A
30.   Etat* etatFinal; //A
31.   Etat* etatInitial; //A
32.   Etat*** ensembleTransitions; //A
33.
34.   int tailleEnsemleGlobal;
35.   int tailleAlphabet;
36.
37. public:
38. /* Constructeurs */
39.   Automate();
40.   Automate(char* n_alphabet ,int nbSalles,int nbLettresAlphabet);
41.
42. /*Algorithmes générations aléatoire */
43. Automate generationAleatoire1(int nbSalles , float densite, char* n_alphabet,int nbL
    ettresAlphabet);
44. Automate generationAleatoire2(int nbSalles , float densite, char* n_alphabet,int nbL
    ettresAlphabet);
45. Automate generationAleatoire3(int nbSalles , float densite, char* n_alphabet,int nbL
    ettresAlphabet);
46.
47. std::string motAleatoire(int longeurmot);
48. /* Fonctions */
49.   Etat* fonctionDeTransition( Etat a,char lettre);
50.   bool motReconnu(std::string mot);
51.
52.   bool existecheminInter(Etat a);
53.   bool existechemin();
54.   bool existeEF(Etat a);
55.   bool existeCheminEntre(Etat* depart);
```

```
/* set & getters */
  std::map< char, int> getAlphabet();
  int getId(char a);
  void setTransition(int etat , char lettre , int etatf);
  Etat* getEnsembleGlobal();
  Etat*** getEnsembleTransitions();
  void afficherAutomate();

/* Fonctions a finir */

  //std::string
  void pluspetitcheminInter(std::vector<Etat*> depart);
  std::string pluspetitchemin();

};
//Automate generationAleatoire1(int nbSalles , float densite, char* n_alphabet,int nbLettresAlphabet);

#endif
```

Automate.cpp

```
#include "automate.h"
#include <iostream>
#include <fstream>


using namespace std;

/* ----- Constructeurs ----- */

Automate::Automate(){
  //alphabet = NULL;
  alphabet2=NULL;
  ensembleGlobal = NULL;
  etatFinal = NULL;
  etatInitial =NULL;
  ensembleTransitions =NULL;
};

Automate::Automate(char* n_alphabet ,int nbSalles ,int nbLettresAlphabet){
  tailleAlphabet = nbLettresAlphabet;
  tailleEnsemleGlobal = nbSalles;

  alphabet2=n_alphabet;

  Etat* tab= new Etat[nbSalles];
  ensembleGlobal= tab;
  for(int i=0;i<nbSalles;i++){
    ensembleGlobal[i].setEtat(i);
  }
  alphabet = map< char, int>();
  for(int j=0;j< nbLettresAlphabet;j++){  // creation de l'alphabet ayant comme a ==> 0 etc...
    alphabet[n_alphabet[j]]=j;
  }


  ensembleTransitions = new Etat**[nbSalles];
  for (int i =0; i <nbSalles;i++){
    ensembleTransitions[i] = new Etat*[nbLettresAlphabet];
  }

  for(int i = 0; i<nbSalles;i++){
    for(int j=0; j<nbLettresAlphabet;j++){
```

```
43.       ensembleTransitions[i][j]= NULL;
44.     }
45.   }
46.   etatInitial = &ensembleGlobal[0];
47.   etatFinal = &ensembleGlobal[nbSalles-1];
48. }
49.
50. /* ------ Generation aléatoire -----------*/
51. Automate Automate::generationAleatoire1(int nbSalles , float densite, char* n_alphab
    et ,int nbLettresAlphabet){
52.   Automate retour(n_alphabet,nbSalles,nbLettresAlphabet);
53.   // densité = nbcouloirs/nbsalles => nbcouloirs = nbsalles * densité
54.   int nbTransitions = nbSalles*densite;
55.   int etatAleat1 , etatAleat2; int indiceLettre;
56.   Etat* ensembleGlobalRetour = retour.getEnsembleGlobal();
57.   srand(time(NULL));
58.   while(nbTransitions!= 0){
59.     indiceLettre=rand()%nbLettresAlphabet;
60.     etatAleat1=rand()%nbSalles; //cout <<"etatAleat1 : " << etatAleat1 << endl;
61.     etatAleat2=rand()%nbSalles; //cout <<"etatAleat2 : " << etatAleat2 << endl;
62.
63.     if(retour.fonctionDeTransition(ensembleGlobalRetour[etatAleat1],n_alphabet[indic
    eLettre])==NULL && etatAleat2!=0 && etatAleat1!=nbSalles-
    1 && (etatAleat1!=etatAleat2)){
64.       retour.setTransition(etatAleat1,n_alphabet[indiceLettre],etatAleat2);
65.       nbTransitions--;
66.     }
67.   }
68.   return retour;
69. }
70.
71. Automate Automate::generationAleatoire2(int nbSalles , float densite, char* n_alphab
    et,int nbLettresAlphabet){
72.   Automate retour(n_alphabet,nbSalles,nbLettresAlphabet);
73.   // densité = nbcouloirs/nbsalles => nbcouloirs = nbsalles * densité
74.   int nbTransitions = nbSalles*densite;
75.   int nbTransitionsSortantes, nbTransitionsEntrantes;
76.
77.   nbTransitionsSortantes = nbTransitions/2;
78.   nbTransitionsEntrantes= nbTransitions-nbTransitionsSortantes;
79.
80.   Etat* ensembleGlobalRetour = retour.getEnsembleGlobal();
81.   int etatAleat1 , etatAleat2; int indiceLettre;
82.   srand(time(NULL));
83.
84.   while(nbTransitionsSortantes!= 0 && nbTransitionsEntrantes!=0){
85.     indiceLettre=rand()%nbLettresAlphabet;
86.     etatAleat1=rand()%nbSalles;
87.     etatAleat2=rand()%nbSalles;
88.
89.     if(etatAleat1!=etatAleat2){
90.
91.       if(etatAleat1<etatAleat2 && retour.fonctionDeTransition(ensembleGlobalRetour[e
    tatAleat1],n_alphabet[indiceLettre])==NULL){
92.         if(etatAleat2!=0 && etatAleat1!=nbSalles-1){
93.           retour.setTransition(etatAleat1,n_alphabet[indiceLettre],etatAleat2);
94.           nbTransitionsSortantes--;
95.         }
96.       }
97.
98.       if(etatAleat2>etatAleat1 && retour.fonctionDeTransition(ensembleGlobalRetour[e
    tatAleat2],n_alphabet[indiceLettre])==NULL){
99.         if(etatAleat2!=(nbSalles-1) && etatAleat1!=0){
100.                retour.setTransition(etatAleat2,n_alphabet[indiceLettre],etatAleat1
    );
101.                nbTransitionsEntrantes--;
```

```
102.               }
103.             }
104.
105.           }
106.         }
107.         return retour;
108.       }
109.
110.       Automate Automate::generationAleatoire3(int nbSalles , float densite, char* n
    _alphabet ,int nbLettresAlphabet){ // a ameliorer
111.         Automate retour(n_alphabet,nbSalles,nbLettresAlphabet);
112.         // densité = nbcouloirs/nbsalles => nbcouloirs = nbsalles * densité
113.         int nbTransitions = nbSalles*densite;
114.         int etatAleat1 , etatAleat2; int indiceLettre;
115.         Etat* ensembleGlobalRetour = retour.getEnsembleGlobal();
116.         srand(time(NULL));
117.
118.         while(nbTransitions!= 0){
119.           indiceLettre=rand()%nbLettresAlphabet;
120.           etatAleat1=rand()%nbSalles; //cout <<"etatAleat1 : " << etatAleat1 << end
    l;
121.           etatAleat2=rand()%nbSalles; //cout <<"etatAleat2 : " << etatAleat2 << end
    l;
122.           if(nbTransitions == 1){
123.             if(retour.fonctionDeTransition(ensembleGlobalRetour[etatAleat1],n_alpha
    bet[indiceLettre])==NULL && etatAleat2!=0 && etatAleat1!=nbSalles-
    1 && (etatAleat1!=etatAleat2)){
124.               retour.setTransition(etatAleat1,n_alphabet[indiceLettre],nbSalles-
    1);
125.               nbTransitions--;
126.             }
127.           }
128.           else{
129.             if(retour.fonctionDeTransition(ensembleGlobalRetour[etatAleat1],n_alpha
    bet[indiceLettre])==NULL && etatAleat2!=0 && etatAleat1!=nbSalles-
    1 && (etatAleat1!=etatAleat2)){
130.               retour.setTransition(etatAleat1,n_alphabet[indiceLettre],etatAleat2);

131.               nbTransitions--;
132.             }
133.           }
134.         }
135.
136.         return retour;
137.       }
138.
139.       string Automate::motAleatoire(int longeurmot){
140.         srand(time(NULL));
141.         string mot = "";
142.         int nb;
143.         while(mot.size()<=longeurmot){
144.           nb = rand()%tailleAlphabet;
145.           mot.append(1,alphabet2[nb]);
146.         }
147.         return mot;
148.       }
149.
150.       /* ----- Fonctions ----- */
151.
152.       Etat* Automate::fonctionDeTransition( Etat a , char lettre){
153.         int t = a.getEtat();int idxlettre = alphabet[lettre];
154.         return ensembleTransitions[t][idxlettre];
155.       }
156.
157.       bool Automate::motReconnu(string mot){ //O(longeurmot) char* mot,int longeurm
    ot
```

```
158.         Etat* actuel = etatInitial;
159.         int longeurmot = mot.size();
160.         for(int i = 0 ;i<longeurmot;i++){
161.
162.           Etat actuelderef = *actuel;
163.
164.           Etat* pourVerif = fonctionDeTransition(actuelderef,mot[i]);
165.
166.           if(i == longeurmot-1){
167.               if(pourVerif==etatFinal){
168.                 return true;}
169.               else{return false;}}
170.           else{
171.              pourVerif = fonctionDeTransition(actuelderef,mot[i]);
172.              if(pourVerif != NULL){
173.
174.               actuel = pourVerif;
175.
176.             }
177.             else{return false;}}
178.         }
179.         return false;
180.       }
181.
182.       bool Automate::existecheminInter(Etat d){
183.         int j=0;
184.         if(existeEF(d)==false){
185.           Etat* T[tailleAlphabet];
186.           for(int i=0;i<tailleAlphabet;i++){
187.             T[i]=fonctionDeTransition(d,alphabet2[i]);
188.           }
189.           while(j<2 && (etatFinal->getSalle_Traverse()==false)){
190.             if(T[j]!=NULL){
191.
192.           if(T[j]==etatFinal){
193.             etatFinal->setSalle_Traverse(true);
194.           }
195.           if(T[j]->getSalle_Traverse()==false){
196.             T[j]->setSalle_Traverse(true);
197.             existecheminInter(*T[j]);
198.           }
199.             }
200.             j++;
201.           }
202.         }
203.
204.         if(etatFinal->getSalle_Traverse())
205.           return true;
206.
207.         else
208.           return false;
209.       }
210.
211.       bool Automate::existechemin(){
212.         bool res;
213.         res=existecheminInter(*etatInitial);
214.         if(res)
215.           return true;
216.         else
217.           return false;
218.       }
219.
220.       bool Automate::existeEF(Etat a){
221.         bool res=false;
222.         int i=0;
223.         while((i<tailleAlphabet) && !res){
```

```
224.           if(etatFinal==fonctionDeTransition(a,alphabet2[i])){
225.             etatFinal->setSalle_Traverse(true);
226.             res=true;
227.           }
228.           i++;
229.         }
230.         return res;
231.       }
232.
233.       void Automate::pluspetitcheminInter(vector<Etat*> depart){
234.         string petitchemin ="";
235.         vector<Etat*> etats;
236.
237.         int k;
238.         if(etatFinal->getPere()== NULL){
239.           k =0;
240.           for(int j =0 ;j<depart.size();j++){
241.             for(int i =0 ; i<tailleAlphabet;i++){
242.               //cout << j << "    " << i << endl;
243.
244.               if(fonctionDeTransition(*depart[j],alphabet2[i]) !=NULL){
245.                 etats.push_back(fonctionDeTransition(*depart[j],alphabet2[i]));
246.                 if(etats[k]!= NULL && etats[k]->getPere()==NULL){
247.                   etats[k]->setPere(depart[j]);
248.                 }
249.                 k++;
250.               }
251.             }
252.           }
253.           pluspetitcheminInter(etats);
254.         }
255.       }
256.
257.       string Automate::pluspetitchemin(){
258.         string petitchemin ="";
259.         vector<Etat*> initial;initial.push_back(etatInitial);
260.
261.
262.         if(existechemin()){
263.           this->pluspetitcheminInter(initial);
264.           /*
265.           for(int i =0 ; i < tailleEnsemleGlobal;i++){
266.             cout << ensembleGlobal[i].getEtat() << " a pour pere : " ;
267.             if(ensembleGlobal[i].getPere() != NULL){
268.               cout << ensembleGlobal[i].getPere()->getEtat()<<endl;
269.             }
270.             else{
271.               cout << "Pas de pere" << endl;
272.             }
273.           }
274.           */
275.           Etat* temp = etatFinal;
276.           while(etatInitial!=temp){
277.             for(int i =0 ; i<tailleEnsemleGlobal ;i++){
278.               for(int j =0 ; j< tailleAlphabet;j++){
279.                 if(fonctionDeTransition(ensembleGlobal[i],alphabet2[j]) == temp &&
    (&ensembleGlobal[i]==temp->getPere())){
280.                   petitchemin.push_back(alphabet2[j]);
281.                   temp = &ensembleGlobal[i];
282.                 }
283.               }
284.             }
285.           }
286.           reverse(petitchemin.begin(),petitchemin.end());// reverse petit chemin av
    ant de renvoyer
287.         }
```

```
288.
289.         return petitchemin;
290.
291.       }
292.
293.       void Automate::afficherAutomate(){
294.
295.         ofstream monFlux("AfficherAutomate.tex",ios::trunc);
296.
297.         if(monFlux){
298.
299.           monFlux << "\\documentclass[12pt]{article}" << endl;
300.           monFlux << "\\usepackage[english]{babel}" << endl;
301.           monFlux << "\\usepackage[utf8x]{inputenc}" << endl;
302.           monFlux << "\\usepackage{amsmath}" << endl;
303.           monFlux << "\\usepackage{tikz}" << endl;
304.           monFlux << "\\usetikzlibrary{arrows,automata}" << endl;
305.           monFlux << "\\begin{document}" << endl;
306.           monFlux << "\\begin{tikzpicture}[-
    >,>=stealth',shorten >=1pt,auto,node distance=4cm, scale = 1.1,transform shape]" <<
    endl;
307.
308.           string T[3]={"below right","below left","below"};
309.           if(tailleEnsemleGlobal <= 10){
310.             // faut parcourir l'ensemble des etats initiaux
311.             monFlux << "\\node[state,initial] ("<<etatInitial-
    >getEtat()<<")   {$1$};" << endl;
312.             // le reste des etats
313.             int j = 2; int m = 0;
314.             for(int i = 1; i < (tailleEnsemleGlobal-1); i++){
315.               if(m < 2){
316.                 if(i%2 != 0){
317.                 monFlux << "\\node[state]  ("<<ensembleGlobal[i].getEtat()<<") ["<<
    T[m]<<" of = "<<ensembleGlobal[i].getEtat()-1 <<"] {$"<<j<<"$};" << endl;
318.                 j++;
319.                 m++; }
320.                else{ monFlux << "\\node[state]  ("<<ensembleGlobal[i].getEtat()<<")
    ["<<T[m]<<" of = "<<ensembleGlobal[i].getEtat()-2 <<"] {$"<<j<<"$};" << endl;
321.               j++; m++;  } }
322.               else {monFlux << "\\node[state]  ("<<ensembleGlobal[i].getEtat()<<")
    ["<<T[m]<<" of = "<<ensembleGlobal[i].getEtat()-2 <<"] {$"<<j<<"$};" << endl;
323.              j++;}
324.
325.                   // if {monFlux << "\\node[state]  ("<<ensembleGlobal[i].getEtat()
    <<") ["<<T[m]<<" of = "<<ensembleGlobal[i].getEtat()-2 <<"] {$"<<j<<"$};" << endl;
326.                   // j++; }
327.             }
328.
329.             monFlux <<"\\node[state,accepting] ("<<etatFinal-
    >getEtat()<<") [below right of = "<<ensembleGlobal[tailleEnsemleGlobal-
    2].getEtat()<<"] {$"<<tailleEnsemleGlobal<<"$};" << endl;
330.
331.
332.             // faut parcourir l'ensemble des etats finaux.
333.             int supercool = 0;
334.             // faire les transitions.
335.             for(int k = 0; k < tailleEnsemleGlobal; k++){ //Parcourir l'ensemble de
    s transitions.
336.               for(int j=0 ; j < tailleAlphabet;j++){
337.                 if(fonctionDeTransition(ensembleGlobal[k],alphabet2[j]) != NULL &&
    supercool==0){
338.                   monFlux <<"\\path ("<<ensembleGlobal[k].getEtat()<< ") edge node
    {$" <<alphabet2[j]<<"$}("<<fonctionDeTransition(ensembleGlobal[k],alphabet2[j])-
    >getEtat()<<")" << endl;
339.                   supercool++;
340.                 }
```

```
341.                 else if(fonctionDeTransition(ensembleGlobal[k],alphabet2[j]) != NUL
    L && supercool!=0){
342.                   monFlux <<"   ("<<ensembleGlobal[k].getEtat()<< ") edge [bend lef
    t] node {$" <<alphabet2[j]<<"$}("<<fonctionDeTransition(ensembleGlobal[k],alphabet2[
    j])->getEtat()<<")" << endl;
343.
344.                 }
345.
346.               }
347.             }
348.             monFlux << ";" << endl;
349.             monFlux << "\\end{tikzpicture}" << endl;
350.             monFlux << "\\end{document}" << endl;
351.             monFlux.close();
352.             system("pdflatex AfficherAutomate.tex");
353.           }
354.         else {
355.           monFlux << "le nombre de salles doit etre inferieur à 10" << endl;
356.           monFlux.close();}
357.
358.         }
359.         else { cout << "ERREUR: Impossible d'ouvrir le fichier." << endl; }
360.       }
361.
362.       /* ----- set & getters -----*/
363.       std::map< char, int> Automate::getAlphabet(){return alphabet;}
364.
365.       int Automate::getId(char a){return alphabet[a];}
366.
367.       void Automate::setTransition(int idxEtat , char lettre , int idxEtatf){ // pb
    ici ?
368.         int idxlettre = alphabet[lettre];
369.         ensembleTransitions[idxEtat][idxlettre] = &(ensembleGlobal[idxEtatf]);
370.       }
371.
372.       Etat* Automate::getEnsembleGlobal(){return ensembleGlobal;}
373.
374.       Etat*** Automate::getEnsembleTransitions(){return ensembleTransitions;}
```

Etat.h

```
1.  #ifndef ETAT_H
2.  #define ETAT_H
3.
4.  //class Transition;
5.  #include <iostream>
6.  #include <string>
7.  //#include "transition.h"
8.
9.
10.
11. class Etat{
12. private:
13.   int etat;
14.   bool salle_traverse;
15.   Etat* pere;
16. public:
17.   Etat();
18.   Etat(int nom);
19.
20.   void setEtat(int a);
21.   int getEtat();
22.
23.   void setSalle_Traverse(bool b);
24.   bool getSalle_Traverse();
```

```
25.
26.   void setPere(Etat* n_pere);
27.   Etat* getPere();
28. };
29.
30. #endif
```

Etat.cpp

```
1.  #include "Etat.h"
2.  using namespace std;
3.
4.  Etat::Etat(int etat):etat(etat),salle_traverse(false),pere(NULL){};
5.
6.  Etat::Etat():etat(0),salle_traverse(false),pere(NULL){};
7.
8.  void Etat::setEtat(int a){
9.    this->etat = a;
10. }
11.
12. int Etat::getEtat(){
13.   return etat;
14. }
15.
16. void Etat::setSalle_Traverse(bool b){
17.   this->salle_traverse=b;
18. }
19.
20. bool Etat::getSalle_Traverse(){
21.   return salle_traverse;
22. }
23.
24. void Etat::setPere(Etat* n_pere){
25.   this->pere = n_pere;
26. }
27. Etat* Etat::getPere(){
28.   return pere;
29. }
```

Automate indéterministe :

Automate.h

```
1.  #ifndef AUTOMATE_H
2.  #define AUTOMATE_H
3.
4.  #include "Etat.h"
5.  #include <map>
6.  #include <stdlib.h>
7.  #include <iostream>
8.  #include <algorithm>
9.  #include <vector>
10. #include <string>
11. #include <fstream>
12. /*
13.   Automate représenté sous forme de quintuplet
14.   A = alphabet
15.   Q = enseble des etats
16.   D = ensemble etats de dédepart
17.   F = ensemble des etats etatsFinaux
18.   T = ensembleDesTransitions
19. */
20.
21. class Automate{
```

```
22. private:
23.   //quintuplet
24.   std::map< char, int> alphabet ;
25.   //char* alphabet; /*A*/
26.   char* alphabet2;
27.   Etat* ensembleGlobal; //A
28.   Etat* etatFinal; //A
29.   Etat* etatInitial; //A
30.   std::vector<Etat*>*** ensembleTransitions; //A tableau a deux dimensions de pointe
    urs de vector contenant des pointeurs d'etats
31.
32.   int tailleEnsemleGlobal;
33.   int tailleAlphabet;
34. public:
35. /* Constructeurs */
36.   Automate();
37.   Automate(char* n_alphabet ,int nbSalles,int nbLettresAlphabet);
38.
39.   /*Algorithmes générations aléatoire */
40.   Automate generationAleatoire1(int nbSalles , float densite, char* n_alphabet,int n
    bLettresAlphabet);
41.   Automate generationAleatoire2(int nbSalles , float densite, char* n_alphabet,int n
    bLettresAlphabet);
42.   Automate generationAleatoire3(int nbSalles , float densite, char* n_alphabet,int n
    bLettresAlphabet);
43.
44.   std::string motAleatoire(int longeurmot);
45.
46. /* Fonctions */
47.   std::vector<Etat*>* fonctionDeTransition(Etat a,char lettre);
48.   bool motReconnu(std::string mot);
49.   std::vector<Etat*> unionEtats(std::vector<Etat*> ensembleDepart, char lettre);
50.
51.   void existechemininter(std::vector<Etat*> ensembleDepart);
52.   bool existechemin();
53.
54.   void pluspetitcheminInter(std::vector<Etat*> depart);
55.   std::string pluspetitchemin();
56.   void afficherAutomate();
57.
58. /* set & getters */
59.   std::map< char, int> getAlphabet();
60.   int getId(char a);
61.   void setTransition(int etat , char lettre , int etatf);
62.   Etat* getEnsembleGlobal();
63.   std::vector<Etat*>*** getEnsembleTransitions();
64.
65.   bool estDansTransition(std::vector<Etat*> transition , Etat* eval);
66.
67.
68. };
69.
70. #endif
```

Automate.cpp

```
1.  #include "automate.h"
2.
3.  #include <iostream>
4.
5.  using namespace std;
6.
7.  /* ----- Constructeurs ----- */
8.
9.  Automate::Automate(){
```

```
  //alphabet = NULL;
  alphabet2=NULL;
  ensembleGlobal = NULL;
  etatFinal = NULL;
  etatInitial =NULL;
  ensembleTransitions =NULL;
};

Automate::Automate(char* n_alphabet ,int nbSalles ,int nbLettresAlphabet){
  //int nbLettresAlphabet = 2;//sizeof(n_alphabet[0])/sizeof(n_alphabet);
  tailleAlphabet = nbLettresAlphabet;
  tailleEnsemleGlobal = nbSalles;

  alphabet2=n_alphabet;

  Etat* tab= new Etat[nbSalles];
  ensembleGlobal= tab;
  //string a ="";
  for(int i=0;i<nbSalles;i++){
    //a =  to_string(i);
    //ensembleGlobal[i]= Etat(i);
    ensembleGlobal[i].setEtat(i);
    //cout << ensembleGlobal[i].getEtat() << endl; // affiche la bonne chose 0,1,2..
.
  }
  alphabet = map< char, int>();
  for(int j=0;j< nbLettresAlphabet;j++){  // creation de l'alphabet ayant comme a ==
> 0 etc...
    alphabet[n_alphabet[j]]=j;
  }

  //initialisation ensemble transitions
  ensembleTransitions = new vector<Etat*>**[nbSalles];
  for (int i =0; i <nbSalles;i++){
    ensembleTransitions[i] = new vector<Etat*>*[nbLettresAlphabet];
  }

  for(int i = 0; i<nbSalles;i++){
    for(int j=0; j<nbLettresAlphabet;j++){
      ensembleTransitions[i][j]= new vector<Etat*>(0); // tableau a NULL
    }
  }


  etatInitial = &ensembleGlobal[0];
  etatFinal = &ensembleGlobal[nbSalles-1];
}

/* Generations Aleatoires */

/* ------ Generation aléatoire ------------*/
Automate Automate::generationAleatoire1(int nbSalles , float densite, char* n_alphab
et ,int nbLettresAlphabet){
  Automate retour(n_alphabet,nbSalles,nbLettresAlphabet);
  // densité = nbcouloirs/nbsalles => nbcouloirs = nbsalles * densité
  int nbTransitions = nbSalles*densite;
  int etatAleat1 , etatAleat2; int indiceLettre;
  Etat* ensembleGlobalRetour = retour.getEnsembleGlobal();
  srand(time(NULL));
  while(nbTransitions!= 0){
    indiceLettre=rand()%nbLettresAlphabet;
    etatAleat1=rand()%nbSalles; //cout <<"etatAleat1 : " << etatAleat1 << endl;
    etatAleat2=rand()%nbSalles; //cout <<"etatAleat2 : " << etatAleat2 << endl;

    vector<Etat*> transition = *retour.fonctionDeTransition(ensembleGlobalRetour[eta
tAleat1],n_alphabet[indiceLettre]);
```

```
72.
73.     if(etatAleat2!=0 && etatAleat1!=nbSalles-
    1 && (etatAleat1!=etatAleat2) && !estDansTransition(transition,&ensembleGlobalRetour
    [etatAleat2])) {
74.       cout<< "teeest" << endl;
75.       retour.setTransition(etatAleat1,n_alphabet[indiceLettre],etatAleat2);
76.       nbTransitions--;
77.     }
78.   }
79.   return retour;
80. }
81.
82. Automate Automate::generationAleatoire2(int nbSalles , float densite, char* n_alphab
    et,int nbLettresAlphabet){
83.   Automate retour(n_alphabet,nbSalles,nbLettresAlphabet);
84.   // densité = nbcouloirs/nbsalles => nbcouloirs = nbsalles * densité
85.   int nbTransitions = nbSalles*densite;
86.   int nbTransitionsSortantes, nbTransitionsEntrantes;
87.
88.   nbTransitionsSortantes = nbTransitions/2;
89.   nbTransitionsEntrantes= nbTransitions-nbTransitionsSortantes;
90.
91.   Etat* ensembleGlobalRetour = retour.getEnsembleGlobal();
92.   int etatAleat1 , etatAleat2; int indiceLettre;
93.   srand(time(NULL));
94.
95.   vector<Etat*> transition;
96.   while(nbTransitionsSortantes!= 0 && nbTransitionsEntrantes!=0){
97.     indiceLettre=rand()%nbLettresAlphabet;
98.     etatAleat1=rand()%nbSalles;
99.     etatAleat2=rand()%nbSalles;
100.
101.          if(etatAleat1!=etatAleat2){
102.
103.            if(etatAleat1<etatAleat2){
104.               transition = *retour.fonctionDeTransition(ensembleGlobalRetour[etatA
    leat1],n_alphabet[indiceLettre]);
105.              if(etatAleat2!=0 && etatAleat1!=nbSalles-
    1 && !estDansTransition(transition,&ensembleGlobalRetour[etatAleat2]) ){
106.                retour.setTransition(etatAleat1,n_alphabet[indiceLettre],etatAleat2
    );
107.                nbTransitionsSortantes--;
108.              }
109.            }
110.
111.            if(etatAleat2>etatAleat1){
112.              transition = *retour.fonctionDeTransition(ensembleGlobalRetour[etatAl
    eat2],n_alphabet[indiceLettre]);
113.              if(etatAleat2!=(nbSalles-
    1) && etatAleat1!=0 && !estDansTransition(transition,&ensembleGlobalRetour[etatAleat
    1])){
114.                retour.setTransition(etatAleat2,n_alphabet[indiceLettre],etatAleat1
    );
115.                nbTransitionsEntrantes--;
116.              }
117.            }
118.
119.          }
120.        }
121.        return retour;
122.      }
123.
124.      Automate Automate::generationAleatoire3(int nbSalles , float densite, char* n
    _alphabet ,int nbLettresAlphabet){ // a ameliorer
125.        Automate retour(n_alphabet,nbSalles,nbLettresAlphabet);
126.        // densité = nbcouloirs/nbsalles => nbcouloirs = nbsalles * densité
```

```
127.         int nbTransitions = nbSalles*densite;
128.         int etatAleat1 , etatAleat2; int indiceLettre;
129.         Etat* ensembleGlobalRetour = retour.getEnsembleGlobal();
130.         srand(time(NULL));
131.
132.         while(nbTransitions!= 0){
133.           indiceLettre=rand()%nbLettresAlphabet;
134.           etatAleat1=rand()%nbSalles; //cout <<"etatAleat1 : " << etatAleat1 << end
    l;
135.           etatAleat2=rand()%nbSalles; //cout <<"etatAleat2 : " << etatAleat2 << end
    l;
136.
137.           vector<Etat*> transition = *retour.fonctionDeTransition(ensembleGlobalRet
    our[etatAleat1],n_alphabet[indiceLettre]);
138.
139.           if(nbTransitions == 1){
140.             if(etatAleat1!=nbSalles-
    1 && !estDansTransition(transition,&ensembleGlobalRetour[etatAleat2])){
141.               retour.setTransition(etatAleat1,n_alphabet[indiceLettre],nbSalles-
    1);
142.               nbTransitions--;
143.             }
144.           }
145.           else{
146.             if(etatAleat2!=0 && etatAleat1!=nbSalles-
    1 && (etatAleat1!=etatAleat2) && !estDansTransition(transition,&ensembleGlobalRetour
    [etatAleat2])){
147.               retour.setTransition(etatAleat1,n_alphabet[indiceLettre],etatAleat2);

148.               nbTransitions--;
149.             }
150.           }
151.         }
152.
153.         return retour;
154.       }
155.
156.       string Automate::motAleatoire(int longeurmot){
157.         srand(time(NULL));
158.         string mot = "";
159.         int nb;
160.         while(mot.size()<longeurmot){
161.           nb = rand()%tailleAlphabet;
162.           mot.append(1,alphabet2[nb]);
163.         }
164.         return mot;
165.       }
166.
167.
168.       /* ----- Fonctions ----- */
169.
170.       vector<Etat*>* Automate::fonctionDeTransition( Etat a , char lettre){
171.         int t = a.getEtat();int idxlettre = alphabet[lettre];
172.         return ensembleTransitions[t][idxlettre];
173.       }
174.
175.       vector<Etat*> Automate::unionEtats(vector<Etat*> ensembleDepart, char lettre)
    {
176.         vector<Etat*> inter; // vecteur a renvoyer contenant l'union
177.         vector<Etat*>* temp; // variable temp ou l'on stocke
178.         vector<Etat*> tempderef ;
179.
180.         for(int i =0 ;i<ensembleDepart.size();i++){ // parcours de l'ensemble des é
    tats de l'ensemble de départ
```

```
181.           if(fonctionDeTransition(*ensembleDepart[i],lettre)-
    >size() != 0){ // parcours des transitions existante si la trransition avec la lettr
    e existe
182.             temp = fonctionDeTransition(*ensembleDepart[i],lettre);
183.             tempderef = *temp;
184.             for(int j = 0 ; j<tempderef.size();j++){ // on a joute les elements a i
    nter si existe transition temp est donc une adresse d'un vector
185.               inter.push_back(tempderef[j]);
186.             }
187.           }
188.         }
189.
190.         // tout les elements ateignables sont dans inter , supprimons maintenant le
    s éléments en double
191.         sort( inter.begin(), inter.end() );
192.         inter.erase(unique(inter.begin(), inter.end()) , inter.end());
193.         //nous obtenons alors l'ensemble
194.         return inter;
195.       }
196.
197.       bool Automate::motReconnu(string mot){ //O(longeurmot)
198.
199.         vector<Etat*> actuel(0); actuel.push_back(etatInitial); // set a etat initi
    al;
200.         vector<Etat*> pourVerif;
201.
202.         int i =0 ;
203.         int longeurmot = mot.size();
204.         while(i<longeurmot){
205.           //cout << mot[i]<<endl;
206.           if(longeurmot == 1){
207.             actuel = unionEtats(actuel,mot[i]);
208.             for(int k= 0;k<actuel.size();k++){
209.               if(actuel[k]==etatFinal){
210.                 return true;
211.               }
212.             }
213.             return false;
214.           }
215.           if(i==longeurmot-1){ // verification de la dernière transition
216.             for(int k= 0;k<actuel.size();k++){
217.               if(actuel[k]==etatFinal){
218.                 return true;
219.               }
220.             }
221.             return false;
222.           }
223.           else{
224.             pourVerif = unionEtats(actuel,mot[i]);
225.             if(pourVerif.size()!=0){
226.               actuel = pourVerif;
227.             }
228.             else{return false;}}
229.             i++;
230.           }
231.           return false;
232.       }
233.
234.       void Automate::pluspetitcheminInter(vector<Etat*> depart){
235.         string petitchemin ="";
236.         vector<Etat*> etats;
237.         vector<Etat*> temp;
238.         vector<Etat*> transitionTemp;
239.
240.
241.         if(etatFinal->getPere()== NULL){
```

```
242.           for(int j =0 ;j<depart.size();j++){
243.             for(int i =0 ; i<tailleAlphabet;i++){
244.
245.               if(fonctionDeTransition(*depart[j],alphabet2[i])->size() !=0){
246.                 transitionTemp = *fonctionDeTransition(*depart[j],alphabet2[i]);
247.                 for(int k= 0 ;k< transitionTemp.size();k++){
248.                   if(transitionTemp[k]->getPere()==NULL){
249.                     transitionTemp[k]->setPere(depart[j]);
250.                   }
251.                   temp.push_back(transitionTemp[k]);
252.                 }
253.               }
254.             }
255.           }
256.           //elimination des doublons
257.           sort( temp.begin(), temp.end() );
258.           temp.erase(unique(temp.begin(), temp.end()) , temp.end());
259.           pluspetitcheminInter(temp);
260.         }
261.       }
262.
263.       string Automate::pluspetitchemin(){
264.         string petitchemin ="";
265.         vector<Etat*> initial;initial.push_back(etatInitial);
266.         vector<Etat*> ensPar;
267.
268.
269.           //cout << " bloque ? " << endl;
270.         if(existechemin()){
271.           this->pluspetitcheminInter(initial);
272.           Etat* temp = etatFinal;
273.           while(etatInitial!=temp){
274.             for(int i =0 ; i<tailleEnsemleGlobal ;i++){
275.               for(int j =0 ; j< tailleAlphabet;j++){
276.                 if(fonctionDeTransition(ensembleGlobal[i],alphabet2[j])-
    >size()!= 0){
277.                   ensPar = *fonctionDeTransition(ensembleGlobal[i],alphabet2[j]);
278.
279.                   for(int k =0;k<ensPar.size();k++){
280.                     if(ensPar[k]== temp && (&ensembleGlobal[i]==temp-
    >getPere())){
281.                       petitchemin.push_back(alphabet2[j]);
282.                       temp = &ensembleGlobal[i];
283.                     }
284.                   }
285.                 }
286.               }
287.             }
288.           }
289.           reverse(petitchemin.begin(),petitchemin.end());// reverse petit chemin av
    ant de renvoyer
290.         }
291.         return petitchemin;
292.
293.       }
294.
295.
296.       /* ----- set & getters -----*/
297.       std::map< char, int> Automate::getAlphabet(){return alphabet;}
298.
299.       int Automate::getId(char a){return alphabet[a];}
300.
301.       void Automate::setTransition(int idxEtat , char lettre , int idxEtatf){
302.         int idxlettre = alphabet[lettre];
303.         ensembleTransitions[idxEtat][idxlettre]-
    >push_back(&(ensembleGlobal[idxEtatf]));
```

```
304.       }
305.
306.       Etat* Automate::getEnsembleGlobal(){return ensembleGlobal;}
307.
308.       vector<Etat*>*** Automate::getEnsembleTransitions(){return ensembleTransition
    s;}
309.
310.
311.       void Automate::existechemininter(vector<Etat*> ensembleDepart){
312.         int tailleAlphabet = sizeof(alphabet2)/sizeof(char);
313.         vector<Etat*> ensembletemp; vector<Etat*>* pourVerif;
314.         vector<Etat*> pourVerifderef;
315.         vector<Etat*> ensembleNonTraverse;
316.
317.         for(int i=0 ; i<ensembleDepart.size();i++){
318.           for(int j = 0 ; j<tailleAlphabet;j++){
319.             pourVerif = fonctionDeTransition(*ensembleDepart[i],alphabet2[j]);
320.             pourVerifderef = *pourVerif;
321.             if(pourVerifderef.size() != 0){
322.               for(int k=0;k<pourVerifderef.size();k++){
323.                 ensembletemp.push_back(pourVerifderef[k]);
324.               }
325.             }
326.           }
327.           /*######## on supprime les doubles afin d'obtenir l'union ########*/
328.           sort( ensembletemp.begin(), ensembletemp.end() );
329.           ensembletemp.erase(unique(ensembletemp.begin(), ensembletemp.end()) , ens
    embletemp.end());
330.           for(int x =0 ;x<ensembletemp.size();x++){
331.             if(ensembletemp[x]==etatFinal){
332.               ensembletemp[x]->setSalle_Traverse(true);
333.             }
334.             else{
335.               if(ensembletemp[x]->getSalle_Traverse()==false){
336.                 ensembleNonTraverse.push_back(ensembletemp[x]);
337.                 ensembletemp[x]->setSalle_Traverse(true);
338.               }
339.             }
340.           }
341.           existechemininter(ensembleNonTraverse);
342.         }
343.       }
344.
345.       bool Automate::existechemin(){
346.         vector<Etat*> initial(0); initial.push_back(etatInitial);
347.         existechemininter(initial);
348.         return etatFinal->getSalle_Traverse();
349.       }
350.
351.       void Automate::afficherAutomate(){
352.
353.         ofstream monFlux("AfficherAutomate.tex",ios::trunc);
354.
355.         if(monFlux){
356.
357.           monFlux << "\\documentclass[12pt]{article}" << endl;
358.           monFlux << "\\usepackage[english]{babel}" << endl;
359.           monFlux << "\\usepackage[utf8x]{inputenc}" << endl;
360.           monFlux << "\\usepackage{amsmath}" << endl;
361.           monFlux << "\\usepackage{tikz}" << endl;
362.           monFlux << "\\usetikzlibrary{arrows,automata}" << endl;
363.           monFlux << "\\begin{document}" << endl;
364.           monFlux << "\\begin{tikzpicture}[-
    >,>=stealth',shorten >=0.5pt,auto,node distance=3cm, scale = 1,transform shape]" <<
    endl;
365.
```

```
366.           string T[3]={"below right","below left","below"};
367.           if(sizeof(ensembleGlobal)<= 10){
368.             // faut parcourir l'ensemble des etats initiaux
369.             monFlux << "\\node[state,initial] ("<<etatInitial-
    >getEtat()<<")   {$1$};" << endl;
370.             // le reste des etats
371.             int j = 2; int m = 0;
372.             for(int i = 1; i < tailleEnsemleGlobal-1; i++){
373.               if(m < 2){
374.                 if(i%2 != 0){
375.                 monFlux << "\\node[state]  ("<<ensembleGlobal[i].getEtat()<<") ["<<
    T[m]<<" of = "<<ensembleGlobal[i].getEtat()-1 <<"] {$"<<j<<"$};" << endl;
376.                 j++;
377.                 m++; }
378.                else{ monFlux << "\\node[state]  ("<<ensembleGlobal[i].getEtat()<<")
    ["<<T[m]<<" of = "<<ensembleGlobal[i].getEtat()-2 <<"] {$"<<j<<"$};" << endl;
379.               j++; m++;  } }
380.               else {monFlux << "\\node[state]  ("<<ensembleGlobal[i].getEtat()<<")
    ["<<T[m]<<" of = "<<ensembleGlobal[i].getEtat()-2 <<"] {$"<<j<<"$};" << endl;
381.              j++;}
382.
383.                   // if {monFlux << "\\node[state]  ("<<ensembleGlobal[i].getEtat()
    <<") ["<<T[m]<<" of = "<<ensembleGlobal[i].getEtat()-2 <<"] {$"<<j<<"$};" << endl;
384.                   // j++; }
385.             }
386.
387.             monFlux <<"\\node[state,accepting] ("<<etatFinal-
    >getEtat()<<") [below right of = "<<ensembleGlobal[tailleEnsemleGlobal-
    2].getEtat()<<"] {$"<< tailleEnsemleGlobal<<"$};" << endl;
388.
389.
390.             // faut parcourir l'ensemble des etats finaux.
391.             int supercool = 0;
392.             // faire les transitions.
393.             for(int k = 0; k < tailleEnsemleGlobal; k++){ //Parcourir l'ensemble de
    s transitions.
394.               for(int j=0 ; j < tailleAlphabet;j++){
395.                 if(fonctionDeTransition(ensembleGlobal[k],alphabet2[j])-
    >size() != 0 ){
396.                   for(unsigned int i=0; i< fonctionDeTransition(ensembleGlobal[k],a
    lphabet2[j])->size(); i++){
397.
398.                     if(supercool == 0){
399.                       monFlux <<"\\path ("<<ensembleGlobal[k].getEtat()<< ") edge [
    bend right] node {$" <<alphabet2[j]<<"$}("<<(fonctionDeTransition(ensembleGlobal[k],
    alphabet2[j]))->at(i)->getEtat()<<")" << endl;
400.                       supercool++;
401.
402.                   }
403.                 else { monFlux <<"   ("<<ensembleGlobal[k].getEtat()<< ") edge [ben
    d left] node {$" <<alphabet2[j]<<"$}("<<(fonctionDeTransition(ensembleGlobal[k],alph
    abet2[j]))->at(i)->getEtat()<<")" << endl; i++;}
404.               }
405.                 // else if(fonctionDeTransition(ensembleGlobal[k],alphabet2[j]) !=
    NULL){
406.                 //   for(unsigned int i=0; i<(fonctionDeTransition(ensembleGlobal[k
    ],alphabet2[j]))->size(); i++){
407.                 //   monFlux <<"   ("<<ensembleGlobal[k].getEtat()<< ") edge [bend
    left] node {$" <<alphabet2[j]<<"$}("<<(fonctionDeTransition(ensembleGlobal[k],alphab
    et2[j]))->at(i)->getEtat()<<")" << endl;
408.
409.               }}}
410.
411.
412.
413.             monFlux << ";" << endl;
```

```
414.             monFlux << "\\end{tikzpicture}" << endl;
415.             monFlux << "\\end{document}" << endl;
416.             monFlux.close();
417.             system("pdflatex AfficherAutomate.tex");
418.           }
419.         else {
420.           monFlux << "le nombre de salles doit etre inferieur à 10" << endl;
421.           monFlux.close();}
422.
423.         }
424.         else { cout << "ERREUR: Impossible d'ouvrir le fichier." << endl; }
425.
426.       }
427.
428.       bool Automate::estDansTransition(vector<Etat*> transition , Etat* eval){
429.         for (int i = 0; i < transition.size(); i++) {
430.           if(transition[i]==eval){
431.             return true;
432.           }
433.         }
434.         return false;
435.
436.       }
```

Etat.h

```
1.  #ifndef ETAT_H
2.  #define ETAT_H
3.
4.  //class Transition;
5.  #include <iostream>
6.  #include <string>
7.  //#include "transition.h"
8.
9.
10.
11. class Etat{
12. private:
13.   int etat;
14.   bool salle_traverse;
15.   Etat* pere;
16. public:
17.   Etat();
18.   Etat(int nom);
19.
20.   void setEtat(int a);
21.   int getEtat();
22.
23.   void setSalle_Traverse(bool b);
24.   bool getSalle_Traverse();
25.
26.   void setPere(Etat* n_pere);
27.   Etat* getPere();
28. };
29.
30. #endif
```

Etat.cpp

```
1.  #include "Etat.h"
2.  using namespace std;
3.
4.  Etat::Etat(int etat):etat(etat),salle_traverse(false),pere(NULL){};
5.
```

```
6.  Etat::Etat():etat(0),salle_traverse(false),pere(NULL){};
7.
8.  void Etat::setEtat(int a){
9.    this->etat = a;
10. }
11.
12. int Etat::getEtat(){
13.   return etat;
14. }
15.
16. void Etat::setSalle_Traverse(bool b){
17.   this->salle_traverse=b;
18. }
19.
20. bool Etat::getSalle_Traverse(){
21.   return salle_traverse;
22. }
23.
24. void Etat::setPere(Etat* n_pere){
25.   this->pere = n_pere;
26. }
27. Etat* Etat::getPere(){
28.   return pere;
29. }
```