École Publique d'Ingénieurs en 3 ans

Rapport de projet

CONCEPTION D'INTERFACE GRAPHIQUE JEU DE AKARI

le 20 mars 2023,

version 1.1

David Guo
david.guo@ecole.ensicaen.fr

Alexandre Ninassi
alexandre.ninassi@ensicaen.fr

Thomas Seng
thomas.seng@ecole.ensicaen.fr

Sebastien Fourey
sebastien.fourey@ensicaen.fr

www.ensicaen.fr

Remerciements

Je voudrais également remercier chaleureusement le professeur encadrant de TP, Alexandre Ninassi, pour son soutien et ses précieux commentaires tout au long de mes travaux pratiques. Votre expertise et votre disponibilité ont été d'une aide précieuse pour la mise en place de mes expériences et la collecte de données.

Enfin, je tiens à remercier mon binôme de TP, Thomas Seng, pour notre collaboration fructueuse tout au long de ce projet. Votre implication, votre rigueur et votre esprit d'équipe ont grandement contribué à la qualité de nos résultats et à la réussite de ce travail.

Table des matières

Remerciements 2

I Cadre du projet et méthodologie pour la gestion de projet 5

1. Position du problème 7
2. Méthode utilisée pour la gestion de projet 7
3. Identification des acteurs 8

II Analyse et spécification des besoins 9

4. Analyse des risques 11
5. Besoins fonctionnels 11
6. Besoins non-fonctionnels 12

III Réalisation 13

7. Architecture logicielle de la solution 15
8. Environnement de développement logiciel 15
9. Implémentation des caractéristiques 15
   - 9.1. Enumération *state* 15
   - 9.2. Modèle 16
     - 9.2.1. Classe GridModel 16
     - 9.2.2. Classe CommandHistory 17
   - 9.3. Vue 18
     - 9.3.1. Classe GridView 18
     - 9.3.2. Classe Box 19
   - 9.4. Présentateur 19
   - 9.5. Widget Stopwatch 20
   - 9.6. Améliorations apportées depuis la présentation 21

IV Conclusion 23

10. Analyse des résultats produits 25
11. Bilan du projet 26
12. Suite du projet 27

Table des figures

1	Lancement de l’application	25
2	En appuyant sur le bouton "Start"	25
3	Grille 10 x 10	25
4	Grille 14 x 14	25
5	Possibilité de mettre des lampes, indications visuelles des cases noires validées (chiffre en vert) et des lampes mal placées (lampes rouges)	25
6	Message "Règles"	25
7	Message "A propos"	26

Première partie

Cadre du projet et méthodologie pour la gestion de projet

1. Position du problème

L'objectif du projet est de programmer une interface permettant de jouer au Akari. Un Akari est un jeu mathématique de logique sous forme d'une grille comprenant des cases noires, parfois avec des valeurs, et des cases blanches. Le but du jeu est d'éclairer chaque case blanche en plaçant des lampes aux endroits adéquats.

Les règles du jeu sont les suivantes :

— Chaque case blanche doit être éclairée.

— Chaque lampe diffuse verticalement et horizontalement un rayon lumineux qui éclaire toutes les cases blanches de sa ligne et de sa colonne jusqu'à ce qu'il atteigne une case noire.

— Une case noire ne peut pas être éclairée, ni contenir une lampe.

— Une case noire arborant une valeur doit vérifier la condition suivante : le nombre de cases adjacentes (horizontalement ou verticalement) contenant une lampe doit être égal à la valeur affichée sur la case. Cette valeur est toujours comprise entre 0 et 4.

— Une lampe ne peut pas en éclairer une autre (le rayon lumineux diffusé par une lampe ne peut pas atteindre une case occupée par une autre lampe).

2. Méthode utilisée pour la gestion de projet

Nous avons fait face à un projet dans lequel nous devions apporté une note personnelle.

Parmi les méthodes itératives, nous avons utilisé une des méthodes agiles, qui sont celles utilisées par les ingénieurs logiciel de manière générale. La méthode agile assure une meilleure communication avec le client et une certaine visibilité du produit livrable.

Parmi les méthodes agiles, nous avons utilisé la méthode scrum.

L'un des piliers de la méthode scrum est de développer un logiciel de manière incrémentale en listant de manière transparente une liste de tâches à effectuer.

Un autre pilier de cette méthode est le suivant : plus nous avançons dans le projet, plus le logiciel est complet, et plus il possède de fonctionnalités.

3. Identification des acteurs

La réalisation d'un projet débute par la détermination des différentes parties prenantes. Nous y distinguons notamment :

- Utilisateur : Alexandre Ninassi
- Client : Alexandre Ninassi
- Chef de projet : Thomas Seng
- Gestionnaire de version : David Guo
- Developpeurs : Thomas Seng, David Guo

Deuxième partie

Analyse et spécification des besoins

4. Analyse des risques

Risque	Gravité	Mesure de prévention
Risque humain : Absentéisme des développeurs dans le groupe : lié à la période de l'année (maladies)	Importante	Prévoir le planning des séances en amont
Risque humain : Conflits dans le groupe	Limitée	Organiser un événement en commun.
Risque intrinsèque au projet : mauvaise affectation des tâches	Importante	Connaitre les forces et les faiblesses de chacun
Risque technique : Absence du gestionnaire de version	Négligeable	Affecter plusieurs gestionnaire de version au projet

5. Besoins fonctionnels

Voici les exigences du client sur le jeu :

- Choix de la taille et du niveau de difficulté
- Indication des lampes mal placées
- Chronomètre indiquant le temps de jeu
- Mise en évidence des cases noires ayant le bon nombre de lampes voisines
- Indication chiffrée du nombre de cases illuminées et du nombre requis.
- Indication chiffrée du nombre de cases noires ayant un nombre correct de lampes voisines, par rapport au nombre à atteindre.
- Décompte du nombre de retours en arrière effectués par le joueur
- Ergonomie et Convivialité : l'application fournira une interface conviviale et simple d'utilisation, qui ne requiert pas de pré-requis, c'est-à-dire qu'elle pourra être utilisée par n'importe qui (même des personnes n'ayant pas suivi de cours d'informatique)

6. Besoins non-fonctionnels

— Extensibilité : l'architecture de l'application permettra l'évolution et la maintenance (ajout ou suppression des fonctionnalités) d'une manière flexible, tel que le requièrent les méthodes agiles

Troisième partie

Réalisation

7. Architecture logicielle de la solution

Durant ce projet, nous avons adopté une architecture de type Modèle - Vue - Présentateur (MVP). Elle consiste à distinguer trois entités distinctes qui sont :

- Le modèle qui est totalement centré sur la logique métier
- La vue qui concerne la gestion graphique sans connaissance du modèle
- Le présentateur qui contient toute la logique de présentation et qui fait le pont entre le modèle et la vue

Ces trois entités distinctes jouent un rôle précis dans l'interface

Pour executer l'application, il faut se mettre à la racine du projet et taper la commande suivante :

```
./tp4/build-Akari-Desktop-Debug/Akari
```

8. Environnement de développement logiciel

Pour pouvoir développer notre logiciel, nous avons utilisé :

- Linux comme environnement de travail
- Qt Creator pour le développement de l'application
- Qt Linguist pour l'internationalisation
- TexMaker et Overleaf pour la rédaction de ce rapport

9. Implémentation des caractéristiques

9.1. Enumération *state*

Nous avons défini un type énuméré *state* dans un fichier State.h :

- Off
- On
- Black
- Black_0
- Black_1
- Black_2
- Black_3

— Black_4
— Black_0_checked
— Black_1_checked
— Black_2_checked
— Black_3_checked
— Black_4_checked
— Light
— Red_light

9.2. Modèle

Le modèle est composée de deux classes distinctes : GridModel.cpp et CommandHistory.cpp

9.2.1. Classe GridModel

La classe GridModel possède la logique métier du jeu Akari :

— la méthode **coordToIndex** permet de convertir les coordonnées d'une case dans la grille en un indice, unique dans la QMultiHash. Elle prend en paramètre les coordonnées de la case.
— la méthode **displayGrid** affiche la grille sur la sortie standard.
— la fonction **loadMap** charge une carte de jeu à partir d'une chaîne de caractères représentant la grille. Elle initialise les compteurs utiles à l'affichage, parcourt chaque caractère de la chaîne et, en fonction de sa valeur, modifie le type de case correspondant dans la grille.
— la méthode **isABlackCase** permet de vérifier si la case courante est une case noire, avec ou sans valeur.
— la méthode **isThereALightInTheRowUntilWall** vérifie s'il y a une lumière dans la ligne jusqu'aux murs ou premières cases noires rencontrées.
— la méthode **isThereALightInTheColumnUntilWall** vérifie s'il y a une lumière dans la colonne jusqu'aux murs ou premières cases noires rencontrées.
— la fonction **clickOnBox** prend en paramètres les coordonnées d'une case de la grille. Si la case n'est pas une case noire, elle vérifie si elle contient une lampe. Si la case n'est ni une lampe, ni une lampe rouge, son état devient "Light". Sinon, elle repasse à l'état "Off". Ensuite, elle rafraîchit la grille pour actualiser les différentes cases en fonction du nouveau contexte.
— la méthode **refreshBox** met à jour l'état de cette case en fonction de la présence ou non d'une lumière dans la colonne ou la ligne correspondante jusqu'au mur.
— la méthode **refreshLight** permet de rafraîchir une case qui contient une lampe, afin de savoir s'il s'agit d'une lampe normale ou d'une lampe rouge.
— la méthode **refreshBlackBoxWithValue** met à jour une case noire avec une valeur. Si la condition de la case noire avec valeur est vérifiée, son état change pour signifier que la condition est

validée. Sinon, l'état de la case est ou devient "non vérifiée".

- la méthode **refreshGrid** met à jour la grille en parcourant chaque case et en appelant les méthodes refreshCase(), refreshLight() et refreshBlackCaseWithValue() en fonction du type de case.
- la fonction **checkGrid** vérifie si la grille est correctement remplie en parcourant chaque case.
- la méthode **isABlackBoxWithValue** vérifie si une case est une case noire avec une valeur.
- la méthode **checkBlackBoxWithValueCondition** vérifie si la condition d'une case noire avec une valeur est respectée.
- la méthode **isALight** vérifie si une case contient une lampe, normale ou rouge.
- la fonction **manualChange** permet de changer manuellement l'état d'une case en lui attribuant une nouvelle valeur. Il s'agit d'une fonction pour les tests.
- les fonctions **getOnBoxesNumber**, **getCheckedBlackBoxesNumber**, **getEmptyBoxesNumber** et **getBlackBoxesWithValueNumber** retournent le nombre de cases allumées, le nombre de cases noires vérifiées, le nombre de cases éteintes et le nombre de cases noires avec une valeur respectivement.
- la fonction **getGrid** retourne une copie du tableau de la grille sous la forme d'un QMultiHash, qui est une structure de données Qt permettant de stocker des paires clé/valeur multiples
- la fonction **getGridChanges** renvoie uniquement les cases où un changement d'état a été effectué, avec les nouveaux états correspondant
- la fonction **getValueBox** permet d'obtenir la valeur d'une case

9.2.2. Classe CommandHistory

La classe CommandHistory permet d'implémenter les fonctionnalités de défaire et refaire une action

- la fonction **pushUndo** ajoute une QPair, représentant les coordonnées de la case sur laquelle le joueur clique, à la liste de commandes des annulations possibles.
- la fonction **popUndo** extrait la dernière commande enregistrée dans la liste d'annulations, l'ajoute à la fin de la liste des commandes annulées et retourne cette commande. Cette fonction est utilisée pour annuler la dernière action effectuée dans la grille du jeu.
- la fonction **popRedo** permet de récupérer la dernière commande de la liste des annulations possibles.
- la fonction **isUndoListEmpty** retourne true si la liste des commandes "undo" est vide et false sinon.
- la fonction **isRedoListEmpty** retourne true si la liste des commandes "redo" est vide et false sinon.

9.3. Vue

La vue est composée de deux classes distinctes : GridView.cpp et Box.cpp

9.3.1. Classe GridView

La classe GridView, permet de dessiner et mettre à jour la grille affichée dans l'interface :

- la fonction **setSize** permet de définir la taille de la grille. Elle prend en paramètre un entier "size".
- la fonction **coordToIndex** permet de convertir les coordonnées d'une case dans la grille en un index unique dans un tableau à une dimension. Elle prend en paramètre les coordonnées de la case.
- la fonction **generateGrid** qui permet de dessiner une grille à l'aide de la classe Case. Une grille est un agrégat d'instances de Box mis dans un QGridLayout.
- la fonction **refreshGrid** qui permet de mettre à jour la grille lorsqu'une modification est appliquée à celle-ci.
- la fonction **displayGrid** qui affiche une grille dans la console.
- la fonction **resizeEvent** qui est appelée chaque fois que la fenêtre contenant la grille est redimensionnée.
- la fonction **handleClickOnBox** est appelée lorsque nous cliquons sur une case de la grille. Cette méthode prend en paramètre les coordonnées de la case cliquée.
- la fonction **getBox** permet de récupérer un objet "Box" à partir de ses coordonnées dans la grille. Cette méthode prend en paramètre les coordonnées de la case cliquée.

9.3.2. Classe Box

La classe Box permet de dessiner une case :

— la fonction **setState** permet de définir l'état d'une case. Elle prend en paramètre un type *state* qui représente le nouvel état de la case.
— la fonction **getState** permet de récupérer l'état actuel d'une case.
— la fonction **paintEvent** permet de dessiner la représentation de la case, en fonction de son état, dans l'interface de jeu.
— la fonction **drawBlackSquareWithNumber** est utilisée pour dessiner un carré noir avec un numéro de couleur spécifiée à l'intérieur.
— la fonction **drawWhiteSquare** est utilisée pour dessiner un carré blanc.
— la fonction **drawYellowSquare** est utilisée pour dessiner un carré jaune.
— la fonction **drawYellowSquareWithPicture** est utilisée pour dessiner un carré jaune avec une image spécifiée à l'intérieur (lampe normale ou lampe rouge).
— La fonction **mousePressEvent** est appelée chaque fois que nous cliquons sur la case avec la souris et émet un signal clickedWithPosition avec la position de la case.

9.4. Présentateur

Le fichier MainWindow.cpp nous sert de présentateur, il possède les éléments suivants :

— le slot **onRules** permet d'afficher les règles du jeu lorsque nous cliquons sur l'onglet "Rules" ou "Règles"
— le slot **onAbout** permet d'afficher divers informations sur les créateurs de ce logiciel lorsque nous cliquons sur l'onglet correspondant
— le slot **onQuitAbout** permet d'afficher une fênetre de confirmation pour quitter l'application lorsque nous cliquons sur l'onglet correspondant
— le slot **onStartClick** permet de commencer le chronomètre lorsque nous cliquons sur le bouton "Start"
— le slot **handleClick** est appelée lorsque nous cliquons sur une case de la grille de jeu.
— le slot de test **onTestClick** est appelé lorsque nous cliquons sur un bouton de test dans l'interface graphique. Elle simule une série de clics de l'utilisateur sur des cases de la grille, et met à jour l'état de l'application en conséquence (cette fonctionnalité est destinée à l'examinateur pour la présentation de notre projet). Il faut donc décommenter une ligne qui symbolise la carte dans le slot onStart, et décommenter l'intégration dans l'interface graphique du bouton Test dans le constructeur de la MainWindow.
— le slot **onResetClick** permet d'enlever toutes les actions d'un utilisateur sur la grille.
— le slot **undo** permet d'annuler une action faite par l'utilisateur.

— le slot **redo** permet de refaire une action annulée par l'utilisateur
— le slot **onCheckClick** permet de vérifier si une grille a été correctement remplie par l'utilisateur ou non lorsque nous cliquons sur le bouton "Check"
— la méthode **readFile** lit un fichier texte contenant une carte de jeu. Le chemin du fichier est construit en fonction de la taille de la carte et de la difficulté du jeu. Le fichier est ouvert en mode lecture, puis un objet QTextStream est créé pour lire le contenu du fichier ligne par ligne.
— la méthode **getSize** renvoie la taille sélectionnée dans un objet QComboBox nommé textitsizeComboBox. La taille est stockée dans l'objet itemData du QComboBox, qui est une valeur associée à l'élément sélectionné dans le QComboBox. Cette valeur est retournée sous forme d'un objet QVariant qui peut contenir différents types de données.
— la méthode **getDifficultyString** renvoie une chaîne de caractères représentant la difficulté sélectionnée dans une autre méthode **getDifficulty**. La valeur de difficulté est convertie en entier à l'aide de la fonction **toInt** et est utilisée dans une instruction switch.
— la méthode **refreshStatus** met à jour l'affichage du statut de la grille de jeu dans l'interface graphique.
— la méthode **closeEvent** est appelée lorsque l'utilisateur essaie de fermer la fenêtre principale de l'application. La fonction enregistre les dernières dimensions de la fenêtre dans les paramètres de l'application en utilisant la classe QSettings. La méthode **setValue** de la classe QSettings est utilisée pour enregistrer la clé "size" avec la valeur de la méthode **size**, qui renvoie la taille actuelle de la fenêtre. Cette taille est enregistrée sous forme d'objet QSize contenant la largeur et la hauteur de la fenêtre.

9.5. Widget Stopwatch

Nous avons fait un chronomètre de sorte à ce qu'il soit réutilisable et extensible (une sorte d'équivalent de JavaBeans). Un singleton est implémentée dans cette classe, afin de ne pas avoir plusieurs instance du chronomètre

— la fonction **getInstance** permet d'avoir une unique instance de ce chronomètre
— la fonction **updateDisplay** permet de raffraichir la vue du chronomètre
— la fonction **start** permet de commencer le chronomètre (d'un point de vue logique)
— la fonction **stop** permet d'arrêter le chronomètre (d'un point de vue logique)

9.6. Améliorations apportées depuis la présentation

Nous avons apporté des modifications dans notre code depuis la présentation ayant eu lieu durant la dernière séance :

- Dans la première version, toute la grille était envoyé du modèle à la vue. A présent, la QMultiHash transmise ne comporte plus que les cases de la ligne et de la colonne (jusqu'aux premières cases noires comprises, ou jusqu'aux bords de la grille) de la case cliquée.
- Ajout des bulles d'information sur les boutons situés sous la grille.
- Suppression du menu Options qui était vide
- Ajout d'un message très synthétique décrivant les règles du jeu.
- Ajout des raccourcis sur les boutons suivants :
  - "Vérifier !" : Ctrl+C (Check !)
  - "Commencer !" : Ctrl+S (Start !)
  - "Réinitialiser !" : Ctrl+R (Reset !)

  (Le raccourci Ctrl+Q était déjà présent)

Quatrième partie

Conclusion

10. Analyse des résultats produits

Voici quelques illustrations de notre logiciel final

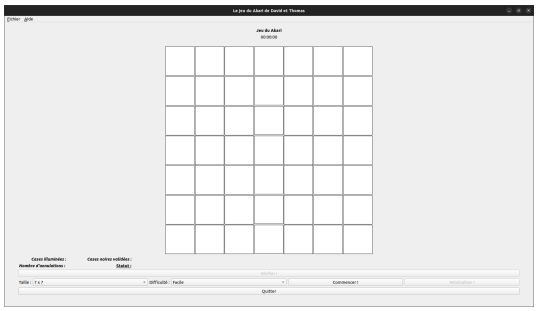

Figure 1. *Lancement de l'application*

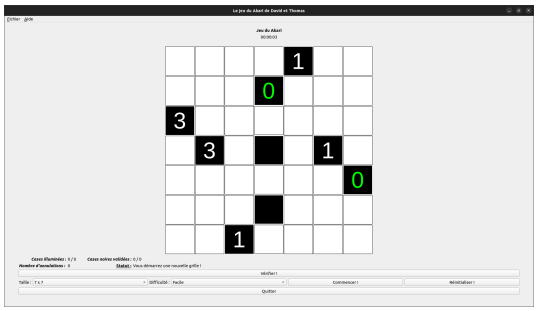

Figure 2. *En appuyant sur le bouton "Start"*

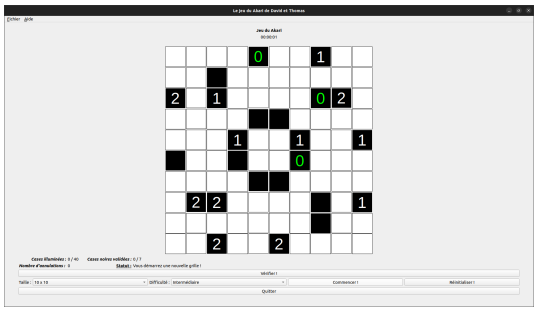

Figure 3. *Grille 10 x 10*

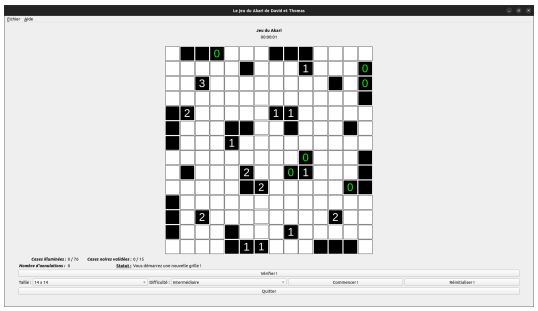

Figure 4. *Grille 14 x 14*

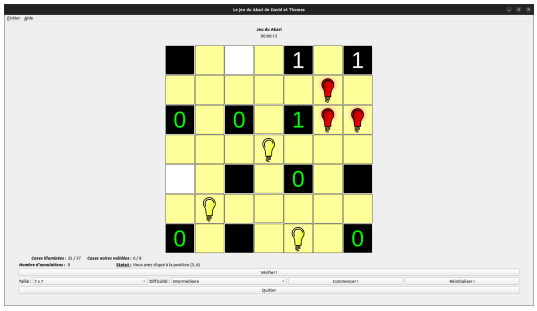

Figure 5. *Possibilité de mettre des lampes, indications visuelles des cases noires validées (chiffre en vert) et des lampes mal placées (lampes rouges)*

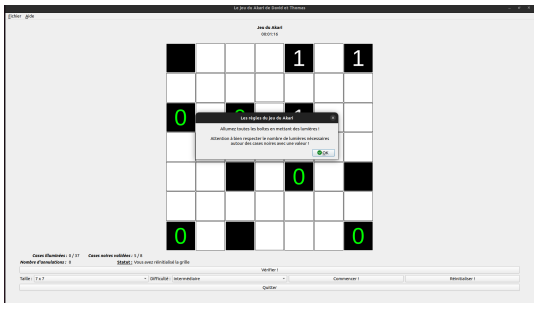

Figure 6. *Message "Règles"*

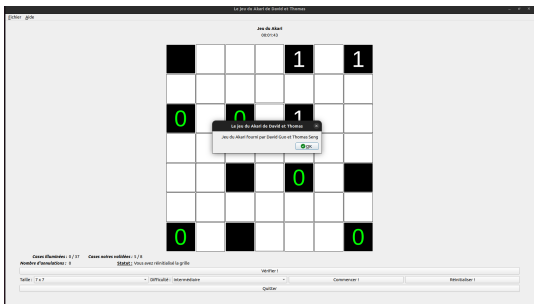

Figure 7. *Message "A propos"*

11. Bilan du projet

Les objectifs que nous nous étions initialement fixés ont tous été remplis. Notre jeu est fonctionnel, notre application est ergonomique, responsive et internationalisée, toutes les fonctionnalités minimales attendues par les enseignants sont présentes, et des fonctionnalités optionnelles (également précisées par les enseignants) ont été ajoutées.

L'architecture Modèle-Vue-Présentateur a été, à notre sens, respectée au mieux.

Nous avons étudié des patrons de conception pour les reproduire comme le singleton pour le chronomètre, ou produire des versions simplifiées de ceux-ci, comme le patron Commande pour intégrer les fonctionnalités **undo** et **redo**.

Les remarques de notre examinateur Alexandre Ninassi lors de la présentation durant la dernière séance de TP ont été prise en considération : les corrections évoquées ont été réalisées et les fonctionnalités manquantes implémentées.

Nous avons renforcé notre maîtrise du langage Qt en mettant en pratique les enseignements vus en cours, et en découvrant de nouveaux outils lors de recherches personnelles.

Nous sommes particulièrement satisfaits de notre travail sur l'implémentation de la grille, se faisant par l'introduction d'une QGridLayout qui se remplit d'instances de Box.

Nous avons su répartir efficacement la charge de travail entre les deux membres de notre binôme, tout en informant l'autre de nos avancées.

12. Suite du projet

Voici quelques idées pour poursuivre ce projet :

— Implémenter un thème sombre

— Ajouter une IA solveur du jeu

— Améliorer esthétiquement l'interface de jeu

— Ajouter une fênetre Menu

— Ajouter une page d'accueil

— Ajouter des icônes

— Implémenter les états Croix sur cases éteintes et Croix sur cases allumées

BIBLIOGRAPHIE

[1] Sujet Akari : https ://foad.ensicaen.fr/mod/page/view.php ?id=30649

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

École Publique d'Ingénieurs en 3 ans

6 boulevard Maréchal Juin, CS 4505

14050 CAEN cedex 04

ministère éducation nationale enseignement supérieur recherche

Normandie Université

Cti Commission des titres d'ingénieur

cdefi Conférence des Directeurs des Écoles Françaises d'Ingénieurs

European Accreditation of Engineering Programmes EUR-ACE®

CONFÉRENCE DES GRANDES ÉCOLES