SAE S1 02

Comparaison d'approches algorithmiques

CRAZY CIRCUS

Groupe 7

Amina El Houari 106

Badis Rahli 101

Table des matières

Brève présentation du projet :

Ce projet dans le cadre de la SAE S102, avait pour but de réaliser en langage C une copie de jeu de société Crazy Circus crée par Dominique Ehrhard. L'application se présente sous forme textuelle ce qui permet à plusieurs joueurs de s'affronter.

L'idée du jeu repose sur la manipulation de piles via des commandes spécifiques :

- KI : L'animal situé au sommet du podium **BLEU** saute pour se placer au sommet du podium **ROUGE**

- LO : L'animal situé au sommet du podium **ROUGE** saute pour se placer au sommet du podium **BLEU**

- SO : Les animaux situés aux sommets respectifs des deux podiums échangent leurs places

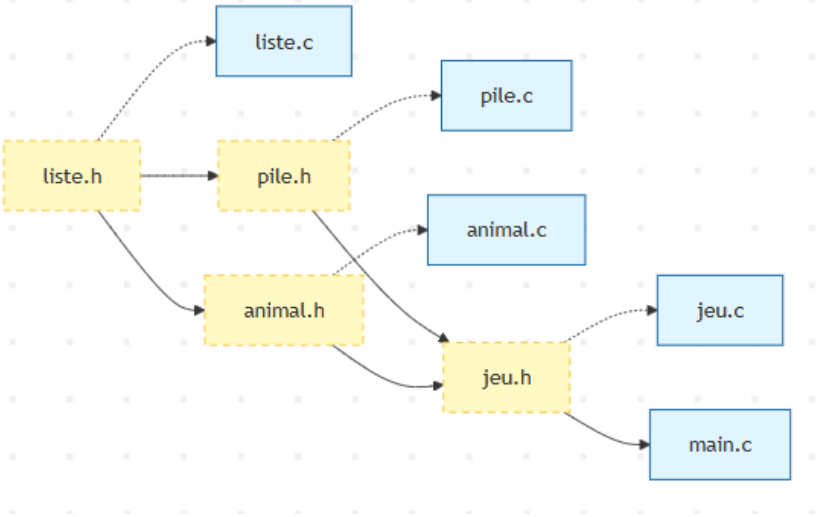
- NI : L'animal situé à la base du podium **BLEU** quitte sa position pour venir se placer au sommet de ce même podium

- MA : L'animal situé à la base du podium **ROUGE** quitte sa position pour venir se placer au sommet de ce même podium

La particularité de notre application est le fait que le lancement du jeu se fait grâce à la lecture d'un fichier de configuration externe nommé crazy.cfg dans lequel se trouve les noms des ordres et des animaux

Graphe de dépendance :

L'application a été conçue selon une architecture stricte. Le projet est structuré autour de plusieurs composants : Liste, pile, animal et jeu . Voici son grap



Test unitaires :

Les composants Liste et pile ont été tester à l'aide de plusieurs fonctions :

Test pour liste :

```
void testListe() {
    printf("Debut du test Liste...\n");
    Liste l;
    initListe(&l);
    assert(listeEstVide(&l) == 1);
    assert(l.taille == 0);
    assert(teteListe(&l) == NULL);
    insererTete(&l, "LION");
    assert(listeEstVide(&l) == 0);
    assert(l.taille == 1);
    assert(strcmp(teteListe(&l), "LION") == 0);
    insererTete(&l, "OURS");
    assert(l.taille == 2);
    assert(strcmp(teteListe(&l), "OURS") == 0);
    char* val = supprimerTete(&l);
    assert(strcmp(val, "OURS") == 0);
    assert(l.taille == 1);
    assert(strcmp(teteListe(&l), "LION") == 0);
    detruireListe(&l);
    assert(listeEstVide(&l) == 1);
    assert(l.taille == 0);
    printf("Test Liste OK !\n");
}
Test pour pile :
void testPile() {
    printf("Debut du test Pile...\n");
    Pile p;
    initPile(&p);
    assert(pileEstVide(&p) == 1);
    assert(taillePile(&p) == 0);
    empiler(&p, "ROUGE");
    assert(pileEstVide(&p) == 0);
    assert(taillePile(&p) == 1);
    assert(strcmp(sommet(&p), "ROUGE") == 0);
```

```
    empiler(&p, "BLEU");
    assert(taillePile(&p) == 2);
    assert(strcmp(sommet(&p), "BLEU") == 0);
    char* val = depiler(&p);
    assert(strcmp(val, "BLEU") == 0);
    assert(taillePile(&p) == 1);
    assert(strcmp(sommet(&p), "ROUGE") == 0);
    val = depiler(&p);
    assert(strcmp(val, "ROUGE") == 0);
    assert(pileEstVide(&p) == 1);
    empiler(&p, "VERT");
    detruirePile(&p);
    assert(pileEstVide(&p) == 1);
    printf("Test Pile OK !\n");
}
```

Bilan du projet :

Les difficultés rencontrées :

La compréhension générale du sujet et des règles du *Crazy Circus* a été assez rapide pour nous. Cependant, la difficulté majeure de ce projet n'a pas été l'algorithmique, mais la coordination technique.

Lors de nos précédents projets, nous travaillions sur un fichier unique, ce qui nous permettait d'utiliser des outils de collaboration instantanée comme "LiveShare" pour coder en temps réel. Pour cette SAÉ, la structure imposée en multiples fichiers sources .c et en-têtes .h a rendu cette méthode obsolète. Nous avons donc dû apprendre à utiliser GitHub pour centraliser et fusionner notre code. Cette transition a nécessité un temps d'adaptation, notamment pour comprendre la gestion des versions et la résolution des conflits entre nos modifications respectives.

Ce qui a été réussi :

Malgré ces défis organisationnels, nous sommes parvenus à rendre un projet fonctionnel qui répond aux exigences du sujet.

Notre hiérarchie de fichier avec ses multiples composants a été un succès. Notre code a été structurer proprement.

Notre programme respecte le formatage des différents affichages (alignement, espace etcc). Il respecte également toutes les sorties attendues garantissant sa compatibilité avec les tests automatiques.

Ce qui pourrait être améliorés :

Bien que le projet remplisse ses fonctionnalités principales, plusieurs axes d'amélioration pourraient renforcer sa robustesse et sa maintenabilité. Premièrement,

la **sécurisation des entrées utilisateur** est primordiale : l'utilisation actuelle de strcmp dans le main.c pour analyser les commandes ne gère pas le cas des chaînes vides ou mal formées, ce qui expose le programme à des risques de plantage ,si un utilisateur entre simplement des espaces. Une validation plus stricte en amont est recommandée.

Deuxièmement, la **gestion des erreurs** gagnerait à être découplée de l'affichage. Actuellement, certaines fonctions de logique (comme dans animal.c ou jeu.c) écrivent directement les messages d'erreur dans la console (printf). Il serait préférable qu'elles retournent des codes d'erreur précis, laissant à l'interface (le main) la responsabilité d'afficher les messages appropriés à l'utilisateur.

Enfin, en termes de **portabilité**, la présence de directives spécifiques à Windows (_CRT_SECURE_NO_WARNINGS) pourrait être évitée en utilisant les fonctions standard sécurisées ou en configurant le compilateur différemment, assurant ainsi une compilation sans avertissement sur tous les systèmes (Linux, MacOS). Une meilleure modularisation, séparant plus nettement la logique du jeu de l'interface textuelle, faciliterait également une éventuelle évolution vers une interface graphique.

Annexes :

animal.h :

```c
#pragma once

#include "liste.h"

/**
 * @def MAX_ANIMAUX
 * @brief Nombre maximum d'animaux supportés par le programme.
 */
#define MAX_ANIMAUX 10

 /**
  * @def MAX_ORDRES
  * @brief Nombre maximum d'ordres configurables
  */
#define MAX_ORDRES 10

  /**
   * @def TAILLE_NOM
   * @brief Taille max pour un nom d'animal
   */
#define TAILLE_NOM 50

/**
 * @def DEF_FICHIER_CONFIG
 * @brief Nom par défaut du fichier de configuration à charger.
 */
#define DEF_FICHIER_CONFIG "crazy.cfg"

/**
 * @struct ConfigJeu
 * @brief Structure représentant la configuration complète du jeu
 */
typedef struct {
    char* nomsAnimaux[MAX_ANIMAUX]; /* Tableau de chaines pour les noms */
```

```
    int nbAnimaux;                  /* Nombre reel d'animaux lus */
    char* ordres[MAX_ORDRES];       /* Tableau de chaines pour les ordres (KI,
LO...) */
    int nbOrdres;                   /* Nombre reel d'ordres lus */
} ConfigJeu;



/**
 * @brief Charge la configuration du jeu depuis un fichier et remplit la
structure ConfigJeu
 * @param[out] config Pointeur vers la structure ConfigJeu
 * @param[in] nomFichier Chemin du fichier à lire
 * @return int  Renvoie 1 si le chargement est réussi et 0 en cas d'erreur
 */
int chargerConfiguration(ConfigJeu* config, const char* nomFichier);


/**
 * @brief Libère la mémoire allouée dynamiquement pour la configuration
 * @param[in,out] config Pointeur vers la structure ConfigJeu à nettoyer
 */
void libererConfiguration(ConfigJeu* config);


/**
 * @brief Affiche le contenu de la configuration chargée
 * @param[in] config Pointeur vers la structure ConfigJeu à afficher
 */
void afficherConfiguration(const ConfigJeu* config);


/**
* @brief Vérifie la validité des règles du jeu chargées. Au moins 2 animaux et
3 ordres
* @param[in] config Pointeur vers la structure ConfigJeu à vérifier
* @return int Renvoie 1 si la configuration est valide, 0 sinon
*/
int validerConfiguration(const ConfigJeu* config);
```

animal.c :

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "animal.h"


 /**
  * @brief Duplique une chaîne de caractères
  * @param[in] s La chaîne de caractères à copier
  * @return char* Un pointeur vers la nouvelle chaîne allouée (ou NULL si échec
d'allocation)
  */
static char* monStrDup(const char* s) {
    if (s == NULL) return NULL;
    size_t len = strlen(s);
    char* d = (char*)malloc(len + 1);
    if (d == NULL) return NULL;
```

```
    strcpy(d, s);
    return d;
}

/**
 * @brief Charge la configuration du jeu depuis un fichier
 * @param[out] config Pointeur vers la structure ConfigJeu à remplir
 * @param[in] nomFichier Chemin vers le fichier de configuration
 * @return int 1 si le chargement technique est réussi, 0 en cas d'erreur
d'ouverture
 */
int chargerConfiguration(ConfigJeu* config, const char* nomFichier) {
    FILE* fichier = fopen(nomFichier, "r");
    if (fichier == NULL) {
        return 0;
    }

    config->nbAnimaux = 0;
    config->nbOrdres = 0;

    char ligne[1024];

    /* Lecture des animaux */
    if (fgets(ligne, sizeof(ligne), fichier) != NULL) {
        ligne[strcspn(ligne, "\n")] = '\0';
        ligne[strcspn(ligne, "\r")] = '\0';

        char* token = strtok(ligne, " ");
        while (token != NULL && config->nbAnimaux < MAX_ANIMAUX) {

            if (strlen(token) > 0) {
                config->nomsAnimaux[config->nbAnimaux] = monStrDup(token);
                config->nbAnimaux++;
            }
            token = strtok(NULL, " ");
        }
    }
    else {
        fclose(fichier);
        return 0;
    }

    /* Lecture des ordres */
    if (fgets(ligne, sizeof(ligne), fichier) != NULL) {
        ligne[strcspn(ligne, "\n")] = '\0';
        ligne[strcspn(ligne, "\r")] = '\0';

        char* token = strtok(ligne, " ");
        while (token != NULL && config->nbOrdres < MAX_ORDRES) {
            if (strlen(token) > 0) {
                config->ordres[config->nbOrdres] = monStrDup(token);
                config->nbOrdres++;
            }
            token = strtok(NULL, " ");
        }
    }

    fclose(fichier);
    return 1;
}

/**
 * @brief Libère la mémoire allouée pour la configuration
```

```
 * @param[in,out] config Pointeur vers la structure ConfigJeu à nettoyer
 */
void libererConfiguration(ConfigJeu* config) {
    if (config == NULL) return;

    for (int i = 0; i < config->nbAnimaux; i++) {
        free(config->nomsAnimaux[i]);
        config->nomsAnimaux[i] = NULL;
    }
    config->nbAnimaux = 0;

    for (int i = 0; i < config->nbOrdres; i++) {
        free(config->ordres[i]);
        config->ordres[i] = NULL;
    }
    config->nbOrdres = 0;
}

/**
 * @brief Affiche le contenu de la configuration chargée
 * @param[in] config Pointeur vers la structure ConfigJeu à afficher
 */
void afficherConfiguration(const ConfigJeu* config) {
    if (config == NULL) return;

    printf("Animaux chargees (%d) : ", config->nbAnimaux);
    for (int i = 0; i < config->nbAnimaux; i++) {
        printf("%s ", config->nomsAnimaux[i]);
    }
    printf("\n");

    printf("Ordres charges (%d)   : ", config->nbOrdres);
    for (int i = 0; i < config->nbOrdres; i++) {
        printf("%s ", config->ordres[i]);
    }
    printf("\n");
}

/**
 * @brief Vérifie les règles imposées par le jeu
 * @param[in] config Pointeur vers la structure ConfigJeu à vérifier
 * @return int 1 si la configuration est valide, 0 sinon
 */
int validerConfiguration(const ConfigJeu* config) {
    /* Les animaux doivent être deux au minimum  */
    if (config->nbAnimaux < 2) {
        printf("Erreur Config : Il faut au moins 2 animaux (trouve : %d).\n",
config->nbAnimaux);
        return 0;
    }

    /* Les ordres doivent être au moins trois  */
    if (config->nbOrdres < 3) {
        printf("Erreur Config : Il faut au moins 3 ordres (trouve : %d).\n",
config->nbOrdres);
        return 0;
    }

    /* Les ordres sont nécessairement choisis parmi les 5 ordres connus */
    const char* ordresValides[] = { "KI", "LO", "SO", "NI", "MA" };
    int nbValidesConnus = 5;

    for (int i = 0; i < config->nbOrdres; i++) {
```

```
        int estReconnu = 0;
        for (int j = 0; j < nbValidesConnus; j++) {
            if (strcmp(config->ordres[i], ordresValides[j]) == 0) {
                estReconnu = 1;
                break;
            }
        }
        if (!estReconnu) {
            printf("Erreur Config : L'ordre '%s' n'est pas un ordre valide (KI,
LO, SO, NI, MA).\n", config->ordres[i]);
            return 0;
        }
    }

    return 1;
}
```

jeu.h :

```
#pragma once

#include "pile.h"
#include "animal.h"


/**
 * @struct EtatJeu
 * @brief Représente l'état du jeu à un instant T. Il contient les deux piles
d'animaux représentants les podiums
 */
typedef struct {
    Pile podiumBleu;
    Pile podiumRouge;
} EtatJeu;


/**
 * @struct Deck
 * @brief Structure qui stock toutes les positions possibles (Les cartes
objectifs)
 */
typedef struct {
    EtatJeu* positions; /* Tableau dynamique de toutes les positions possibles
*/
    int nbPositions;    /* Nombre total de positions stockées */
    int* estUtilisee;   /* Tableau de booléens pour savoir si une carte a déjà
été tirée */
} Deck;



/**
 * @brief Initialise un état de jeu avec des piles vides
 * @param[out] e Pointeur vers la structure EtatJeu à initialiser
 */
void initEtat(EtatJeu* e);


/**
 * @brief Libère la mémoire d'un état de jeu
 * @param[in,out] e Pointeur vers la structure EtatJeu à nettoyer
```

```c
 */
void libererEtat(EtatJeu* e);



/**
 * @brief Effectue une copie d'un état vers un autre
 * @param[in] src Pointeur vers la structure EtatJeu source
 * @param[out] dest Pointeur vers la structure EtatJeu de destination
 */
void copierEtat(const EtatJeu* src, EtatJeu* dest);



/**
 * @brief Compare deux états de jeu
 * @param[in] e1 Pointeur vers la strcuture EtatJeu du premier état
 * @param[in] e2 Pointeur vers la strcuture EtatJeu du deuxième état
 * @return
 */
int estMemeEtat(const EtatJeu* e1, const EtatJeu* e2);


/**
 * @brief Execute l'ordre KI. L'animal au sommet du podium bleu saute vers le
sommet du podium rouge
 * @param[in,out] e Pointeur vers l'état du jeu à modifier
 * @return int 1 si le mouvement a été effectué, 0 si c'est impossible
 */
int commandeKI(EtatJeu* e);

/**
 * @brief Execute l'ordre LO. L'animal au sommet du podium rouge saute vers le
sommet du podium bleu
 * @param[in,out] e Pointeur vers l'état du jeu à modifier
 * @return int 1 si le mouvement a été effectué, 0 si c'est impossible
 */
int commandeLO(EtatJeu* e);

/**
 * @brief Execute l'ordre SO. Les deux animaux aux sommets des deux podiums
échangent leurs places
 * @param[in,out] e Pointeur vers l'état du jeu à modifier
 * @return int 1 si le mouvement a été effectué, 0 si c'est impossible
 */
int commandeSO(EtatJeu* e);

/**
 * @brief Execute l'ordre NI. L'animal tout en bas du podium bleu passe tout en
haut de ce même podium
 * @param[in,out] e Pointeur vers l'état du jeu à modifier
 * @return int 1 si le mouvement a été effectué, 0 si c'est impossible
 */
int commandeNI(EtatJeu* e);

/**
 * @brief Execute l'ordre NMA. L'animal tout en bas du podium rouge passe tout
en haut de ce même podium
 * @param[in,out] e Pointeur vers l'état du jeu à modifier
 * @return int 1 si le mouvement a été effectué, 0 si c'est impossible
 */
int commandeMA(EtatJeu* e);
```

```c
/**
 * @brief Execute une séquence de commande donnée sous forme de chaines. Elle
appelle les fonctions des commandes tour à tour
 * @param[in,out] e Pointeur vers l'état du jeu qui sera modifié en place
 * @param[in] seq Chaîne de caractères représentant la séquence d'ordres
 * @return int 1 si toute la séquence ets valide, 0 si un ordre est inconnu ou
impossible
 */
int executerSequence(EtatJeu* e, const char* seq);



/**
 * @brief Génère l'ensemble des positions possibles du jeu
 * @param[in] config Configuration contenant les animaux disponibles
 * @param[out] deck Pointeur vers le deck à remplir
 */
void genererToutesPositions(const ConfigJeu* config, Deck* deck);

/* Tire une nouvelle position cible al□atoire qui n'a pas encore □t□ jou□e */
/* Retourne un pointeur vers l'□tat cible, ou NULL si plus de cartes */

/**
 * @brief Tire une nouvelle carte objectif au hasard parmis celles non
utilisés. La carte sera marqué comme tiré arpès
 * @param[in,out] deck Le paquet dans lequel piocher
 * @return EtatJeu* Pointeur vers l'état cible tiré ou NULL si il y a plus de
cartes
 */
EtatJeu* tirerNouvelleCarte(Deck* deck);

/**
 * @brief Affiche le duel entre la position actuel et l'objectif
 * @param[in] depart Etat actuel du jeu (Les podiums à gauche)
 * @param[in] arrivee Etat objectif à atteindre (Les podiums à droite)
 */
void afficherDuel(const EtatJeu* depart, const EtatJeu* arrivee);

/**
* @brief Affiche la liste des ordres possibles au début du jeu
* @param[in] config La configuration qui contient les noms des ordres
*/
void afficherOrdresPossibles(const ConfigJeu* config);
```

jeu.c :

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "jeu.h"


 /**
  * @brief Initialise les podiums d'un état de jeu.
  * @param[out] e Pointeur vers l'état à initialiser.
  */
void initEtat(EtatJeu* e) {
    initPile(&e->podiumBleu);
```

```c
    initPile(&e->podiumRouge);
}

/**
 * @brief Libère la mémoire des piles contenues dans un état.
 * @param[in,out] e Pointeur vers l'état à nettoyer.
 */
void libererEtat(EtatJeu* e) {
    detruirePile(&e->podiumBleu);
    detruirePile(&e->podiumRouge);
}

/**
 * @brief Copie le contenu d'un état de jeu source vers un état destination
 * @param[in] src État de jeu source
 * @param[out] dest État destination
 */
void copierEtat(const EtatJeu* src, EtatJeu* dest) {
    detruirePile(&dest->podiumBleu);
    detruirePile(&dest->podiumRouge);
    initPile(&dest->podiumBleu);
    initPile(&dest->podiumRouge);


    Cellule* courant = src->podiumBleu.tete;
    char* temp[MAX_ANIMAUX];
    int n = 0;

    while (courant != NULL && n < MAX_ANIMAUX) {
        temp[n++] = courant->valeur;
        courant = courant->suivant;
    }

    for (int i = n - 1; i >= 0; i--) {
        empiler(&dest->podiumBleu, temp[i]);
    }

    courant = src->podiumRouge.tete;
    n = 0;
    while (courant != NULL && n < MAX_ANIMAUX) {
        temp[n++] = courant->valeur;
        courant = courant->suivant;
    }
    for (int i = n - 1; i >= 0; i--) {
        empiler(&dest->podiumRouge, temp[i]);
    }
}

/**
 * @brief Compare deux états de jeu pour savoir s'ils sont identiques
 * @param[in] e1 Premier état de jeu
 * @param[in] e2 Deuxième état de jeu
 * @return int 1 si identiques, 0 sinon
 */
int estMemeEtat(const EtatJeu* e1, const EtatJeu* e2) {

    if (taillePile(&e1->podiumBleu) != taillePile(&e2->podiumBleu)) return 0;
    if (taillePile(&e1->podiumRouge) != taillePile(&e2->podiumRouge)) return 0;


    Cellule* c1 = e1->podiumBleu.tete;
    Cellule* c2 = e2->podiumBleu.tete;
    while (c1 != NULL) {
```

```
        if (strcmp(c1->valeur, c2->valeur) != 0) return 0;
        c1 = c1->suivant;
        c2 = c2->suivant;
    }


    c1 = e1->podiumRouge.tete;
    c2 = e2->podiumRouge.tete;
    while (c1 != NULL) {
        if (strcmp(c1->valeur, c2->valeur) != 0) return 0;
        c1 = c1->suivant;
        c2 = c2->suivant;
    }

    return 1;
}



/**
 * @brief Exécute l'ordre KI
 * @param[in,out] e État du jeu
 * @return int 1 si succès, 0 si podium bleu vide
 */
int commandeKI(EtatJeu* e) {
    if (pileEstVide(&e->podiumBleu)) return 0;
    empiler(&e->podiumRouge, depiler(&e->podiumBleu));
    return 1;
}

/**
 * @brief Exécute l'ordre LO
 * @param[in,out] e État du jeu
 * @return int 1 si succès, 0 si podium rouge vide
 */
int commandeLO(EtatJeu* e) {
    if (pileEstVide(&e->podiumRouge)) return 0;
    empiler(&e->podiumBleu, depiler(&e->podiumRouge));
    return 1;
}

/**
 * @brief Exécute l'ordre SO
 * @param[in,out] e État du jeu
 * @return int 1 si succès, 0 si l'un des podiums est vide
 */
int commandeSO(EtatJeu* e) {
    if (pileEstVide(&e->podiumBleu) || pileEstVide(&e->podiumRouge)) return 0;

    char* valBleu = depiler(&e->podiumBleu);
    char* valRouge = depiler(&e->podiumRouge);

    empiler(&e->podiumBleu, valRouge);
    empiler(&e->podiumRouge, valBleu);
    return 1;
}

/**
 * @brief Effectue une rotation Bas vers Haut sur une pile
 * @param[in,out] p La pile à manipuler
 * @return int 1 si succès, 0 si pile vide
 */
static int rotationBasVersHaut(Pile* p) {
```

```
    if (pileEstVide(p)) return 0;
    if (taillePile(p) == 1) return 1;

    Pile temp;
    initPile(&temp);


    while (taillePile(p) > 1) {
        empiler(&temp, depiler(p));
    }

    char* bas = depiler(p);

    while (!pileEstVide(&temp)) {
        empiler(p, depiler(&temp));
    }

    empiler(p, bas);

    detruirePile(&temp);
    return 1;
}

/**
 * @brief Exécute l'ordre NI
 * @param[in,out] e État du jeu
 * @return int 1 si succès
 */
int commandeNI(EtatJeu* e) {
    return rotationBasVersHaut(&e->podiumBleu);
}

/**
 * @brief Exécute l'ordre MA
 * @param[in,out] e État du jeu
 * @return int 1 si succès
 */
int commandeMA(EtatJeu* e) {
    return rotationBasVersHaut(&e->podiumRouge);
}

/**
 * @brief Parse et exécute une séquence de commandes
 * @param[in,out] e État du jeu à modifier
 * @param[in] seq Chaîne de caractères contenant les ordres
 * @return int 1 si OK, 0 si mouvement impossible, -1 si ordre inconnu
 */
int executerSequence(EtatJeu* e, const char* seq) {
    int len = strlen(seq);

    if (len % 2 != 0) return 0;

    for (int i = 0; i < len; i += 2) {
        char cmd[3];
        cmd[0] = seq[i];
        cmd[1] = seq[i + 1];
        cmd[2] = '\0';

        int res = 0;

        if (strcmp(cmd, "KI") == 0)      res = commandeKI(e);
        else if (strcmp(cmd, "LO") == 0) res = commandeLO(e);
        else if (strcmp(cmd, "SO") == 0) res = commandeSO(e);
```

```
        else if (strcmp(cmd, "NI") == 0) res = commandeNI(e);
        else if (strcmp(cmd, "MA") == 0) res = commandeMA(e);
        else return -1;

        if (res == 0) return 0;
    }

    return 1;
}

/**
 * @brief Enregistre une permutation avec une coupure donnée
 * @param[in,out] deck Le deck de cartes à remplir
 * @param[in,out] animaux Tableau des animaux dans l'ordre de la permutation
actuelle
 * @param[in] n Nombre total d'animaux
 * @param[in] coupure Index de séparation (0 à n)
 */
static void enregistrerConfiguration(Deck* deck, char** animaux, int n, int
coupure) {
    EtatJeu* nouv = &deck->positions[deck->nbPositions];
    initEtat(nouv);

    for (int i = 0; i < coupure; i++) {
        empiler(&nouv->podiumBleu, animaux[i]);
    }


    for (int i = coupure; i < n; i++) {
        empiler(&nouv->podiumRouge, animaux[i]);
    }

    deck->nbPositions++;
}

/**
 * @brief Calcule la factorielle d'un nombre (n!) pour l'algorithme de Heap
 */
static int factorielle(int n) {
    if (n <= 1) return 1;
    int res = 1;
    for (int i = 2; i <= n; i++) res *= i;
    return res;
}

/**
 * @brief Génère toutes les positions possibles du jeu grâce à l'algorithme de
Heap
 * @param[in] config Configuration contenant les animaux
 * @param[out] deck Structure Deck à allouer et remplir
 */
void genererToutesPositions(const ConfigJeu* config, Deck* deck) {
    int n = config->nbAnimaux;
    int factN = factorielle(n);
    int capaciteMax = factN * (n + 1);

    deck->positions = (EtatJeu*)malloc(sizeof(EtatJeu) * capaciteMax);
    if (deck->positions == NULL) {
        fprintf(stderr, "Erreur fatale : Memoire insuffisante pour le
deck.\n");
        exit(EXIT_FAILURE);
    }
    deck->nbPositions = 0;
```

```
    char* A[MAX_ANIMAUX];
    for (int i = 0; i < n; i++) A[i] = config->nomsAnimaux[i];

    int c[MAX_ANIMAUX];
    for (int i = 0; i < n; i++) c[i] = 0;

    for (int k = 0; k <= n; k++) enregistrerConfiguration(deck, A, n, k);

    int i = 0;
    while (i < n) {
        if (c[i] < i) {
            if (i % 2 == 0) {
                char* tmp = A[0]; A[0] = A[i]; A[i] = tmp;
            }
            else {
                char* tmp = A[c[i]]; A[c[i]] = A[i]; A[i] = tmp;
            }

            for (int k = 0; k <= n; k++) enregistrerConfiguration(deck, A, n,
k);

            c[i] += 1;
            i = 0;
        }
        else {
            c[i] = 0;
            i += 1;
        }
    }

    deck->estUtilisee = (int*)calloc(deck->nbPositions, sizeof(int));
}

/**
 * @brief Tire une carte objectif aléatoire non encore jouée
 * @param[in,out] deck Le paquet de cartes
 * @return EtatJeu* Pointeur vers l'état cible, ou NULL si le deck est épuisé
 */
EtatJeu* tirerNouvelleCarte(Deck* deck) {
    int dispo = 0;
    /* Compter les cartes restantes */
    for (int i = 0; i < deck->nbPositions; i++) {
        if (!deck->estUtilisee[i]) dispo++;
    }

    if (dispo == 0) return NULL;

    /* Tirage au sort parmi les disponibles */
    int choix = rand() % dispo;
    int compteur = 0;

    for (int i = 0; i < deck->nbPositions; i++) {
        if (!deck->estUtilisee[i]) {
            if (compteur == choix) {
                deck->estUtilisee[i] = 1; /* Marquer comme utilisée */
                return &deck->positions[i];
            }
            compteur++;
        }
    }
    return NULL;
}
```

```c
/**
 * @brief Affiche la liste des ordres disponibles
 * @param[in] config le fichier de configuration
 */
void afficherOrdresPossibles(const ConfigJeu* config) {
    for (int i = 0; i < config->nbOrdres; i++) {
        char* o = config->ordres[i];
        printf("%s ", o);

        if (strcmp(o, "KI") == 0)      printf("(B -> R)");
        else if (strcmp(o, "LO") == 0) printf("(B <- R)");
        else if (strcmp(o, "SO") == 0) printf("(B <-> R)");
        else if (strcmp(o, "NI") == 0) printf("(B ^)");
        else if (strcmp(o, "MA") == 0) printf("(R ^)");

        if (i < config->nbOrdres - 1) printf(" | ");
    }
    printf("\n\n");
}


/**
 * @brief  Convertit une pile en tableau pour faciliter l'affichage
 * @param[in] p Constante d'un Pointeur vers une pile
 * @param[in,out] tab pointeur d'un tableau de chaine de caractères
 * @return int
 */
static int pileVersTableau(const Pile* p, char* tab[]) {
    int h = 0;
    Cellule* c = p->tete;
    while (c != NULL) {
        tab[h++] = c->valeur;
        c = c->suivant;
    }
    return h;
}

/**
 * @brief Affiche le duel
 * @param[in] depart État actuel du joueur
 * @param[in] arrivee État objectif à atteindre
 */
void afficherDuel(const EtatJeu* depart, const EtatJeu* arrivee) {
    char* b1[MAX_ANIMAUX]; int hB1 = pileVersTableau(&depart->podiumBleu, b1);
    char* r1[MAX_ANIMAUX]; int hR1 = pileVersTableau(&depart->podiumRouge, r1);

    char* b2[MAX_ANIMAUX]; int hB2 = pileVersTableau(&arrivee->podiumBleu, b2);
    char* r2[MAX_ANIMAUX]; int hR2 = pileVersTableau(&arrivee->podiumRouge,
r2);

    /* Calcul de la hauteur maximale pour l'affichage */
    int maxH = 0;
    if (hB1 > maxH) maxH = hB1;
    if (hR1 > maxH) maxH = hR1;
    if (hB2 > maxH) maxH = hB2;
    if (hR2 > maxH) maxH = hR2;

    int colW = 12; /* Largeur de colonne fixe pour l'alignement */
```

```
    for (int k = maxH - 1; k >= 0; k--) {
        /* Partie Gauche */

        /* Podium Bleu  */
        if (k < hB1) printf("%-*s", colW, b1[hB1 - 1 - k]);
        else         printf("%-*s", colW, "");

        /* Podium Rouge */
        if (k < hR1) printf("%-*s", colW, r1[hR1 - 1 - k]);
        else         printf("%-*s", colW, "");

        /* Espacement central */
        printf("          ");

        /* Partie Droite */

        /* Podium Bleu */
        if (k < hB2) printf("%-*s", colW, b2[hB2 - 1 - k]);
        else         printf("%-*s", colW, "");

        /* Podium Rouge */
        if (k < hR2) printf("%s", r2[hR2 - 1 - k]);

        printf("\n");
    }

    /* Affichage des socles et de la flèche centrale */
    printf("%-*s%-*s   ==>    %-*s%-*s\n", colW, "----", colW, "----", colW, "-
---", colW, "----");
    printf("%-*s%-*s          %-*s%-*s\n", colW, "BLEU", colW, "ROUGE", colW,
"BLEU", colW, "ROUGE");
    printf("\n");
}
```

liste.h :

```
#pragma once

/**
 * @typedef Element
 * @brief Type des données dans la liste
 */
typedef char* Element;

/**
* @struct Cellule
 * @brief Structure représentant un maillon de la liste
 */
typedef struct Cellule {
    Element valeur;          /* La donnee stockee (le nom de l'animal) */
    struct Cellule* suivant; /* Pointeur vers le maillon suivant */
} Cellule;

/**
* @struct Liste
 * @brief Structure representant la liste en elle-meme
 */
typedef struct {
    Cellule* tete; /* Pointeur vers le sommet de la liste */
    int taille;    /* Nombre d'elements dans la liste */
} Liste;
```

```c
/**
 * @brief Initialise une liste vide en mettant la taille à 0 et la tete à NULL
 * @param[out] l Pointeur vers la structure Liste à initialiser
 */
void initListe(Liste* l);


/**
 * @brief Vérifie si la liste est vide
 * @param[in] l Constante du pointeur vers la liste
 * @return int 1 si la liste ne contient aucun élément, sinon 0
 */
int listeEstVide(const Liste* l);

/**
 * @brief Insère un nouvel élement en tête de liste
 * @param[in,out] l Pointeur vers la liste
 * @param[in] val La valeur (une chaine de caractères) à ajouter
 */
void insererTete(Liste* l, Element val);

/**
 * @brief Supprime l'élément en tete de liste et le renvoie
 * @param[in,out] l Pointeur vers la liste
 * @return Element La valeur qui était en tête (le nom de l'animal)
 */
Element supprimerTete(Liste* l);

/**
 * @brief Consulte l'élement de tete sans le supprimer
 * @param[in] l Constante du pointeur vers la liste
 * @return Element La valeur située en tête
 */
Element teteListe(const Liste* l);

/**
* @brief Vide la liste pour libérer la mémoire
* @param[in,out] l Pointeur vers la liste à détruire
*/
void detruireListe(Liste* l);
```

liste.c :

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "liste.h"


 /**
  * @brief Initialise une liste à l'état vide
  * @param[out] l Pointeur vers la liste à initialiser
  */
void initListe(Liste* l) {
    l->tete = NULL;
```

```c
    l->taille = 0;
}

/**
 * @brief Vérifie si la liste ne contient aucun élément
 * @param[in] l Pointeur vers la liste à tester.
 * @return int 1 si la liste est vide, 0 sinon
 */
int listeEstVide(const Liste* l) {
    return (l->tete == NULL);
}

/**
 * @brief Ajoute un élément au début de la liste
 * @param[in,out] l Pointeur vers la liste
 * @param[in] val La valeur à stocker
 */
void insererTete(Liste* l, Element val) {
    Cellule* nouv = (Cellule*)malloc(sizeof(Cellule));

    if (nouv == NULL) {
        fprintf(stderr, "Erreur fatale : Echec d'allocation memoire dans
insererTete\n");
        exit(EXIT_FAILURE);
    }

    nouv->valeur = val;
    nouv->suivant = l->tete;
    l->tete = nouv;
    l->taille++;
}

/**
 * @brief Supprime l'élément en tête de liste
 * @param[in,out] l Pointeur vers la liste
 * @return Element La valeur qui était stockée en tête ou NULL si liste vide
 */
Element supprimerTete(Liste* l) {
    if (listeEstVide(l)) {
        return NULL;
    }

    Cellule* aSupprimer = l->tete;
    Element valeur = aSupprimer->valeur;

    l->tete = aSupprimer->suivant;
    free(aSupprimer);
    l->taille--;

    return valeur;
}

/**
 * @brief Renvoie l'élément en tête sans le supprimer
 * @param[in] l Pointeur vers la liste
 * @return Element La valeur située en tête ou NULL si vide
 */
Element teteListe(const Liste* l) {
    if (listeEstVide(l)) {
        return NULL;
    }
    return l->tete->valeur;
}
```

```c
/**
 * @brief Vide toute la liste et libère la mémoire des cellules
 * @param[in,out] l Pointeur vers la liste à détruire
 */
void detruireListe(Liste* l) {
    while (!listeEstVide(l)) {
        supprimerTete(l);
    }
}
```

pile.h :

```c
#pragma once

#include "liste.h"



/**
* @typedef Pile
 * @brief Le typpe Pile vient un alias du type Liste
 */
typedef Liste Pile;

/**
 * @brief Initialise une pile vide
 * @param[out] p Pointeur vers la structure Pile à initialiser
 */
void initPile(Pile* p);

/**
 * @brief Vérifie si la pile est vide
 * @param[in] p Pointeur vers la pile à tester
 * @return int 1 si la pile ne contient aucun élément, 0 sinon
 */
int pileEstVide(const Pile* p);

/**
 * @brief Empile un élément en l'ajoutant au sommet de la pile
 * @param[in,out] p Pointeur vers la pile
 * @param[in] val La valeur (nom de l'animal) à ajouter au sommet
 */
void empiler(Pile* p, Element val);

/**
 * @brief Dépile un élément, retire l'élement situé au sommet, retire l'élément
situé au sommet de la pile et le renvoie.
 * @param[in,out] p Pointeur vers la pile
 * @return Element L'élément qui était au sommet
 */
Element depiler(Pile* p);

/**
 * @brief Consulte le sommet de la pile
 * @param[in] p Pointeur vers la pile
 * @return Element L'élément au sommet
 */
Element sommet(const Pile* p);
```

```c
/**
 * @brief Renvoie la taille de la pile
 * @param[in] p Pointeur vers la pile
 * @return int Le nombre d'éléments
 */
int taillePile(const Pile* p);

/**
 * @brief Vide la pile et libère la mémoire.
 * @param[in,out] p Pointeur vers la pile à détruire.
 */
void detruirePile(Pile* p);
```

pile.c :

```c


#define _CRT_SECURE_NO_WARNINGS
#include "pile.h"


 /**
  * @brief Initialise une pile vide (délègue l'initialisation à la fonction
initListe)
  * @param[out] p Pointeur vers la pile à initialiser
  */
void initPile(Pile* p) {
    initListe(p);
}

/**
 * @brief Vérifie si la pile est vide
 * @param[in] p Pointeur vers la pile
 * @return int 1 si vide, 0 sinon
 */
int pileEstVide(const Pile* p) {
    return listeEstVide(p);
}

/**
 * @brief Empile un élément au sommet
 * @param[in,out] p Pointeur vers la pile
 * @param[in] val L'élément à ajouter
 */
void empiler(Pile* p, Element val) {
    insererTete(p, val);
}

/**
 * @brief Dépile l'élément du sommet
 * @param[in,out] p Pointeur vers la pile
 * @return Element L'élément retiré
 */
Element depiler(Pile* p) {
    return supprimerTete(p);
}

/**
 * @brief Regarde l'élément au sommet sans le retirer
 * @param[in] p Pointeur vers la pile
 * @return Element L'élément au sommet
```

```c
 */
Element sommet(const Pile* p) {
    return teteListe(p);
}

/**
 * @brief Renvoie la hauteur actuelle de la pile
 * @param[in] p Pointeur vers la pile
 * @return int Le nombre d'éléments
 */
int taillePile(const Pile* p) {
    return p->taille;
}

/**
 * @brief Vide la pile et libère la mémoire (délègue le nettoyage à
detruireListe)
 * @param[in,out] p Pointeur vers la pile à détruire
 */
void detruirePile(Pile* p) {
    detruireListe(p);
}
```

main.c :

```c


#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "jeu.h"


#define MAX_JOUEURS 10
#define LEN_CMD 256
#define CONFIG_FILENAME "crazy.cfg"

/**
 * @struct Joueur
 * @brief Représente un participant à la partie
 */
typedef struct {
    char* nom;  /* Pointeur vers le nom (argv) */
    int score;  /*Score courant */
} Joueur;


/**
 * @brief Fonction de comparaison pour le tri des scores
 */
int compareJoueurs(const void* a, const void* b) {
    const Joueur* j1 = (const Joueur*)a;
    const Joueur* j2 = (const Joueur*)b;

    if (j1->score != j2->score) {
        return j2->score - j1->score;
    }
    return strcmp(j1->nom, j2->nom);
}
```

```c
int main(int argc, char* argv[]) {
    if (argc < 3) {
        printf("Usage: %s <Joueur1> <Joueur2> ...\n", argv[0]);
        printf("Erreur : Il faut au moins 2 joueurs pour lancer la partie.\n");
        return EXIT_FAILURE;
    }

    Joueur joueurs[MAX_JOUEURS];
    int nbJoueurs = 0;

    /* Récupération des noms depuis la ligne de commande */
    for (int i = 1; i < argc; i++) {
        if (nbJoueurs >= MAX_JOUEURS) {
            printf("Attention: Nombre maximum de joueurs (%d) atteint. Les
suivants sont ignores.\n", MAX_JOUEURS);
            break;
        }

        /* Vérification des doublons de noms */
        int existe = 0;
        for (int k = 0; k < nbJoueurs; k++) {
            if (strcmp(joueurs[k].nom, argv[i]) == 0) {
                existe = 1;
                break;
            }
        }
        if (existe) {
            printf("Erreur: Les noms des joueurs doivent etre distincts
(%s).\n", argv[i]);
            return EXIT_FAILURE;
        }

        /* Initialisation du joueur */
        joueurs[nbJoueurs].nom = argv[i];
        joueurs[nbJoueurs].score = 0;
        nbJoueurs++;
    }



    ConfigJeu config;

    if (!chargerConfiguration(&config, CONFIG_FILENAME)) {
        fprintf(stderr, "Erreur fatale : Impossible de lire le fichier de
configuration %s.\n", CONFIG_FILENAME);
        return EXIT_FAILURE;
    }
    if (!validerConfiguration(&config)) {
        libererConfiguration(&config);
        return EXIT_FAILURE;
    }




    srand((unsigned int)time(NULL));

    /* Génération du paquet complet des positions */
    Deck deck;
    genererToutesPositions(&config, &deck);
```

```c
    /* Affichage des ordres disponibles*/
    afficherOrdresPossibles(&config);

    /* Tirage des états initiaux */
    EtatJeu* courant = tirerNouvelleCarte(&deck); /* Position de départ */
    EtatJeu* objectif = tirerNouvelleCarte(&deck); /* Position objectif */

    if (courant == NULL || objectif == NULL) {
        printf("Erreur : Pas assez de combinaisons pour jouer.\n");
        return EXIT_FAILURE;
    }

    int* peutJouer = (int*)malloc(sizeof(int) * nbJoueurs);
    for (int i = 0; i < nbJoueurs; i++) peutJouer[i] = 1;
    int nbJoueursEnLice = nbJoueurs;



    while (objectif != NULL) {

        afficherDuel(courant, objectif);

        int tourTermine = 0;


        while (!tourTermine) {
            char buffer[LEN_CMD];

            if (fgets(buffer, sizeof(buffer), stdin) == NULL) {
                tourTermine = 1;
                objectif = NULL;
                break;
            }
            buffer[strcspn(buffer, "\n")] = 0;

            if (strlen(buffer) == 0) continue;

            char* nomJoueur = strtok(buffer, " ");
            char* sequence = strtok(NULL, " ");

            int idJoueur = -1;
            for (int i = 0; i < nbJoueurs; i++) {
                if (strcmp(joueurs[i].nom, nomJoueur) == 0) {
                    idJoueur = i;
                    break;
                }
            }

            if (idJoueur == -1) {
                printf("Joueur inconnu (%s)\n", nomJoueur);
                continue;
            }

            if (!peutJouer[idJoueur]) {
                printf("%s ne peut pas jouer durant ce tour\n", nomJoueur);
                continue;
            }

            if (sequence == NULL) {
                continue;
            }
```

```
            EtatJeu testState;
            initEtat(&testState);
            copierEtat(courant, &testState);

            int codeRetour = executerSequence(&testState, sequence);

            if (codeRetour == -1) {

                char ordreFaux[3] = "??";
                for (size_t i = 0; i < strlen(sequence); i += 2) {
                    char sub[3];
                    sub[0] = sequence[i];
                    sub[1] = sequence[i + 1];
                    sub[2] = '\0';

                    int estValide = 0;
                    if (strcmp(sub, "KI") == 0 || strcmp(sub, "LO") == 0 ||
                        strcmp(sub, "SO") == 0 || strcmp(sub, "NI") == 0 ||
                        strcmp(sub, "MA") == 0) {
                        estValide = 1;
                    }

                    if (!estValide) {
                        strcpy(ordreFaux, sub);
                        break;
                    }
                }
                printf("L'ordre %s n'existe pas\n", ordreFaux);
            }

            /* Vérification de la victoire */
            int bonneSolution = 0;
            if (codeRetour == 1) {
                if (estMemeEtat(&testState, objectif)) {
                    bonneSolution = 1;
                }
            }

            libererEtat(&testState);

            if (bonneSolution) {
                /* VICTOIRE DU JOUEUR */
                printf("%s gagne un point\n\n", nomJoueur);
                joueurs[idJoueur].score++;
                tourTermine = 1;
            }
            else {

                if (codeRetour != -1) {
                    printf("La sequence ne conduit pas a la situation
attendue\n");
                    printf("%s ne peut plus jouer durant ce tour\n",
nomJoueur);
                    peutJouer[idJoueur] = 0;
                    nbJoueursEnLice--;
                }


                if (nbJoueursEnLice == 1) {
                    int survivant = -1;
                    for (int i = 0; i < nbJoueurs; i++) {
```

```
                        if (peutJouer[i]) {
                            survivant = i;
                            break;
                        }
                    }
                    /* agne par forfait */
                    printf("%s gagne un point car lui seul peut encore jouer
durant ce tour\n\n", joueurs[survivant].nom);
                    joueurs[survivant].score++;
                    tourTermine = 1;
                }
                else if (nbJoueursEnLice == 0) {
                    printf("Tous les joueurs ont echoue. Fin du tour sans
vainqueur.\n\n");
                    tourTermine = 1;
                }
            }

        }


        /* Préparation du Tour Suivant */
        if (objectif != NULL) {
            /* L'objectif atteint devient le nouveau point de départ */
            courant = objectif;
             /* Tirage d'un nouvel objectif */
            objectif = tirerNouvelleCarte(&deck);

            /* Réinitialisation des droits de jeu pour tous */
            for (int i = 0; i < nbJoueurs; i++) peutJouer[i] = 1;
            nbJoueursEnLice = nbJoueurs;
        }

    }


    /*Fin de Partie et score */

    /* Tri des joueurs selon le score */
    qsort(joueurs, nbJoueurs, sizeof(Joueur), compareJoueurs);

    /* Affichage du classement final */
    for (int i = 0; i < nbJoueurs; i++) {
        printf("%s %d\n", joueurs[i].nom, joueurs[i].score);
    }



    free(peutJouer);

    for (int i = 0; i < deck.nbPositions; i++) {
        libererEtat(&deck.positions[i]);
    }
    free(deck.positions);
    free(deck.estUtilisee);

    libererConfiguration(&config);

    return EXIT_SUCCESS;
}
```