Traitement d'image et Analyse comportementale d'Agents dans différents Environnements

Arthur Boit

Candidat n°50920
INFORMATIQUE (Informatique pratique)
MATHEMATIQUES (Mathématiques Appliquées)

Juin 2025

Pourquoi ce sujet ?

- Transformation réel-virtuel par la photographie
- Aspect pratique (vie quotidienne)
- Des résultats visuels motivants

Love-Parade (2010) causant 21 morts et des centaines de blessés

RER-B à Châtelet

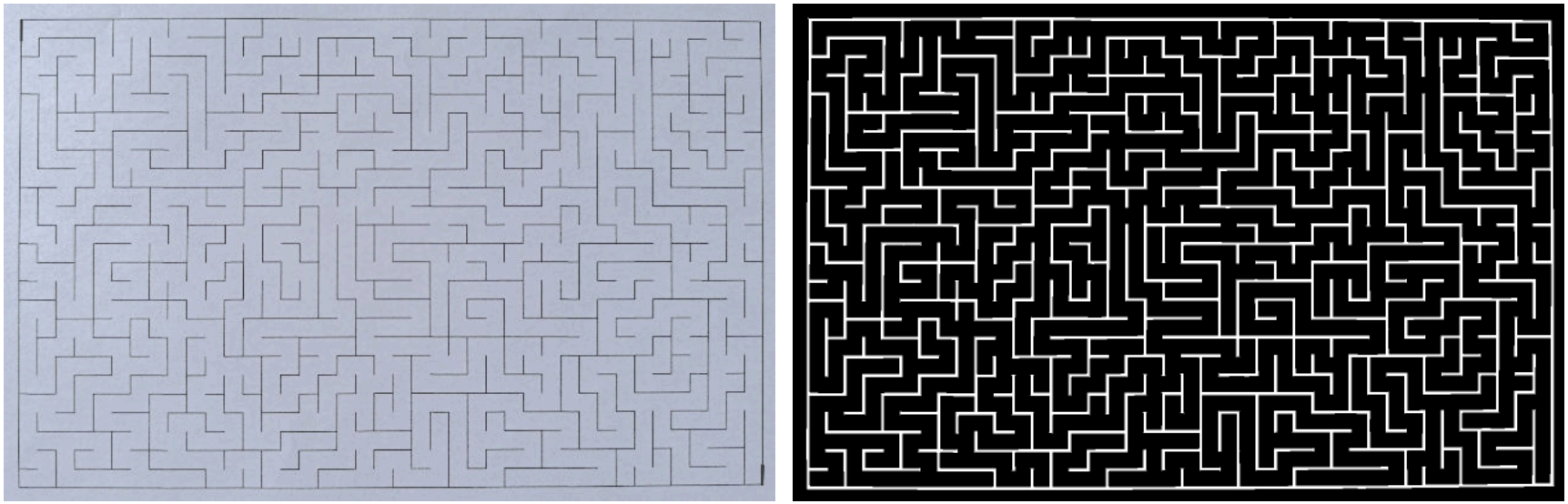
Plan de l'étude

Comment utiliser une simple photo pour simuler le mouvement des foules ?

1. Convertir une image en un environnement : filtre de Canny
2. Exploitation de l'environnement
3. Interprétation

Filtre de Canny

1. Floutage Gaussien
2. Filtre de Sobel
3. Suppression des non-maximaux
4. Double seuillage à l'hystérésis



Un labyrinthe avant et après le filtre de Canny

Floutage Gaussien

$$\mathrm{G}(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Exemple d'un noyau de taille 5×5 pour $\sigma = 1,4$:

$$K = \frac{1}{159}\begin{bmatrix} 2 & 5 & 6 & 5 & 2 \\ 5 & 10 & 13 & 10 & 5 \\ 6 & 13 & 17 & 13 & 6 \\ 5 & 10 & 13 & 10 & 5 \\ 2 & 5 & 6 & 5 & 2 \end{bmatrix}$$

Chaque pixel de l'image devient une moyenne pondérée de ses voisins.

Filtre de Sobel

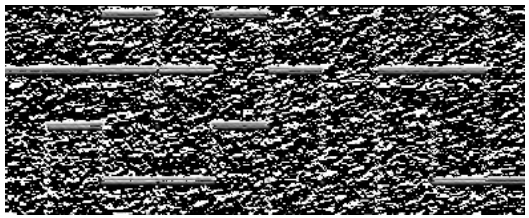
Un masque par axe : approximation du gradient.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

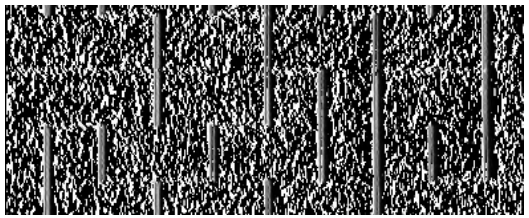
$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Application du filtre de Sobel

Approximation du gradient en chacun des pixels de l'image



Gradient selon x



Gradient selon y

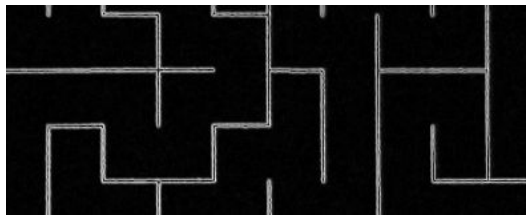


Image déduite des 2 précédentes

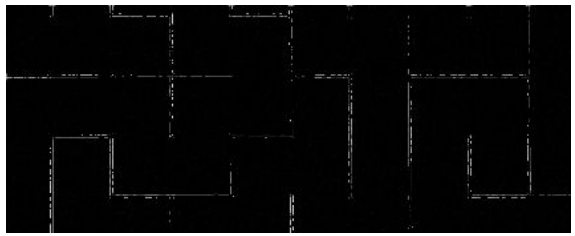
$$m = \sqrt{\mathrm{G}_x^2 + \mathrm{G}_y^2}$$

$$m_{\text{norm}} = \frac{m - m_{\text{min}}}{m_{\text{max}} - m_{\text{min}}}$$

Suppression des non-maximaux

Contours sont épais (bruit, filtre brutal...) $\Longrightarrow$ affinage.

$$\theta = \mathrm{atan2}\left(\mathrm{G}_x,\, \mathrm{G}_y\right)$$



Portion de labyrinthe après suppression des non maximaux

Double seuillage à l'hystérésis

$$m \leftarrow \begin{cases} 1 & \text{si } m \geqslant t_{max} \\ 0 & \text{si } m \leqslant t_{min} \\ \frac{1}{2} & \text{sinon} \end{cases}$$

Puis si le pixel est connecté à un pixel d'intensité 1 alors il devient blanc, sinon il devient noir.

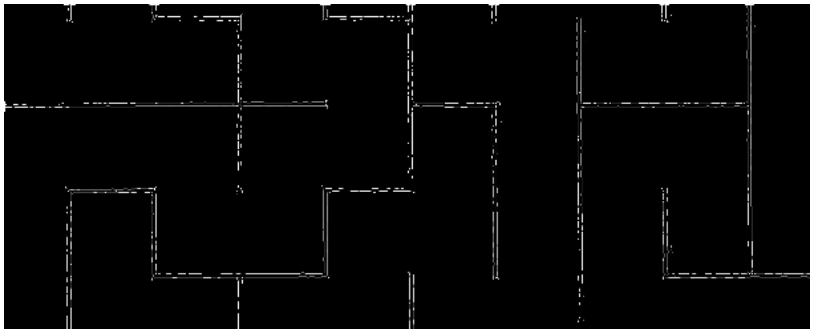
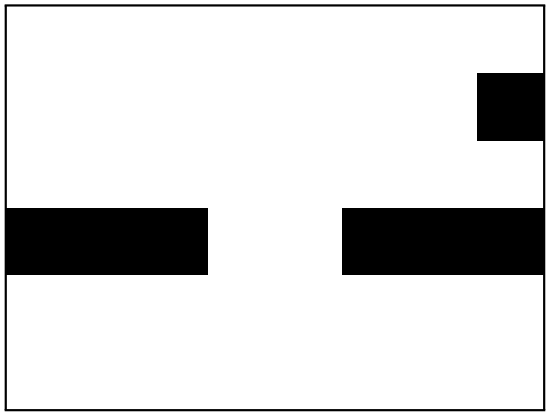


Image finale de la portion de labyrinthe

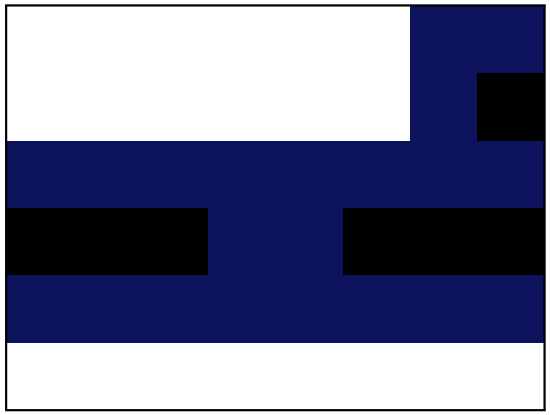
Fermeture Morphologique

Trous dans les murs : souci pour un potentiel parcours

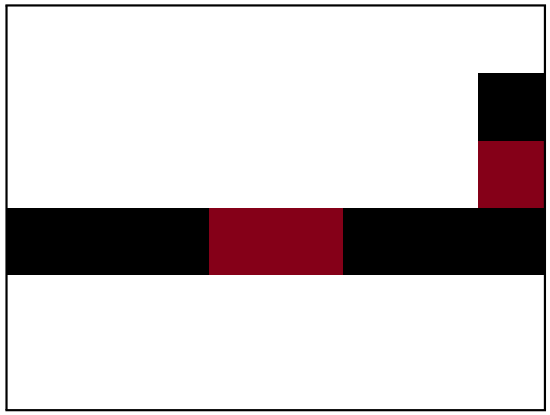
$\Longrightarrow$ Application d'une fermeture morphologique



Exemple d'un contour discontinu



Dilatation



Érosion

Parcours de l'image

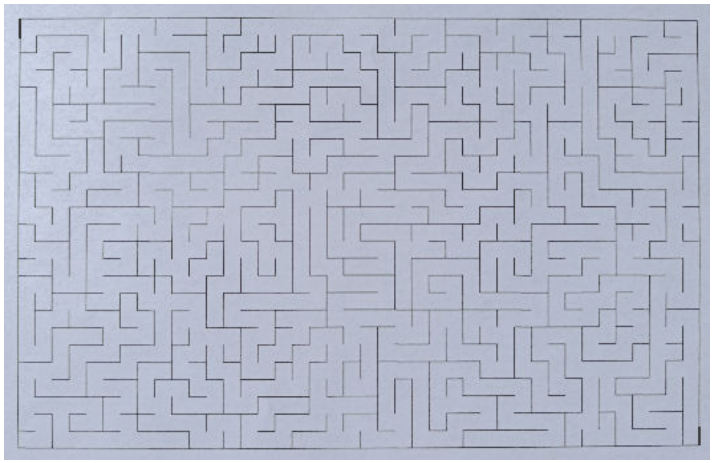
$\rightarrow$ Approche égoïste

Assimilation à un graphe :

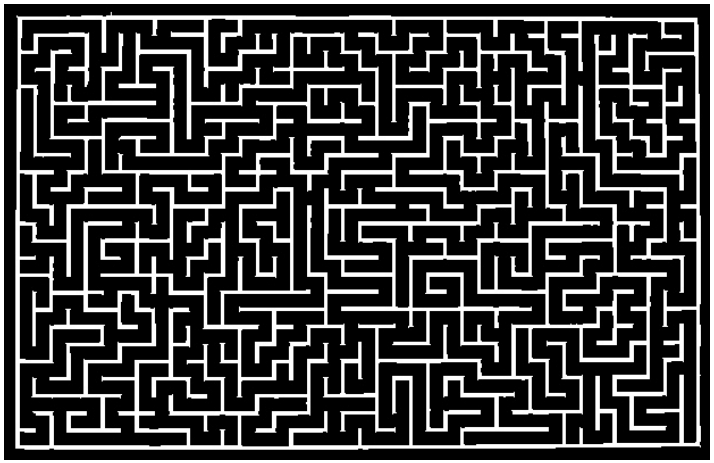
- Les sommets sont les pixels
- Les arcs relient les pixels en contact s'ils sont noirs
- Le sommet de destination porte le poids

Idée : recherche du plus court chemin répétée en boucle.

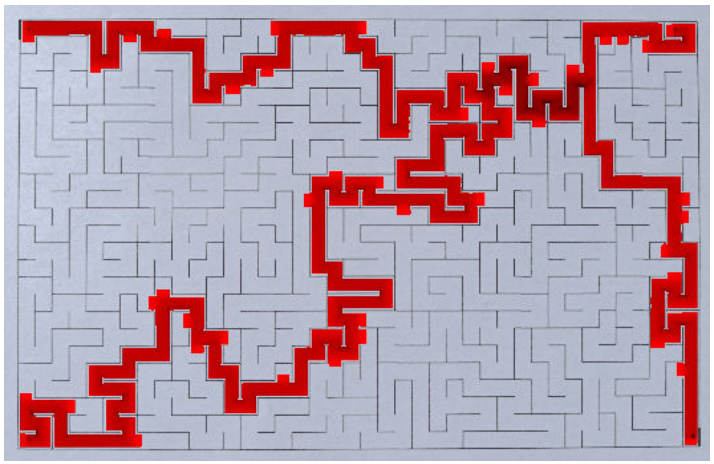
Parcours d'un labyrinthe



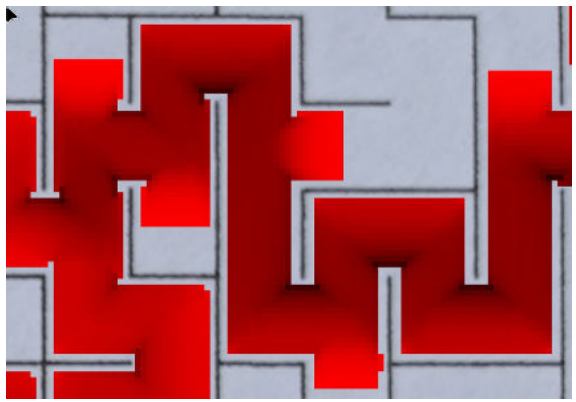
Labyrinthe original



Contours détectés



Parcours de 2000 agents



Zoom sur l'image

Optimisation du parcours

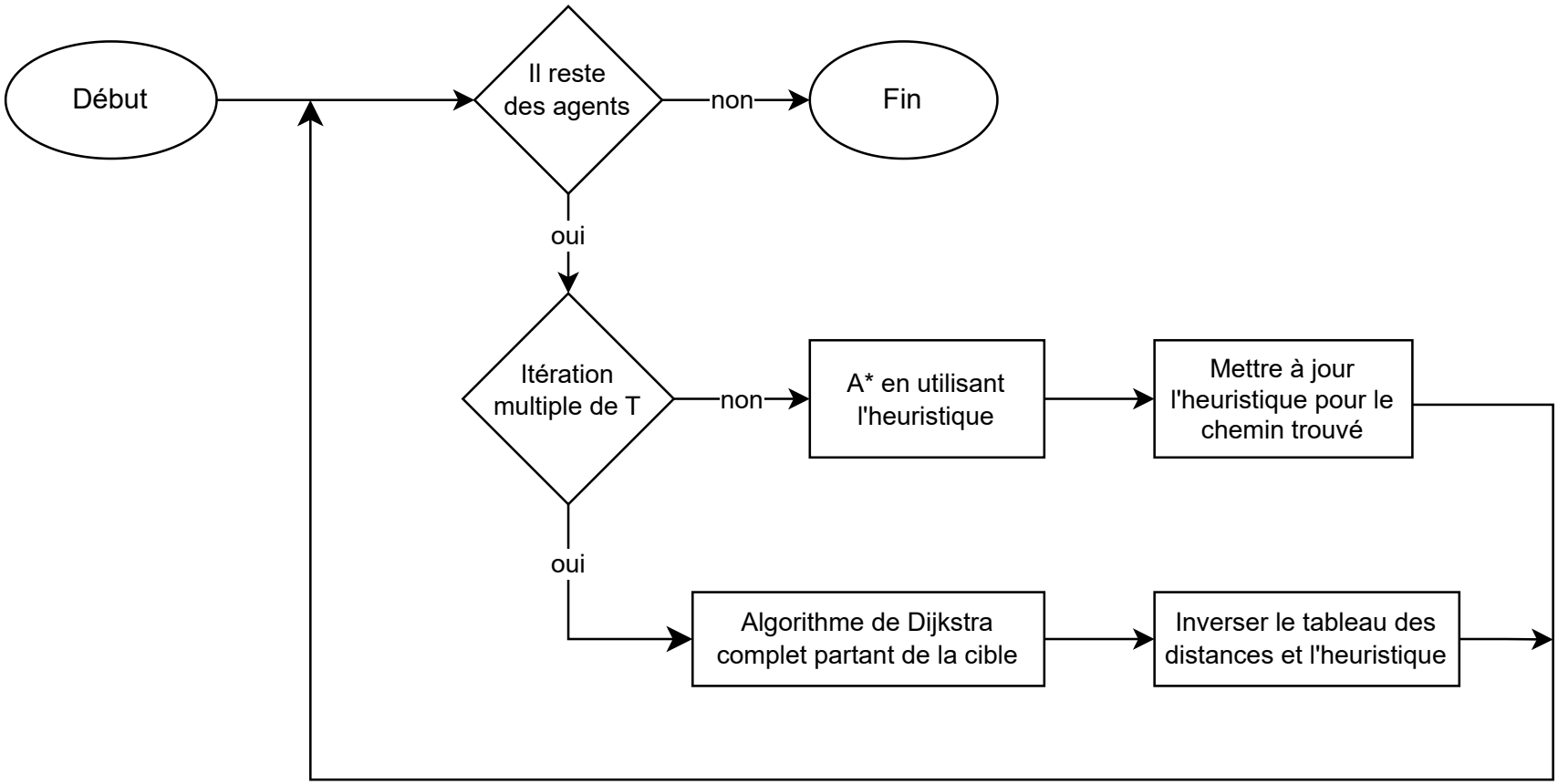
$\Longrightarrow$ Exécution longue et beaucoup de calculs perdus.

Meilleure idée : A* avec une heuristique évolutive.

- Quelle heuristique choisir au départ ?
- Comment la faire évoluer pour améliorer les performances ?

Optimisation du parcours

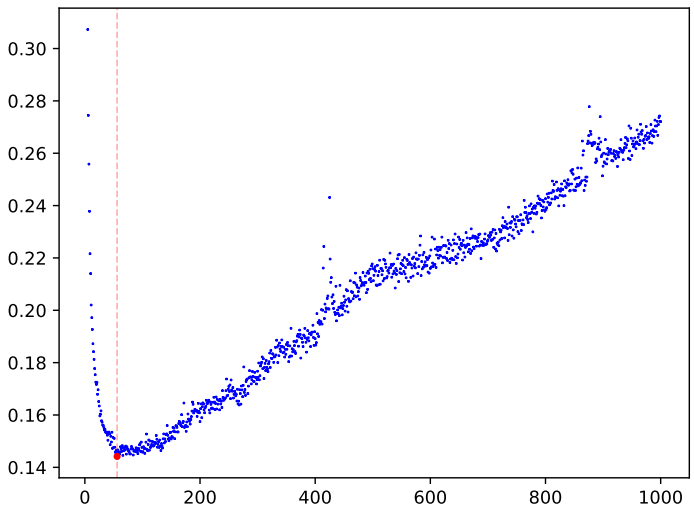
A* itératif à heuristique évolutive



Optimisation du parcours

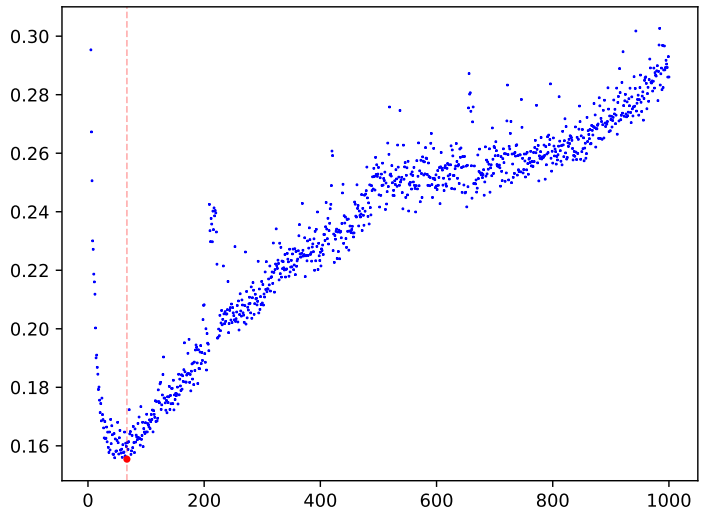
- w : poids initial
- k : facteur de compression
- abscisses : période T
- ordonnées : temps / temps max

$w = 100,\ k = 10$



Minimum en T = 56

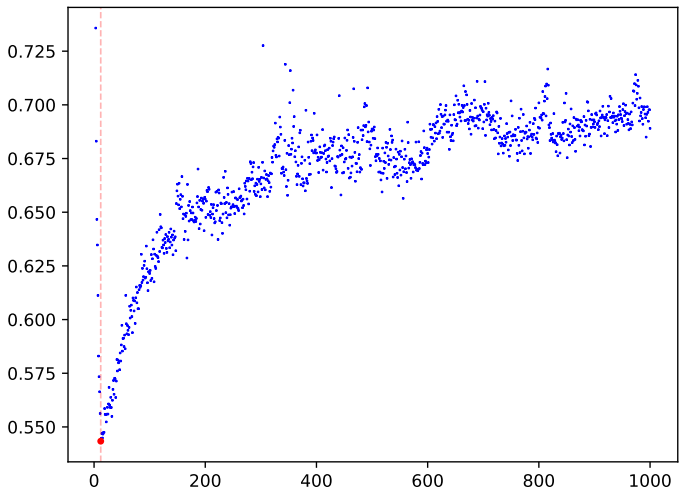
$w = 100,\ k = 15$



Minimum en T = 67

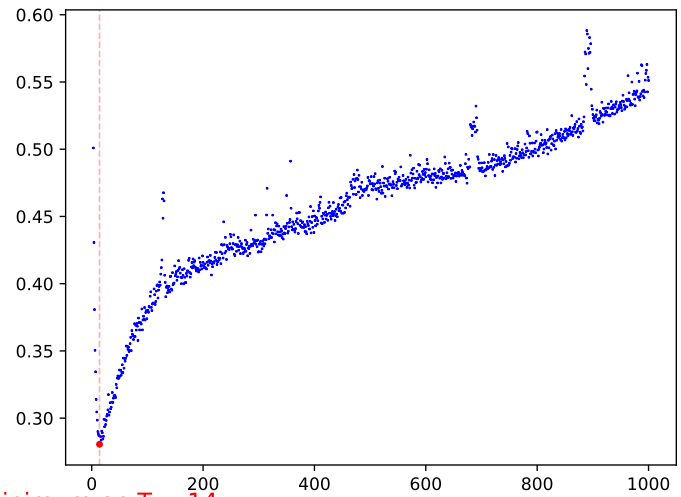
Optimisation du parcours

$w = 1,\ k = 10$



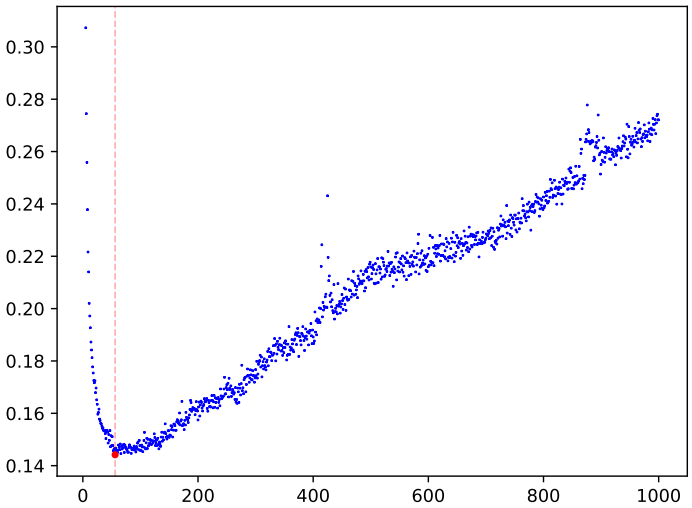
Minimum en T = 12

$w = 10,\ k = 10$



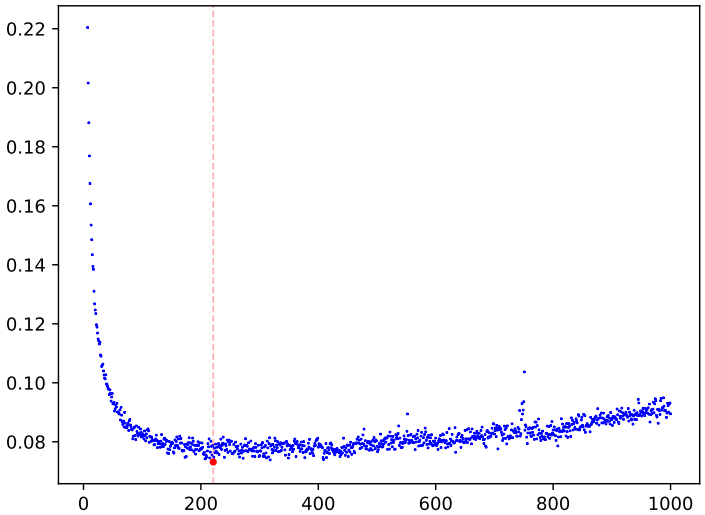
Minimum en T = 14

$w = 100,\ k = 10$



Minimum en T = 56

$w = 1000,\ k = 10$



Minimum en T = 221

Optimisation du parcours

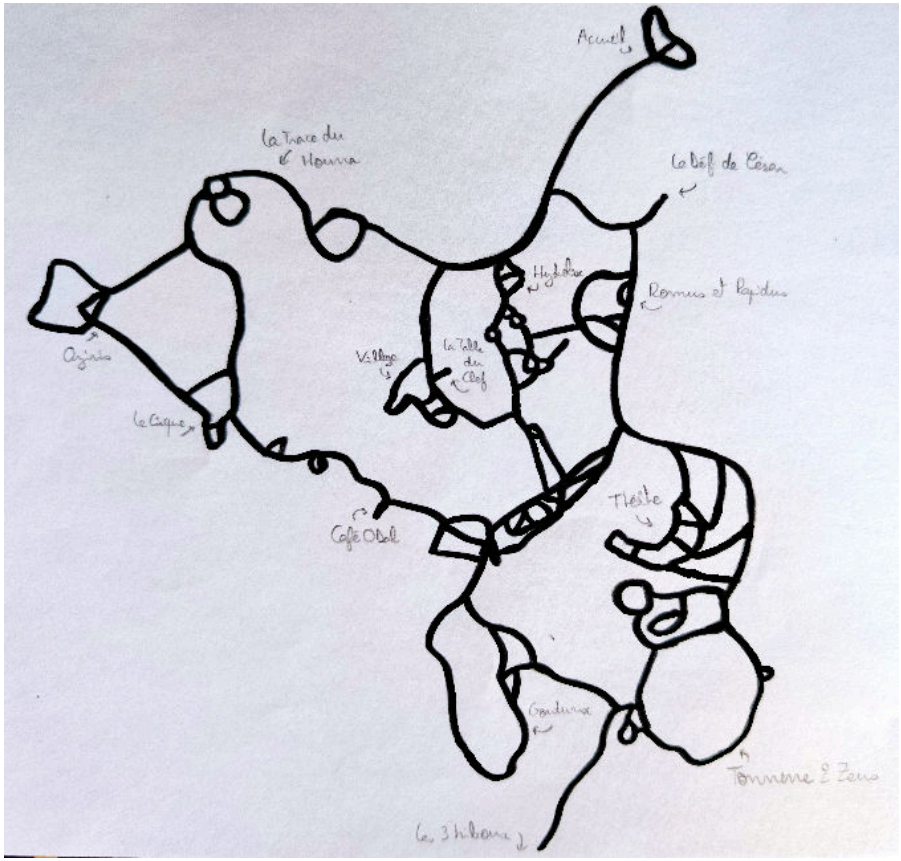
Poids initial	Compression	T optimal
1	10	12
10	10	14
100	10	56
100	15	67
1000	10	221

- Chemins étroits (compression)
- Difficulté à changer d'itinéraire (poids initial)

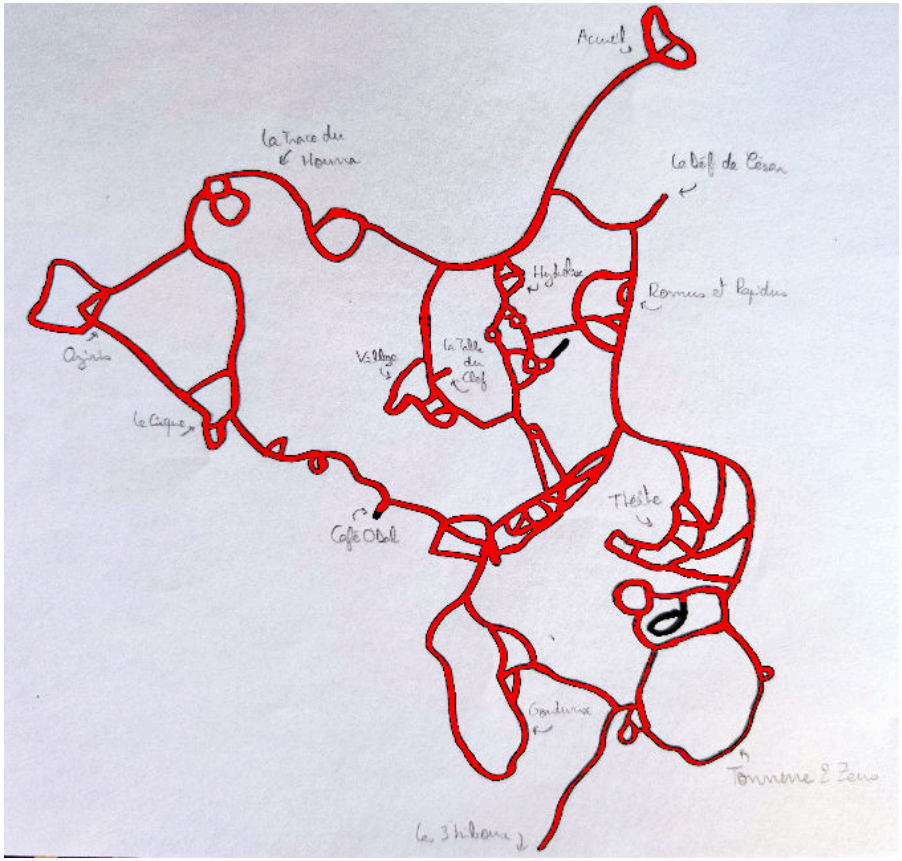
$\Longrightarrow$ Moins de chemins différents empruntés

$\Longrightarrow$ Mise à jour complète plus rare

Dernière application : Parc d'attraction



Dessin du parc Astérix



Parc Astérix après le parcours de 4000 visiteurs

Conclusion

- Filtre de Canny pour transformer une image en un environnement exploitable
- Algorithme A* itératif avec une heuristique évolutive
- Un paramètre difficile à définir
- Un résultat exploitable

Annexe | Liste des fichiers principaux

Fichier source	Fichier en-tête
main.c	
circular_list.c	circular_list.h
common.c	common.h
config.c	config.h
crowd.c	crowd.h
csv.c	csv.h
image.c	image.h
image_usage.c	image_usage.h
logging.c	logging.h
priority_queue.c	priority_queue.h
queue.c	queue.h

Annexe | fichier main.c

```c
#include <opencv2/opencv.hpp>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

#include "priority_queue.h"
#include "queue.h"
#include "image_usage.h"
#include "image.h"
#include "logging.h"
#include "config.h"
#include "crowd.h"
#include "circular_list.h"
#include "common.h"
#include "csv.h"

int main(int argc, char** argv) {
    // Chargement de la configuration
    config_load("config.conf");
    clock_t start, end;
    double cpu_time_used;

    if (argc < 5 || 6 < argc) {
        log_fatal("Usage : %s <image> <movements-file> <weight0> <alpha> [compression]", argv[0]);
    }
    const char* movements_file_path = argv[2];
    int weight0 = atoi(argv[3]);
    int alpha = atoi(argv[4]);
    if (weight0 < 0 || alpha < 0) {
        log_fatal("Les poids doivent être positifs");
```

Annexe | fichier main.c

```
}
if (weight0 == 0 && alpha == 0) {
    log_fatal("Les poids ne peuvent pas être tous les deux nuls");
}
int n = 1;

colored_image_t colored_image = image_read(argv[1]);
image_t image = image_from_colored_image(colored_image);

if (argc == 6) {
    n = atoi(argv[5]);
    image_t past = image;
    image = image_resize(image, n);
    image_free(past);
}

// Application du filtre de Canny
image_t canny_image = canny(image, 0.1, 0.2);

// Epaississement de l'image
image_t image_morpho = image_fermeture_morphologique(canny_image, 30/n);

image_write(image_morpho, "presentation/image_morpho.jpg");
image_write(image, "presentation/grey.jpg");
colored_image_write(colored_image, "presentation/original.jpg");

// Test sur les environnements
environment_t env;
circular_list_t* movements;

env = env_from_image(image_morpho);
env_initialiser_tableaux(&env);
```

Annexe | fichier main.c

```
    movements = load_movements(movements_file_path, n);
    start = clock();
    multiple_move_env_iterative_a_star(movements, &env, weight0, alpha, 10);
    end = clock();
    cpu_time_used = ((double) (end-start)) / CLOCKS_PER_SEC;
    log_info("A* modulo %d : %.3f secondes", 10, cpu_time_used);

    env_image_edit(image_morpho, env, 1);
    image_write(image_morpho, "pictures/image_resultat0.jpg");
    env_image_colored_edit(colored_image, env, n);
    colored_image_write(colored_image, "pictures/image_resultat.jpg");
    log_info("Image resultante ecrite dans pictures/image_resultat.jpg");

    free_movements(movements);
    env_liberer_tableaux(&env);
    env_free(env);


    image_free(canny_image);
    image_free(image_morpho);
    colored_image_free(colored_image);
    image_free(image);

    return 0;
}
```

Annexe | fichier circular_list.c

```c
#include <stdlib.h>
#include <stdbool.h>

#include "circular_list.h"

// Créer une liste circulaire
circular_list_t* cl_create() {
    circular_list_t* cl = (circular_list_t*) malloc(sizeof(circular_list_t));
    cl->head = NULL;
    cl->size = 0;
    return cl;
}

// Ajouter un élément à la liste circulaire (placé en tête)
void cl_add(circular_list_t* cl, void* value) {
    circular_list_node_t* node = (circular_list_node_t*) malloc(sizeof(circular_list_node_t));
    node->value = value;

    if (cl_is_empty(cl)) {
        node->next = node;
        node->prev = node;
        cl->head = node;
    }
    else {
        node->next = cl->head;
        node->prev = cl->head->prev;
        cl->head->prev->next = node;
        cl->head->prev = node;
    }
    cl->head = node;
    cl->size++;
}
```

Annexe | fichier circular_list.c

```c
// Supprimer un élément de la liste circulaire (celui en tête), puis passage au suivant
void cl_remove(circular_list_t* cl) {
    if (cl_is_empty(cl)) return;

    circular_list_node_t* node = cl->head;
    if (cl->size == 1) {
        cl->head = NULL;
    }
    else {
        node->prev->next = node->next;
        node->next->prev = node->prev;
        cl->head = node->next;
    }
    free(node);
    cl->size--;
}

// Libérer la mémoire occupée par la liste circulaire
void cl_free(circular_list_t* cl) {
    circular_list_node_t* current = cl->head;
    while (current != NULL) {
        circular_list_node_t* next = current->next;
        free(current);
        current = next;
    }
    free(cl);
}

// Récupérer la valeur d'un élément de la liste circulaire (celui en tête)
void* cl_get(circular_list_t* cl) {
    if (cl_is_empty(cl)) return NULL;
```

Annexe | fichier circular_list.c

```
    return cl->head->value;
}

// Définir la valeur d'un élément de la liste circulaire (celui en tête)
void cl_set(circular_list_t* cl, void* value) {
    if (cl_is_empty(cl)) return;
    cl->head->value = value;
}

// Passer à l'élément suivant de la liste circulaire
void cl_next(circular_list_t* cl) {
    if (cl_is_empty(cl)) return;
    cl->head = cl->head->next;
}

// Passer à l'élément précédent de la liste circulaire
void cl_prev(circular_list_t* cl) {
    if (cl_is_empty(cl)) return;
    cl->head = cl->head->prev;
}

// Vérifier si la liste circulaire est vide
bool cl_is_empty(circular_list_t* cl) {
    return cl->size == 0;
}
```

Annexe | fichier circular_list.h

```c
#ifndef CIRCULAR_LIST_H
#define CIRCULAR_LIST_H

#include <stdbool.h>

struct circular_list_node_s {
    struct circular_list_node_s* next;
    struct circular_list_node_s* prev;
    void* value;
};
typedef struct circular_list_node_s circular_list_node_t;

struct circular_list_s {
    circular_list_node_t* head;
    int size;
};
typedef struct circular_list_s circular_list_t;

// Créer une liste circulaire vide
circular_list_t* cl_create();

// Ajouter un élément à la liste circulaire (placé en tête)
void cl_add(circular_list_t* cl, void* value);

// Supprimer un élément de la liste circulaire (celui en tête)
void cl_remove(circular_list_t* cl);

// Libérer la mémoire occupée par la liste circulaire
void cl_free(circular_list_t* cl);

// Récupérer la valeur d'un élément de la liste circulaire (celui en tête)
void* cl_get(circular_list_t* cl);
```

Annexe | fichier circular_list.h

```c
// Définir la valeur d'un élément de la liste circulaire (celui en tête)
void cl_set(circular_list_t* cl, void* value);

// Passer à l'élément suivant de la liste circulaire
void cl_next(circular_list_t* cl);

// Passer à l'élément précédent de la liste circulaire
void cl_prev(circular_list_t* cl);

// Vérifier si la liste circulaire est vide
bool cl_is_empty(circular_list_t* cl);

#endif
```

Annexe | fichier common.c

```c
#include <stdlib.h>

#include "common.h"

position_t** pred = NULL;
double** dis = NULL;
double** heuristique = NULL;
int** heuristique_propagation = NULL;
int** heuristique_in_queue = NULL;
int** visited = NULL;
position_t*** ptrs = NULL;
```

Annexe | fichier common.h

```c
#ifndef COMMON_H
#define COMMON_H


typedef struct position_s {
    int i;
    int j;
} position_t;

struct movement_s {
    position_t start;
    position_t target;
    int agents;
};
typedef struct movement_s movement_t;

// Directions possibles
const int directions[4][2] = {
    {-1, 0}, {1, 0}, {0, -1}, {0, 1}, // Haut, Bas, Gauche, Droite
};

// Tableaux
extern position_t** pred;
extern double** dis;
extern double** heuristique;
extern int** heuristique_propagation;
extern int** heuristique_in_queue;
extern int** visited;
extern position_t*** ptrs;


#endif
```

Annexe | fichier config.c

```c
#include <stdio.h>

#include "config.h"

int DEBUG_MODE = 0; // Mode de débogage par défaut

// Charger une configuration à partir d'un fichier
void config_load(const char* filename) {
    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        fprintf(stderr, "Erreur lors de l'ouverture du fichier de configuration : %s\n", filename);
        return;
    }
    fscanf(file, "DEBUG_MODE==%d\n", &DEBUG_MODE);
    fclose(file);
    fprintf(stderr, "debug mode : %d\n", DEBUG_MODE);
}
```

Annexe | fichier config.h

```c
#ifndef CONFIG_H
#define CONFIG_H


extern int DEBUG_MODE; // 0 = no debug, 1 = debug, 2 = verbose debug

// Charger une configuration à partir d'un fichier
void config_load(const char* filename);

#endif
```

Annexe | fichier crowd.c

```c
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>

#include "crowd.h"
#include "image.h"
#include "image_usage.h"
#include "circular_list.h"
#include "logging.h"
#include "common.h"

// Créer un environnement à partir d'une image
environment_t env_from_image(image_t image) {
    log_debug("Création d'un environnement à partir de l'image : %s", image.name);

    environment_t env {
        .rows = image.rows,
        .cols = image.cols,
        .agents = (int**) malloc(sizeof(int*) * env.rows),
        .max = 0
    };
    for (int i = 0; i < env.rows; i++) {
        env.agents[i] = (int*) malloc(sizeof(int) * env.cols);
        for (int j = 0; j < env.cols; j++) {
            if (image.pixels[i][j] == 1.) env.agents[i][j] = -1;
            else env.agents[i][j] = 0;
        }
    }
    log_debug("Environnement créé à partir de l'image : %s", image.name);

    return env;
}
```

Annexe | fichier crowd.c

```c
// Libérer la mémoire occupée par un environnement
void env_free(environment_t env) {
    log_debug("Libération de la mémoire d'un environnement");

    for (int i = 0; i < env.rows; i++) {
        free(env.agents[i]);
    }
    free(env.agents);

    log_debug("Mémoire de l'environnement libérée");
}

// Modifier une image en fonction de l'environnement
void env_image_edit(image_t image, environment_t env, int n) {
    log_debug("Modification de l'image en fonction de l'environnement : %s", image.name);

    if (env.max == 0) return;
    for (int i = 0; i < env.rows; i++) {
        for (int j = 0; j < env.cols; j++) {

            if (env.agents[i][j] > -1) {
                pixel_t pix = (double) env.agents[i][j] / (double) env.max;

                for (int k = 0; k < n; k++) {
                    if (i * n + k >= image.rows) break;
                    for (int l = 0; l < n; l++) {
                        if (j * n + l >= image.cols) break;
                        image.pixels[i * n + k][j * n + l] = pix;
                    }
                }
            }
```

```
        }
    }
    log_debug("Image modifiée en fonction de l'environnement : %s", image.name);
}

// Modifier une image colorée en fonction de l'environnement
void env_image_colored_edit(colored_image_t image, environment_t env, int n) {
    log_debug("Modification de l'image colorée en fonction de l'environnement : %s", image.name);

    if (env.max == 0) return;
    for (int i = 0; i < env.rows; i++) {
        for (int j = 0; j < env.cols; j++) {

            if (env.agents[i][j] > 0) {
                double alpha = 1. - (double) env.agents[i][j] / (double) env.max;
                colored_pixel_t pix {
                    .r = 255. * alpha * alpha * alpha,
                    .g = 0,
                    .b = 0
                };

                for (int k = 0; k < n; k++) {
                    if (i * n + k >= image.rows) break;
                    for (int l = 0; l < n; l++) {
                        if (j * n + l >= image.cols) break;
                        image.pixels[i * n + k][j * n + l] = pix;
                    }
                }
            }
        }
    }
    log_debug("Image colorée modifiée en fonction de l'environnement : %s", image.name);
```

```c
}

// Distance norme 1
int distance_norme1(position_t p1, position_t p2) {
    return abs(p1.i - p2.i) + abs(p1.j - p2.j);
}

// Parcourir un environnement avec un A* itératif
void move_env_iterative_a_star(movement_t movement, environment_t* env, int weight0, int alpha, int modulo) {
    log_debug("Déplacement de %d agents dans un environnement avec A* itératif", movement.agents);
    position_t start = movement.start;
    position_t target = movement.target;
    int agents = movement.agents;

    // Initialiser les tableaux
    for (int i = 0; i < env->rows; i++) {
        for (int j = 0; j < env->cols; j++) {
            visited[i][j] = -1;
        }
    }

    // Créer une file de priorité
    priority_queue_t* pq = pq_create(env->rows * env->cols);

    // Boucle principale
    position_t* s;
    position_t* t;
    int iteration = 0;
    while (agents > 0) {
        if (iteration % modulo != 0) {
            dis[start.i][start.j] = 0.;
            visited[start.i][start.j] = iteration;
```

```
        s = ptrs[start.i][start.j];
        t = ptrs[target.i][target.j];
    }
    else {
        dis[target.i][target.j] = 0.;
        visited[target.i][target.j] = iteration;
        s = ptrs[target.i][target.j];
        t = ptrs[start.i][start.j];
    }
    pq_push(pq, 0, (void*) s);

    while (!pq_is_empty(pq)) {
        position_t* u = (position_t*) pq_pop(pq);
        if ((iteration % modulo != 0 || agents == 1) && u->i == t->i && u->j == t->j) break;

        for (int d = 0; d < 4; d++) {
            int ni = u->i + directions[d][0];
            int nj = u->j + directions[d][1];

            if (ni >= 0 && ni < env->rows && nj >= 0 && nj < env->cols
                    && visited[ni][nj] < iteration && env->agents[ni][nj] != -1) {

                double dis_n = (visited[ni][nj] == iteration) ? dis[ni][nj] : INFINITY;
                double new_dist = dis[u->i][u->j] + (env->agents[ni][nj]*alpha + weight0);

                if (new_dist < dis_n) {
                    dis[ni][nj] = new_dist;
                    pred[ni][nj] = *u;

                    position_t* pos = ptrs[ni][nj];

                    visited[ni][nj] = iteration;
```

```
                int total_cost = new_dist + heuristique[ni][nj];

                pq_push(pq, total_cost, (void*) pos);
            }
        }
    }
    visited[u->i][u->j] = iteration;
}
while (!pq_is_empty(pq)) {
    s = (position_t*) pq_pop(pq);
}
// Agir sur l'environnement

if (iteration % modulo == 0) {
    double** temps = heuristique;
    heuristique = dis;
    dis = temps;
}
else {
    position_t current = target;
    while ((current.i != start.i || current.j != start.j)
            && visited[current.i][current.j] == iteration) {
        env->agents[current.i][current.j]++;
        heuristique[current.i][current.j] = dis[target.i][target.j]-dis[current.i][current.j];
        if (env->agents[current.i][current.j] > env->max) {
            env->max = env->agents[current.i][current.j];
        }
        current = pred[current.i][current.j];
    }
    env->agents[start.i][start.j]++;
    if (env->agents[start.i][start.j] > env->max) {
```

```
                env->max = env->agents[start.i][start.j];
            }
        }

        iteration++;
        agents--;
    }
    log_debug("Tous les agents ont été déplacés");
    // Libérer les ressources
    pq_free(pq);

    log_debug("Déplacement des %d agents dans un environnement avec A* itératif terminé", movement.agents);
}

// Appliquer plusieurs mouvements à un environnement avec A* itératif
void multiple_move_env_iterative_a_star(circular_list_t* movements, environment_t* env,
                                        int weight0, int alpha, int modulo) {
    log_debug("Déplacement d'agents dans un environnement avec A* itératif");
    int n = movements->size;
    while (!cl_is_empty(movements)) {
        movement_t* m = (movement_t*) cl_get(movements);
        move_env_iterative_a_star(*m, env, weight0, alpha, modulo);
        free(m);
        cl_remove(movements);
    }
    log_debug("Déplacement d'agents dans un environnement avec A* itératif terminé");
}

// Initialiser les tableaux nécessaires pour les déplacements dans un environnement
void env_initialiser_tableaux(environment_t* env) {
    pred = (position_t**) malloc(sizeof(position_t*) * env->rows);
    dis = (double**) malloc(sizeof(double*) * env->rows);
```

```
    heuristique = (double**) malloc(sizeof(double*) * env->rows);
    heuristique_propagation = (int**) malloc(sizeof(int*) * env->rows);
    heuristique_in_queue = (int**) malloc(sizeof(int*) * env->rows);
    visited = (int**) malloc(sizeof(int*) * env->rows);
    ptrs = (position_t***) malloc(sizeof(position_t**) * env->rows);
    for (int i = 0; i < env->rows; i++) {
        pred[i] = (position_t*) malloc(sizeof(position_t) * env->cols);
        heuristique[i] = (double*) malloc(sizeof(double) * env->cols);
        heuristique_propagation[i] = (int*) malloc(sizeof(int) * env->cols);
        heuristique_in_queue[i] = (int*) malloc(sizeof(int) * env->cols);
        dis[i] = (double*) malloc(sizeof(double) * env->cols);
        visited[i] = (int*) malloc(sizeof(int*) * env->cols);
        ptrs[i] = (position_t**) malloc(sizeof(position_t*) * env->cols);
        for (int j = 0; j < env->cols; j++) {
            position_t pos = {.i = i, .j = j};
            pred[i][j] = (position_t) {.i = -1, .j = -1};
            heuristique[i][j] = 0;
            heuristique_propagation[i][j] = 0;
            heuristique_in_queue[i][j] = false;
            visited[i][j] = -1;
            ptrs[i][j] = (position_t*) malloc(sizeof(position_t));
            ptrs[i][j]->i = i;
            ptrs[i][j]->j = j;
        }
    }
}

// Libérer les ressources allouées pour les tableaux
void env_liberer_tableaux(environment_t* env) {
    for (int i = 0; i < env->rows; i++) {
        free(pred[i]);
        free(dis[i]);
```

```
        free(heuristique[i]);
        free(heuristique_propagation[i]);
        free(heuristique_in_queue[i]);
        free(visited[i]);
        for (int j = 0; j < env->cols; j++) {
            free(ptrs[i][j]);
        }
        free(ptrs[i]);
    }
    free(ptrs);
    free(pred);
    free(dis);
    free(heuristique);
    free(heuristique_propagation);
    free(heuristique_in_queue);
    free(visited);
}
```

Annexe | fichier crowd.h

```c
#ifndef CROWD_H
#define CROWD_H


#include "image.h"
#include "image_usage.h"
#include "circular_list.h"
#include "common.h"

struct environment_s {
    int rows;
    int cols;
    int** agents;
    int max;
};
typedef struct environment_s environment_t;

// Créer un environnement à partir d'une image
environment_t env_from_image(image_t image);

// Libérer la mémoire occupée par un environnement
void env_free(environment_t env);

// Modifier une image en fonction de l'environnement
void env_image_edit(image_t image, environment_t env, int n);

// Modifier une image colorée en fonction de l'environnement
void env_image_colored_edit(colored_image_t image, environment_t env, int n);

// Parcourir un environnement avec un A* itératif
void move_env_iterative_a_star(movement_t movement, environment_t* env, int weight0, int alpha, int n);
```

Annexe | fichier crowd.h

```c
// Appliquer plusieurs mouvements à un environnement avec A* itératif
void multiple_move_env_iterative_a_star(circular_list_t* movements, environment_t* env,
                                        int weight0, int alpha, int modulo);

// Initialiser les tableaux nécessaires pour les déplacements dans un environnement
void env_initialiser_tableaux(environment_t* env);

// Libérer les ressources allouées pour les tableaux
void env_liberer_tableaux(environment_t* env);

#endif
```

Annexe | fichier csv.c

```c
#include <stdio.h>
#include <stdlib.h>

#include "csv.h"
#include "common.h"
#include "circular_list.h"

circular_list_t* load_movements(const char* filename, int n) {
    FILE* file = fopen(filename, "r");
    if (!file) return NULL;

    char line[4096];
    // Sauter l'en-tête
    if (!fgets(line, sizeof(line), file)) {
        fclose(file);
        return NULL;
    }

    circular_list_t* cl = cl_create();

    while (fgets(line, sizeof(line), file)) {
        position_t start, target;
        int agents;

        if (sscanf(line, "%d:%d,%d:%d,%d", &start.i, &start.j, &target.i, &target.j, &agents) == 5) {
            start.i /= n;
            start.j /= n;
            target.i /= n;
            target.j /= n;

            movement_t* m = (movement_t*) malloc(sizeof(movement_t));
            m->start = start;
```

```
            m->target = target;
            m->agents = agents;
            cl_add(cl, (void*) m);
        }
    }

    fclose(file);
    return cl;
}

void free_movements(circular_list_t* cl) {
    if (cl_is_empty(cl)) {
        cl_free(cl);
        return;
    }
    movement_t* m = (movement_t*) cl_get(cl);
    free(m);
    cl_remove(cl);
    free_movements(cl);
}

// Écrire le résultat en performance d'une exécution de parcours dans un fichier CSV
void write_result(const char* filename, int modulo, int clocks, double time) {
    fopen(filename, "a");
    FILE* file = fopen(filename, "a");
    if (!file) return;
    fprintf(file, "%d;%d;%lf\n", modulo, clocks, time);
    fclose(file);
}
```

Annexe | fichier csv.h

```c
#ifndef CSV_H
#define CSV_H

#include "common.h"
#include "circular_list.h"

// Charge les mouvements depuis un fichier CSV et retourne la tête de la liste circulaire
circular_list_t* load_movements(const char* filename, int n);

// Libère la mémoire de la liste circulaire
void free_movements(circular_list_t* cl);

// Écrire le résultat en performance d'une exécution de parcours dans un fichier CSV
void write_result(const char* filename, int modulo, int clocks, double time);

#endif // CSV_H
```

Annexe | fichier image.c

```c
#include <opencv2/opencv.hpp>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include "image.h"
#include "logging.h"

// Fonctions pratiques

// Copier une image en niveaux de gris
image_t image_copy(image_t image) {
    log_debug("Copie de l'image : %s", image.name);
    image_t copy = {
        .name = image.name,
        .rows = image.rows,
        .cols = image.cols,
        .pixels = (pixel_t**) malloc(sizeof(pixel_t*) * image.rows),
    };
    for (int i = 0; i < image.rows; i++) {
        copy.pixels[i] = (pixel_t*) malloc(sizeof(pixel_t) * image.cols);
        for (int j = 0; j < image.cols; j++) {
            copy.pixels[i][j] = image.pixels[i][j];
        }
    }
    log_debug("Image copiée : %s", image.name);
    return copy;
}

// Copier une image colorée
colored_image_t colored_image_copy(colored_image_t image) {
```

```c
    log_debug("Copie de l'image colorée : %s", image.name);
    colored_image_t copy = {
        .name = image.name,
        .rows = image.rows,
        .cols = image.cols,
        .pixels = (colored_pixel_t**) malloc(sizeof(colored_pixel_t*) * image.rows),
    };
    for (int i = 0; i < image.rows; i++) {
        copy.pixels[i] = (colored_pixel_t*) malloc(sizeof(colored_pixel_t) * image.cols);
        for (int j = 0; j < image.cols; j++) {
            copy.pixels[i][j] = image.pixels[i][j];
        }
    }
    log_debug("Image colorée copiée : %s", image.name);
    return copy;
}

// Fonctions de gestion de la mémoire

// Libérer la mémoire d'une image en niveaux de gris
void image_free(image_t image) {
    log_debug("Libération de la mémoire de l'image : %s", image.name);
    for (int i = 0; i < image.rows; i++) {
        free(image.pixels[i]);
    }
    free(image.pixels);
    log_debug("Mémoire de l'image libérée : %s", image.name);
}

// Libérer la mémoire d'une image colorée
void colored_image_free(colored_image_t image) {
    log_debug("Libération de la mémoire de l'image colorée : %s", image.name);
```

```
    for (int i = 0; i < image.rows; i++) {
        free(image.pixels[i]);
    }
    free(image.pixels);
    log_debug("Mémoire de l'image colorée libérée : %s", image.name);
}

// Libérer la mémoire d'un noyau de convolution
void kernel_free(kernel_t kernel) {
    log_debug("Libération de la mémoire d'un noyau de convolution");
    for (int i = 0; i < kernel.size; i++) {
        free(kernel.data[i]);
    }
    free(kernel.data);
    log_debug("Mémoire du noyau de convolution libérée");
}

// Fonctions de conversion

// Convertir un cv::Mat en colored_image_t
colored_image_t colored_image_from_mat(cv::Mat mat) {
    log_debug("Conversion d'un cv::Mat en colored_image_t");
    if (mat.empty()) log_fatal("Erreur lors de la conversion : cv::Mat vide");

    colored_image_t image = {
        .name = NULL,
        .rows = mat.rows,
        .cols = mat.cols,
        .pixels = (colored_pixel_t**) malloc(sizeof(colored_pixel_t*) * mat.rows),
    };
    for (int i = 0; i < mat.rows; i++) {
        image.pixels[i] = (colored_pixel_t*) malloc(sizeof(colored_pixel_t) * mat.cols);
```

```
        for (int j = 0; j < mat.cols; j++) {
            cv::Vec3b color = mat.at<cv::Vec3b>(i, j);
            image.pixels[i][j] = (colored_pixel_t) {
                .r = (double) color[2],
                .g = (double) color[1],
                .b = (double) color[0]
            };
        }
    }
    log_debug("Conversion réussie : colored_image_t créé");
    return image;
}

// Convertir un colored_image_t en cv::Mat
cv::Mat cvmat_from_colored_image(colored_image_t colored_image) {
    log_debug("Conversion d'un colored_image_t en cv::Mat");

    cv::Mat mat(colored_image.rows, colored_image.cols, CV_8UC3);

    if (mat.empty()) log_fatal("Erreur lors de la conversion : cv::Mat vide");

    for (int i = 0; i < colored_image.rows; i++) {
        for (int j = 0; j < colored_image.cols; j++) {
            colored_pixel_t pixel = colored_image.pixels[i][j];
            cv::Vec3b color;
            color[0] = (uchar) (pixel.b); // Bleu
            color[1] = (uchar) (pixel.g); // Vert
            color[2] = (uchar) (pixel.r); // Rouge
            mat.at<cv::Vec3b>(i, j) = color;
        }
    }
```

```
    log_debug("Conversion réussie : cv::Mat créé");

    return mat;
}

// Convertir un image_t en cv::Mat
cv::Mat cvmat_from_image(image_t image) {
    log_debug("Conversion d'un image_t en cv::Mat");

    cv::Mat mat(image.rows, image.cols, CV_8UC1);

    if (mat.empty()) log_fatal("Erreur lors de la conversion : cv::Mat vide");

    for (int i = 0; i < image.rows; i++) {
        for (int j = 0; j < image.cols; j++) {
            // Convertir les pixels en valeurs entre 0 et 255
            mat.at<uchar>(i, j) = (uchar)(image.pixels[i][j] * 255.0);
        }
    }
    log_debug("Conversion réussie : cv::Mat créé");

    return mat;
}

// Convertir un colored_image_t en image_t (niveaux de gris avec pondérations)
image_t image_from_colored_image(colored_image_t colored_image) {
    log_debug("Conversion d'un colored_image_t en image_t : %s", colored_image.name);
    image_t image = {
        .name = colored_image.name,
        .rows = colored_image.rows,
        .cols = colored_image.cols,
        .pixels = (pixel_t**) malloc(sizeof(pixel_t*) * colored_image.rows),
```

```
    };

    for (int i = 0; i < colored_image.rows; i++) {
        image.pixels[i] = (pixel_t*) malloc(sizeof(pixel_t) * colored_image.cols);
        for (int j = 0; j < colored_image.cols; j++) {
            colored_pixel_t pixel = colored_image.pixels[i][j];
            // Pondérations standard pour convertir en niveaux de gris et normalisation
            image.pixels[i][j] = (0.299 * pixel.r + 0.587 * pixel.g + 0.114 * pixel.b) / 255.0;
        }
    }
    log_debug("Conversion réussie : %s", colored_image.name);
    return image;
}

// Fonctions de lecture et d'écriture d'images

// Lire une image en niveaux de gris
colored_image_t image_read(const char* path) {
    // Lire l'image avec OpenCV
    log_debug("Lecture de l'image : %s", path);

    cv::Mat mat = cv::imread(path, cv::IMREAD_COLOR);
    if (mat.empty()) log_fatal("Erreur lors de la lecture de l'image : %s", path);

    colored_image_t image = colored_image_from_mat(mat);
    image.name = strdup(path); // Dupliquer le nom du fichier
    return image;
}

// Ecrire une image en niveaux de gris
void image_write(image_t image, const char* path) {
    log_debug("Écriture de l'image : %s dans %s", image.name, path);
```

```
    cv::Mat img = cvmat_from_image(image);
    cv::imwrite(path, img);

    log_debug("Image écrite : %s dans %s", image.name, path);
}

// Ecrire une image colorée
void colored_image_write(colored_image_t image, const char* path) {
    log_debug("Écriture de l'image colorée : %s dans %s", image.name, path);

    cv::Mat img = cvmat_from_colored_image(image);
    cv::imwrite(path, img);

    log_debug("Image colorée écrite : %s dans %s", image.name, path);
}

// Fonctions d'affichage

// Afficher une image colorée
void colored_image_show(colored_image_t image) {
    log_debug("Affichage de l'image colorée : %s", image.name);

    cv::Mat img = cvmat_from_colored_image(image);
    cv::namedWindow(image.name, cv::WINDOW_NORMAL);
    cv::resizeWindow(image.name, 600, 600);
    cv::imshow(image.name, img);

    log_debug("Image colorée affichée (en attente d'action) : %s", image.name);
    cv::waitKey(0); // se ferme après un appui sur une touche
    log_debug("Fermeture de l'image colorée : %s", image.name);
}
```

Annexe | fichier image.c

```c
// Afficher une image en niveaux de gris
void image_show(image_t image) {
    log_debug("Affichage de l'image en niveaux de gris : %s", image.name);

    // Convertir l'image_t en cv::Mat
    cv::Mat mat = cvmat_from_image(image);

    // Afficher l'image avec OpenCV
    cv::namedWindow(image.name, cv::WINDOW_NORMAL);
    cv::resizeWindow(image.name, 600, 600);
    cv::imshow(image.name, mat);
    log_debug("Image en niveaux de gris affichée (en attente d'action): %s", image.name);
    cv::waitKey(0); // Se ferme après un appui sur une touche
    log_debug("Fermeture de l'image en niveaux de gris : %s", image.name);
}

// Fonctions de manipulation d'images

// Réduire la taille d'une image
image_t image_resize(image_t image, int scale) {
    log_debug("Redimensionnement de l'image : %s avec un facteur de réduction de %d", image.name, scale);
    if (scale <= 0) log_fatal("Erreur de redimensionnement : facteur de réduction invalide (%d)", scale);

    int new_rows = image.rows / scale;
    int new_cols = image.cols / scale;
    image_t scaled = {
        .name = image.name,
        .rows = new_rows,
        .cols = new_cols,
        .pixels = (pixel_t**) malloc(sizeof(pixel_t*) * new_rows),
    };
```

```
    // Parcours des pixels
    for (int i = 0; i < new_rows; i++) {
        scaled.pixels[i] = (pixel_t*) malloc(sizeof(pixel_t) * new_cols);
        for (int j = 0; j < new_cols; j++) {
            // Moyenne des pixels voisins
            pixel_t pixel_moyen = 0.;
            int count = 0;
            for (int x = 0; x < scale && i * scale + x < image.rows; x++) {
                for (int y = 0; y < scale && j * scale + y < image.cols; y++) {
                    pixel_t p = image.pixels[i * scale + x][j * scale + y];
                    pixel_moyen += p;
                    count++;
                }
            }
            pixel_moyen /= count;
            scaled.pixels[i][j] = pixel_moyen;
        }
    }
    log_debug("Image redimensionnée : %s avec un facteur de réduction de %d", image.name, scale);
    return scaled;
}

// Appliquer un filtre à une image (réflexion de l'image aux bords)
image_t image_apply_filter(image_t image, kernel_t kernel) {
    log_debug("Application d'un filtre à l'image : %s", image.name);
    image_t result = image_copy(image);

    int border = kernel.size / 2;
    for (int i = 0; i < image.rows; i++) {
        for (int j = 0; j < image.cols; j++) {
            pixel_t intensity = 0;
            for (int x = 0; x < kernel.size; x++) {
```

```
                for (int y = 0; y < kernel.size; y++) {
                    int ni = i + x - border;
                    int nj = j + y - border;

                    if (ni < 0) ni = -ni; // Réflexion aux bords
                    if (nj < 0) nj = -nj; // Réflexion aux bords
                    if (ni >= image.rows) ni = 2 * image.rows - ni - 1; // Réflexion aux bords
                    if (nj >= image.cols) nj = 2 * image.cols - nj - 1; // Réflexion aux bords

                    pixel_t pixel = image.pixels[ni][nj];
                    intensity += pixel * kernel.data[x][y];
                }
            }
            result.pixels[i][j] = intensity;
        }
    }
    log_debug("Filtre appliqué à l'image : %s", image.name);
    return result;
}
```

Annexe | fichier image.h

```c
#ifndef IMAGE_H
#define IMAGE_H

#include <opencv2/opencv.hpp>

// Définition des structures

// Structure représentant un pixel coloré (RGB)
typedef struct colored_pixel_s {
    double r; // Rouge
    double g; // Vert
    double b; // Bleu
} colored_pixel_t;

// Structure représentant une image colorée
typedef struct colored_image_s {
    char* name;
    int rows;
    int cols;
    colored_pixel_t** pixels;
} colored_image_t;

// Structure représentant un pixel en niveaux de gris (double entre 0 et 1)
typedef double pixel_t;

// Structure représentant une image en niveaux de gris
typedef struct image_s {
    char* name;
    int rows;
    int cols;
    pixel_t** pixels;
} image_t;
```

Annexe | fichier image.h

```c
// Structure représentant un noyau de convolution
typedef struct kernel_s {
    int size;
    double** data; // tableau de taille size*size
} kernel_t;

// Fonctions pratiques
image_t image_copy(image_t image);
colored_image_t colored_image_copy(colored_image_t image);

// Fonctions de gestion de la mémoire
void image_free(image_t image);
void colored_image_free(colored_image_t image);
void kernel_free(kernel_t kernel);

// Fonctions de conversion
colored_image_t colored_image_from_mat(cv::Mat mat);
cv::Mat cvmat_from_colored_image(colored_image_t colored_image);
cv::Mat cvmat_from_image(image_t image);
image_t image_from_colored_image(colored_image_t colored_image);

// Fonctions de lecture et d'écriture d'images
colored_image_t image_read(const char* filename);
void image_write(image_t image, const char* filename);
void colored_image_write(colored_image_t image, const char* filename);

// Fonctions d'affichage
void colored_image_show(colored_image_t image);
void image_show(image_t image);

// Fonctions de manipulation d'images
```

Annexe | fichier image.h

```
image_t image_resize(image_t image, int scale);
image_t image_apply_filter(image_t image, kernel_t kernel);

#endif // IMAGE_H
```

Annexe | fichier image_usage.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>


#include "image.h"
#include "image_usage.h"
#include "queue.h"
#include "priority_queue.h"
#include "logging.h"
#include "common.h"


// Créer un noyau gaussien de taille size et d'écart-type sigma
kernel_t create_gaussian_kernel(int size, double sigma) {
    log_debug("Création d'un noyau gaussien de taille %d et d'écart-type %.2f", size, sigma);
    kernel_t kernel = {
        .size = size,
        .data = (double**) malloc(sizeof(double*) * size)
    };
    double mean = size / 2;
    double sum = 0.0;
    for (int x = 0; x < size; x++) {
        kernel.data[x] = (double*) malloc(sizeof(double) * size);
        for (int y = 0; y < size; y++) {
            kernel.data[x][y] = exp(-0.5 * (pow((x - mean) / sigma, 2.0) + pow((y - mean) / sigma, 2.0)))
                                / (2 * M_PI * sigma * sigma);
            sum += kernel.data[x][y];
        }
    }
    // Normalisation du noyau
```

```c
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            kernel.data[i][j] /= sum;
        }
    }
    log_debug("Noyau gaussien créé de taille %d et d'écart-type %.2f", size, sigma);
    return kernel;
}

// Créer le noyau de Sobel selon l'axe des x
kernel_t create_sobel_kernel_x() {
    log_debug("Création du noyau de Sobel selon l'axe des x");
    kernel_t kernel = {
        .size = 3,
        .data = (double**) malloc(sizeof(double*) * 3)
    };
    for (int i = 0; i < 3; i++) {
        kernel.data[i] = (double*) malloc(sizeof(double) * 3);
    }

    kernel.data[0][0] = -1; kernel.data[0][1] = -2; kernel.data[0][2] = -1;
    kernel.data[1][0] = 0; kernel.data[1][1] = 0; kernel.data[1][2] = 0;
    kernel.data[2][0] = 1; kernel.data[2][1] = 2; kernel.data[2][2] = 1;

    log_debug("Noyau de Sobel créé selon l'axe des x");

    return kernel;
}

// Créer le noyau de Sobel selon l'axe des y
kernel_t create_sobel_kernel_y() {
    log_debug("Création du noyau de Sobel selon l'axe des y");
```

Annexe | fichier image_usage.c

```
    kernel_t kernel = {
        .size = 3,
        .data = (double**) malloc(sizeof(double*) * 3)
    };
    for (int i = 0; i < 3; i++) {
        kernel.data[i] = (double*) malloc(sizeof(double) * 3);
    }

    kernel.data[0][0] = -1; kernel.data[0][1] = 0; kernel.data[0][2] = 1;
    kernel.data[1][0] = -2; kernel.data[1][1] = 0; kernel.data[1][2] = 2;
    kernel.data[2][0] = -1; kernel.data[2][1] = 0; kernel.data[2][2] = 1;

    log_debug("Noyau de Sobel créé selon l'axe des y");

    return kernel;
}

// Appliquer un filtre de Sobel à un image_t (avec extension de l'image) et calcule les gradients
void image_apply_sobel(image_t image, image_t* gradient_x, image_t* gradient_y) {
    log_debug("Application du filtre de Sobel à l'image : %s", image.name);
    kernel_t kernel_x = create_sobel_kernel_x();
    kernel_t kernel_y = create_sobel_kernel_y();

    image_t copy1 = image_copy(image);
    image_t copy2 = image_copy(image);

    *gradient_x = image_apply_filter(copy1, kernel_x);
    *gradient_y = image_apply_filter(copy2, kernel_y);

    // Variables pour la normalisation
    double g_max = 0.;
    double g_min = 1.;
```

Annexe | fichier image_usage.c

```
    // Première étape de calcul
    for (int i = 0; i < image.rows; i++) {
        for (int j = 0; j < image.cols; j++) {
            image.pixels[i][j] = sqrt(pow(gradient_x->pixels[i][j], 2) + pow(gradient_y->pixels[i][j], 2));
        }
    }

    // Normalisation
    for (int i = 0; i < image.rows; i++) {
        for (int j = 0; j < image.cols; j++) {
            if (image.pixels[i][j] < g_min) g_min = image.pixels[i][j];
            if (image.pixels[i][j] > g_max) g_max = image.pixels[i][j];
        }
    }
    if (g_max != g_min) {
        for (int i = 0; i < image.rows; i++) {
            for (int j = 0; j < image.cols; j++) {
                image.pixels[i][j] = (image.pixels[i][j] - g_min)/(g_max - g_min);
            }
        }
    }


    image_free(copy1);
    image_free(copy2);
    kernel_free(kernel_x);
    kernel_free(kernel_y);

    log_debug("Filtre de Sobel appliqué à l'image : %s", image.name);
}
```

Annexe | fichier image_usage.c

```c
// Calculer la direction des gradients
image_t image_compute_gradient_direction(image_t gradient_x, image_t gradient_y) {
    log_debug("Calcul de la direction des gradients");
    image_t direction = image_copy(gradient_x);

    for (int i = 0; i < gradient_x.rows; i++) {
        for (int j = 0; j < gradient_x.cols; j++) {
            direction.pixels[i][j] = atan2(gradient_x.pixels[i][j], gradient_y.pixels[i][j]);
        }
    }
    log_debug("Direction des gradients calculée");

    return direction;
}


// Suppression des non-maxima locaux
image_t image_non_maxima_suppression(image_t image, image_t direction) {
    log_debug("Suppression des non-maxima locaux");
    image_t result = image_copy(image);

    for (int i = 1; i < image.rows-1; i++) {
        for (int j = 1; j < image.cols-1; j++) {
            double angle = direction.pixels[i][j]; // angle en radians
            angle = fmod(angle + M_PI, M_PI); // angle entre 0 et pi

            double q = 0.0;
            double r = 0.0;

            if ((angle >= 0 && angle < M_PI/8) || (angle >= 7*M_PI/8 && angle < M_PI)) {
                q = image.pixels[i][j+1];
```

```
                r = image.pixels[i][j-1];
            } else if (angle >= M_PI/8 && angle < 3*M_PI/8) {
                q = image.pixels[i-1][j+1];
                r = image.pixels[i+1][j-1];
            } else if (angle >= 3*M_PI/8 && angle < 5*M_PI/8) {
                q = image.pixels[i-1][j];
                r = image.pixels[i+1][j];
            } else if (angle >= 5*M_PI/8 && angle < 7*M_PI/8) {
                q = image.pixels[i-1][j-1];
                r = image.pixels[i+1][j+1];
            }

            if (image.pixels[i][j] >= q && image.pixels[i][j] >= r) {
                result.pixels[i][j] = image.pixels[i][j];
            } else {
                result.pixels[i][j] = 0;
            }
        }
    }
    image_free(direction);

    log_debug("Suppression des non-maxima locaux terminée");
    return result;
}


// Appliquer un double seuil
void image_double_threshold(image_t image, double t_max, double t_min) {
    log_debug("Application d'un double seuil : t_max = %.2f, t_min = %.2f", t_max, t_min);
    for (int i = 0; i < image.rows; i++) {
        for (int j = 0; j < image.cols; j++) {
            // Si l'intensité est assez forte on garde le pixel
```

```
            if (image.pixels[i][j] > t_max) {
                image.pixels[i][j] = 1.;
            }
            // Si elle est trop faible on le supprime
            else if (image.pixels[i][j] < t_min) {
                image.pixels[i][j] = 0.;
            }
            // Si elle est entre les deux seuils on regardera si un voisin est assez fort
            else {
                image.pixels[i][j] = 1/2.;
            }
        }
    }
    log_debug("Double seuil appliqué");
}

// Tracer les contours en contact avec un pixel fort
void image_hysteresis_aux(image_t image, bool** visited, int i, int j, queue_t* queue) {
    position_t* pos = (position_t*) malloc(sizeof(position_t));
    pos->i = i;
    pos->j = j;
    queue_enqueue(queue, (void*) pos);
    visited[i][j] = true;

    while (!queue_is_empty(queue)) {
        position_t* p = (position_t*) queue_dequeue(queue);
        image.pixels[p->i][p->j] = 1.; // Marquer le pixel comme fort

        // Vérifier les voisins
        for (int di = -1; di <= 1; di++) {
            for (int dj = -1; dj <= 1; dj++) {
                if (di == 0 && dj == 0) continue; // Ignorer le pixel central
```

```
                int ni = p->i + di;
                int nj = p->j + dj;

                if (ni >= 0 && ni < image.rows && nj >= 0 && nj < image.cols &&
                    !visited[ni][nj] && image.pixels[ni][nj] != 0) {
                    visited[ni][nj] = true;

                    position_t* new_pos = (position_t*) malloc(sizeof(position_t));
                    new_pos->i = ni;
                    new_pos->j = nj;

                    queue_enqueue(queue, (void*) new_pos);
                }
            }
        }
        free(p);
    }
}

// Tracer les contours d'une image avec une hystérésis
void image_hysteresis(image_t image) {
    log_debug("Application de l'hystérésis sur l'image : %s", image.name);
    // Initialiser la matrice des pixels visités
    bool** visited = (bool**) malloc(sizeof(bool*) * image.rows);
    for (int i = 0; i < image.rows; i++) {
        visited[i] = (bool*) malloc(sizeof(bool) * image.cols);
        for (int j = 0; j < image.cols; j++) {
            visited[i][j] = false;
        }
    }

    // Initialiser la queue
```

```c
    queue_t* queue = queue_create();

    // Pour chaque pixel fort, rendre fort les pixels faibles connectés
    for (int i = 0; i < image.rows; i++) {
        for (int j = 0; j < image.cols; j++) {
            if (!visited[i][j] && image.pixels[i][j] == 1.) {
                image_hysteresis_aux(image, visited, i, j, queue);
            }
        }
    }

    // Supprimer les pixels faibles restant
    for (int i = 0; i < image.rows; i++) {
        for (int j = 0; j < image.cols; j++) {
            if (image.pixels[i][j] != 1.) image.pixels[i][j] = 0.;
        }
    }

    // Libérer les ressources
    for (int i = 0; i < image.rows; i++) {
        free(visited[i]);
    }
    free(visited);
    queue_free(queue);

    log_debug("Hystérésis appliquée sur l'image : %s", image.name);
}

// Application du filtre de Canny
image_t canny(image_t image, double t_max, double t_min) {
    log_debug("Application du filtre de Canny sur l'image : %s", image.name);
```

Annexe | fichier image_usage.c

```c
    // Flou gaussien
    kernel_t kernel = create_gaussian_kernel(5, 1.0);
    image_t blured_image = image_apply_filter(image, kernel);
    kernel_free(kernel);

    // Appliquer le filtre de Sobel
    image_t gradient_x, gradient_y;
    image_apply_sobel(blured_image, &gradient_x, &gradient_y);

    // Calculer la direction des gradients
    image_t direction = image_compute_gradient_direction(gradient_x, gradient_y);
    image_free(gradient_x);
    image_free(gradient_y);

    // Suppression des non-maxima locaux
    image_t non_maxima = image_non_maxima_suppression(blured_image, direction);
    image_free(blured_image);

    // Appliquer un double seuil
    image_double_threshold(non_maxima, t_max, t_min);

    // Appliquer l'hystérésis
    image_hysteresis(non_maxima);

    log_debug("Filtre de Canny appliqué sur l'image : %s", image.name);

    return non_maxima;
}

// Rendre continue les contours de l'image
image_t image_fermeture_morphologique(image_t image, int size) {
    log_debug("Application de la fermeture morphologique sur l'image : %s", image.name);
```

Annexe | fichier image_usage.c

```
image_t result_dilatation = image_copy(image);

// Dilatation
int mean = size / 2;
for (int i = 0; i < image.rows; i++) {
    for (int j = 0; j < image.cols; j++) {
        bool to_dilate = false;
        for (int x = -mean; x <= mean && !to_dilate; x++) {
            for (int y = -mean; y <= mean; y++) {
                int ni = i + x;
                int nj = j + y;
                if (!(ni >= 0 && ni < image.rows && nj >= 0 && nj < image.cols)) continue;
                if (image.pixels[ni][nj] > 0) {
                    to_dilate = true;
                    break;
                }
            }
        }
        if (to_dilate) {
            result_dilatation.pixels[i][j] = 1.0; // Dilater le pixel
        }
        else {
            result_dilatation.pixels[i][j] = image.pixels[i][j]; // Garder le pixel
        }
    }
}

// Erosion
image_t result_erosion = image_copy(result_dilatation);
for (int i = 0; i < image.rows; i++) {
    for (int j = 0; j < image.cols; j++) {
        bool to_erode = false;
```

Annexe | fichier image_usage.c

```
            for (int x = -mean; x <= mean && !to_erode; x++) {
                for (int y = -mean; y <= mean; y++) {
                    int ni = i + x;
                    int nj = j + y;
                    if (!(ni >= 0 && ni < image.rows && nj >= 0 && nj < image.cols)) continue;
                    if (result_dilatation.pixels[ni][nj] < 1.0) {
                        to_erode = true;
                        break;
                    }
                }
            }
            if (to_erode) {
                result_erosion.pixels[i][j] = 0.0; // Eroder le pixel
            }
            else {
                result_erosion.pixels[i][j] = result_dilatation.pixels[i][j]; // Garder le pixel
            }
        }
    }
    image_free(result_dilatation);

    log_debug("Fermeture morphologique appliquée sur l'image : %s", image.name);
    return result_erosion;
}
```

Annexe | fichier image_usage.h

```c
#ifndef IMAGE_USAGE_H
#define IMAGE_USAGE_H

#include "image.h"
#include "queue.h"
#include "priority_queue.h"
#include "common.h"


// Crée un noyau gaussien
kernel_t create_gaussian_kernel(int size, double sigma);

// Crée le noyau de Sobel pour l'axe x
kernel_t create_sobel_kernel_x();

// Crée le noyau de Sobel pour l'axe y
kernel_t create_sobel_kernel_y();

// Applique un filtre de Sobel pour calculer les gradients
void image_apply_sobel(image_t image, image_t* gradient_x, image_t* gradient_y);

// Calcule la direction des gradients
image_t image_compute_gradient_direction(image_t gradient_x, image_t gradient_y);

// Supprime les non-maxima locaux
image_t image_non_maxima_suppression(image_t image, image_t direction);

// Applique un double seuil à une image
void image_double_threshold(image_t image, double t_max, double t_min);

// Applique une hystérésis pour tracer les contours
void image_hysteresis(image_t image);
```

Annexe | fichier image_usage.h

```c
// Application du filtre de Canny
image_t canny(image_t image, double t_max, double t_min);

// Rendre continue les contours de l'image
image_t image_fermeture_morphologique(image_t image, int size);

#endif // IMAGE_USAGE_H
```

Annexe | fichier logging.c

```c
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>

#include "logging.h"
#include "config.h"

#define LOG_BUFFER_SIZE 4096 // On suppose que 4096 est suffisant pour la plupart des messages

// Affiche un message de debug (niveau 2)
void log_debug(const char* message, ...) {
    if (DEBUG_MODE >= 2) {
        char buffer[LOG_BUFFER_SIZE];
        va_list args;
        va_start(args, message);
        vsnprintf(buffer, LOG_BUFFER_SIZE, message, args);
        va_end(args);
        fprintf(stderr, "[DEBUG] %s\n", buffer);
    }
}

// Affiche un message d'erreur (niveau 1)
void log_error(const char* message, ...) {
    if (DEBUG_MODE >= 1) {
        char buffer[LOG_BUFFER_SIZE];
        va_list args;
        va_start(args, message);
        vsnprintf(buffer, LOG_BUFFER_SIZE, message, args);
        va_end(args);
        fprintf(stderr, "[ERROR] %s\n", buffer);
    }
}
```

Annexe | fichier logging.c

```c
// Affiche un message d'information (niveau 0)
void log_info(const char* message, ...) {
    if (DEBUG_MODE >= 0) {
        char buffer[LOG_BUFFER_SIZE];
        va_list args;
        va_start(args, message);
        vsnprintf(buffer, LOG_BUFFER_SIZE, message, args);
        va_end(args);
        fprintf(stderr, "[INFO] %s\n", buffer);
    }
}

// Affiche un message d'avertissement (niveau 0)
void log_warning(const char* message, ...) {
    if (DEBUG_MODE >= 0) {
        char buffer[LOG_BUFFER_SIZE];
        va_list args;
        va_start(args, message);
        vsnprintf(buffer, LOG_BUFFER_SIZE, message, args);
        va_end(args);
        fprintf(stderr, "[WARNING] %s\n", buffer);
    }
}

// Affiche un message fatal et termine le programme
void log_fatal(const char* message, ...) {
    char buffer[LOG_BUFFER_SIZE];
    va_list args;
    va_start(args, message);
    vsnprintf(buffer, LOG_BUFFER_SIZE, message, args);
    va_end(args);
```

Annexe | fichier logging.c

```
    fprintf(stderr, "[FATAL] %s\n", buffer);
    exit(EXIT_FAILURE);
}
```

Annexe | fichier logging.h

```c
#ifndef LOGGING_H
#define LOGGING_H


void log_debug(const char* message, ...);
void log_error(const char* message, ...);
void log_info(const char* message, ...);
void log_warning(const char* message, ...);
void log_fatal(const char* message, ...);


#endif
```

Annexe | fichier priority_queue.c

```c
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>

#include "priority_queue.h"


// Créer une file de priorité
priority_queue_t* pq_create(int capacity) {
    priority_queue_t* pq = (priority_queue_t*) malloc(sizeof(priority_queue_t));
    pq->nodes = (heap_node_t*) malloc(sizeof(heap_node_t) * capacity);
    pq->len = 0;
    pq->capacity = capacity;
    return pq;
}

// Libérer une file de priorité
void pq_free(priority_queue_t* pq) {
    free(pq->nodes);
    free(pq);
}

// Échanger deux noeuds dans le tas
void swap(heap_node_t* a, heap_node_t* b) {
    heap_node_t temp = *a;
    *a = *b;
    *b = temp;
}

// Maintenir la propriété du tas (min-heap) en descendant
void percolate_down(priority_queue_t* pq, int index) {
    int smallest = index;
```

```c
    int left = 2 * index + 1;
    int right = 2 * index + 2;

    if (left < pq->len && pq->nodes[left].priority < pq->nodes[smallest].priority) {
        smallest = left;
    }
    if (right < pq->len && pq->nodes[right].priority < pq->nodes[smallest].priority) {
        smallest = right;
    }
    if (smallest != index) {
        swap(&pq->nodes[index], &pq->nodes[smallest]);
        percolate_down(pq, smallest);
    }
}

// Maintenir la propriété du tas (min-heap) en remontant
void percolate_up(priority_queue_t* pq, int index) {
    int parent = (index - 1) / 2;

    if (index > 0 && pq->nodes[index].priority < pq->nodes[parent].priority) {
        swap(&pq->nodes[index], &pq->nodes[parent]);
        percolate_up(pq, parent);
    }
}

// Ajouter un élément à la file de priorité
void pq_push(priority_queue_t* pq, double priority, void* value) {
    if (pq->len == pq->capacity) {
        printf("Erreur : capacité maximale atteinte dans la file de priorité.\n");
        exit(-1);
    }
```

```
    pq->nodes[pq->len].priority = priority;
    pq->nodes[pq->len].value = value;
    pq->len++;
    percolate_up(pq, pq->len - 1);
}

// Extraire l'élément avec la plus petite priorité
void* pq_pop(priority_queue_t* pq) {
    if (pq->len == 0) {
        fprintf(stderr, "Erreur : la file de priorité est vide.\n");
        exit(-1);
    }

    void* min_position = pq->nodes[0].value;
    pq->nodes[0] = pq->nodes[pq->len - 1];
    pq->len--;
    percolate_down(pq, 0);

    return min_position;
}

// Vérifier si la file de priorité est vide
bool pq_is_empty(priority_queue_t* pq) {
    return pq->len == 0;
}
```

Annexe | fichier priority_queue.h

```c
#ifndef PRIORITY_QUEUE_H
#define PRIORITY_QUEUE_H

#include <stdbool.h>

// Définition des structures
typedef struct heap_node_s {
    double priority; // La priorité (plus petite est meilleure)
    void* value;     // La valeur associée
} heap_node_t;

typedef struct priority_queue_s {
    heap_node_t* nodes; // Tableau de nœuds
    int len;            // Nombre d'éléments dans la file
    int capacity;       // Capacité maximale de la file
} priority_queue_t;

// Fonctions pour manipuler la file de priorité
priority_queue_t* pq_create(int capacity); // Créer une file de priorité
void pq_free(priority_queue_t* pq); // Libérer une file de priorité
void pq_push(priority_queue_t* pq, double priority, void* value); // Ajouter un élément
void* pq_pop(priority_queue_t* pq); // Extraire l'élément avec la plus petite priorité
bool pq_is_empty(priority_queue_t* pq); // Vérifier si la file est vide

#endif // PRIORITY_QUEUE_H
```

Annexe | fichier queue.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include "queue.h"


struct maillon_s {
    void* value;
    struct maillon_s* next;
};
typedef struct maillon_s maillon_t;

struct queue_s {
    maillon_t* head;
    maillon_t* tail;
    int len;
};
typedef struct queue_s queue_t;

// Créer un maillon
maillon_t* maillon_create(void* value) {
    maillon_t* m = (maillon_t*) malloc(sizeof(maillon_t));
    m->value = value;
    m->next = NULL;
    return m;
}

// Initialiser une queue
queue_t* queue_create() {
    queue_t* q = (queue_t*) malloc(sizeof(queue_t));
    q->head = NULL;
```

```
    q->tail = NULL;
    q->len = 0;
    return q;
}

// Ajouter un élément à la queue (enqueue)
void queue_enqueue(queue_t* q, void* value) {
    maillon_t* new_maillon = maillon_create(value);
    if (q->tail == NULL) {
        q->head = new_maillon;
        q->tail = new_maillon;
    } else {
        q->tail->next = new_maillon;
        q->tail = new_maillon;
    }
    q->len++;
}

// Retirer un élément de la queue (dequeue)
void* queue_dequeue(queue_t* q) {
    if (q->head == NULL) {
        fprintf(stderr, "Erreur : la queue est vide.\n");
        exit(-1);
    }
    maillon_t* temp = q->head;
    void* value = temp->value;
    q->head = q->head->next;
    if (q->head == NULL) {
        q->tail = NULL;
    }
    q->len--;
    free(temp);
```

Annexe | fichier queue.c

```
    return value;
}

// Vérifier si la queue est vide
bool queue_is_empty(queue_t* q) {
    return q->head == NULL;
}

// Libérer une queue
void queue_free(queue_t* q) {
    while (!queue_is_empty(q)) {
        queue_dequeue(q);
    }
    free(q);
}
```

Annexe | fichier queue.h

```c
#ifndef QUEUE_H
#define QUEUE_H

#include <stdbool.h>

// Définition des structures
typedef struct queue_s queue_t;

// Fonctions pour manipuler la queue
queue_t* queue_create(); // Créer une nouvelle queue
void queue_enqueue(queue_t* q, void* value); // Ajouter un élément à la queue
void* queue_dequeue(queue_t* q); // Retirer un élément de la queue
bool queue_is_empty(queue_t* q); // Vérifier si la queue est vide
void queue_free(queue_t* q); // Libérer la mémoire de la queue

#endif // QUEUE_H
```