TIPE : Modélisation d'une foule d'usagers du métro parisien

Louis Thevenet

Épreuve de TIPE

Session 2023

Plan de l'exposé

Craig Reynolds (1987) : Premier modèle de mouvement de foule : les boids

- Représentation de nuées d'oiseaux
- Règles très simples
- Vitesses et trajectoires des agents liées aux voisins proches

Craig Reynolds (1987) : Premier modèle de mouvement de foule : les boids

- Représentation de nuées d'oiseaux
- Règles très simples
- Vitesses et trajectoires des agents liées aux voisins proches

Craig Reynolds (1987) : Premier modèle de mouvement de foule : les boids

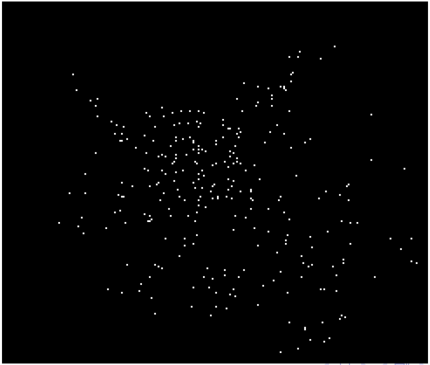
- Représentation de nuées d'oiseaux
- Règles très simples
- Vitesses et trajectoires des agents liées aux voisins proches

Craig Reynolds (1987) : Premier modèle de mouvement de foule : les boids

- Représentation de nuées d'oiseaux
- Règles très simples
- Vitesses et trajectoires des agents liées aux voisins proches

Craig Reynolds (1987) : Premier modèle de mouvement de foule : les boids

- Représentation de nuées d'oiseaux
- Règles très simples
- Vitesses et trajectoires des agents liées aux voisins proches



Dirk Helbing et Péter Molnar (1998) : Le concept de "forces sociales"

- Force d'attraction sociale : les trajectoires peuvent être influencées
- Force de répulsion : les contacts physiques sont évités

Julien Pettre et Wouter Van Toll (2021) : Evitement de collision

- Les trajectoires sont adoucies et plus réalistes en fonction des obstacles

Dirk Helbing et Péter Molnar (1998) : Le concept de "forces sociales"

- Force d'attraction sociale : les trajectoires peuvent être influencées
- Force de répulsion : les contacts physiques sont évités

Julien Pettre et Wouter Van Toll (2021) : Evitement de collision

- Les trajectoires sont adoucies et plus réalistes en fonction des obstacles

Dirk Helbing et Péter Molnar (1998) : Le concept de "forces sociales"

- Force d'attraction sociale : les trajectoires peuvent être influencées
- Force de répulsion : les contacts physiques sont évités

Julien Pettre et Wouter Van Toll (2021) : Evitement de collision

- Les trajectoires sont adoucies et plus réalistes en fonction des obstacles

Dirk Helbing et Péter Molnar (1998) : Le concept de "forces sociales"

- Force d'attraction sociale : les trajectoires peuvent être influencées
- Force de répulsion : les contacts physiques sont évités

Julien Pettre et Wouter Van Toll (2021) : Evitement de collision

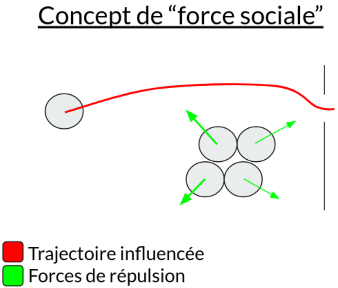
- Les trajectoires sont adoucies et plus réalistes en fonction des obstacles

Dirk Helbing et Péter Molnar (1998) : Le concept de "forces sociales"

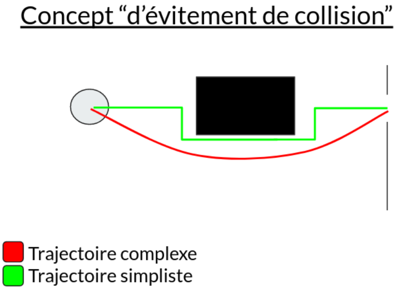
- Force d'attraction sociale : les trajectoires peuvent être influencées
- Force de répulsion : les contacts physiques sont évités

Julien Pettre et Wouter Van Toll (2021) : Evitement de collision

- Les trajectoires sont adoucies et plus réalistes en fonction des obstacles



Concept de “force sociale”
Trajectoire influencée
Forces de répulsion



Concept “d’évitement de collision”
Trajectoire complexe
Trajectoire simpliste

Problématisation

- Quelles sont les lois essentielles qui permettent de décrire le mouvement d'une foule à l'échelle de l'individu ?
- Comment implémenter une telle modélisation afin de vérifier la cohérence architecturale de lieux très fréquentés ?

Problématisation

- Quelles sont les lois essentielles qui permettent de décrire le mouvement d'une foule à l'échelle de l'individu ?
- Comment implémenter une telle modélisation afin de vérifier la cohérence architecturale de lieux très fréquentés ?

Problématisation

- Quelles sont les lois essentielles qui permettent de décrire le mouvement d'une foule à l'échelle de l'individu ?
- Comment implémenter une telle modélisation afin de vérifier la cohérence architecturale de lieux très fréquentés ?

Modélisation

- Modèle mathématique simple
- Adaptation informatique
- Amélioration du modèle

Modélisation

- Modèle mathématique simple
- Adaptation informatique
- Amélioration du modèle

Modélisation

- Modèle mathématique simple
- Adaptation informatique
- Amélioration du modèle

Modélisation

- Modèle mathématique simple
- Adaptation informatique
- Amélioration du modèle

On restreint l'étude à deux dimensions, en **discrétisant** le temps et le plan :

- Carte $\leftrightarrow P_{i,j} \in M_{n,p}(\mathbb{N}), (n,p) \in \mathbb{N}^2$
- Agent $\leftrightarrow position, but \in [\![1,n]\!] \times [\![1,p]\!]$

On restreint l'étude à deux dimensions, en **discrétisant** le temps et le plan :

- Carte $\leftrightarrow P_{i,j} \in M_{n,p}(\mathbb{N}), (n,p) \in \mathbb{N}^2$
- Agent $\leftrightarrow position, but \in [\![1,n]\!] \times [\![1,p]\!]$

On restreint l'étude à deux dimensions, en **discrétisant** le temps et le plan :

- Carte $\leftrightarrow P_{i,j} \in M_{n,p}(\mathbb{N}), (n,p) \in \mathbb{N}^2$
- Agent $\leftrightarrow position, but \in [\![1, n]\!] \times [\![1, p]\!]$

```c
struct location {
  int y;
  int x;
}

struct person {
  struct location pos;
  struct location goal;
}
```

```c
struct map {
int **level;       // plan de la simulation
int start_nb;      // nombre d'entrées
location *starts;  // et leurs positions
int exit_nb;       // nombre de sorties
location *exits;   // et leurs positions
int width;         // largeur
int height;        // hauteur
}
```

$$map \rightarrow level[i][j] \leftrightarrow \text{ nombre de personnes à la position } (i, j)$$

On définit des fonctions :

- Simulation
  - *charger_carte*
  - *intention*
  - *deplacement*
- Génération des résultats
  - *creer_image*
  - *sauvegarder_image*

On définit des fonctions :

- Simulation
  - *charger_carte*
  - *intention*
  - *deplacement*
- Génération des résultats
  - *creer_image*
  - *sauvegarder_image*

On définit des fonctions :

- Simulation
  - *charger_carte*
  - *intention*
  - *deplacement*
- Génération des résultats
  - *creer_image*
  - *sauvegarder_image*

On définit des fonctions :

- Simulation
  - *charger_carte*
  - *intention*
  - *deplacement*
- Génération des résultats
  - *creer_image*
  - *sauvegarder_image*

On définit des fonctions :

- Simulation
  - *charger_carte*
  - *intention*
  - *deplacement*
- Génération des résultats
  - *creer_image*
  - *sauvegarder_image*

On définit des fonctions :

- Simulation
  - *charger_carte*
  - *intention*
  - *deplacement*
- Génération des résultats
  - *creer_image*
  - *sauvegarder_image*

On définit des fonctions :

- Simulation
  - *charger_carte*
  - *intention*
  - *deplacement*
- Génération des résultats
  - *creer_image*
  - *sauvegarder_image*

Notant $\mathbb{A}$ l'ensemble des agents de la simulation, *intention* et *deplacement* se définissent par :

$$intention : \begin{cases} \mathbb{A} & \to \mathbb{N}^2 \\ x & \mapsto intention(x) \end{cases}$$

$$deplacement : \begin{cases} M_{n,p}(\mathbb{N}) \times \mathbb{A} & \to M_{n,p}(\mathbb{N}) \\ (M, x) & \mapsto deplacement(M, x) \end{cases}$$

Implémentation

- Choix d'implémentation
- Construction de la matrice d'adjacence
- Implémentation des algorithmes

Implémentation

- Choix d'implémentation
- Construction de la matrice d'adjacence
- Implémentation des algorithmes

Implémentation

- Choix d'implémentation
- Construction de la matrice d'adjacence
- Implémentation des algorithmes

Implémentation

- Choix d'implémentation
- Construction de la matrice d'adjacence
- Implémentation des algorithmes

Deux algorithmes connus de recherche du plus court chemin :

- Dijkstra
- A*

Ceux-ci sont basés sur l'étude d'un **graphe associé** au problème

Implémentation

Choix

Deux algorithmes connus de recherche du plus court chemin :

- Dijkstra
- A^*

Ceux-ci sont basés sur l'étude d'un **graphe associé** au problème

Deux algorithmes connus de recherche du plus court chemin :

- Dijkstra
- A*

Ceux-ci sont basés sur l'étude d'un **graphe associé** au problème

On définit le graphe associé à une carte M par :

$$\begin{aligned} G_M = &(S, A) \\ &\text{avec } S = \{u \in M : u \text{ est une position valide}\}, \\ &A = \{(u,\ v) \in S : u \text{ et } v \text{ sont voisins}\} \end{aligned}$$

On définit le graphe associé à une carte M par :

$$
\begin{aligned}
G_M =& (S, A) \\
&\text{avec } S = \{u \in M : u \text{ est une position valide}\}, \\
&A = \{(u,\ v) \in S : u \text{ et } v \text{ sont voisins}\}
\end{aligned}
$$

On définit le graphe associé à une carte M par :

$$\begin{aligned} G_M =& (S, A) \\ \text{avec } S =& \{u \in M : u \text{ est une position valide}\}, \\ A =& \{(u,\ v) \in S : u \text{ et } v \text{ sont voisins}\} \end{aligned}$$

On définit le graphe associé à une carte M par :

$$\begin{aligned} G_M =& (S, A) \\ \text{avec } S =& \{u \in M \colon u \text{ est une position valide}\}, \\ A =& \{(u,\ v) \in S \colon u \text{ et } v \text{ sont voisins}\} \end{aligned}$$

Dans le programme, le graphe G_M est représenté par une matrice d'adjacence A_G définie par :

$$S = \{0, 1, \ldots, n \times p - 1\}$$
$$\forall (i,j) \in [\![0, n-1]\!] \times [\![0, p-1]\!], A_{G_{i,j}} = \mathbb{1}_A(i,j)$$

Dans le programme, le graphe G_M est représenté par une matrice d'adjacence A_G définie par :

$$S = \{0, 1, \ldots, n \times p - 1\}$$

$$\forall (i, j) \in [\![0, n-1]\!] \times [\![0, p-1]\!], A_{G_{i,j}} = \mathbb{1}_A(i, j)$$

Dans le programme, le graphe G_M est représenté par une matrice d'adjacence A_G définie par :

$$S = \{0, 1, \ldots, n \times p - 1\}$$

$$\forall (i, j) \in [\![0, n-1]\!] \times [\![0, p-1]\!], A_{G_{i,j}} = \mathbb{1}_A(i, j)$$

Algorithme utilisé : **A***

- Approxime le plus court chemin à l'aide d'une heuristique
- Explore le graphe en estimant la pertinence de chaque nœud pour se rendre à l'objectif

Algorithme utilisé : **A***

- Approxime le plus court chemin à l'aide d'une heuristique
- Explore le graphe en estimant la pertinence de chaque nœud pour se rendre à l'objectif

Algorithme utilisé : **A***

- Approxime le plus court chemin à l'aide d'une heuristique
- Explore le graphe en estimant la pertinence de chaque nœud pour se rendre à l'objectif

Une heuristique est une application de la forme :

$$h: \begin{cases} \mathbb{R}^2 \times \mathbb{R}^2 \times M_{n,p}(\mathbb{N}) \to \mathbb{N} \\ (pos, pos_{but}, M) \mapsto h(pos, pos_{but}, M) \end{cases}$$

Exemple : la norme euclidienne

$$h: \begin{cases} \mathbb{R}^2 \times \mathbb{R}^2 \times M_{n,p}(\mathbb{N}) \to \mathbb{N} \\ ((x, y), (x', y'), M) \mapsto \sqrt{(x' - x)^2 + (y - y')^2} \end{cases}$$

Une heuristique est une application de la forme :

$$h\colon \begin{cases} \mathbb{R}^2 \times \mathbb{R}^2 \times M_{n,p}(\mathbb{N}) \to \mathbb{N} \\ (pos, pos_{but}, M) \mapsto h(pos, pos_{but}, M) \end{cases}$$

Exemple : la norme euclidienne

$$h\colon \begin{cases} \mathbb{R}^2 \times \mathbb{R}^2 \times M_{n,p}(\mathbb{N}) \to \mathbb{N} \\ ((x,y),(x',y'),M) \mapsto \sqrt{(x'-x)^2 + (y-y')^2} \end{cases}$$

Une heuristique est une application de la forme :

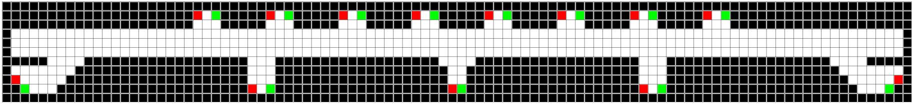
$$h\colon \begin{cases} \mathbb{R}^2 \times \mathbb{R}^2 \times M_{n,p}(\mathbb{N}) \to \mathbb{N} \\ (pos, pos_{but}, M) \mapsto h(pos, pos_{but}, M) \end{cases}$$

Exemple : la norme euclidienne

$$h\colon \begin{cases} \mathbb{R}^2 \times \mathbb{R}^2 \times M_{n,p}(\mathbb{N}) \to \mathbb{N} \\ ((x,y),(x',y'),M) \mapsto \sqrt{(x'-x)^2 + (y-y')^2} \end{cases}$$

Cadre : quai de métro à l'arrivée d'un train (partie supérieure) et disposant de plusieurs accès (partie inférieure)

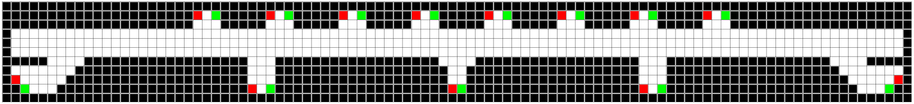
Une case correspond à 1m^2, avec une limite de 6 personnes/m^2



Mur (case infranchissable)
Quai (case franchissable)
Sortie
Entrée

Cadre : quai de métro à l'arrivée d'un train (partie supérieure) et disposant de plusieurs accès (partie inférieure)

Une case correspond à 1m^2, avec une limite de 6 personnes/m^2



Mur (case infranchissable)

Quai (case franchissable)

Sortie

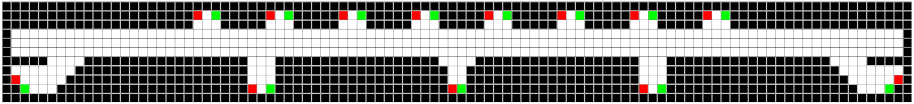
Entrée

Implémentation

Cadre des tests

Cadre : quai de métro à l'arrivée d'un train (partie supérieure) et disposant de plusieurs accès (partie inférieure)

Une case correspond à $1m^2$, avec une limite de 6 personnes/m^2



Mur (case infranchissable)

Quai (case franchissable)

Sortie

Entrée

Résultats du programme :

Pour de petites populations initiales la simulation se **termine correctement**

Pour une population initiale de **600 agents**, la simulation n'aboutit pas à une évacuation complète :

Résultats du programme :

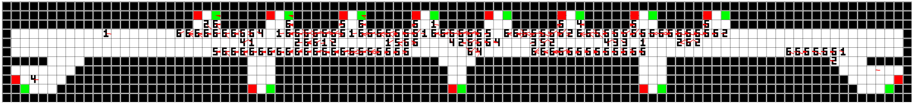
Pour de petites populations initiales la simulation se **termine correctement**

Pour une population initiale de **600 agents**, la simulation n'aboutit pas à une évacuation complète :

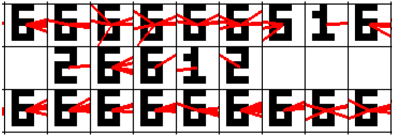
Résultats du programme :

Pour de petites populations initiales la simulation se **termine correctement**

Pour une population initiale de **600 agents**, la simulation n'aboutit pas à une évacuation complète :



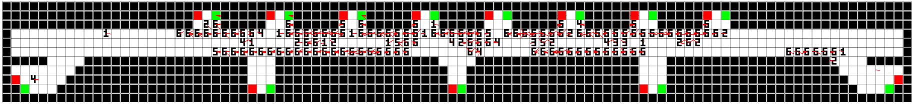
La simulation est **figée** de manière non réaliste :



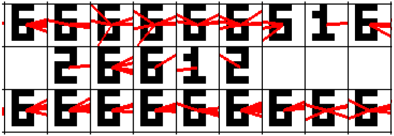
Résultats du programme :

Pour de petites populations initiales la simulation se **termine correctement**

Pour une population initiale de **600 agents**, la simulation n'aboutit pas à une évacuation complète :



La simulation est **figée** de manière non réaliste :



Cette heuristique n'est correcte que dans le cas d'un graphe constant. Or notre graphe varie en fonction des agents à proximité.

On cherche alors à modifier notre heuristique pour

- la rendre plus réaliste
- prendre en compte les agents à proximité

On cherche alors à modifier notre heuristique pour

- la rendre plus réaliste
- prendre en compte les agents à proximité

On cherche alors à modifier notre heuristique pour

- la rendre plus réaliste
- prendre en compte les agents à proximité

On ajoute pour cela un paramètre *REPULSION* à la simulation :

```
int h(location pos, location goal, map *m) {
    int f = 1;
    for (int i=0; i<REPULSION; i++) {
        f*=m->level[pos.y][pos.x];
    }
    return f
        + (goal.x - pos.x) * (goal.x - pos.x)
        + (goal.y - pos.y) * (goal.y - pos.y);
}
```

Après avoir testé différentes valeurs, on a estimé que la valeur 4 renvoyait un résultat convenable.

On ajoute pour cela un paramètre *REPULSION* à la simulation :

```
int h(location pos, location goal, map *m) {
    int f = 1;
    for (int i=0; i<REPULSION; i++) {
        f*=m->level[pos.y][pos.x];
    }
    return f
            + (goal.x - pos.x) * (goal.x - pos.x)
            + (goal.y - pos.y) * (goal.y - pos.y);
}
```

Après avoir testé différentes valeurs, on a estimé que la valeur **4** renvoyait un résultat convenable.

Résultats du programme :

Pour une population initiale de **600 agents**, la simulation ne rencontre plus de blocage.

Le nouveau blocage est rencontré pour des populations initiales **supérieures à 1000 agents.**

Le mouvement est maintenant limité par l'agencement même du lieu :

Résultats du programme :

Pour une population initiale de **600 agents**, la simulation ne rencontre plus de blocage.

Le nouveau blocage est rencontré pour des populations initiales **supérieures à 1000 agents.**

Le mouvement est maintenant limité par l'agencement même du lieu :

Résultats du programme :

Pour une population initiale de **600 agents**, la simulation ne rencontre plus de blocage.

Le nouveau blocage est rencontré pour des populations initiales **supérieures à 1000 agents.**

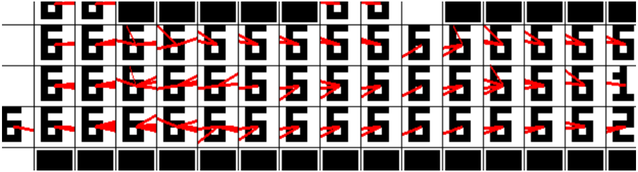
Le mouvement est maintenant limité par l'agencement même du lieu :

Résultats du programme :

Pour une population initiale de **600 agents**, la simulation ne rencontre plus de blocage.

Le nouveau blocage est rencontré pour des populations initiales **supérieures à 1000 agents.**

Le mouvement est maintenant limité par l'agencement même du lieu :



On cherche maintenant à obtenir un déplacement plus réaliste en introduisant des limites de densité :

- Densité acceptable : 3 p/m²
- Densité maximale : 6 p/m²

Si la case visée contient N agents, le déplacement est possible si :

- $N < 3$
- $2 < N < 6$ et l'agent subit des pressions extérieures de la part d'au moins deux agents

On cherche maintenant à obtenir un déplacement plus réaliste en introduisant des limites de densité :

- Densité acceptable : 3 p/m²
- Densité maximale : 6 p/m²

Si la case visée contient N agents, le déplacement est possible si :

- $N < 3$
- $2 < N < 6$ et l'agent subit des pressions extérieures de la part d'au moins deux agents

On cherche maintenant à obtenir un déplacement plus réaliste en introduisant des limites de densité :

- Densité acceptable : 3 p/m²
- Densité maximale : 6 p/m²

Si la case visée contient N agents, le déplacement est possible si :

- $N < 3$
- $2 < N < 6$ et l'agent subit des pressions extérieures de la part d'au moins deux agents

On cherche maintenant à obtenir un déplacement plus réaliste en introduisant des limites de densité :

- Densité acceptable : 3 p/m²
- Densité maximale : 6 p/m²

Si la case visée contient N agents, le déplacement est possible si :

- $N < 3$
- $2 < N < 6$ et l'agent subit des pressions extérieures de la part d'au moins deux agents

On cherche maintenant à obtenir un déplacement plus réaliste en introduisant des limites de densité :

- Densité acceptable : 3 p/m²
- Densité maximale : 6 p/m²

Si la case visée contient N agents, le déplacement est possible si :

- $N < 3$
- $2 < N < 6$ et l'agent subit des pressions extérieures de la part d'au moins deux agents

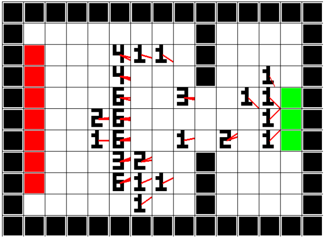
On cherche maintenant à obtenir un déplacement plus réaliste en introduisant des limites de densité :

- Densité acceptable : 3 p/m²
- Densité maximale : 6 p/m²

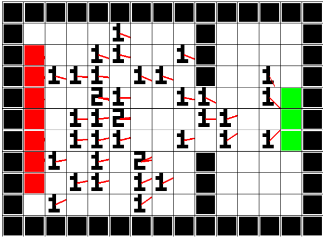
Si la case visée contient N agents, le déplacement est possible si :

- $N < 3$
- $2 < N < 6$ et l'agent subit des pressions extérieures de la part d'au moins deux agents

La nouvelle fonction permet un déplacement plus réaliste des agents qui s'éparpillent dans l'espace après être entrés dans la pièce.



(a) Ancienne fonction



(b) Nouvelle fonction

Sortie

Mur (case infranchissable)

Entrée

Air (case franchissable)

Deux applications du programme

- Evaluation du trafic maximal d'un quai de métro
- Vérification de résultats scientifiques connus

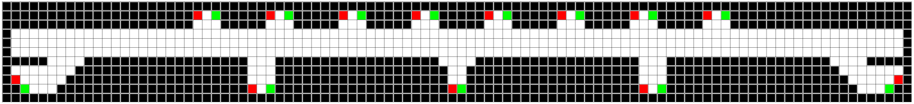
Deux applications du programme

- Evaluation du trafic maximal d'un quai de métro
- Vérification de résultats scientifiques connus

Deux applications du programme

- Evaluation du trafic maximal d'un quai de métro
- Vérification de résultats scientifiques connus

Cadre : quai de métro à l'arrivée d'un train



Longueur du quai : 100 m
Largeur du quai : 3 m

Mur (case infranchissable)

Quai (case franchissable)

Sortie

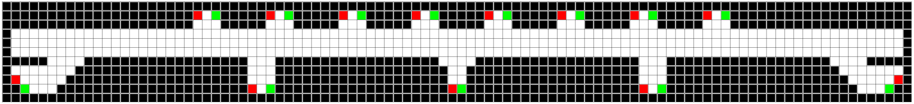
Entrée

Surface de la partie centrale : $50 \times 3 = 150\ \text{m}^2$

Ainsi, le quai est dimensionné pour recevoir

- dans un cas normal : $150 \times 3 = 450$ usagers
- dans un cas critique : $150 \times 6 = 900$ usagers

Cadre : quai de métro à l'arrivée d'un train



Longueur du quai : 100 m

Largeur du quai : 3 m

Mur (case infranchissable)

Quai (case franchissable)

Sortie

Entrée

Surface de la partie centrale : $50 \times 3 = 150$ m^2

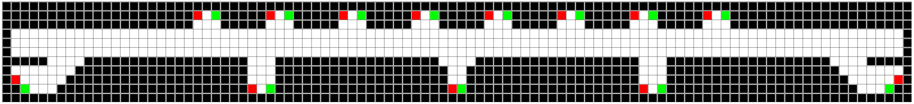
Ainsi, le quai est dimensionné pour recevoir

- dans un cas normal : $150 \times 3 = 450$ usagers
- dans un cas critique : $150 \times 6 = 900$ usagers

Evaluation du trafic maximal

Cadre des tests

Cadre : quai de métro à l'arrivée d'un train



Longueur du quai : 100 m
Largeur du quai : 3 m

■	Mur (case infranchissable)
□	Quai (case franchissable)
(vert)	Sortie
(rouge)	Entrée

Surface de la partie centrale : $50 \times 3 = 150$ m^2

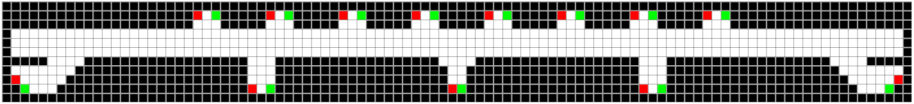
Ainsi, le quai est dimensionné pour recevoir

- dans un cas normal : $150 \times 3 = 450$ usagers
- dans un cas critique : $150 \times 6 = 900$ usagers

Evaluation du trafic maximal

Cadre des tests

Cadre : quai de métro à l'arrivée d'un train



Longueur du quai : 100 m
Largeur du quai : 3 m

Surface de la partie centrale : $50 \times 3 = 150\ \text{m}^2$

Mur (case infranchissable)

Quai (case franchissable)

Sortie

Entrée

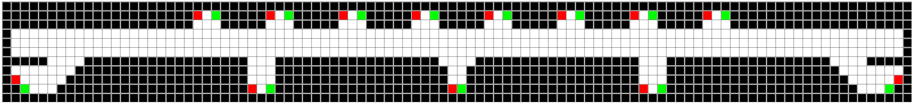
Ainsi, le quai est dimensionné pour recevoir

- dans un cas normal : $150 \times 3 = 450$ usagers
- dans un cas critique : $150 \times 6 = 900$ usagers

Evaluation du trafic maximal

Cadre des tests

Cadre : quai de métro à l'arrivée d'un train



Longueur du quai : 100 m
Largeur du quai : 3 m

Mur (case infranchissable)

Quai (case franchissable)

Sortie

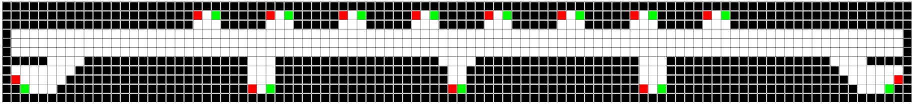
Entrée

Surface de la partie centrale : $50 \times 3 = 150$ m^2

Ainsi, le quai est dimensionné pour recevoir

- dans un cas normal : $150 \times 3 = 450$ usagers
- dans un cas critique : $150 \times 6 = 900$ usagers

Cadre : quai de métro à l'arrivée d'un train



Longueur du quai : 100 m
Largeur du quai : 3 m

Surface de la partie centrale : $50 \times 3 = 150$ m^2

Mur (case infranchissable)

Quai (case franchissable)

Sortie

Entrée

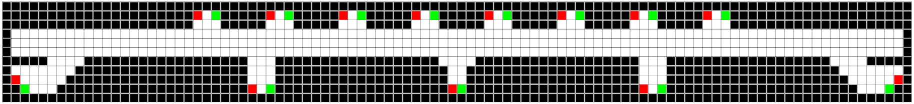
Ainsi, le quai est dimensionné pour recevoir

- dans un cas normal : $150 \times 3 = 450$ usagers
- dans un cas critique : $150 \times 6 = 900$ usagers

Evaluation du trafic maximal

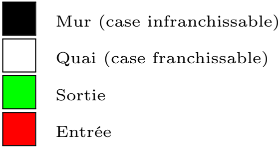
Cadre des tests

Cadre : quai de métro à l'arrivée d'un train



Longueur du quai : 100 m
Largeur du quai : 3 m

Surface de la partie centrale : $50 \times 3 = 150$ m^2



Ainsi, le quai est dimensionné pour recevoir

- dans un cas normal : $150 \times 3 = 450$ usagers
- dans un cas critique : $150 \times 6 = 900$ usagers

Protocole de test

- relève du nombre d'étapes de simulation pour une population initiale donnée
- moyenne sur 20 mesures pour chaque point
- relève du nombre de simulations qui n'ont pas terminé pour chaque population initiale

Protocole de test

- relève du nombre d'étapes de simulation pour une population initiale donnée
- moyenne sur 20 mesures pour chaque point
- relève du nombre de simulations qui n'ont pas terminé pour chaque population initiale

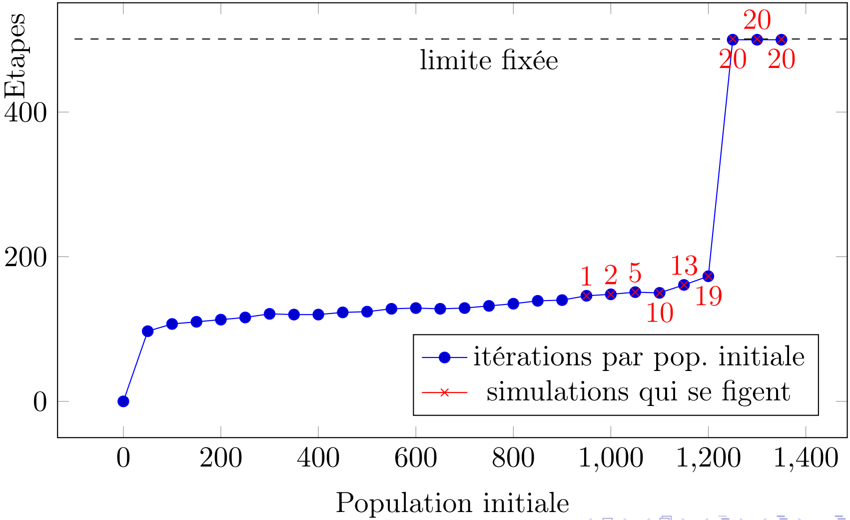
Protocole de test

- relève du nombre d'étapes de simulation pour une population initiale donnée
- moyenne sur 20 mesures pour chaque point
- relève du nombre de simulations qui n'ont pas terminé pour chaque population initiale

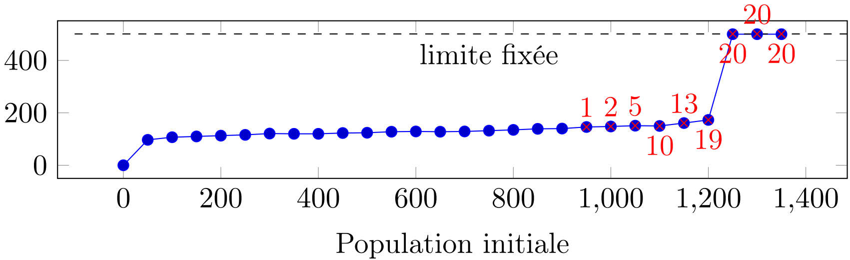
Protocole de test

- relève du nombre d'étapes de simulation pour une population initiale donnée
- moyenne sur 20 mesures pour chaque point
- relève du nombre de simulations qui n'ont pas terminé pour chaque population initiale

Durée de simulation pour une population initiale donnée



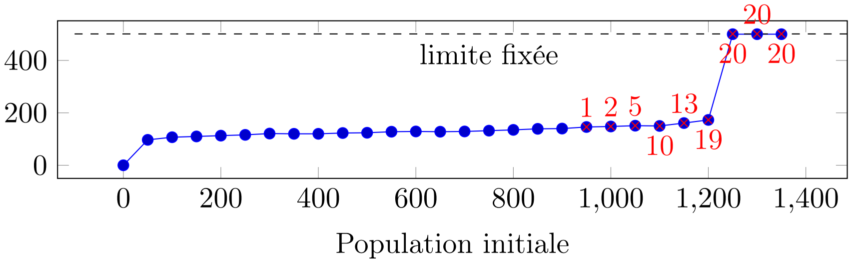
Durée de simulation pour une population initiale donnée



On retrouve bien des blocages à partir d'une population initiale d'environ 950 personnes.

Le modèle renvoie bien un résultat cohérent avec la valeur estimée.

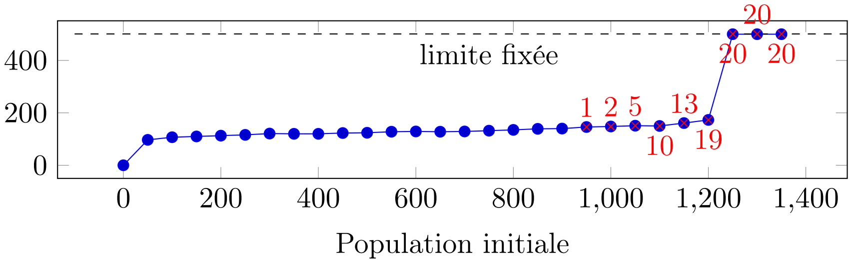
Durée de simulation pour une population initiale donnée



On retrouve bien des blocages à partir d'une population initiale d'environ 950 personnes.

Le modèle renvoie bien un résultat cohérent avec la valeur estimée.

Durée de simulation pour une population initiale donnée



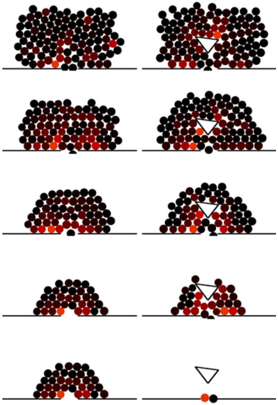
On retrouve bien des blocages à partir d'une population initiale d'environ 950 personnes.

Le modèle renvoie bien un résultat cohérent avec la valeur estimée.

Placement d'un obstacle devant une sortie

Ce phénomène s'interprète comme une division du flux incident

Illustration

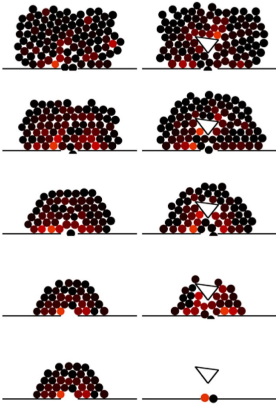


Source : Pour La Science N°501

Placement d'un obstacle devant une sortie

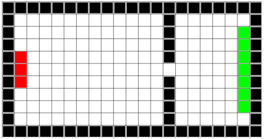
Ce phénomène s'interprète comme une division du flux incident

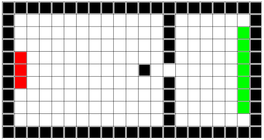
Illustration



Source : Pour La Science N°501

On souhaite mettre en valeur ce phénomène :





Protocole de test

- relève du nombre d'étapes de simulation pour une population initiale donnée
- relève du nombre de cases pour lesquelles la densité a dépassé 5 personnes/m²
- moyenne sur 50 mesures pour chaque point

Protocole de test

- relève du nombre d'étapes de simulation pour une population initiale donnée
- relève du nombre de cases pour lesquelles la densité a dépassé 5 personnes/m^2
- moyenne sur 50 mesures pour chaque point

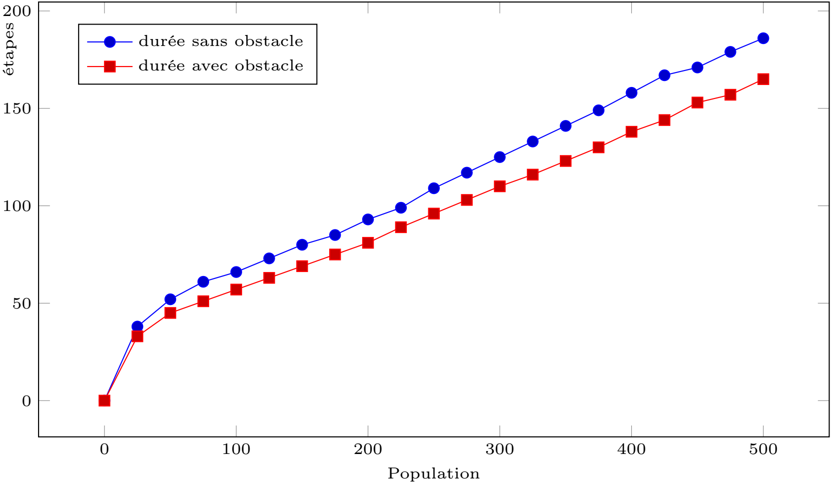
Protocole de test

- relève du nombre d'étapes de simulation pour une population initiale donnée
- relève du nombre de cases pour lesquelles la densité a dépassé 5 personnes/m^2
- moyenne sur 50 mesures pour chaque point

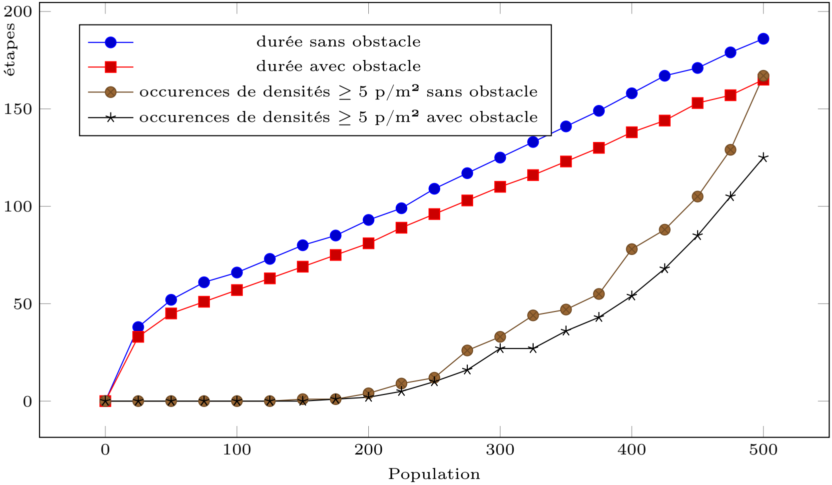
Protocole de test

- relève du nombre d'étapes de simulation pour une population initiale donnée
- relève du nombre de cases pour lesquelles la densité a dépassé 5 personnes/m^2
- moyenne sur 50 mesures pour chaque point

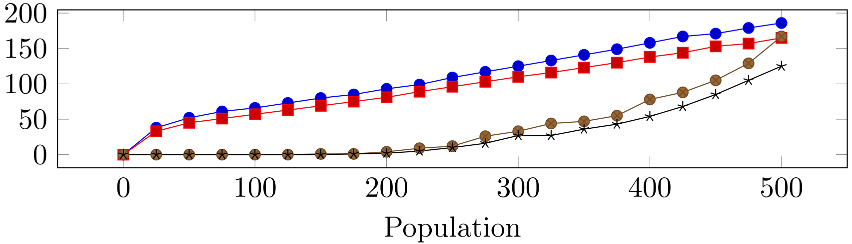
Durée de simulation pour une population initiale donnée



Durée de simulation pour une population initiale donnée



Durée de simulation pour une population initiale donnée



On constate une évacuation plus rapide avec l'obstacle ainsi qu'un nombre moins important de situations dangereuses pour les agents.

Le modèle renvoie bien un résultat en accord avec le phénomène décrit.

Conclusion

La modélisation informatique de foules a de nombreuses applications :

- Sécurité et dimensionnement des infrastructures
- Etude de comportement sociaux
- Cinéma, jeux vidéo

Merci de votre attention.

Conclusion

La modélisation informatique de foules a de nombreuses applications :

- Sécurité et dimensionnement des infrastructures
- Etude de comportement sociaux
- Cinéma, jeux vidéo

Merci de votre attention.

Conclusion

La modélisation informatique de foules a de nombreuses applications :

- Sécurité et dimensionnement des infrastructures
- Etude de comportement sociaux
- Cinéma, jeux vidéo

Merci de votre attention.

Conclusion

La modélisation informatique de foules a de nombreuses applications :

- Sécurité et dimensionnement des infrastructures
- Etude de comportement sociaux
- Cinéma, jeux vidéo

Merci de votre attention.

Annexe : programme

Louis Thevenet

Épreuve de TIPE

Session 2023

map.c I

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

#define DIAG 0


map *load_map(char *filename) {
    FILE *f = fopen(filename, "r");
    struct map *m = (struct map *) malloc(1 * sizeof(struct map));
    m->name = (char *) malloc(40 * sizeof(char));
    fscanf(f, "%s\n", m->name);
    fscanf(f, "%d", &(m->width));
    fscanf(f, "%d\n", &(m->height));
    fscanf(f, "%d", &(m->start_nb));
    fscanf(f, "%d\n", &(m->exit_nb));

    m->starts = malloc(sizeof(location) * m->start_nb);
    int start_cnt = 0;
```

map.c II

```c
m->exits = malloc(sizeof(location) * m->exit_nb);
int exit_cnt = 0;

m->level = (int **) malloc((m->height) * sizeof(int *));
m->vert_nb = 0;
for (int i = 0; i < m->height; i++) {
    m->level[i] = (int *) malloc((m->width) * sizeof(int));
    for (int j = 0; j < m->width; j++) {
        fscanf(f, "%d", &(m->level[i][j]));

        if (m->level[i][j] != Wall)
            m->vert_nb++;
        else m->surface++; // un carré = 1m²

        if (m->level[i][j] == Start) {
            if (start_cnt >= m->start_nb) {
                printf("Found too many starts, %d were specified",
                ↪  m->start_nb);
                exit(0);
            }
            m->starts[start_cnt] = (location) {i, j};
```

```c
            start_cnt++;
        }

        if (m->level[i][j] == Exit) {
            if (exit_cnt >= m->exit_nb) {
                printf("Found too many exits, %d were specified",
                ↪  m->exit_nb);
                exit(0);
            }
            m->exits[exit_cnt] = (location) {i, j};
            exit_cnt++;
        }
    }
}
// adj. matrix of the graph
m->adj = malloc(sizeof(bool *) * (m->height * m->width));
for (int i = 0; i < m->height; i++) {
    for (int j = 0; j < m->width; j++) {
        if (m->level[i][j] != Wall) {
            m->adj[m->width * i + j] = malloc(sizeof(bool) * 8);
            get_neighbours(m, i, j, m->adj[m->width * i + j]);
```

```
            } else {
                m->adj[m->width * i + j] = NULL;
            }
        }
    }
    return m;
}

void get_neighbours(map *m, int y, int x, bool *adj)
/*
 * Return neighbours of position (y,x) for adj. matrix
 * */
{
    /*
    0    1    2
    3  (x,y)  4
    5    6    7
    */

    if (DIAG == 1) {
        adj[0] = ((x != 0) && y != 0 && m->level[y - 1][x - 1] != Wall) ? 1 : 0;
```

```
        adj[1] = (y != 0 && m->level[y - 1][x] != Wall) ? 1 : 0;
        adj[2] = (y != 0 && x != m->width - 1 && m->level[y - 1][x + 1] !=
        ↪  Wall) ? 1 : 0;
        adj[3] = (x != 0 && m->level[y][x - 1] != Wall) ? 1 : 0;
        adj[4] = (x != m->width - 1 && m->level[y][x + 1] != Wall) ? 1 : 0;
        adj[5] = (x != 0 && y != m->height - 1 && m->level[y + 1][x - 1] !=
        ↪  Wall) ? 1 : 0;
        adj[6] = (y != m->height - 1 && m->level[y + 1][x] != Wall) ? 1 : 0;
        adj[7] = (x != m->width - 1 && y != m->height - 1 && m->level[y + 1][x
        ↪  + 1] != Wall) ? 1 : 0;

    } else {
        adj[0] = 0;
        adj[1] = (y != 0 && m->level[y - 1][x] != Wall) ? 1 : 0;
        adj[2] = 0;
        adj[3] = (x != 0 && m->level[y][x - 1] != Wall) ? 1 : 0;
        adj[4] = (x != m->width - 1 && m->level[y][x + 1] != Wall) ? 1 : 0;
        adj[5] = 0;
        adj[6] = (y != m->height - 1 && m->level[y + 1][x] != Wall) ? 1 : 0;
        adj[7] = 0;
    }
```

map.c VI

```c
}

void free_map(map *m) {
    for (int i = 0; i < m->height; i++) {
        free(m->level[i]);
    }
    free(m->level);

    for (int i = 0; i < m->vert_nb; i++) {

        free(m->adj[i]);

    }
    free(m->adj);
    free(m->name);
    free(m->starts);
    free(m->exits);
    free(m);

}
```

```c
#include "person.h"

person **generate_population(map *m, int n) {

    person **res = malloc(sizeof(person) * n);

    for (int i = 0; i < n; i++) {
        res[i] = malloc(sizeof(person));

        if (rand()%10<4) {
            res[i]->pos.x=-1;
            res[i]->pos.y=-1;
            while (res[i]->pos.x == -1 ||res[i]->pos.y==-1
                    || m->level[res[i]->pos.y][res[i]->pos.x] != Air){

                res[i]->pos.x=rand()%m->width;
                res[i]->pos.y=rand()%m->height;
            }
        }
        else {
            res[i]->pos = m->starts[rand() % (m->start_nb)];
```

person.c II

```c
        }

        res[i]->goal = m->exits[rand() % (m->exit_nb)];
    }
    return res;
}

void free_population(person **p, int n) {
    for (int i = 0; i < n; i++) {
        free(p[i]);
    }
    free(p);
}
```

min_heap.c I

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

Heap *CreateHeap(int capacity, int heap_type) {
    Heap *h = (Heap *) malloc(sizeof(Heap));

    if (h == NULL) {
        printf("Memory Error!");
        exit(0);
    }
    h->heap_type = heap_type;
    h->count = 0;
    h->capacity = capacity;
    h->arr = (int *) malloc(capacity * sizeof(int));
    h->pos = (int *) malloc(capacity * sizeof(int));

    if (h->arr == NULL || h->pos == NULL) {
        printf("Memory Error!");
        exit(0);
    }
```

min_heap.c II

```c
    return h;
}

void FreeHeap(Heap *h) {
    free(h->arr);
    free(h->pos);
    free(h);
}

void insert(Heap *h, int key, int *dist) {
    if (h->count < h->capacity) {
        h->arr[h->count] = key;
        h->pos[key] = h->count;
        heapify_bottom_top(h, h->count, dist);
        h->count++;
    }
}

void heapify_bottom_top(Heap *h, int index, int *dist) {
    int temp_val;
    int parent_ind = (index - 1) / 2;
```

```c
    if (dist[h->arr[parent_ind]] > dist[h->arr[index]]) {
        temp_val = h->arr[parent_ind];

        h->arr[parent_ind] = h->arr[index];
        h->arr[index] = temp_val;

        h->pos[h->arr[index]] = index;
        h->pos[h->arr[parent_ind]] = parent_ind;

        heapify_bottom_top(h, parent_ind, dist);
    }
}

void heapify_top_bottom(Heap *h, int parent_node, int *dist) {
    int left = parent_node * 2 + 1;
    int right = parent_node * 2 + 2;
    int min_pos;
    int min;
    int temp;
```

min_heap.c IV

```c
if (left >= h->count || left < 0)
    left = -1;
if (right >= h->count || right < 0)
    right = -1;

if (left != -1 && dist[h->arr[left]] < dist[h->arr[parent_node]]) {
    min_pos = h->pos[h->arr[left]];
    min = left;
} else
    min = parent_node;
if (right != -1 && dist[h->arr[right]] < dist[h->arr[min]]) {
    min_pos = h->pos[h->arr[right]];
    min = right;
}
if (min != parent_node) {
    temp = h->arr[min];
    h->arr[min] = h->arr[parent_node];
    h->arr[parent_node] = temp;

    h->pos[h->arr[min]] = h->pos[h->arr[parent_node]];
    h->pos[h->arr[parent_node]] = min_pos;
```

min_heap.c V

```
        heapify_top_bottom(h, min, dist);
    }
}

int PopMin(Heap *h, int *dist) {
    int pop;
    if (h->count == 0) {
        printf("\n__Heap is Empty__ (Pop)\n");
        return -1;
    }

    pop = h->arr[0];
    h->arr[0] = h->arr[h->count - 1];
    h->pos[0] = h->pos[h->count - 1];
    h->count--;
    heapify_top_bottom(h, 0, dist);
    return pop;
}

int is_in_heap(Heap *h, int u) {
```

min_heap.c VI

```
    if (h->count == 0) {
        return 0;
    }
    for (int i = 0; i < h->capacity; i++) {
        if (h->arr[i] == u)
            return 1;
    }
    return 0;
}

void update_key(Heap *h, int i, int *d) {
    int temp;
    while (i != 0 && d[h->arr[i]] > d[h->arr[(i - 1) / 2]]) {
        temp = h->arr[(i - 1) / 2];
        h->arr[(i - 1) / 2] = h->arr[i];
        h->arr[i] = temp;

        i = (i - 1) / 2;
    }
}
```

move.c I

```c
#include "min_heap.h"
#include "person.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define INT_MAX 100000
#define MAX_PERSON 6
#define MAX_PERSON_REGULAR 3
#define HEUR heur_eucl_pop
#define FORCE_REPULSION 4

int get_y(int i) {
    if (i < 3)
        return -1;
    if (i > 4)
        return 1;
    return 0;
}

int get_x(int i) {
```

```
    if (i == 0 || i == 3 || i == 5)
        return -1;
    if (i == 2 || i == 4 || i == 7)
        return 1;
    return 0;
}

int line_to_grid_x(int i, int width) { return i % width; }

int line_to_grid_y(int i, int width) { return (i - i % width) / width; }

int heur_eucl(int y, int x, location goal, map *m) {
    return sqrt((goal.x - x) * (goal.x - x) + (goal.y - y) * (goal.y - y));
}

int heur_eucl_pop(int y, int x, location goal, map *m) { // prend en compte les
↪   gens
    int f = 1;
    for (int i=0; i<FORCE_REPULSION; i++) {
        f*=m->level[y][x];
    }
```

move.c III

```c
    return ((goal.x - x) * (goal.x - x) + (goal.y - y) * (goal.y - y)+f);
}

int a_star(map *m, location start, location goal) {
    int *pred = malloc(sizeof(int) * (m->height * m->width));
    int *g = malloc(sizeof(int) * m->height * m->width);
    int *f = malloc(sizeof(int) * m->height * m->width);

    int c, u, x_u, y_u, v, x_v, y_v, g_score;

    Heap *Q = CreateHeap(m->height * m->width, 0);

    for (int i = 0; i < m->height * m->width; i++) {
        g[i] = INT_MAX;
        f[i] = INT_MAX;
        pred[i] = -1;
        if (start.y * m->width + start.x == i) {
            g[start.y * m->width + start.x] = 0;
            f[start.y * m->width + start.x] = 0 + HEUR(start.y, start.x, goal,
            ↪  m);
        }
```

move.c IV

```
    if (m->level[line_to_grid_y(i, m->width)][line_to_grid_x(i, m->width)]
    ↪  != Wall) {
        insert(Q, i, f);
    }
}

while (Q->count != 0) {
    u = PopMin(Q, f);
    if (u == goal.y * m->width + goal.x) {
        while (pred[u] != start.y * m->width + start.x && u != -1) {
            u = pred[u];
        }
        free(g);
        free(f);
        free(pred);
        FreeHeap(Q);
        return u;
    }

    x_u = line_to_grid_x(u, m->width);
```

move.c V

```
y_u = line_to_grid_y(u, m->width);
if (m->adj[u] != NULL) { // pas un mur (au cas où)
    bool *adj = m->adj[u];
    for (int i = 0; i < 8; i++) {
        if (adj[i] == true) {
            y_v = y_u + get_y(i);
            x_v = x_u + get_x(i);
            v = y_v * m->width + x_v;
            c = 1;
            g_score = g[u] + c;


            if (g_score < g[v]) {

                g[v] = g_score;
                f[v] = g_score + HEUR(y_v, x_v, goal, m);
                pred[v] = u;
                heapify_bottom_top(Q, Q->pos[y_v * m->width + x_v], f);
            }
        }
    }
```

```c
        }
    }
    free(g);
    free(f);
    free(pred);
    FreeHeap(Q);
    return -1;
}

void move_basic(map *m, person *p, int next) {
    if (next == -1) return;
    int y = line_to_grid_y(next, m->width);
    int x = line_to_grid_x(next, m->width);

    if (m->level[y][x] >= Person + MAX_PERSON - 1) return; //on peut pas bouger
    ↪  ici car trop de gens

    if (m->level[p->pos.y][p->pos.x] != Start && m->level[p->pos.y][p->pos.x]
    ↪  != Exit) {
        m->level[p->pos.y][p->pos.x] = (m->level[p->pos.y][p->pos.x] > Person)
        ↪  ? m->level[p->pos.y][p->pos.x] - 1 : Air;
```

```c
    }
    p->pos = (location) {y, x};
    if (p->goal.y == y && p->goal.x == x) return;

    if (m->level[y][x] != Start && m->level[y][x] != Exit) {
        m->level[y][x] = (m->level[y][x] >= Person) ? m->level[y][x] + 1 :
        ↪  Person;
    }
}

void move_stress(map *m, person **p, int pop, int *pred, int p0) {

    if (pred[p0] == -1) return;
    int y = line_to_grid_y(pred[p0], m->width);
    int x = line_to_grid_x(pred[p0], m->width);

    if (m->level[y][x] >= Person + MAX_PERSON_REGULAR - 1) { // trop de gens
    ↪  pour vouloir y aller en temps normal
        int cnt=0;
        for (int i=p0; i<pop; i++) {
            if (pred[i]!=-1 && line_to_grid_y(pred[i], m->width)==p[i]->pos.y
```

```
                    && line_to_grid_x(pred[i], m->width)==p[i]->pos.x)
                    ↪  cnt++;
    }
    if (cnt<2) { //pas assez de gens poussent pour y aller
        return;
    }
}
if (m->level[y][x] >= Person + MAX_PERSON - 1) return; //on peut pas bouger
↪  ici car trop de gens

if (m->level[p[p0]->pos.y][p[p0]->pos.x] != Start &&
↪  m->level[p[p0]->pos.y][p[p0]->pos.x] != Exit) {

    m->level[p[p0]->pos.y][p[p0]->pos.x] =
    ↪  (m->level[p[p0]->pos.y][p[p0]->pos.x] > Person) ?
                                                m->level[p[p0]->pos.y][␣
                                                ↪  p[p0]->pos.x] - 1 :
                                                ↪  Air;
}
p[p0]->pos = (location) {y, x};
if (p[p0]->goal.y == y && p[p0]->goal.x == x) return;
```

move.c IX

```
    if (m->level[y][x] != Start && m->level[y][x] != Exit) {
        m->level[y][x] = (m->level[y][x] >= Person) ? m->level[y][x] + 1 :
        ↪  Person;
    }
    pred[p0]=-1;
}
```

image.c I

```c
#include "image.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

#define SCALE 40

const int digits[10][5][5] = {
        // chiffre 0
        {
                {0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0}
        },
        // chiffre 1
        {
                {0, 0, 1, 0, 0},
                {0, 1, 1, 0, 0},
```

image.c II

```
        {0, 0, 1, 0, 0},
        {0, 0, 1, 0, 0},
        {0, 1, 1, 1, 0}
    },
    // chiffre 2
    {
        {0, 1, 1, 1, 0},
        {0, 0, 0, 1, 0},
        {0, 1, 1, 1, 0},
        {0, 1, 0, 0, 0},
        {0, 1, 1, 1, 0}
    },
    // chiffre 3
    {
        {0, 1, 1, 1, 0},
        {0, 0, 0, 1, 0},
        {0, 1, 1, 1, 0},
        {0, 0, 0, 1, 0},
        {0, 1, 1, 1, 0}
    },
    // chiffre 4
```

```
{
        {0, 1, 0, 1, 0},
        {0, 1, 0, 1, 0},
        {0, 1, 1, 1, 0},
        {0, 0, 0, 1, 0},
        {0, 0, 0, 1, 0}
},
// chiffre 5
{
        {0,1, 1, 1,  0},
        {0,1, 0, 0,  0},
        {0,1, 1, 1,  0},
        {0,0, 0, 1,  0},
        {0,1, 1, 1,  0}
},
// chiffre 6
{
        {0, 1, 1, 1, 0},
        {0, 1, 0, 0, 0},
        {0, 1, 1, 1, 0},
        {0, 1, 0, 1, 0},
```

image.c IV

```
        {0, 1, 1, 1, 0}
},
// chiffre 7
{
        {0, 1, 1, 1, 0},
        {0, 0, 0, 1, 0},
        {0, 0, 0, 1, 0},
        {0, 0, 0, 1, 0},
        {0, 0, 0, 1, 0}
},
// chiffre 8
{
        {0, 1, 1, 1, 0},
        {1, 0, 0, 0, 1},
        {0, 1, 1, 1, 0},
        {1, 0, 0, 0, 1},
        {0, 1, 1, 1, 0}
},
// chiffre 9
{
        {0, 1, 1, 1, 0},
```

image.c V

```
                {1, 0, 0, 0, 1},
                {0, 1, 1, 1, 1},
                {0, 0, 0, 0, 1},
                {0, 1, 1, 1, 1}
        }
};


void draw_digit(int **tab, int digit, int y0, int x0, int a) {

    int digi_size = 5;
    for (int i=1; i<a; i++) {
        if ((a-i) % digi_size == 0) {
            a=a-i;
            break;
        }
    }

    if (digit > 9) digit=9;
    for (int offset_y=0; offset_y < digi_size; offset_y++) {
        for (int offset_x = 0; offset_x < digi_size; offset_x++) {
```

image.c VI

```
            for (int i = offset_y*(a / digi_size) ; i < (offset_y + 1) * (a /
            ↪  digi_size); i++) {
                for (int j = offset_x*(a / digi_size); j < (offset_x + 1) * (a
                ↪  / digi_size); j++) {

                    tab[y0 + i][x0 + j] = digits[digit][offset_y][offset_x];
                }

            }
        }
    }
}

void draw_arrows(int **img, person **p, int pop) {
    for (int i=0; i<pop; i++) {
        double alpha = ((double) (p[i]->goal.y - p[i]->pos.y)) / ((double)
        ↪  (p[i]->goal.x - p[i]->pos.x));
        double beta = 0;

        int y0=SCALE * p[i]->pos.y + SCALE/2;
```

image.c VII

```c
        int x0=SCALE * p[i]->pos.x + SCALE/2;

        int deb = (p[i]->goal.y - p[i]->pos.y>0 && alpha < 0)
             || ((p[i]->goal.y - p[i]->pos.y)<0 && alpha >0) ? -SCALE/2+1 :
             ↪  0;

        int fin =(p[i]->goal.y - p[i]->pos.y>0 && alpha < 0)
             || ((p[i]->goal.y - p[i]->pos.y)<0 && alpha >0)? 0 : SCALE/2;

        for (int x = deb; x<fin; x++) {
            if (alpha * x + beta > -SCALE/2 && alpha * x + beta < SCALE/2) {
                img[(int)(-1 + y0+ alpha * x + beta)][x+x0] = 2;
                img[(int)(y0+ alpha * x + beta)][x+x0] = 2;
                img[(int)(1 + y0+ alpha * x + beta)][x+x0] = 2;
            }
        }
    }
}

int **create_image(map *m, person **p, int pop) {
```

image.c VIII

```c
int height = m->height * SCALE;
int width = m->width * SCALE;

int **img = malloc(sizeof(int *) * height);
for (int i = 0; i < height; i++) {
    img[i] = malloc(sizeof(int) * width);
    for (int j = 0; j < width; j++) {
        img[i][j]=0;
    }

}
// { Air,   Wall,  Start,  Exit , Person};
double factors[] = {.95, .95, 0.97, 0.97, .6};

for (int y = 0; y < m->height; y++) { // agrandir l'image à SCALE*taille
    for (int x = 0; x < m->width; x++) {
        if (m->level[y][x] < Person) {
            for (int i = ceil((1 - factors[m->level[y][x]]) * SCALE);
                 i < ceil(SCALE * factors[m->level[y][x]]); i++) {
                for (int j = ceil((1 - factors[m->level[y][x]]) * SCALE);
                     j < ceil(SCALE * factors[m->level[y][x]]); j++) {
```

image.c IX

```
                    img[SCALE * y + i][SCALE * x + j] = m->level[y][x];
                }
            }
        }

        else {
            draw_digit(img, m->level[y][x]-3, SCALE * y, SCALE * x, SCALE);
            draw_arrows(img, p, pop);
        }
    }
    for (int i = 0; i < height; i++) { // grille
        for (int j = 0; j < width; j++) {
            if (i % SCALE == 0 || j % SCALE == 0) img[i][j] = 1;
        }
    }
}

return img;
}
```

image.c X

```c
void free_image(int **img, int height) {
    for (int i = 0; i < height; i++) {
        free(img[i]);
    }
    free(img);
}

void save_image(int n, int **img, int width, int height) {
    width *= SCALE;
    height *= SCALE;
    char filename[20];
    FILE *file;
    char colors[4][13] = {
            "255 255 255 \0", //air
            "000 000 000 \0", // mur
            "000 255 000 \0", // sortie
            "255 000 000 \0", //entrée

    };
    mkdir("./ppms", 0777);
    sprintf(filename, "ppms/step-%03d.ppm", n);
```

image.c XI

```
    file = fopen(filename, "w");
    fprintf(file, "P3\n%d %d\n255\n", width, height);
    for (int j = 0; j < height; j++) {
        for (int i = 0; i < width; i++) {
            fprintf(file, colors[img[j][i]]);
        }
    }
    fclose(file);
    free_image(img, height);
}
```

main.c I

```c
#include "image.h"

void simulation(char *map_name, int initial_pop) {
  map *m = load_map(map_name);
  person **p = generate_population(m, initial_pop);
  int *pred = malloc(sizeof(int) * initial_pop);
  int pop = initial_pop;
  int cnt = initial_pop;
  int j = 0;
  double aver_dens = 0;

  while (cnt != 0) {
    cnt = pop;

    for (int i = 0; i < pop; i++) {
      pred[i] = a_star(m, p[i]->pos, p[i]->goal);
      move_stress(m, p, pop, pred, i);

      if (p[i]->pos.y == p[i]->goal.y && p[i]->pos.x == p[i]->goal.x) {
        cnt--;
      }
```

main.c II

```
    }
    save_image(j, create_image(m, p, pop), m->width, m->height);

    printf("#%d, %dp, %fp/m^2\n", j, cnt, (float)cnt / (float)m->surface);
      aver_dens += (float)cnt / (float)m->surface;
    j++;
    if (j > 300)
      return;
  }

  printf("average density : %f\n", aver_dens / (double)j);
  free_population(p, initial_pop);
  free(pred);
  free_map(m);
}

int main(int argc, char **argv) {
  if (argc <= 2 || atoi(argv[2])==0) {
    printf("Missing parameters");
    return 1;
  }
```

main.c III

```
    simulation(argv[1], atoi(argv[2]));
  return 0;
}
```