Évacuation d'un immeuble

Modéliser l'évacuation d'un immeuble en cas de sinistre

LA VILLE

2022-2023

Introduction au problème

Facteurs liés au bâtiment :

- Type de bâtiment
- Architecture
- Type d'activité au sein du bâtiment
- Éléments liés à la sécurité

Introduction au problème

Facteurs liés aux sens :

- Indices visuels
- Indices auditifs
- Indices olfactifs
- Chaleur

Facteurs liés aux personnes :

- Profil
- Expérience
- Contexte
- Personnalité
- Rôle au sein du bâtiment

Problématique

En prenant en compte ces facteurs, comment créer une modélisation fidèle à la réalité afin de déterminer les plans d'évacuation les plus efficaces ?

Objectifs

1. Proposer un modèle mathématique en accord avec des situations réelles d'évacuation.
2. Mettre en place ce modèle par une simulation, à l'aide du langage de programmation OCaml.
3. Modifier l'environnement pour déterminer si le paradoxe de Braess s'applique.
4. Implémenter l'algorithme A* et l'appliquer à la simulation.
5. Conclure sur les plans d'évacuations les plus efficaces.

Sommaire

Éléments de la modélisation

Plans

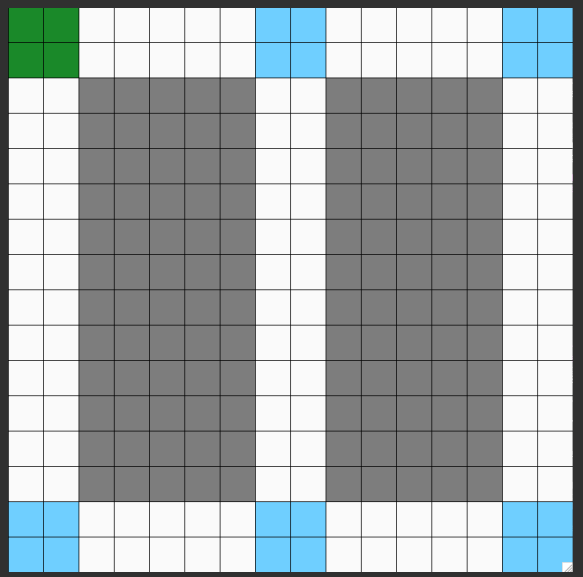
- $\mathcal{P} \subset \mathbb{R}^2 : \{(x, y) \in \mathbb{R}^2 : 0 \leqslant x \leqslant 800$ et $0 \leqslant$ y $\leqslant 800\}$
- $\mathcal{P}' = [\![0\ ;\ 15]\!]^2$

Individu

Un individu $k = ((x, y), (i, j))$ représenté par $B((x, y), csteRayon)$

Condition de collision

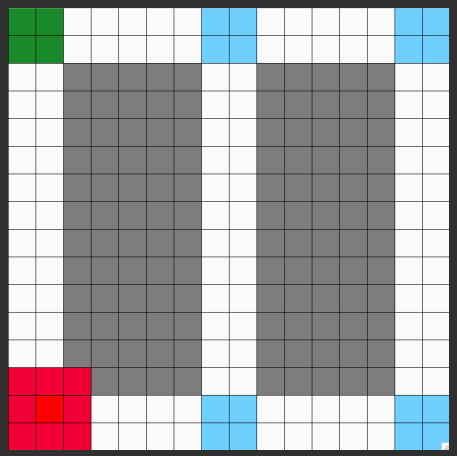
Collision entre (x_1, y_1) et (x_2, y_2) : $||(x_1, y_1) - (x_2, y_2)|| < 2 \times csteRayon$



Carte

Éléments liés au déplacement

- Périmètre de sécurité : $\forall (i,j) \in \mathcal{P}'$,
  $P_s(i,j) = \{(k,l) \in [\![0\ ;\ 15]\!] : |i-k| \leqslant 1 \text{ et } |j-l| \leqslant 1\}$
- Densité d'une cellule : $D((i,j)) = Card(\{k = ((x,y),(i,j))\})$
- Vitesse : $v_{i,j}(t) = v_{course} * (1 - \frac{D(\{P_s(i,j)\}-1}{n_{max}})$
- Ensemble des positions licites : $\Gamma(t) = B((x,y), v(t))$

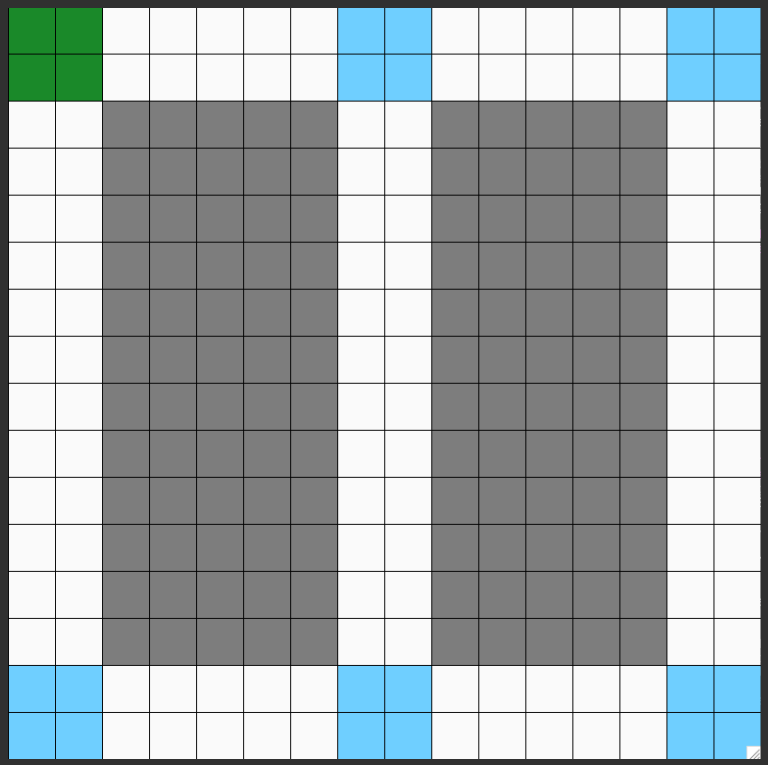


Périmètre de sécurité de la cellule (1, 1) en rouge

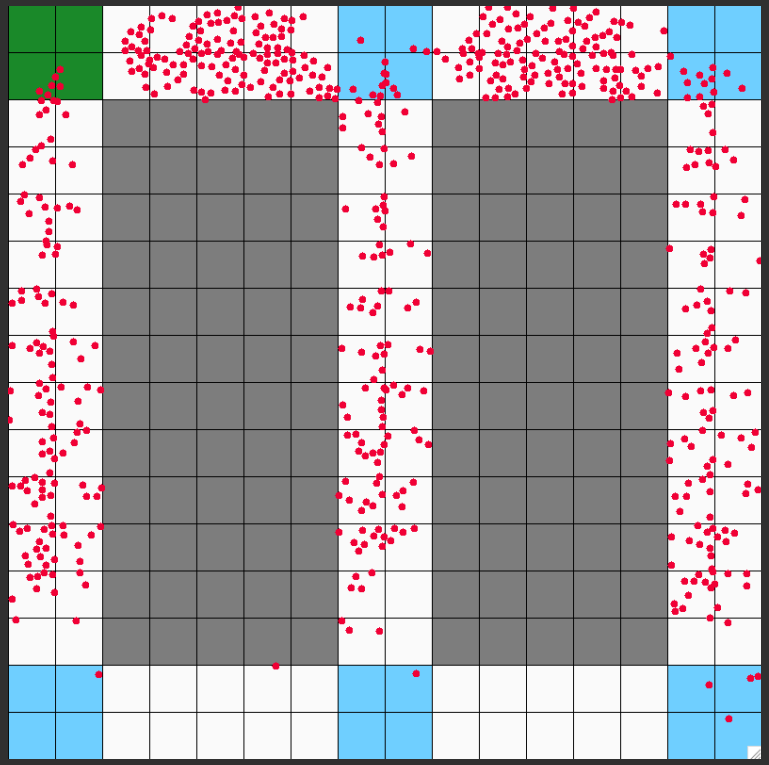
Échelle et proportions

Élément	Réalité	Simulation
/	1m	20 pixels
Plan	40x40m	800x800p
v_{course}	$10 \leqslant v_{course} \leqslant 13$ (km/h)	$55.5 \leqslant v_{course} \leqslant 72.2$ (p/s)
$csteRayon$	environ 19cm	4 pixels

Aspects graphiques



Début de la simulation



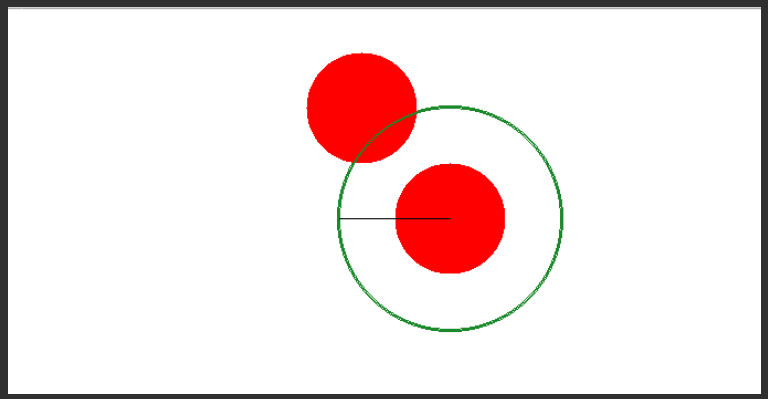
Milieu de la simulation

Répartition du code

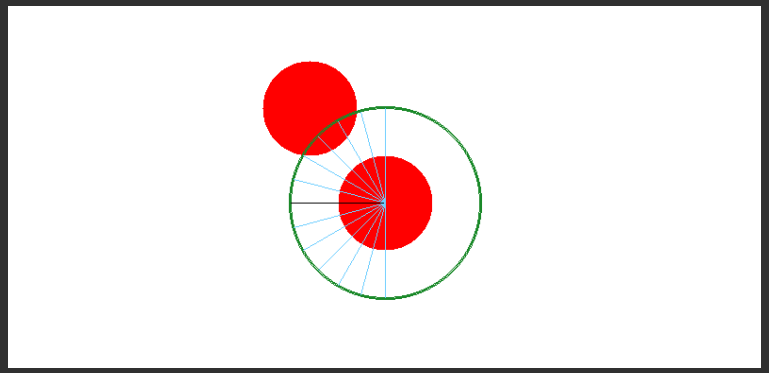
- Mise en place de la file de priorité (tas).
- Mise en place de l'algorithme de Dijkstra et de A*.
- Initialisation des variables, des constantes et des types.
- Fonctions de calculs (de norme ou de rotation d'un vecteur...).
- Fonctions de vérifications (bords atteints...).
- Fonctions de générations (de coordonnées, du chemin...).
- Fonctions de collisions.
- Fonctions de déplacements.
- Fonctions d'affichages.
- Fonction principale.

Déplacement des personnes

Cas théorique : $\Gamma(t) = B((x, y), v(t))$
Cas réel : $k = ((x, y), (i, j))$ se déplace vers la sortie (x_s, y_s)



Vecteur vitesse optimal

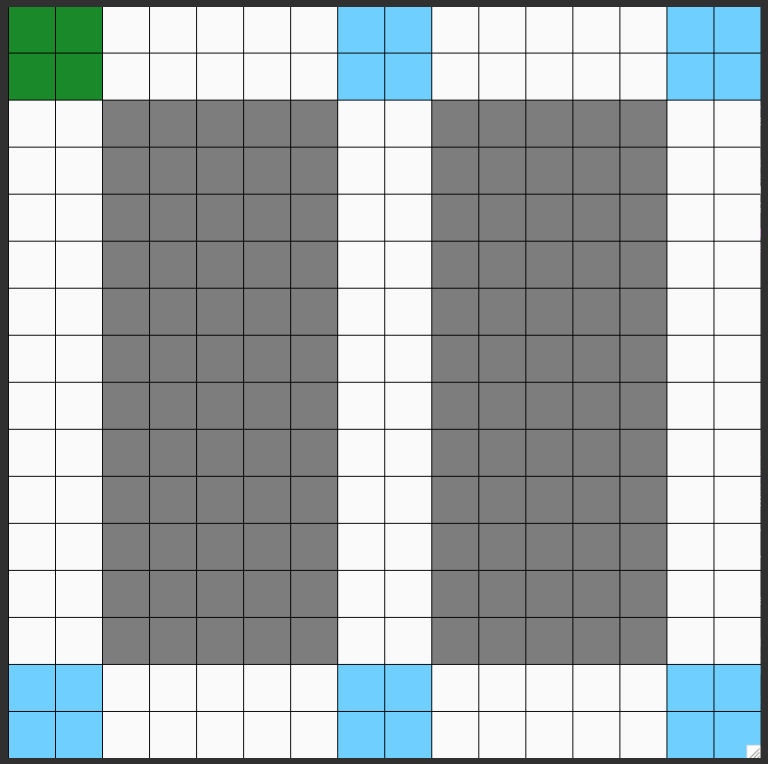


Certains vecteurs de $\Gamma(t)$

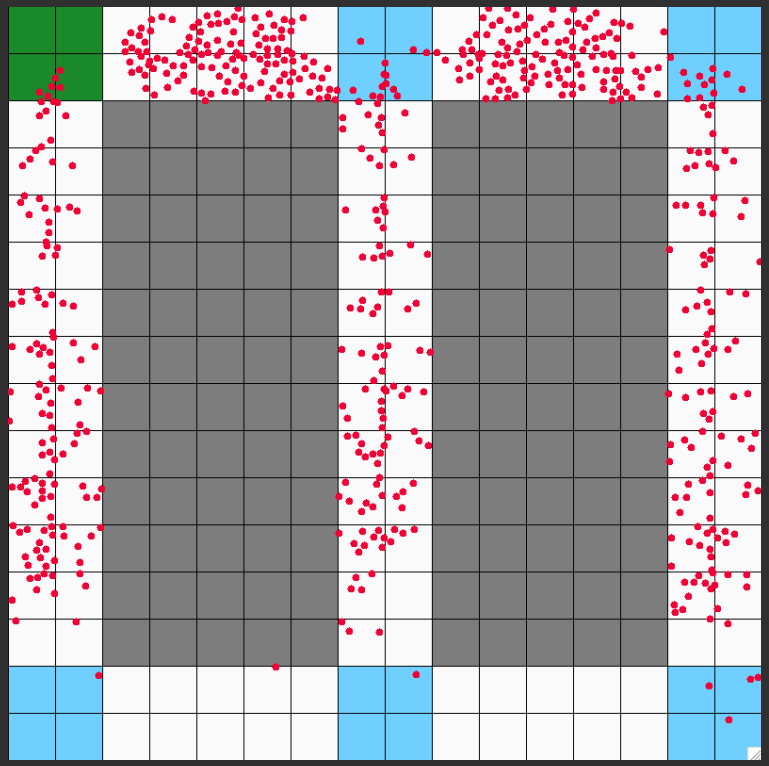
$$\vec{v}_{opt}(t) = \frac{\begin{pmatrix} x_s - x \\ y_s - y \end{pmatrix}}{||(x_s - x, y_s - y)||} * v(t)$$

$$n_{rot} = \#\{0; \theta; -\theta; 2\theta; -2\theta; ...; \frac{(n_{rot}-1)}{2}\theta; -\frac{(n_{rot}-1)}{2}\theta\}$$

Deux configurations



Configuration A :
distance euclidienne
(chemin le plus court)



Configuration B :
densité des couloirs
(chemin le moins dense).

Complexité

Type de fonction	Complexité
File de priorité (tas)	$\theta(\log \lvert S \rvert)$ ou $\theta(1)$
Dijkstra et A*	$\theta((\lvert S \rvert + \lvert A \rvert) \log \lvert S \rvert)$
Affichage	$\theta(N)$
Calculs	$\theta(1)$
Vérifications	$\theta(1)$
Collisions	$\theta(n_{max})$
Générations	$\theta(N * n_{max})$
Opération de rotation de $\vec{v}$	$\theta(n_{rot} * n_{max})$
Déplacement général	$\theta(N * n_{rot} * n_{max})$

N : nombre de personnes.
n_{rot} : nombre de rotations lors d'un déplacement d'une personne.
n_{max} : nombre maximal d'individus qui peuvent co-exister au sein d'un périmètre de sécurité.

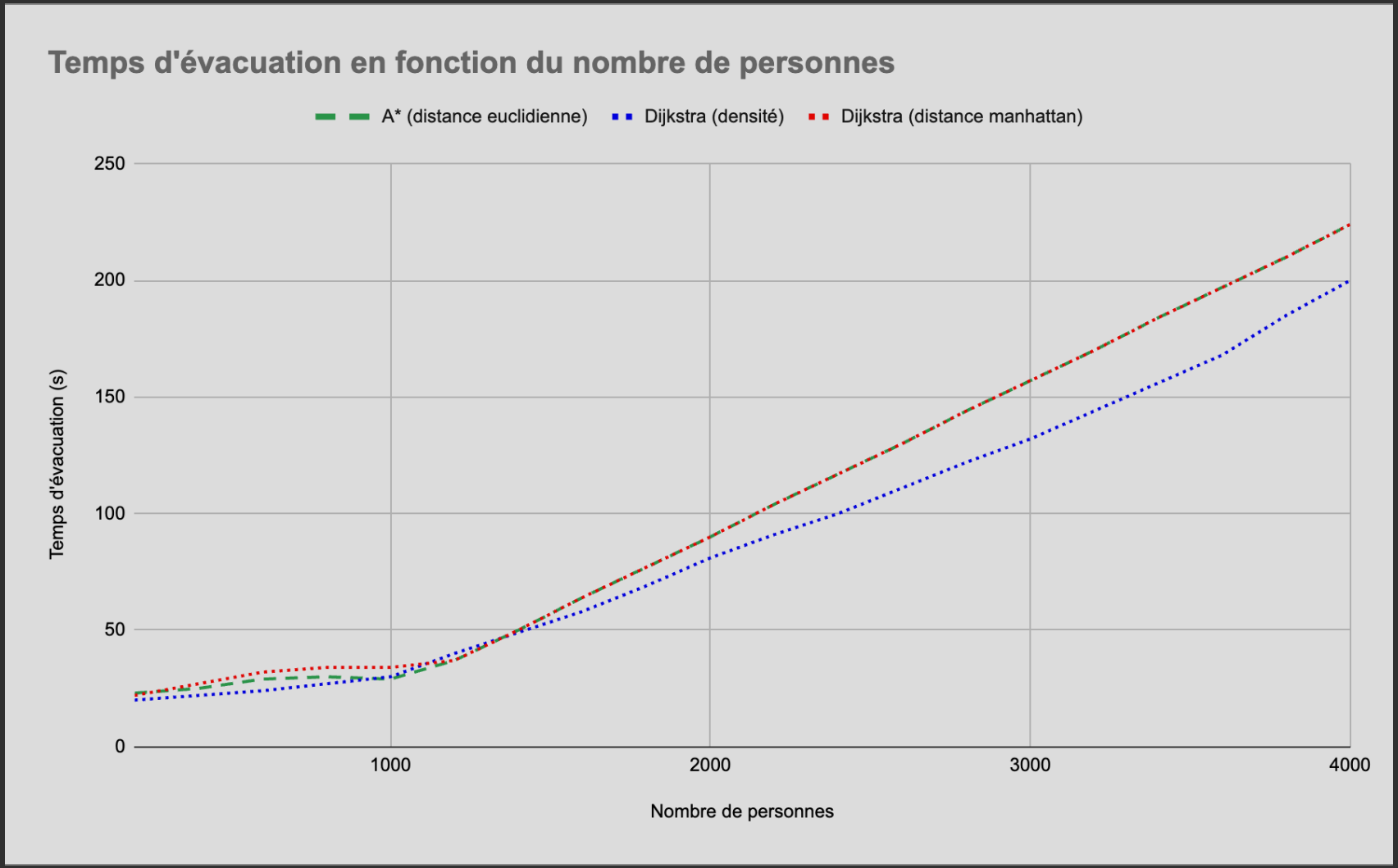
Simuler le modèle

- Augmenter le nombre de sorties.
- Augmenter le nombre de couloirs.
- Prendre en compte les deux types de configuration (chemin le plus court, chemin le moins dense).

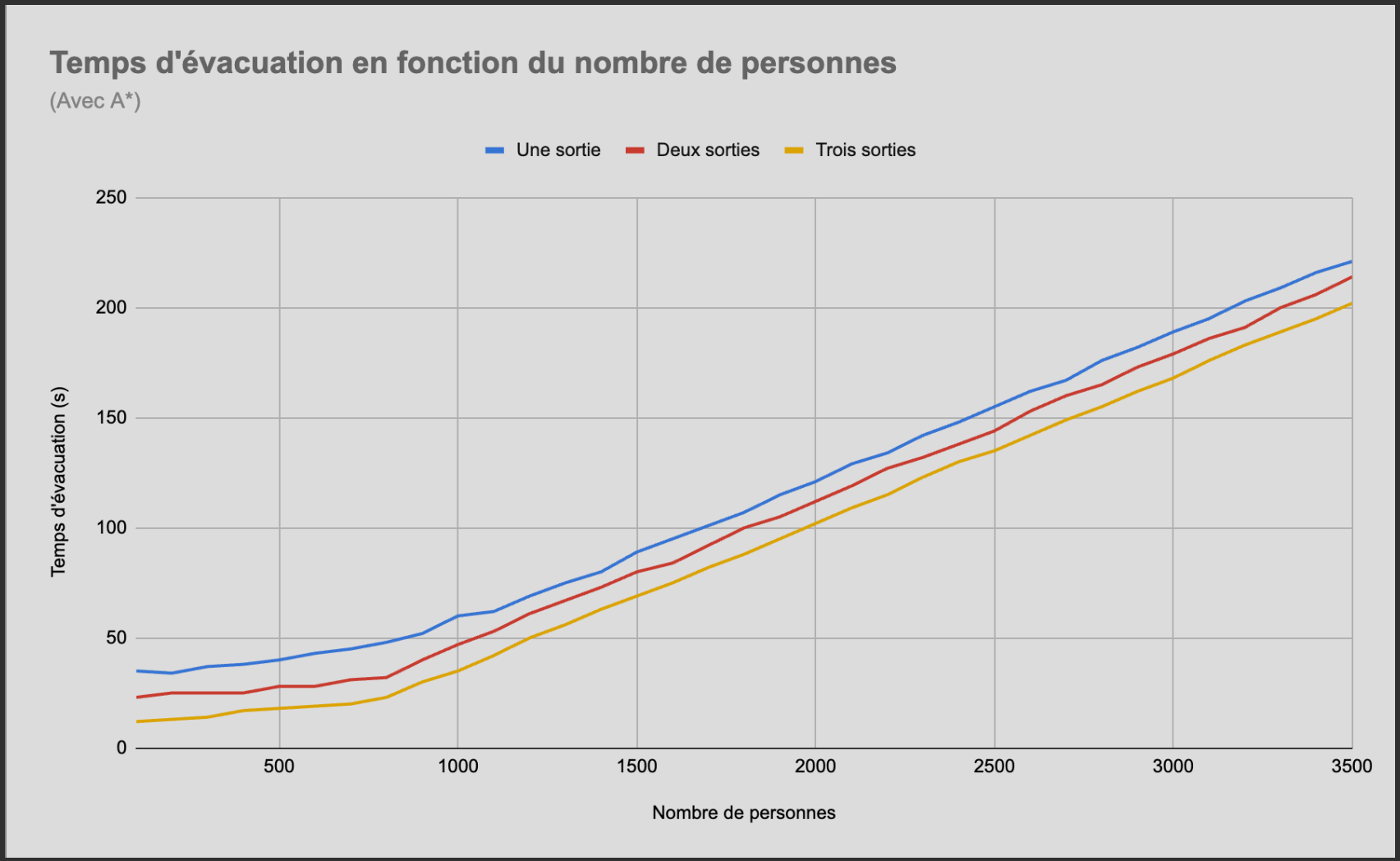
Paradoxe de Braess

L'ajout d'une nouvelle route dans un réseau routier peut réduire la performance globale.

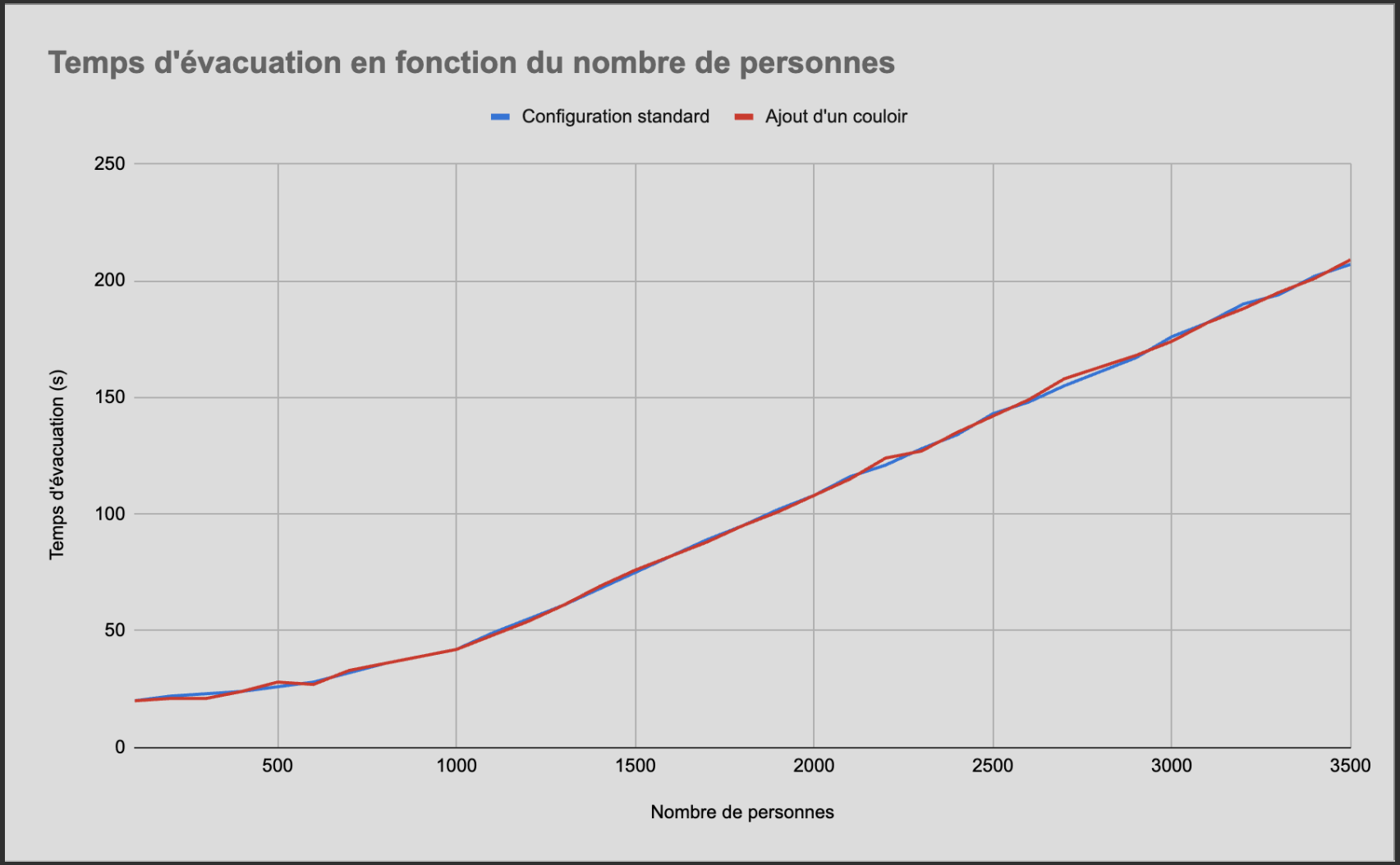
Comparaison entre les différentes configurations



Augmentation du nombre de sorties



Ajout d'un couloir



Conclusions sur les meilleurs plans d'évacuation.

- Mettre en place une meilleure distribution des évacuants au sein des couloirs.
- Augmenter le nombre de sorties.
- L'ajout d'un couloir n'a aucun impact sur le temps d'évacuation.

Initialisation des variables

```
#load "graphics.cma";;
open Graphics;;
open_graph ":0";;
resize_window 800 800;;

(* Couleurs *)
let noir = rgb 0 0 0;;
let blanc = rgb 250 250 250;;
let bleu = rgb 135 206 250;;
let rouge = rgb 220 20 60;;
let vert = rgb 69 139 55;;
let gris = rgb 128 128 128;;

(* Exceptions *)
exception Stop;;
exception DirectionTrouvee of float;;
exception ChangementCellule;;
exception Priorite of float;;
exception Indice of int;;

(* Types *)
type vecteur = {vx:float; vy:float};;
type personne = {mutable x:float; mutable y:float; mutable v:float;
                 mutable chemin:(int*int) list};;
type tas = {mutable tab : ((int * int) * float) array; mutable taille : int};;


```

```
1  (* Variables Globales *)
2  let nombreEvacues = ref 0;;
3  let dt = ref 0;;
4  let csteRayon = 4.;;
5  let sorties = [(0, 0); (1, 1); (2, 0)];;
6  let n = 800/50;; (* nombre de cellules par ligne ou colonne *)
7  let m = 50;; (* taille de la cellule *)
8  let epsilon = 1e-10;;
9  let tailleMax = 650;;
10 let nMAX = m*m/4;;
11 let flux = 15;;
12 let nROT = 13;;
13 let theta = 15.;;
14 let listeAdj =
15 [|
16 [| [(0,1); (1,0)] ; [(0, 0); (1,1)] |];
17 [| [(0,0); (1,1); (2, 0)] ; [(0,1); (1, 0); (2, 1)] |];
18 [| [(1,0); (2,1)] ; [(1, 1); (2,0)] |]
19 |];;
20
```

Mise en place de la file de priorité (minimale)

```
let tasInitialise () =
    let tab = Array.make 10 ((-1, -1), max_float) in {tab = tab; taille = 0};;

let doubleTaille tas =
    let taille = tas.taille in
    let nouveauTab = Array.init (taille*2) (fun i -> if i < taille then tas.tab.(i)
                                                     else ((-1, -1), max_float))
    in tas.tab <- nouveauTab;;

let fils i = (2*i + 1, 2*i + 2);;

let parent i = (i-1)/2;;

let echange tab i j =
    let tmp = tab.(i) in
    tab.(i) <- tab.(j);
    tab.(j) <- tmp;;

let tasVide tas = (tas.taille = 0);;

let rec tasTasseUp tas i =
    let p = parent i in
    let (_, p1) = tas.tab.(i) in
    let (_, p2) =  tas.tab.(p) in
    if p2 > p1 then
        begin echange tas.tab i p; tasTasseUp tas p end;;

```

```
let rec tasTasseDown tas i =
    if i < tas.taille then
        let fg, fd = fils i in
        let fMin = (if fg >= tas.taille then
            if fd < tas.taille then fd else i
            else if fd >= tas.taille then fg
                else
                    let (_, p1) = tas.tab.(fg) in
                    let (_, p2) =  tas.tab.(fd) in
                    if p1 > p2 then fd else fg) in
                    let (_,p1) = tas.tab.(i) in
                    let (_,p2) = tas.tab.(fMin) in
                    if p1 > p2 then
                    begin echange tas.tab i fMin; tasTasseDown tas fMin end;;

let tasAjoute tas e =
    begin
        if tas.taille = Array.length tas.tab then doubleTaille tas;
        tas.tab.(tas.taille) <- e;
        tas.taille <- tas.taille + 1;
        tasTasseUp tas tas.taille;
    end;;

let tasExtrait tas =
    let e = tas.tab.(0) in
        begin
            tas.taille <- tas.taille - 1;
            tas.tab.(0) <- tas.tab.(tas.taille);
            tasTasseDown tas 0;
            end; e;;

```

```
let rec tasPriorite tas sommet =
    try
        for i = 0 to tas.taille-1 do
            let (s, p) = tas.tab.(i) in
            if s = sommet then raise (Priorite p)
        done; failwith "N'est pas dans le tas."
    with Priorite(p) -> p;;

let tasDiminuePriorite tas sommet valeur =
    try
        for i = 0 to tas.taille-1 do
            let (s, p) = tas.tab.(i) in
            if s = sommet then
                begin
                    tas.tab.(i) <- (sommet, valeur);
                    raise (Indice i);
                end
        done; failwith "N'est pas dans le tas."
    with Indice(i) -> tasTasseUp tas i;;

```

Mise en place de Dijkstra

```
1  let abs x = if x < 0 then -x else x;;
2
3  let rec calcul_colonne (i1, j1) (i2, j2) etage = (* i identique *)
4      if j2 > j1 then calcul_colonne (i2, j2) (i1, j1) etage
5      else
6          let compteur = ref 0 in
7          for i = i1 to (i1+1) do
8              for j = j1 to (j2+1) do
9                  compteur := !compteur + List.length etage.(i).(j);
10             done;
11         done; !compteur;;
12
13 let rec calcul_ligne (i1, j1) (i2, j2) etage = (* j identique *)
14     if i2 > i1 then calcul_colonne (i2, j2) (i1, j1) etage
15     else
16         let compteur = ref 0 in
17         for i = i1 to (i2+1) do
18             for j = j1 to (j1+1) do
19                 compteur := !compteur + List.length etage.(i).(j);
20             done;
21         done; !compteur;;
22
23 let calcul_densite (i1, j1) (i2, j2) etage =
24     if i1 = i2 then float_of_int (calcul_colonne (i1, j1) (i2, j2) etage)
25     else if j1 = j2 then float_of_int (calcul_ligne (i1, j1) (i2, j2) etage)
26     else failwith "Problème pour Dijkstra";;
27
28 let distance_manhattan (i1, j1) (i2, j2) = float_of_int ((abs ((i1-i2)*350)) + (abs((j1-j2)
       *700)));;
29
```

```
let dijkstra depart ((iFinal, jFinal) as arrivee) etage =
    let n = Array.length listeAdj in
    let m = Array.length listeAdj.(0) in
    let predecesseur = Array.make_matrix n m (-1,-1) in
    let visite = Array.make_matrix n m false
    and fp = tasInitialise () in
    tasAjoute fp (depart, 0.);
    for i = 0 to n-1 do
        for j = 0 to m-1 do
            if (i, j) <> depart then tasAjoute fp ((i, j), max_float)
        done;
    done;
    while not visite.(iFinal).(jFinal) do
        let ((iS, jS) as sommet, distance) = tasExtrait fp in
            visite.(iS).(jS) <- true;
            if sommet <> arrivee then
                List.iter (fun ((i, j) as s) ->
                let d = calcul_densite s sommet etage in
                if not visite.(i).(j) && tasPriorite fp s > (distance +. d) then
                    begin
                        predecesseur.(i).(j) <- sommet;
                        tasDiminuePriorite fp s (distance+.d);
                    end) listeAdj.(iS).(jS)
    done;
    let chemin = ref [arrivee] in
    while (let (i, j) = List.hd !chemin in predecesseur.(i).(j) <> (-1, -1)) do
        let (iP, jP) = List.hd !chemin in
            chemin := predecesseur.(iP).(jP)::!chemin
        done; !chemin;;
```

Mise en place de A*

```
1  (* Fonctions auxiliaires *)
2  let distance x1 y1 x2 y2 = sqrt((x1 -. x2) ** 2. +. (y1 -. y2)**2.);;
3
4  let plan_of_coordonnees (i, j) = (* P' -> P *)
5  (float_of_int (i*50), float_of_int (j*50));;
6
7  let coordonnees x y = (* P -> P'*)
8      let x = int_of_float x
9      and y = int_of_float y in
10     ((x - (x mod m))/m, (y - (y mod m))/m);;
11
12 let coordonnees_of_sommets (i, j) = (* Plan des sommets-> P' *)
13 ((i*350)/50, (j*700)/50);;
14
15 let sommets_of_coordonnees (i, j) = (* P' -> plan des sommets *)
16 (i/7, j/14);;
17
18 let heuristiqueA couple1 couple2 =
19     let xi, yi = plan_of_coordonnees (coordonnees_of_sommets couple1) in
20     let xj, yj = plan_of_coordonnees (coordonnees_of_sommets couple2) in
21     distance xi yi xj yj;;
22
```

A*

```
let aStar ((i, j) as depart) arrivee etage =
    let n = Array.length listeAdj in and m = Array.length listeAdj.(0) in
    let predecesseur = Array.make_matrix n m (-1,-1) in (* retrouver le chemin *)
    let visite = Array.make_matrix n m false in (* marquer les sommets déjà visités) *)
    let distance = Array.make_matrix n m max_float in
    let filePrio = tasInitialise () in
        tasAjoute filePrio (depart, heuristiqueA depart arrivee);
        distance.(i).(j) <- 0.;
        while not (tasVide filePrio) do
                let ((k,l) as sommet, p) = tasExtrait filePrio in
                visite.(k).(l) <- true;
                if sommet = arrivee then filePrio.taille <- 0
                else List.iter (fun ((vi,vj) as voisin) ->
                    let d = heuristiqueA sommet voisin in
                    if not visite.(vi).(vj) then
                        begin
                            tasAjoute filePrio (voisin, distance.(k).(l) +. d +.
                            heuristiqueA voisin arrivee);
                            predecesseur.(vi).(vj) <- (k, l);
                            distance.(vi).(vj) <- distance.(k).(l) +. d;
                        end)
                listeAdj.(k).(l)
        done;
    let chemin = ref [arrivee] in
    let sommet = ref arrivee in
    while !sommet <> depart do
        let (iS, jS) = !sommet in
            sommet := predecesseur.(iS).(jS);
            chemin := !sommet::!chemin;
    done; !chemin;;

```

Fonctions de calcul

```
let norme vect = sqrt(vect.vx**2. +. vect.vy**2.);;

let normaliser v = let n = norme v in {vx = v.vx/.n; vy = v.vy/.n};;

let radian_of_degre deg = (deg *. Float.pi) /. 180.;;

let rotation angle vect = (* theta en degré *)
    let x = vect.vx in
    let y = vect.vy in
    let a = radian_of_degre angle in
    let x' = (x *. cos a -. y *. sin a) in
    let y' = (x *. sin a +. y *. cos a) in
    if abs x' < epsilon then
        if abs y' < epsilon then (0., 0.) else (0., y')
    else
        if abs y' < epsilon then (x', 0.) else (x', y');;

let densite etage i j = (* calcul de la densité dans le périmètre de sécurité *)
    let d = ref 0 in
    let k = ref 0 in
    for p = max 0 (i-1) to min (i+1) (n-1) do
        for q = max 0 (j-1) to min (j+1) (n-1) do
            incr k;
            d := !d + (List.length etage.(p).(q));
        done;
    done; (!d, !k);;

```

Fonctions de vérifications/collisions

```
1  let appartient_rectangle x0 y0 tailleX tailleY x y =
2  (x0 <= x && x <= (x0+tailleX)) && (y0 <= y && y <= (y0+tailleY));;
3
4  let murs x y =
5  (appartient_rectangle 100 100 250 600 x y) ||
6  (appartient_rectangle 450 100 250 600 x y);;
7
8  let bords x y =
9      let x = int_of_float x
10     and y = int_of_float y in
11     murs x y;;
12
13 let collision x1 y1 x2 y2 = (* collision entre deux personnes *)
14 (distance x1 y1 x2 y2) < 2.*.csteRayon;;
15
16 (* collision entre une personne et une liste de personnes. On prend en compte la possibilité
       que p appartiennent à la cellule en question *)
17 let collision_dans_cellule i j etage p x y =
18 List.exists (fun voisin -> p <> voisin && collision x y voisin.x voisin.y) etage.(i).(j);;
19
```

Fonctions de génération

```
let rec genere_coordonnees i j etage = (* génère des coordonnées dans le sommet (i, j) *)
    let x = Random.float (float_of_int (2*m)) in
    let y = Random.float (float_of_int (2*m)) in
    let xp, yp = (x +. 350.*.float_of_int i, y +. 700.*.float_of_int j) in
    let p, q = coordonnees xp yp in
    if collision_dans_cellule p q etage {x = xp; y = yp; v = 0.; chemin = []} xp yp
        then genere_coordonnees i j etage
    else (xp, yp);;

let genere_chemins couple etage fonction = (* prend en compte toutes les sorties *)
    let chemin = ref (fonction couple (List.hd sorties) etage) in
    List.iter (fun sortie ->
        let c = aStar couple sortie etage in
        if List.length c < List.length !chemin then chemin := c)
    (List.tl sorties); List.tl !chemin;;

let v_course () = (10. +. (Random.float 3.))*.10./.3.6;;

let v_marche () = (3. +. (Random.float 2.))*.10./.3.6;;

let determine_v i j etage =
    let (densite, k) = densite etage i j in (* k = nombre de cellules de P_s *)
    if densite = 1 then v_course ()
    else v_course()*.(1.-.((float_of_int densite)-.1.)/.float_of_int (nMAX*k));;

```

Fonctions de génération

```
let genere_personne etage =
    let i = Random.int 3 in
    let j = Random.int 2 in
    let x, y = genere_coordonnees i j etage in
    {x = x; y = y; v = 0.; chemin = []};;

let rec ajoute_personne etage =
    let p = genere_personne etage in
    let (i, j) = coordonnees p.x p.y in
    etage.(i).(j) <- p::etage.(i).(j);;

let genere_etage nombrePersonnes =
    let etage = Array.make_matrix n n [] in
    for i = 0 to nombrePersonnes-1 do
        ajoute_personne etage;
    done; etage;;

```

Fonctions d'affichage

```
let initialisation_etage () =
    begin
        set_color blanc;
        fill_rect 0 0 800 800;
        set_color gris;
        fill_rect 100 100 250 600;
        fill_rect 450 100 250 600;
        set_color bleu;
        for i = 0 to 2 do
            for j = 0 to 1 do
                fill_rect (i*350) (j*700) 100 100;
            done
        done;

        (* délimitation du plan P' *)
        set_color black;
        for i = 0 to n-1 do
            moveto (i*m) 0;
            lineto (i*m) 800;
        done;
        for i = 0 to n-1 do
            moveto 0 (i*m);
            lineto 800 (i*m);
        done;
    end;;

```

Fonctions d'affichage

```
1  let affiche_personne p = (* affiche une personne*)
2      begin
3          set_color rouge;
4          fill_circle (int_of_float(p.x)) (int_of_float(p.y)) (int_of_float csteRayon);
5      end;;
6
7  let affiche_etage etage = (* affiche etage + emplacement des personnes*)
8      begin
9          initialisation_etage ();
10         Array.iter (fun l -> (Array.iter (fun cellule ->  (List.iter (fun p -> affiche_personne
       p) cellule)) l)) etage
11     end;;
12
13
```

Fonctions de déplacement

```
let actualise_chemin etage =
for i = 0 to n-1 do
    for j = 0 to n-1 do
        let couple = sommets_of_coordonnees (i,j) in
        List.iter (fun p -> p.chemin <- genere_chemins couple etage dijkstra) etage.(i).(j)
    done;
done;;

let modifie_vitesse_etage etage =
    for i = 0 to n-1 do
        for j = 0 to n-1 do
            if List.length etage.(i).(j) > 0 then let v = determine_v i j etage in
                List.iter (fun p -> p.v <- v) etage.(i).(j)
        done;
    done;;

let angles = let a = Array.make nROT 0. and j = ref 1 in
    for i = 1 to (nROT-1)/2 do
        begin
        a.(!j) <- theta *. float_of_int i;
        a.(!j + 1) <- -. a.(!j);
        j := !j + 2;
        end;
    done; a;;

let vecteur_vitesse p = (* renvoie le vecteur vitesse d'une personne *)
    let (xSommet, ySommet) = plan_of_coordonnees (coordonnees_of_sommets (List.hd p.chemin)) in
    let xSommet, ySommet = xSommet+.50., ySommet+.50. in
    let v = normaliser {vx = xSommet -. p.x; vy = ySommet -. p.y} in
    {vx = v.vx*.p.v; vy = v.vy*.p.v};;

```

```
(* vérifie s'il y aura une collisition, si c'est le cas elle opère une rotation adaptée *)
let vecteur_rotation p vect etage =
    try
        Array.iter (fun a ->
            let vx, vy = rotation a vect in
            let (i, j) = coordonnees (p.x +. vx) (p.y +. vy) in
            if i < n && j < n then
            if not (collision_dans_cellule i j etage p (p.x +. vx) (p.y +. vy)) &&
            not (bords (p.x +. vx) (p.y +. vy)) then raise (DirectionTrouvee a))
        angles; (0., 0.);
    with DirectionTrouvee(a) -> rotation a vect;;

let rayonDeRotation = float_of_int m;;

let applique_deplacement_liste etage i j =
let rec aux l = match l with
    | [] -> l
    | p::q -> try
    if List.length p.chemin = 0 then begin incr nombreEvacues; raise ChangementCellule end
    else
        begin
            let prochainSommet = List.hd p.chemin in
            let xS, yS = plan_of_coordonnees (coordonnees_of_sommets prochainSommet) in
            let xS, yS = xS+.50., yS+.50. in
            if distance p.x p.y xS yS < rayonDeRotation then
                if List.length p.chemin = 1 then p.chemin <- [] (* proche du sommet final *)
                else p.chemin <- List.tl p.chemin (* proche d'un sommet intermédiaire *)
            else let vecteurOptimal = vecteur_vitesse p in (* loin du sommet *)
                let vx, vy = vecteur_rotation p vecteurOptimal etage in
                begin p.x <- p.x +. vx; p.y <- p.y +. vy end;
        end;
```

```
1          (* si la personne a changé de cellule, il faut la supprimer de l'ancienne et l'ajouter
       dans la nouvelle *)
2          let (i0, j0) = coordonnees p.x p.y in
3              if (i0 <> i) || (j0 <> j)
4                  then begin etage.(i0).(j0) <- p::etage.(i0).(j0); raise ChangementCellule end
5              else p::(aux q);
6
7          with ChangementCellule -> aux q
8      in etage.(i).(j) <- aux etage.(i).(j);;
9
10 let applique_deplacement_etage etage =
11     for i = 0 to n-1 do
12         for j = 0 to n-1 do
13             applique_deplacement_liste etage i j;
14         done
15     done;;
16
```

Fonction principale

```
let main nombrePersonnes =
    let taillePopulation = ref (min tailleMax nombrePersonnes) in
    let etage = genere_etage !taillePopulation in
        try
            while true do
                begin
                    auto_synchronize false;
                    modifie_vitesse_etage etage;
                    applique_deplacement_etage etage;
                    affiche_etage etage;
                    if !nombreEvacues = nombrePersonnes then raise Stop; (* arrêt *)
                    if 0 < (nombrePersonnes - !taillePopulation) then
                        for i = 1 to min flux (nombrePersonnes - !taillePopulation) do
                            begin ajoute_personne etage; incr taillePopulation end
                        done;
                    incr dt;
                    synchronize ()
                end
            done;
        with Stop -> print_int !dt;;  (* affichage des résultats *)
```