Jeu des Policiers-et-voleur sur les graphes planaires

MARGOT MOTTAIS 23837

Plateau du jeu de société *Scotland Yard*

Motivations

Plateau du jeu de société *Scotland Yard*

Comment capturer un fugitif dans un réseau de transport ?

Problématique

Plan

- Réalisation naïve
- Implémentation d’un second algorithme
- Résultats comparés

Modélisation

Réseau de transport → Graphe

Données :

- Un graphe connexe non orienté
- k policiers
- 1 voleur

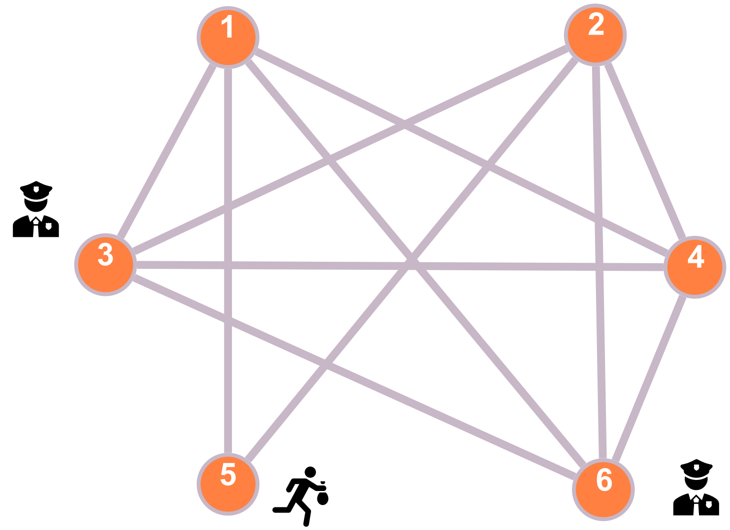
Jeu tour par tour

Initialisation : policiers puis voleur

Capture : un des policiers occupe le sommet du voleur

Voleur :

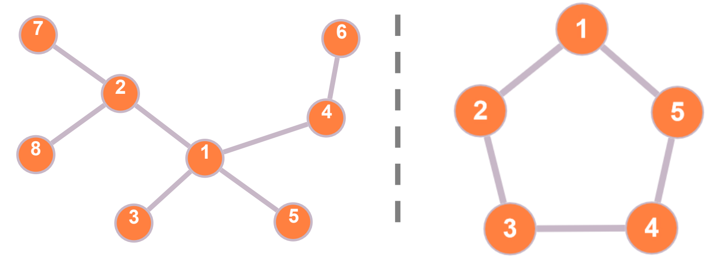
Tirage uniforme parmi les voisins libres



Problème des Cops-and-Robber

1983 — Quilliot, Nowakowski et Wrinkler

Cop-win | Robber-win



1984 — Aigner et Fromme

$$G\ est\ planaire \Rightarrow G\ est\ 3\ cop\ win$$

2016 — Pisantechakool et Tan

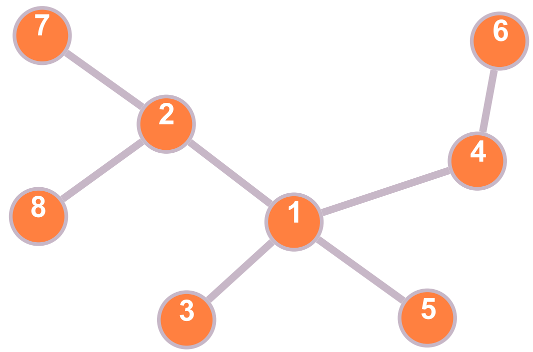
$$G\ est\ planaire \Rightarrow\ capt_3(G) \leq 2|S|$$

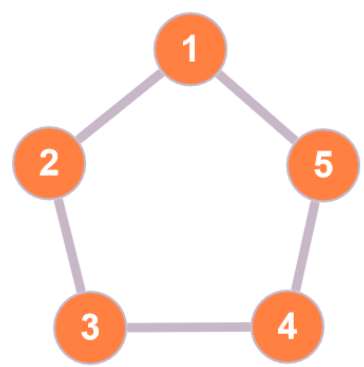
Cas planaire

Aigner et Fromme :

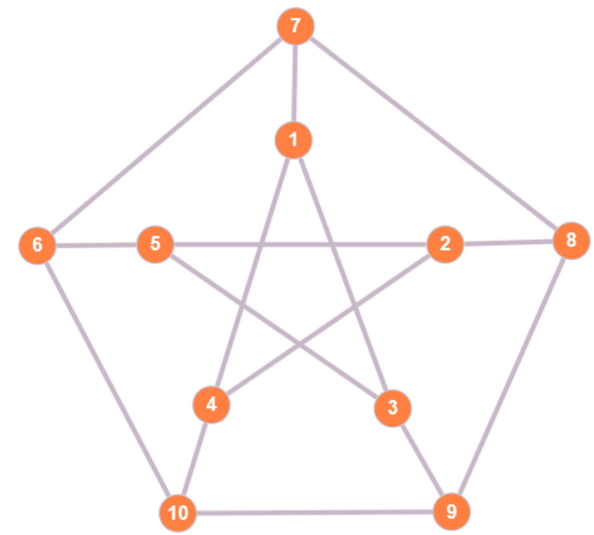
G *est planaire* $\Rightarrow G$ *est 3 cop win*

PLANAIRE





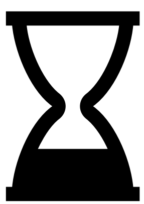
NON-PLANAIRE



Réalisation naïve

POLICIERS

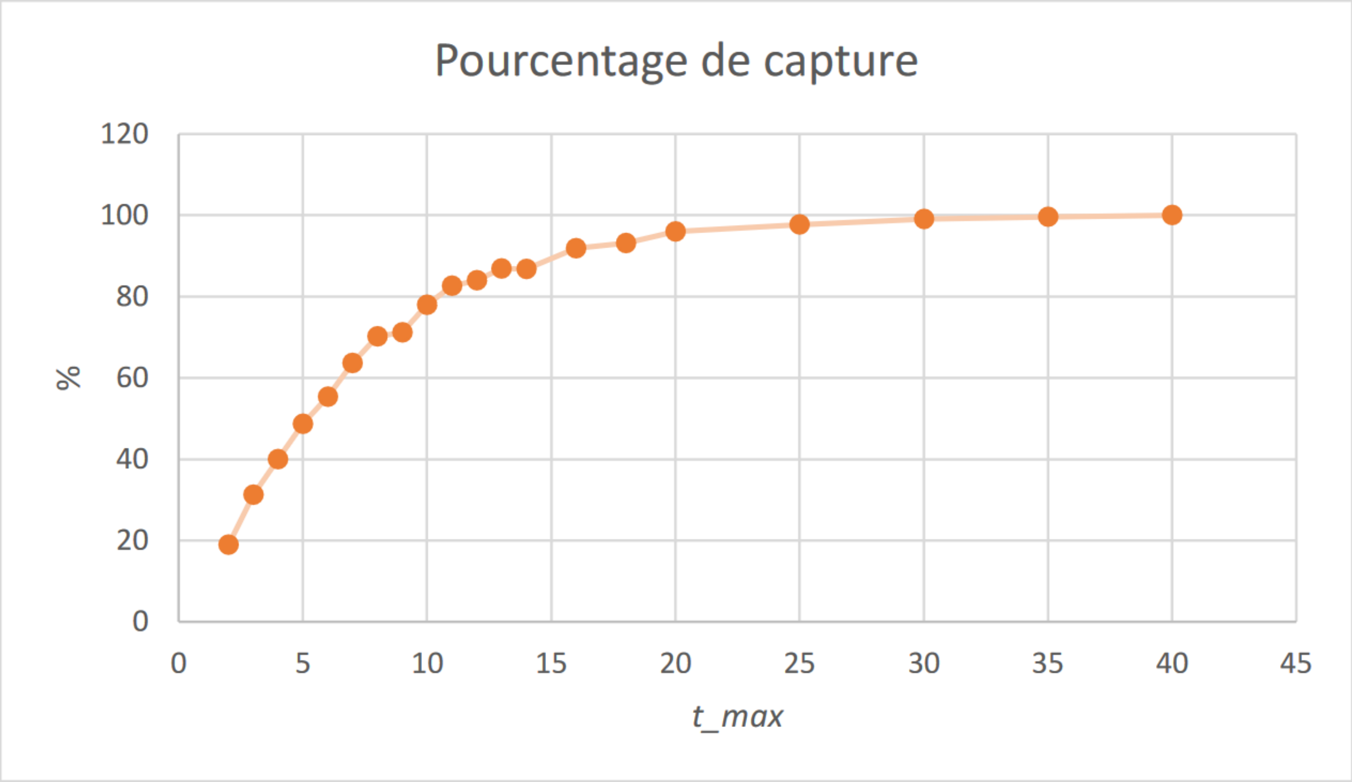
Recherche du chemin le plus court vers le voleur
→ Parcours en largeur



Si $t_{capture} < t_{max}$: victoire des policiers
Sinon : victoire du voleur

où t = nombre de tours

Premiers résultats

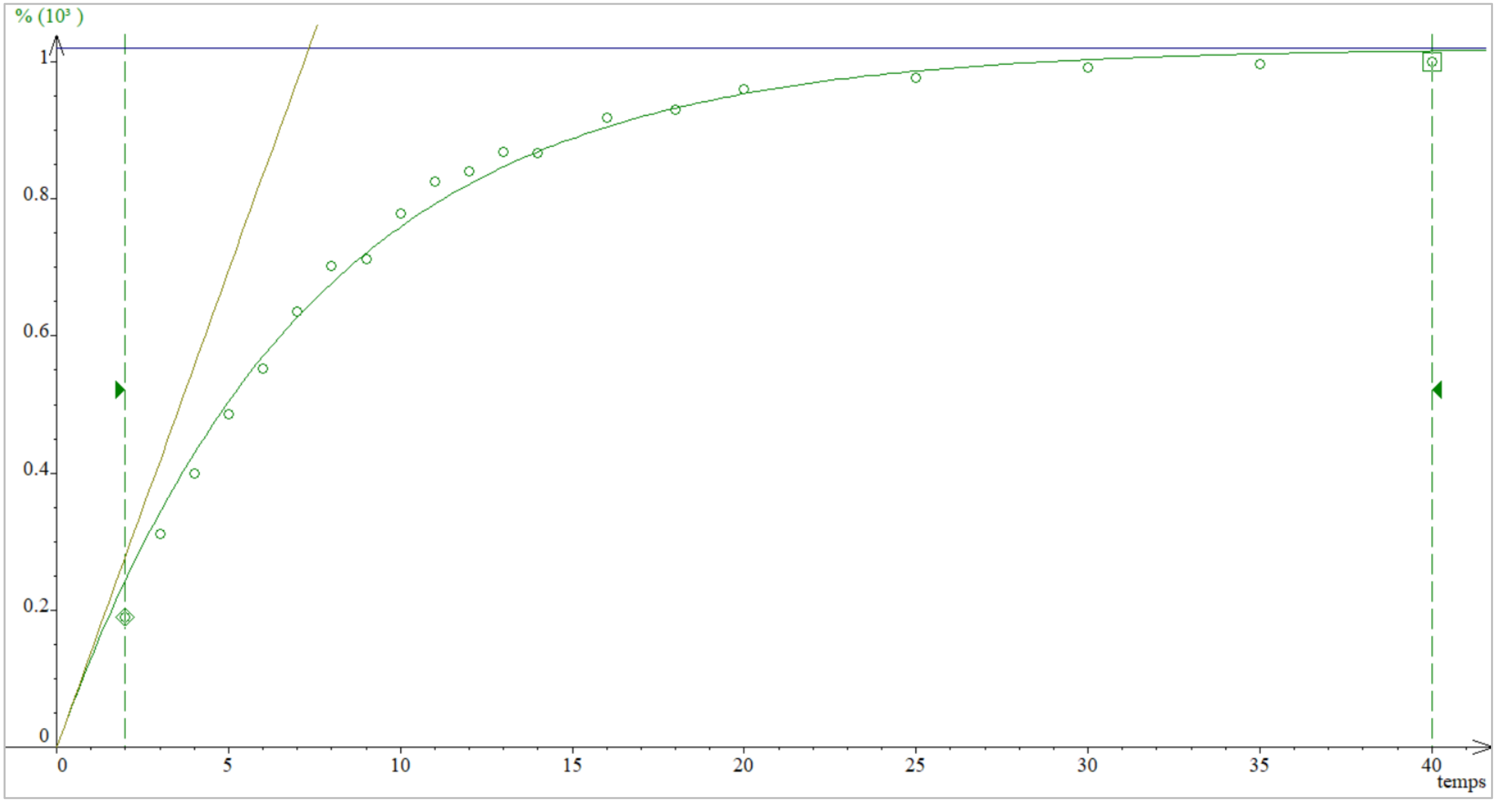


PARAMETRES : à *t_max* fixé

- 10 graphes aléatoires en pavage hexagonal élagué à 95 sommets
- Moyenne sur 100 parties pour chaque graphe

Premiers résultats

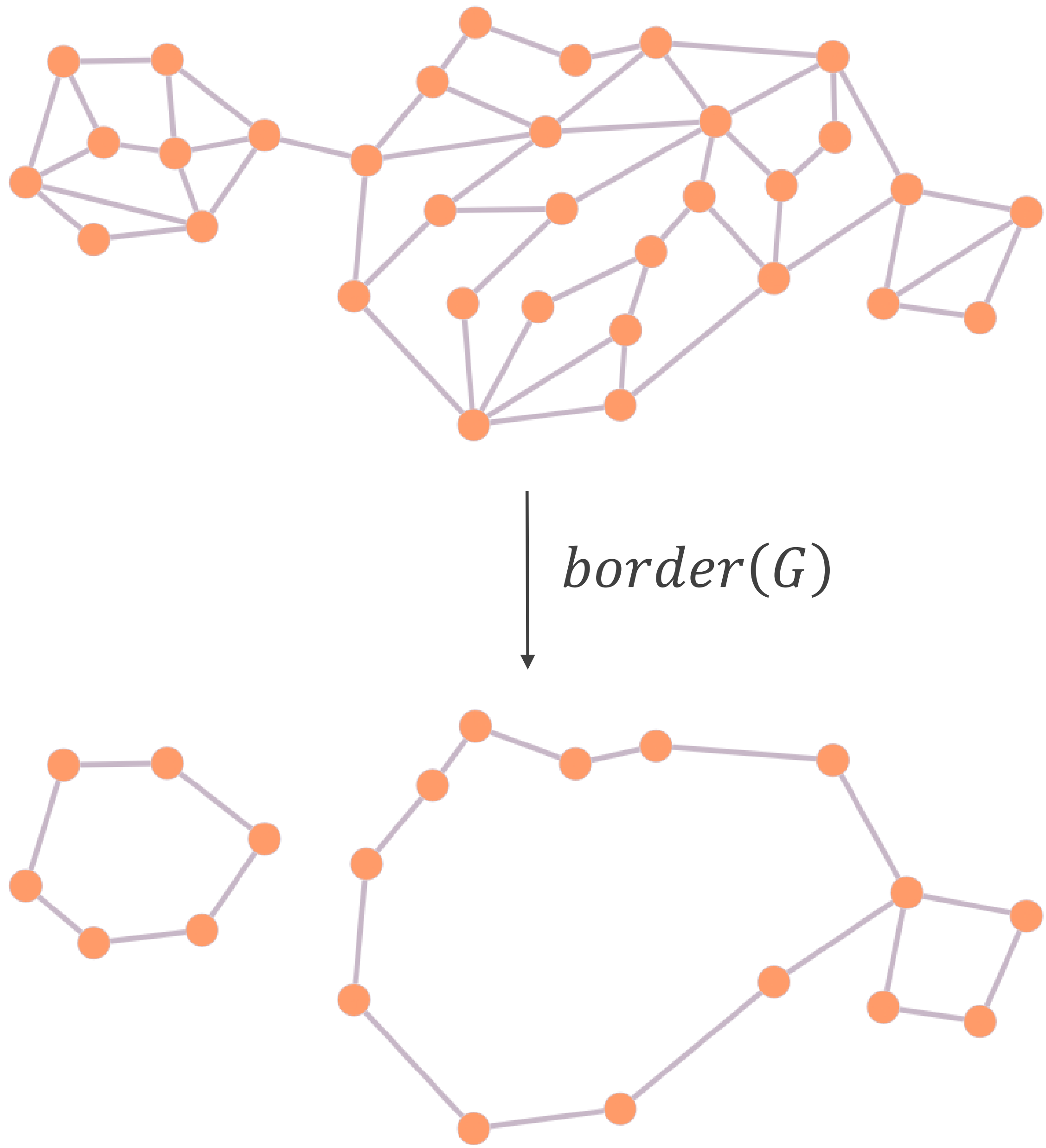
$$p(t\_max) \approx 1 - e^{-\left(t\_max/\tau\right)} \qquad \text{avec } \tau = 7{,}3$$



Nouvelle approche

Algorithme de Pisantechakool et Tan

On the Capture Time of Cops and Robbers Game on a Planar Graph



Algorithme de Pisantechakool et Tan

ENTREES : un graphe connexe planaire G, un pointeur vers un entier nb_{tour}

SORTIE : effet de bord : le pointeur contient le nombre de tours qu'a nécessité la capture

STRATEGIE:

$initialisation\ (G)$:

$R_0 \leftarrow G$

$e0 \leftarrow départ\_policiers()$

$x0 \leftarrow départ\_voleur(e0)$

$R_1 \leftarrow G - \mathrm{e0}$

$nb_{tour} \leftarrow 1$

$phase\_recursive(R_1)$

A l'étape i:

R_i : Territoire du voleur

$P_i^{1,2}$: Chemins 1, 2 gardés par les policiers

$phase\_recursive(R_i)$:

$nb_{tour} \leftarrow nb_{tour} + 1$

Si R_i est un arbre :

$solution\_naive\_arbre(R_i)$

Sinon :

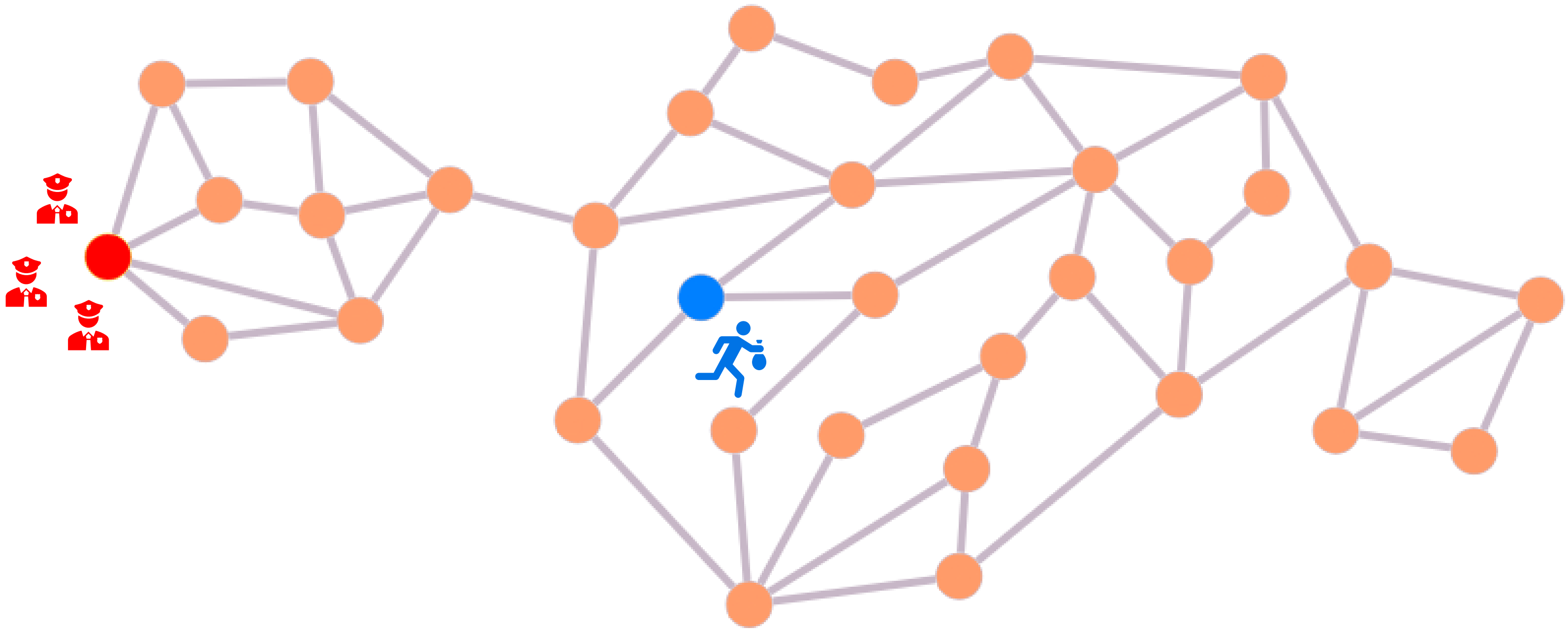
// réduire strictement le territoire du voleur en gardant 1 ou 2 nouveaux chemins

$P_{i+1}^1, P_{i+1}^2 \leftarrow nouveaux\_chemins(R_i)$

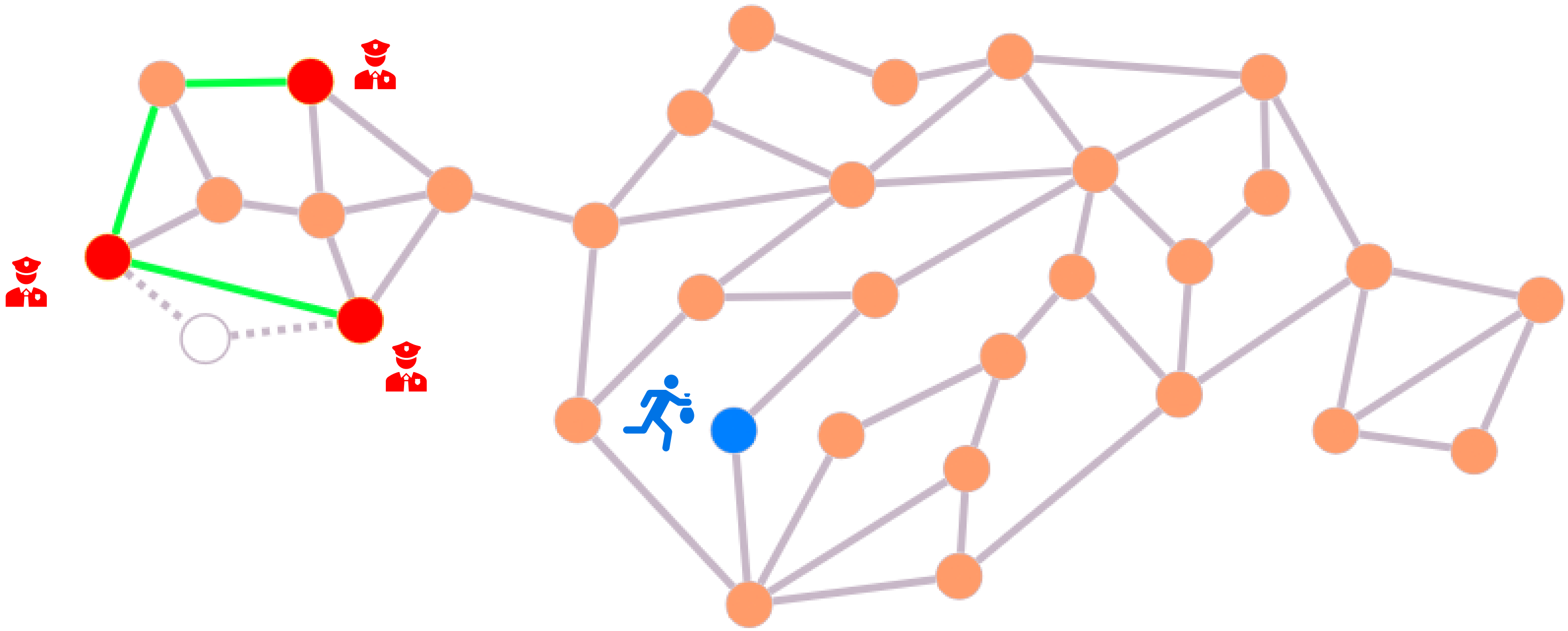
$R_{i+1} \leftarrow nouveau\_territoire(R_i, P_{i+1}^1, P_{i+1}^2)$

$phase\_recursive(R_{i+1})$

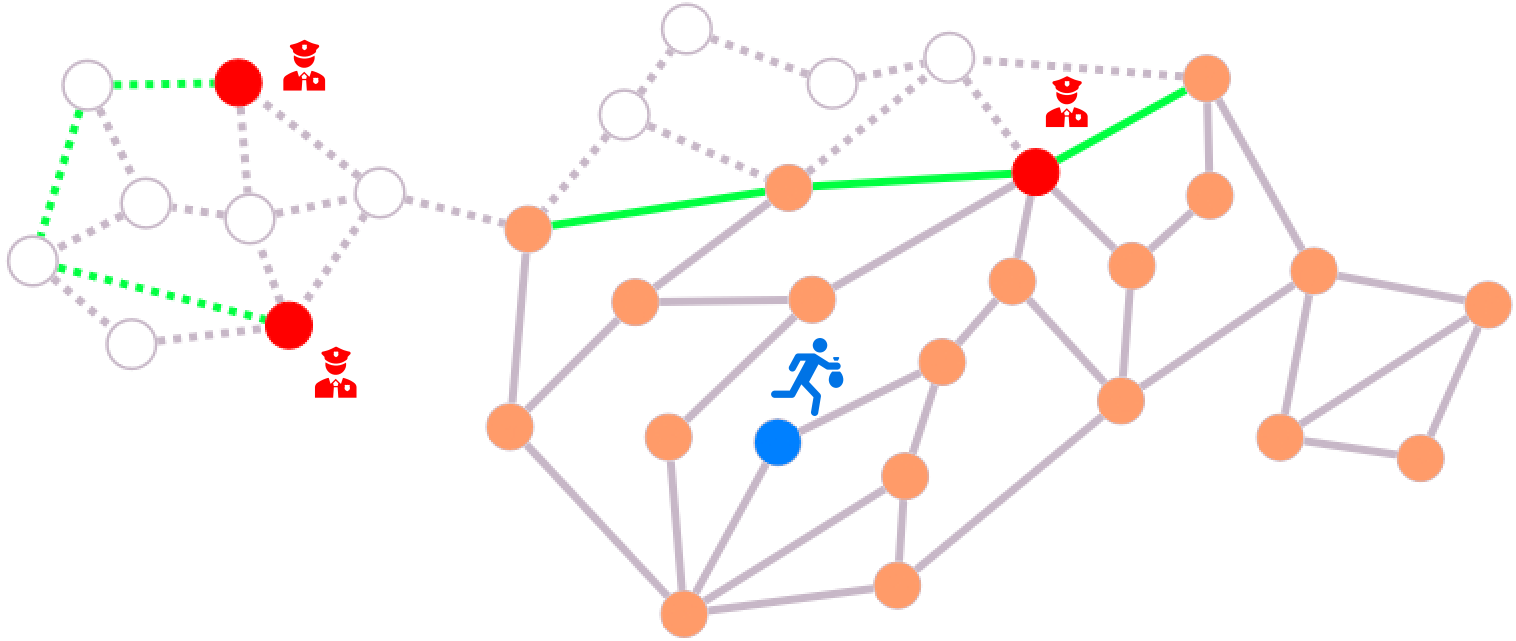
$$initialisation\,(G) : R_1 \leftarrow G \; - e0$$



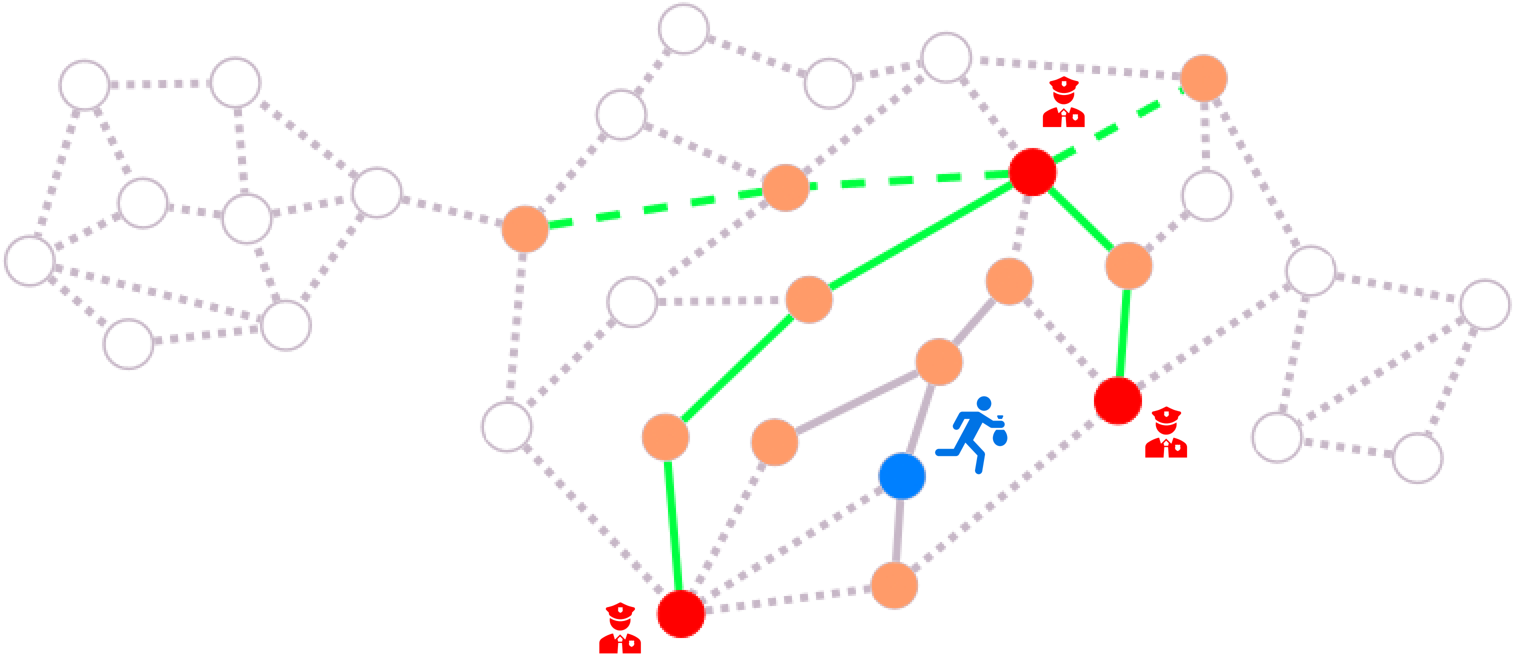
$phase\_recursive(R_1)$



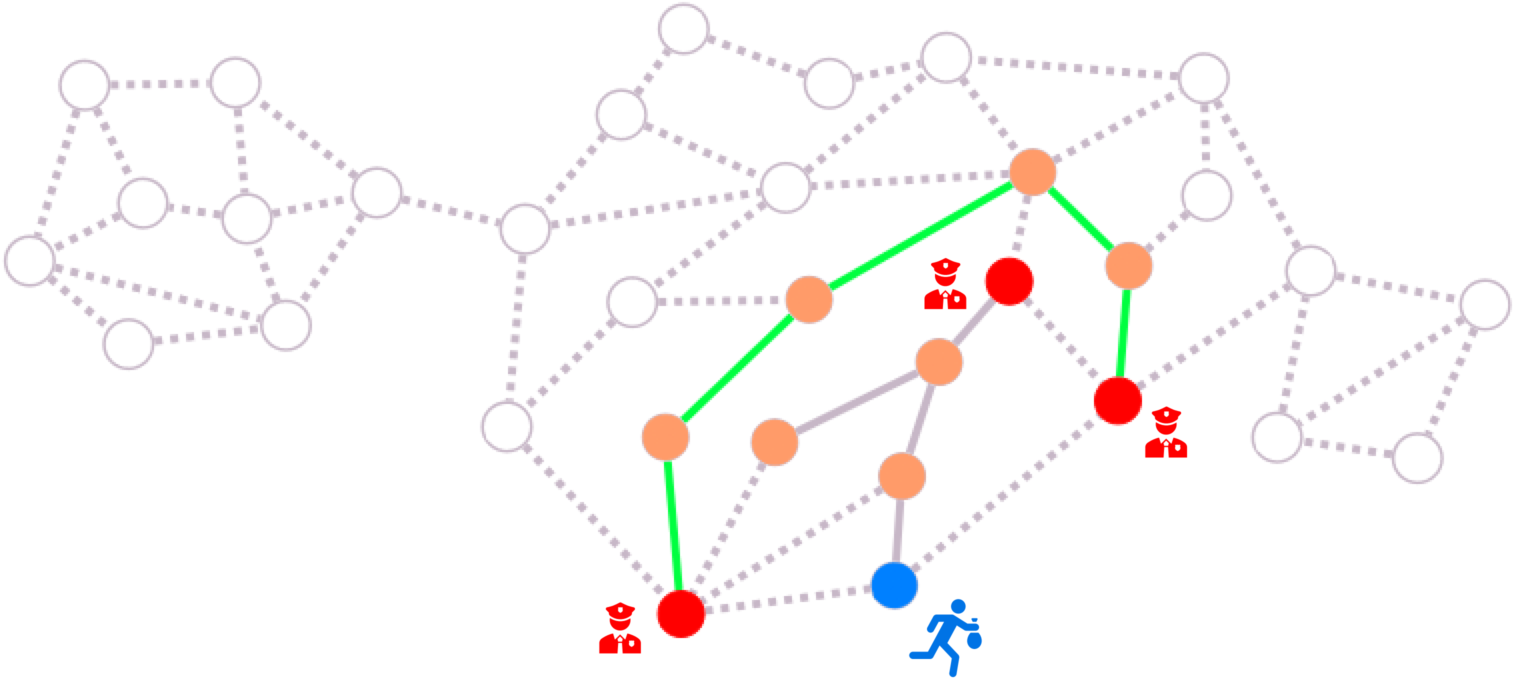
$$phase\_recursive(R_2)$$



$$phase\_recursive(R_3)$$



$solution\_naive\_arbre(R_4)$



Analyse comparée

	TERMINAISON	CORRECTION	COMPLEXITE
Algorithme naïf	Non assurée	/	$O\big(n\,(n+p)\big) = O(n^2)$
Algorithme de Pisantechakool et Tan	$R_{i+1} \subsetneq R_i$	preuve de $G\ est\ planaire \Rightarrow capt_3(G) \leq 2\|S\|$	$O(n^4)$

$G = (S, A)$ où $|S| = n,\ |A| = p$

Implémentation et tests

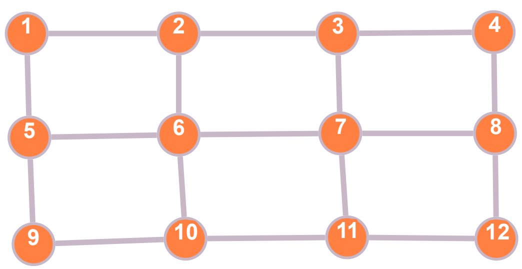
- 20.000 graphes générés par *plantri*

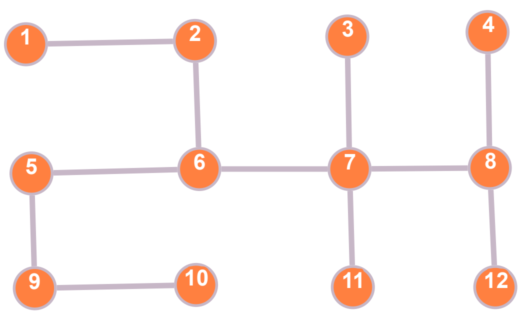
  → Problèmes : $|S| \leq 11$
  Représentation planaire inconnue

- Fabrication de graphes avec leur représentation planaire

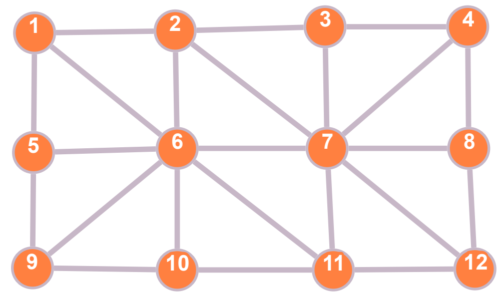
http://users.cecs.anu.edu.au/~bdm/data/formats.html
http://users.cecs.anu.edu.au/~bdm/data/graphs.html

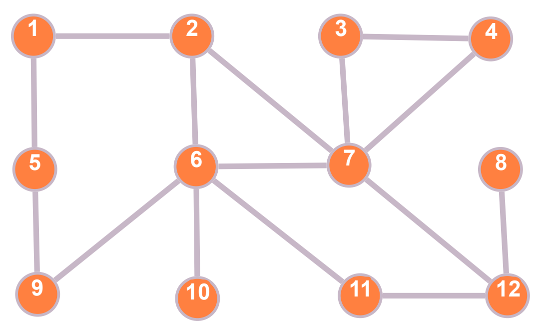
CARRES



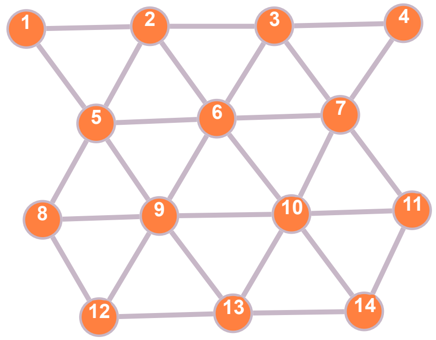


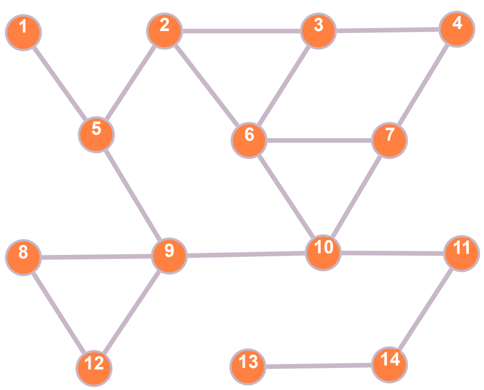
CARRES PLEINS





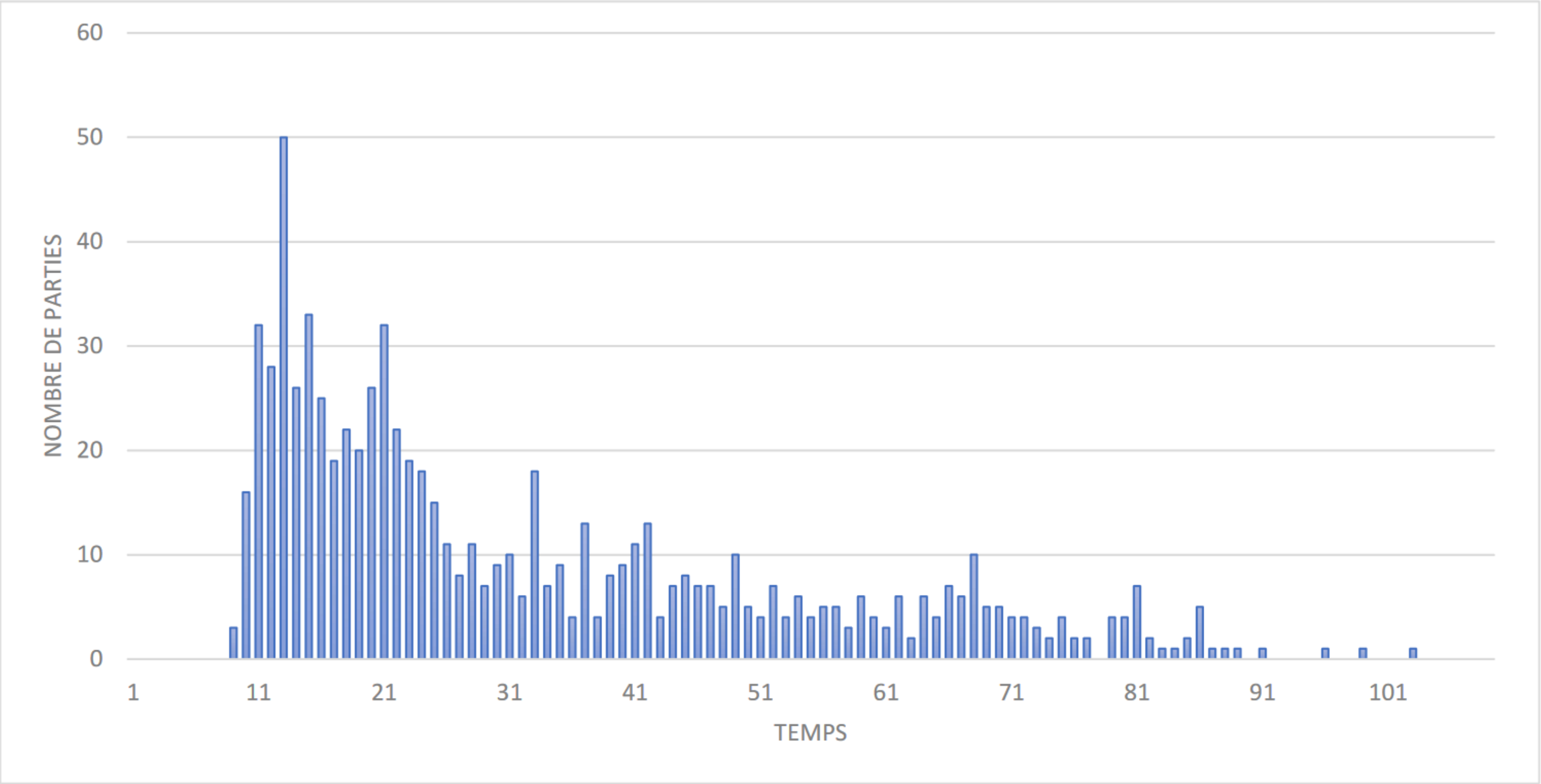
HEXAGONES



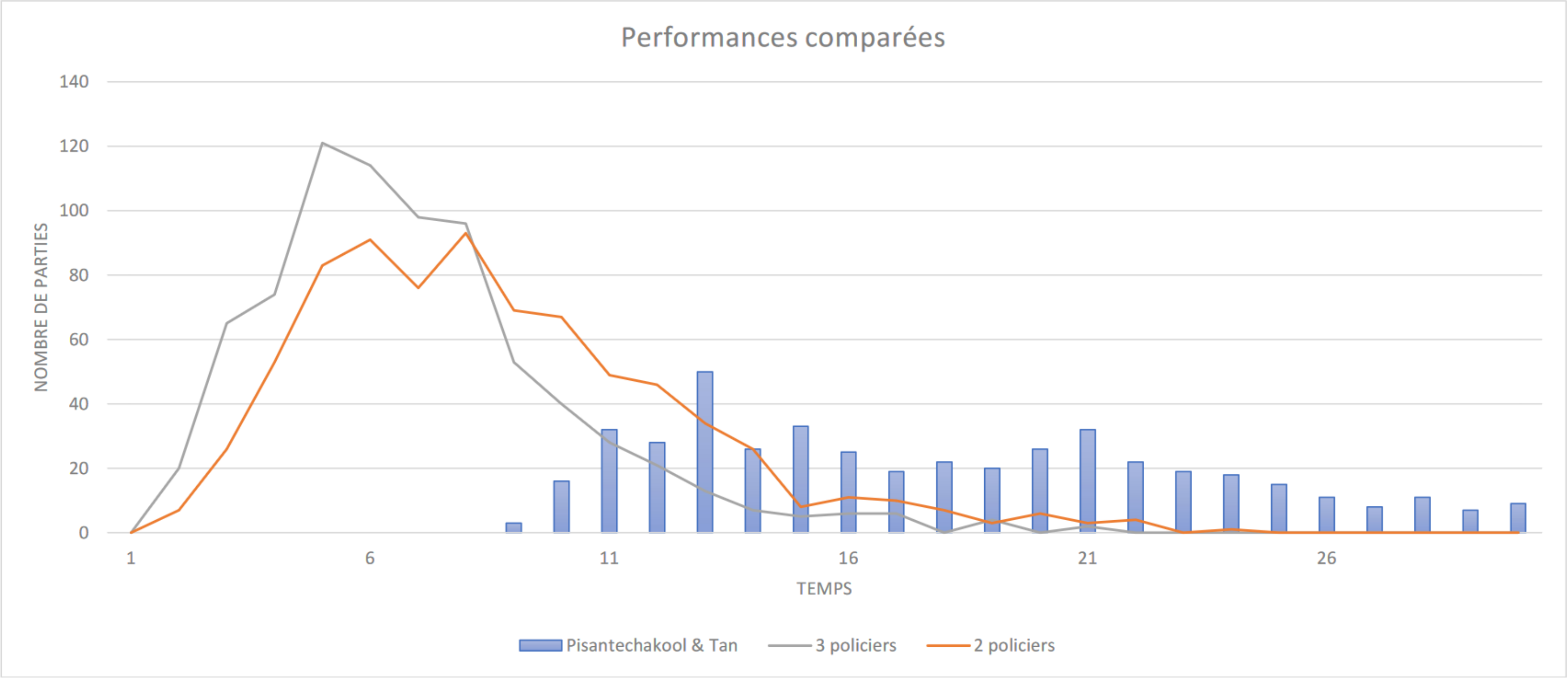


Résultats

Répartition des temps de capture sur 775 tests, grille 7 x 7



Résultats



Performances comparées
140
120
100
80
60
40
20
0
NOMBRE DE PARTIES
1
6
11
16
21
26
TEMPS
Pisantechakool & Tan
3 policiers
2 policiers

Conclusion

ALGORITHME NAÏF

- Satisfaisant pour une majorité de cas
- Cas pathologiques de victoire du voleur

ALGORITHME DE PISANTECHAKOOL ET TAN

- Très fiable…
- … mais très gourmand
- Moins performant en moyenne

```
point -> point -> int
let dist2 ((x1, y1) : point) ((x2, y2) : point) : int = ((x1 - x2) * (x1 - x2)) + ((y1 - y2) * (y1 - y2))
  (*distance euclidienne au carré entre deux points*)


point -> point -> point -> float
let theta ((xp, yp) : point) ((xu, yu) : point) ((xv, yv): point) : float =
  let phi = (yv - yp)*(xu - xp) - (xv - xp)*(yu - yp) in
  let d_uv = float_of_int (dist2 (xu, yu) (xv, yv)) in
  let d_up = float_of_int (dist2 (xu, yu) (xp, yp)) in
  let d_vp = float_of_int (dist2 (xv, yv) (xp, yp)) in
  let theta_0 = acos((d_uv +. d_up -. d_vp) /. (2. *. sqrt(d_uv)  *. sqrt(d_up) )) in
  if phi > 0 then theta_0
  else if phi < 0 then ((2. *. pi ) -. theta_0)
  else pi


point -> point -> int -> int list -> embedded_graph -> int
let voisin_plus_petit_angle (u : point) (p : point) (num_pere : int) (voisins : int list) (e_g : embedded_graph)=
  (* voisins : liste des voisins de u dans G' sans p (père) *)
  let v_min = ref num_pere in
  let theta_min = ref (2. *. pi) in
  let rec aux (voisins : int list) =
    match voisins with
    | [] -> !v_min
    | v :: vs -> begin
      let v_coord = e_g.coord.(v) in
      let theta_v = theta p u v_coord in
      if theta_v < !theta_min then begin
        v_min := v;
        theta_min := theta_v
      end;
      aux vs
    end
  in aux voisins
```

```
embedded_graph -> edge list
let border_cycles_c (e_g : embedded_graph) : edge list =
  (*détermine la bordure cyclique d'un graphe g *)
  (* fonction C(V) définie dans l'article*)
  let g, coord, card, pres = e_g.g, e_g.coord, e_g.card, e_g.pres in
  if card <= 2 then []
  else begin
    (***** étape 1*)
    let d = get_starting_point coord pres in
    let l_exec = ref [d] in
    let x, y = coord.(d) in
    let curr = ref (voisin_plus_petit_angle (x, y) (x, y + 1) (-1) g.(d) e_g ) in
    (* initialisation avec un faux pere de numéro -1 *)
    let pere = ref d in
    l_exec := !curr :: !l_exec;
    while !curr <> d do
      let v_curr = remove !pere g.(!curr) in
      let next = voisin_plus_petit_angle coord.(!curr) coord.(!pere) !pere v_curr e_g in
      l_exec := next :: !l_exec ;
      pere := !curr;
      curr := next
    done;
    (***** étape 2 *)
    let l_border = List.rev !l_exec in
    let res = remove_cut l_border card in
    res
  end
```

```
set -> game_state -> set
let composante_x (esb : set) (s : game_state) : set =
(* parcours en profondeur : prend en argument le set des sommets accessibles par X
 et retroune un set = la composante connexe où est X (= r_i+1)*)
  let ss_eg = set_to_eg esb s in
  let ss_g = ss_eg.g in
  let x0 = s.pos_x in
  let curr_comp = ref [] in
  let found = ref false in
  let vu = Array.make s.e_g.card false in
  let sommets = set_to_list esb in
  let rec dfs (x : int) =
    if not vu.(x) then begin
      vu.(x) <- true;
      curr_comp := x :: !curr_comp;
      if x = x0 then found := true;
      List.iter (fun y -> dfs y) ss_g.(x);
    end
  in
  let rec trouve_comp (l : int list) : int list =
    match l with
    | [] -> failwith "x0 n'est pas dans le territoire de X  \n"
    | x :: xs -> begin
      dfs x;
      if !found = true then !curr_comp
      else begin
        curr_comp := [];
        trouve_comp xs
      end
    end
  in
  let comp_x_list = trouve_comp sommets in
  let res_set = edge_list_to_set comp_x_list s in
  res_set
```

```
edge list -> edge list
let domino_sort (l : edge list) : edge list =
  (* prend en entrée un liste d'arrêtes qui constitue en ensemble de cycles non nécessairement reliés*)
  (* renvoie la liste en mode "domino" , on fait [(a, b); (b, c); (c, d); (e, f)] autant que possible
     pour coller ensemble et mettre dans l'ordre les arêtes des mêmes cycles*)
  let res = ref [] in
  let last = ref (-1) in
  let a_voir =
    match l with
    | [] -> failwith "liste vide domino_sort"
    | x :: xs -> begin
      res := [x];
      last := snd x;
      xs
    end
  in
  let rec aux (l : edge list) : unit =
    match l with
    | [] -> ()
    | (a, b) :: ls -> begin
      let d = find_match !last [] l in
      match d with
      (* on a pas trouvé de domino pour compléter le cycle, on en prend un autre au hasard*)
      | None -> begin
        last := b;
        res := (a, b) :: !res;
        aux ls
      end
      (* on continue un cycle*)
      | Some ((c, d), lreste) -> begin
        if c <> !last then failwith "erreur domino_sort";
        last := d;
        res := (c, d) :: !res;
        aux lreste
      end
    end
  in
  aux a_voir;
  List.rev(!res)
```

```
edge list -> int -> game_state -> int * int
let two_cops_start (c : edge list) (u : int) (s : game_state) : int * int =
  (* Précondition : c est un cycle *)
  let n = List.length c in
  let r_i = r_i s in
  let x = ref u in
  let y = ref u in
  (* floor (n / 3)*)
  let len = (n + 2) / 3 in
  let start_flag = ref true in
  let stop_flag = ref false in
  let rec set_pointer (p : int ref) (l : edge list) (i : int) : unit =
    (* si start_flag = true : on cherche le pointeur pour la premiere fois*)
    (* si stop_flag = true : on a déjà fait le tour une fois de la liste *)
    (* invariant : on a déjà regardé i sommet du cycle après u (u inclus)*)
    (* si x_or_y = true, on est en train de modifier x*)
    match l with
    | [] ->
      if not !stop_flag then begin
      stop_flag := true;
      set_pointer p c i
    end
    | (a, b) :: vs -> begin
      if !start_flag then begin
        (* on veut commencer*)
        if a = u then begin
          start_flag := false;
          set_pointer p vs (i + 1)
        end
        else set_pointer p vs i
      end
      else
        (* on a déjà commencé *)
        if i >= len then begin
          (* on commence à vérifier r_i*)
          if r_i.(a) then p := a
          (* on arrête le programme*)
          else set_pointer p vs (i + 1)
        end
        else set_pointer p vs (i + 1)
      end in
  set_pointer x c 0;
  start_flag := true;
  stop_flag := false;
  let c_rev = List.rev (List.map (fun (a, b) -> (b, a)) c) in
  set_pointer y c_rev 0;
  (!x, !y)
```

```
graph -> set
let tarjan (g : graph) : set =
  (* AP = articulation point = cut vertex *)
  (* renvoie l'ensemble des sommets qui ne sont pas des AP*)
  let n = Array.length g in
  let disc = Array.make n 0 in
  let low = Array.make n (-1) in
  let visited = Array.make n false in
  let isAP = Array.make n false in
  let time = ref 0 in
  let par = ref (-1) in
  let rec aux (x : int) (p : int): unit =
    (* p = parent dans le DFS*)
    let children = ref 0 in
    visited.(x) <- true;
    incr time;
    disc.(x) <- !time;
    low.(x) <- !time;
    List.iter (fun y ->
      if not visited.(y) then begin
      incr children;
      aux y x;
      low.(x) <- min low.(x) low.(y);
      if p <> -1 && low.(y) >= disc.(x) then isAP.(x) <- true;
      end
      else if y <> p then low.(x) <- min low.(x) disc.(y);
    ) g.(x);
    if p = -1 && !children > 1 then isAP.(x) <- true
  in
  aux 0 !par;
  let res = Array.init n (fun i -> not isAP.(i)) in
  res
```