TIPE : Propagation de rumeurs dans un réseau social

Hugo LEVY-FALK

2017

Plan

Rappel de la problématique

Comment propager une rumeur le plus rapidement possible à un maximum de nœuds d'un réseau social ?

Plan

Modélisation

On modélise un réseau social par un graphe.

- Personne $\rightarrow$ Nœud
- Lien social $\rightarrow$ Arête

On ne prend pas en compte la "qualité" de la relation.

Caractéristiques des réseaux simulés

- Stanley Milgram : Six degrés de séparation (Facebook 4.57)
- Algorithme de Watts-Strogatz

Jeu de coordination

- Chaque nœud maximise son gain.
- Un voisin dans l'état "informé" $\rightarrow$ gain a
- Un voisin dans l'état "non-informé" $\rightarrow$ gain b

Si on note p la proportion de voisins informés, le nœud maximise son gain en passant à l'état informé si et seulement si $p \times a > (1 - p) \times b$, ou encore

$$p > \frac{b}{a + b}$$

$\rightarrow$ On caractérise une rumeur par $q = \frac{b}{a+b}$.

Jeu de coordination

$\rightarrow$ On caractérise une rumeur par $q = \frac{b}{a+b}$.

Remarques

Soit un graphe $G = (V, E)$ avec V un ensemble de nœuds et $E \subset V^2$.

- Pas de propagation si $q > 1$;
- Si l'on pose $(V_k)_{k \in \mathbb{N}}$ une suite des nœuds dans l'état "informé" à l'étape k, s'il existe $n \in \mathbb{N}$ tel que $V_n = V_{n+1}$ alors la suite est stationnaire à partir du rang n ;
- La suite étant par ailleurs croissante pour l'inclusion et majorée, la suite converge et on finit une simulation en au plus $|V|$ étapes.

Cluster

Définition : p-cluster

Soit un graphe $G = (V, E)$ avec V un ensemble de nœuds et $E \subset V^2$. On appelle p-cluster tout sous-ensemble $C \subset V$ tel que pour tout $i \in C$ il existe un p-uplet $(v_k)_{k \in [\![1,p]\!]} \in C^p$ deux à deux distincts et tel que pour tout $k \in [\![1, p]\!]$, i et v_k soient voisins.

Remarque

Si le graphe est connexe (cas des graphes étudiés), l'ensemble forme un 1-cluster.

Les clusters sont les seuls obstacles aux rumeurs

Théorème

Les clusters sont les seuls obstacles aux rumeurs.

Comment caractériser une propagation optimale ?

- Capacité à atteindre l'ensemble du graphe ;
- Nombre d'itérations de simulation le plus faible possible ;

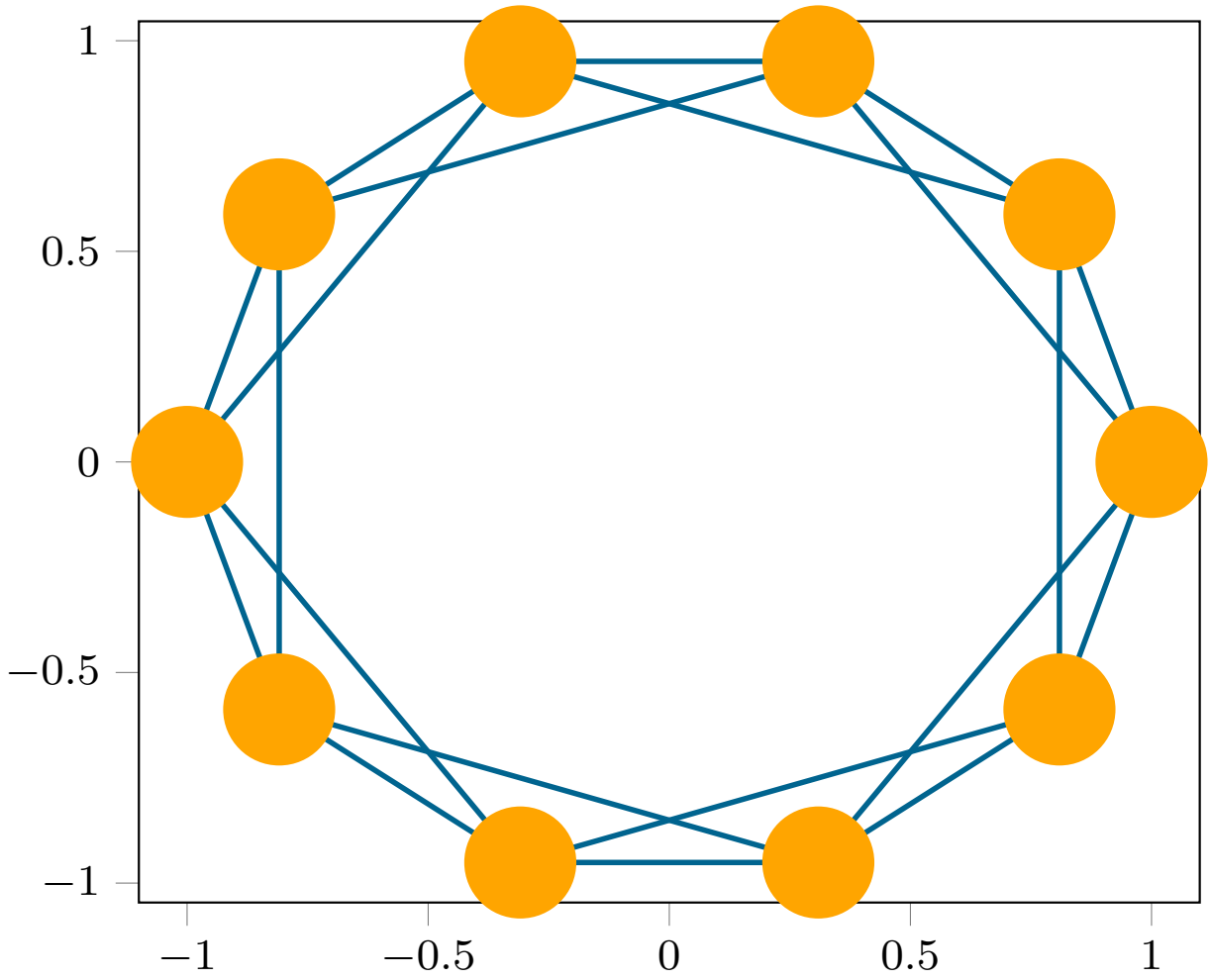
Problème(s) : Unicité de la solution ? identification des propriétés permettant une telle propagation ?
$\rightarrow$ Comparaison de critères arbitraires.

Plan

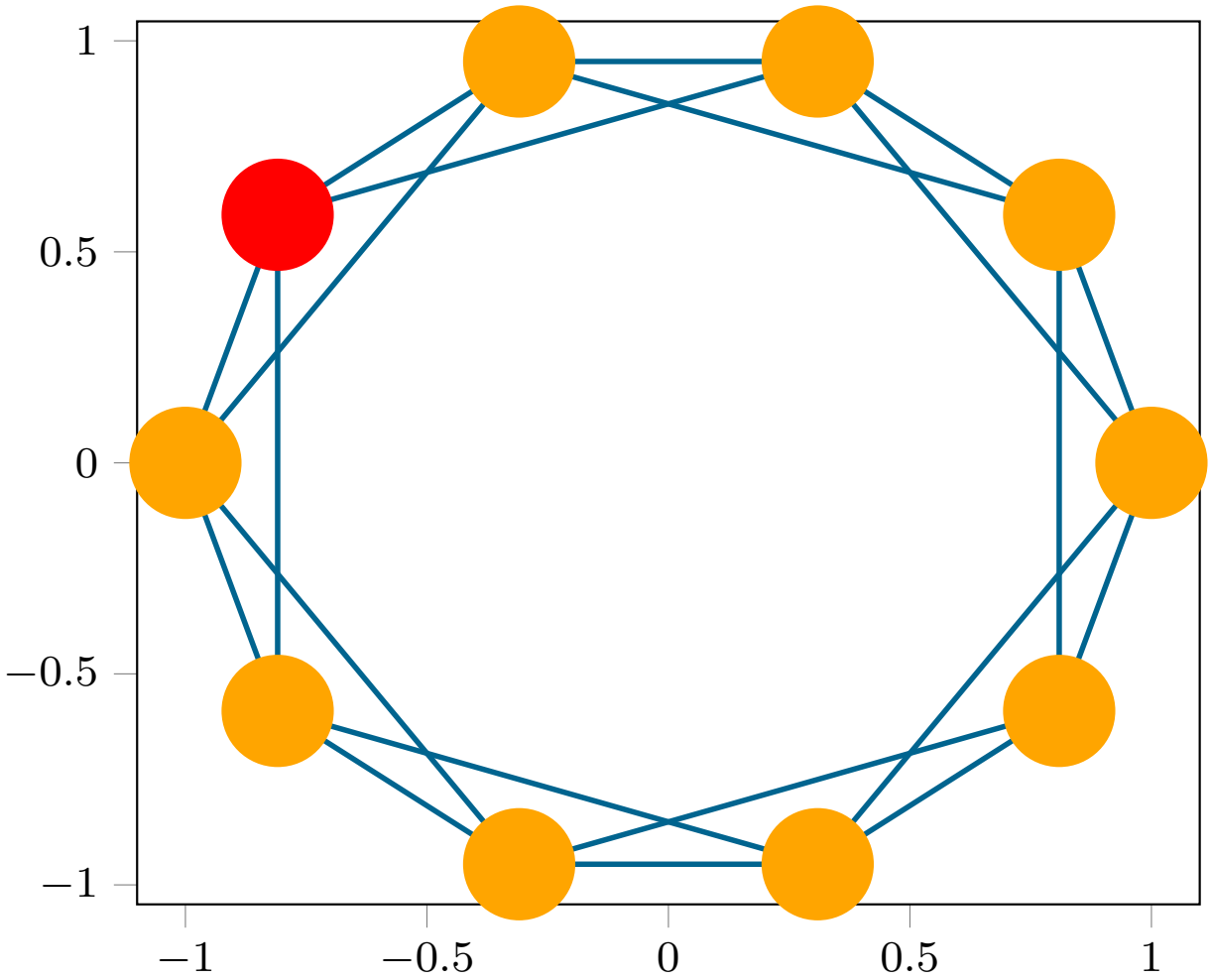
Algorithme de Watts-Strogatz

$$N \in \mathbb{N}, K \in [\![1, \lfloor \frac{N}{2} \rfloor]\!] (N \gg K \gg \ln N), \beta \in [0, 1]$$

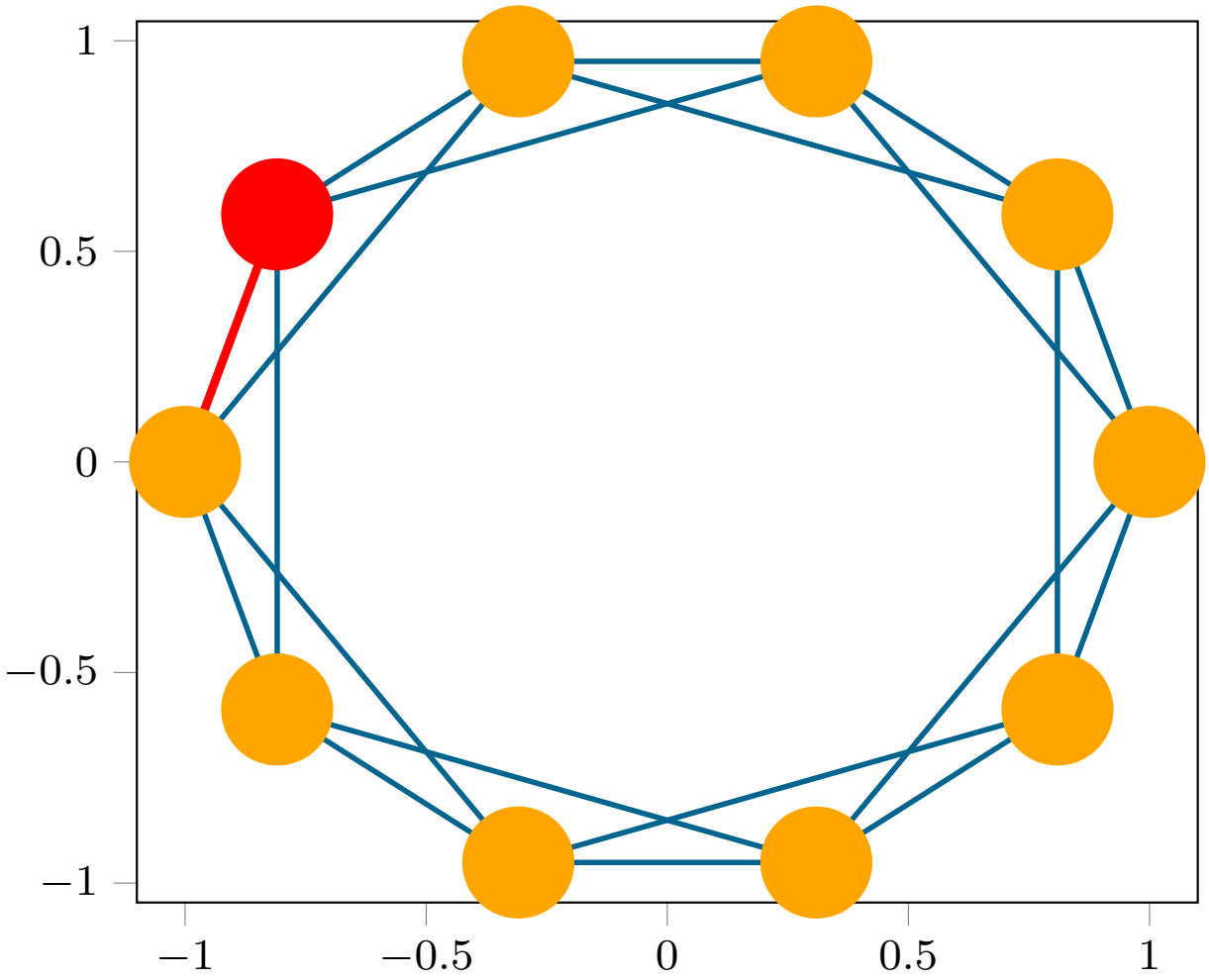
Algorithme de Watts-Strogatz



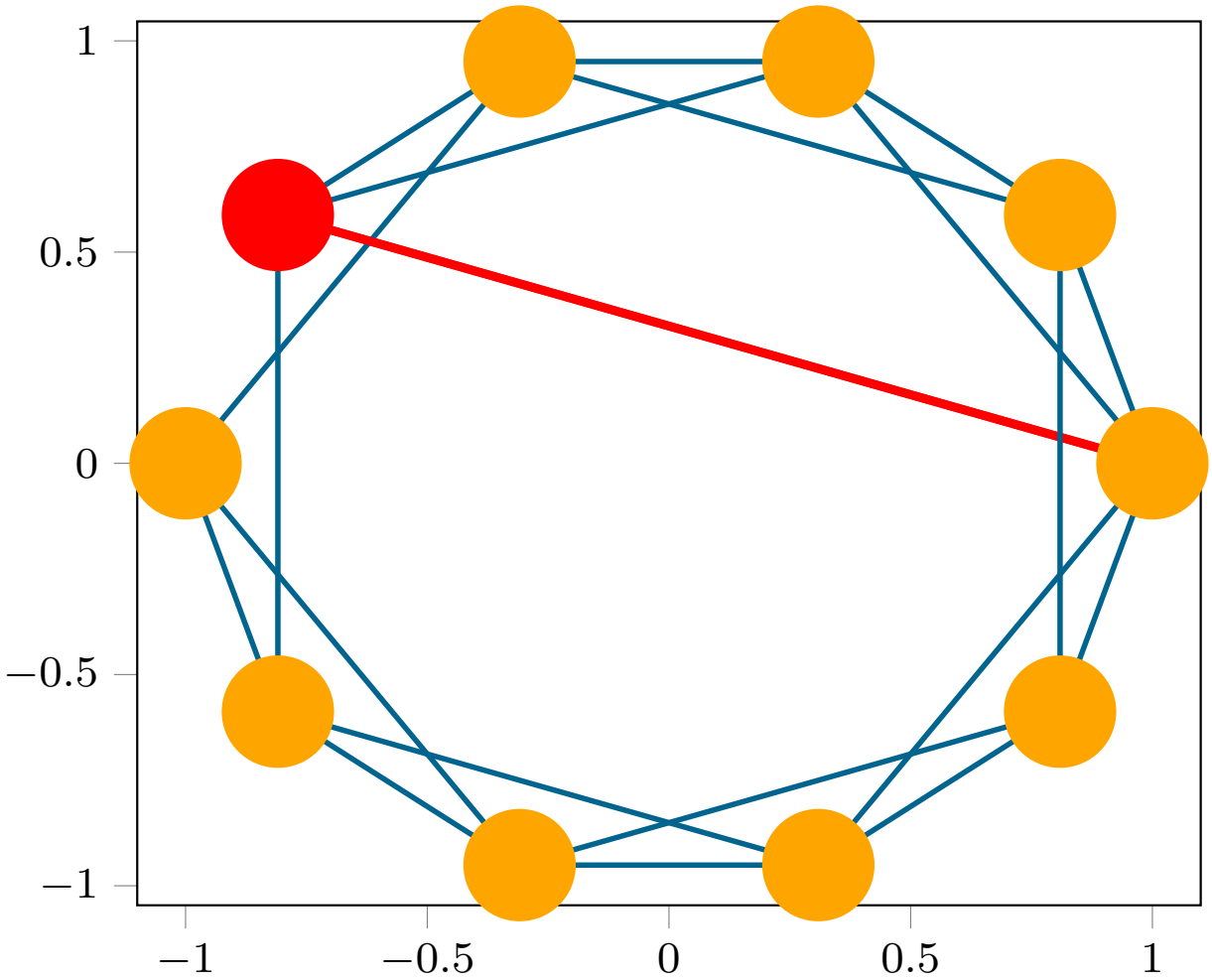
Algorithme de Watts-Strogatz



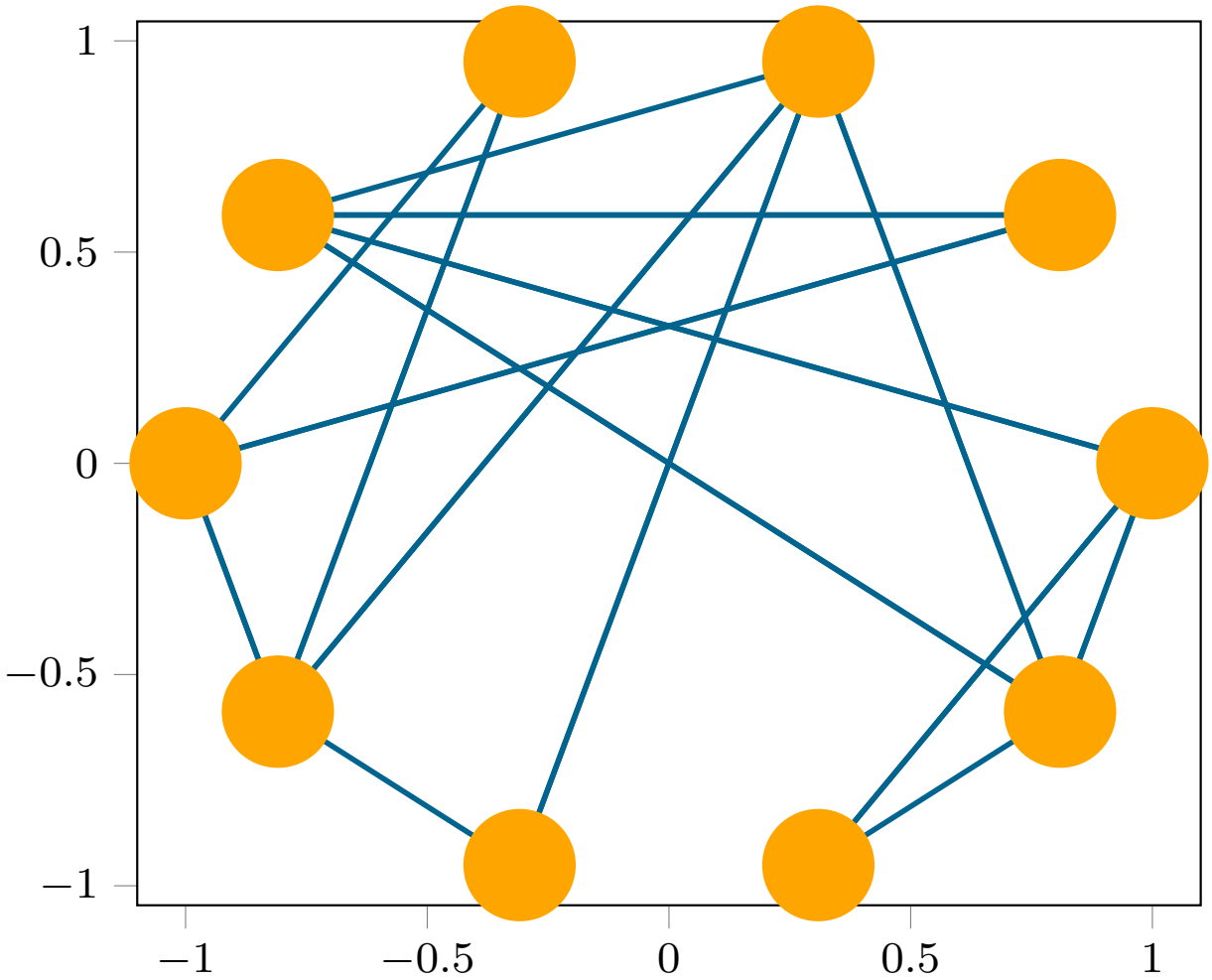
Algorithme de Watts-Strogatz



Algorithme de Watts-Strogatz



Algorithme de Watts-Strogatz



Réseaux simulés

- 500 nœuds ;
- Au plus 500 étapes de simulation ;
- On lance la simulation 100 fois ;
- 3 paramètres à examiner (β, q, proportion initiale d'informés)

$\rightarrow$ Stockage des résultats dans une base de donnée des résultats des calculs afin de pouvoir interrompre l'expérience à tout instant.

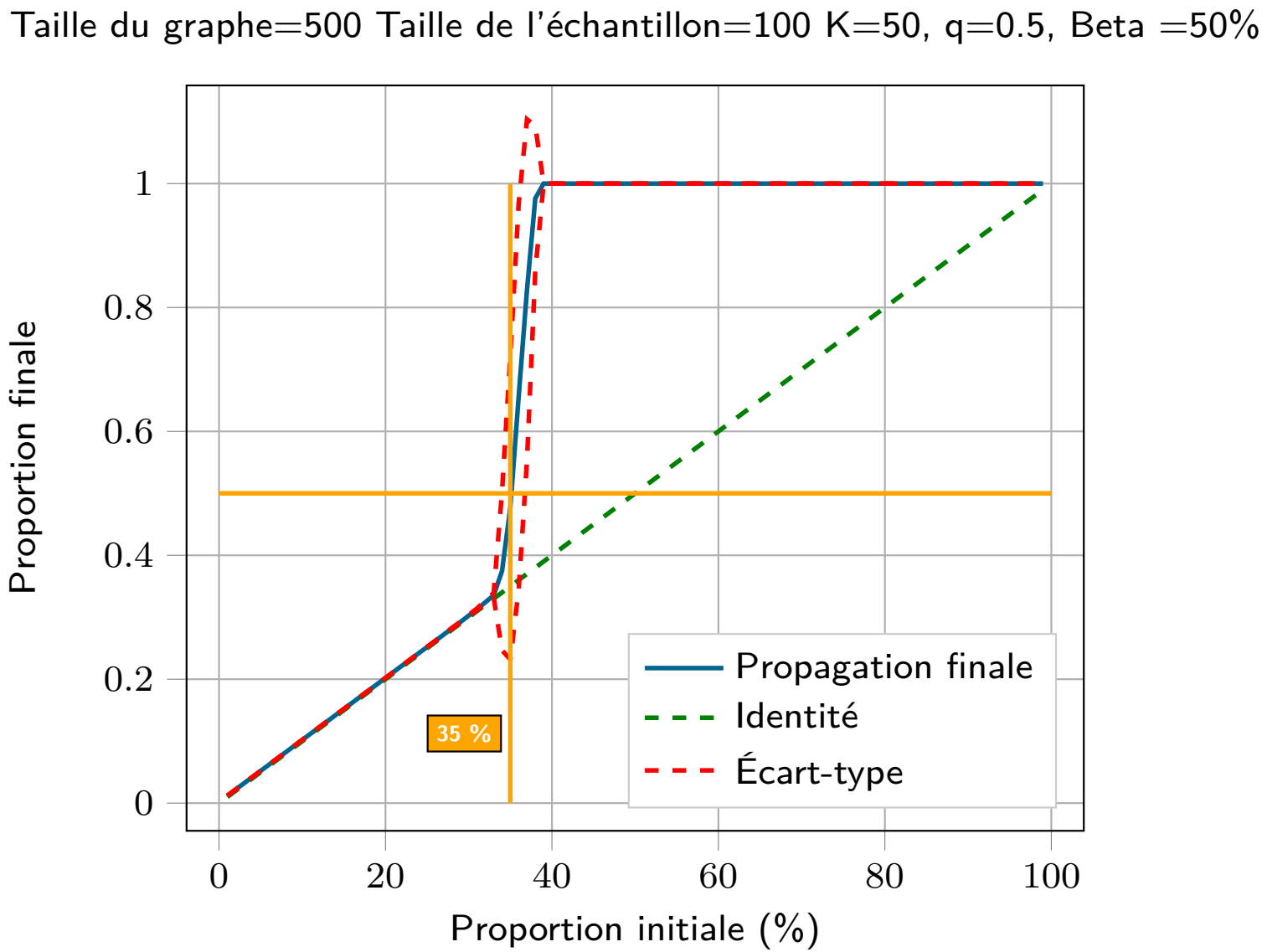
Plan

Protocole expérimental

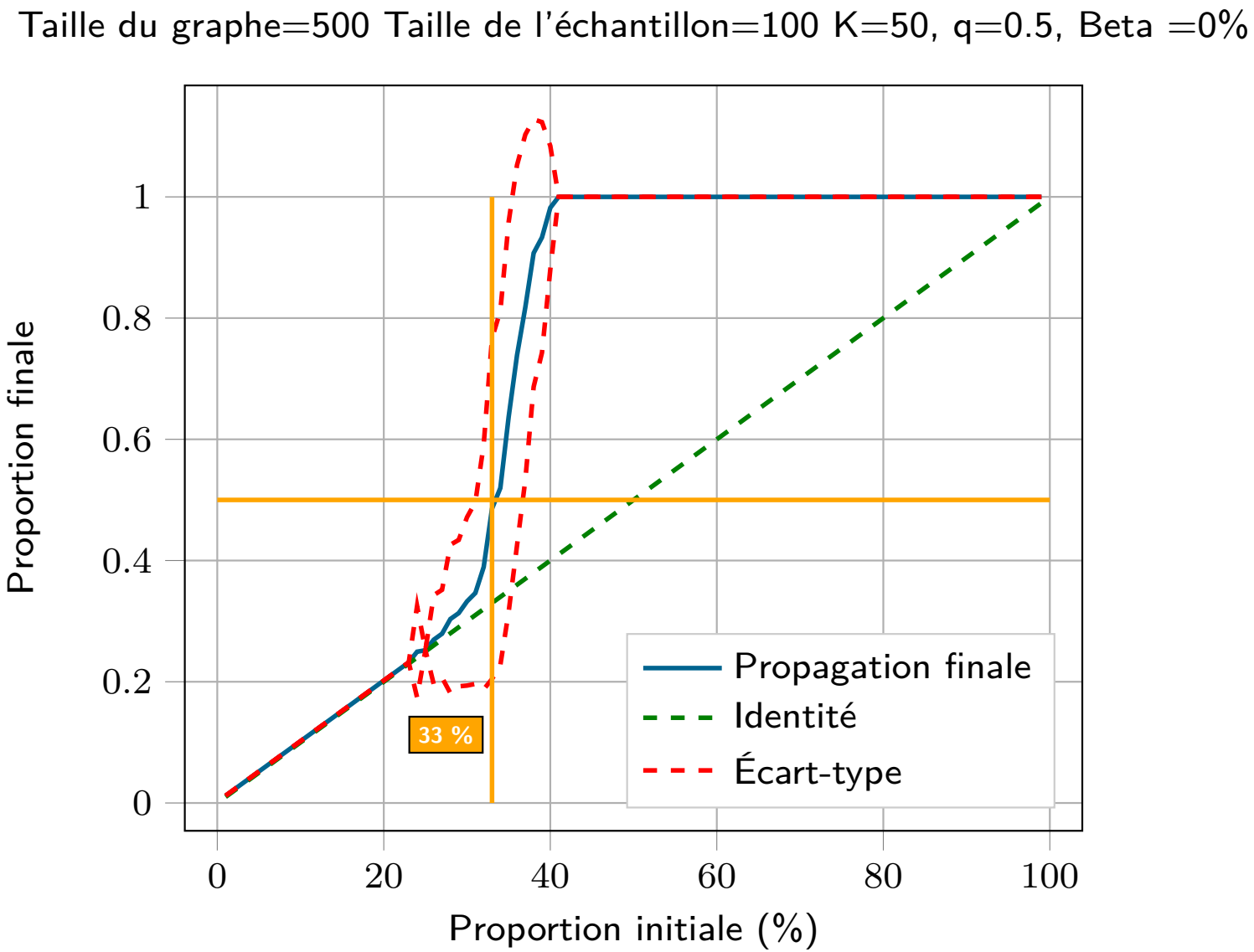
- On fixe $K = 50$;
- $\beta \in \{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$;
- $q \in \{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}\}$;
- proportion initiale de 1% à 99% ;

$\rightarrow$ 100 expériences de propagation en choisissant les éléments initiaux au hasard et stocker la propagation à chaque étape de la simulation. But : pouvoir comparer les résultats des autres expériences, éventuellement fixer certains paramètres qui ont peu d'influence.

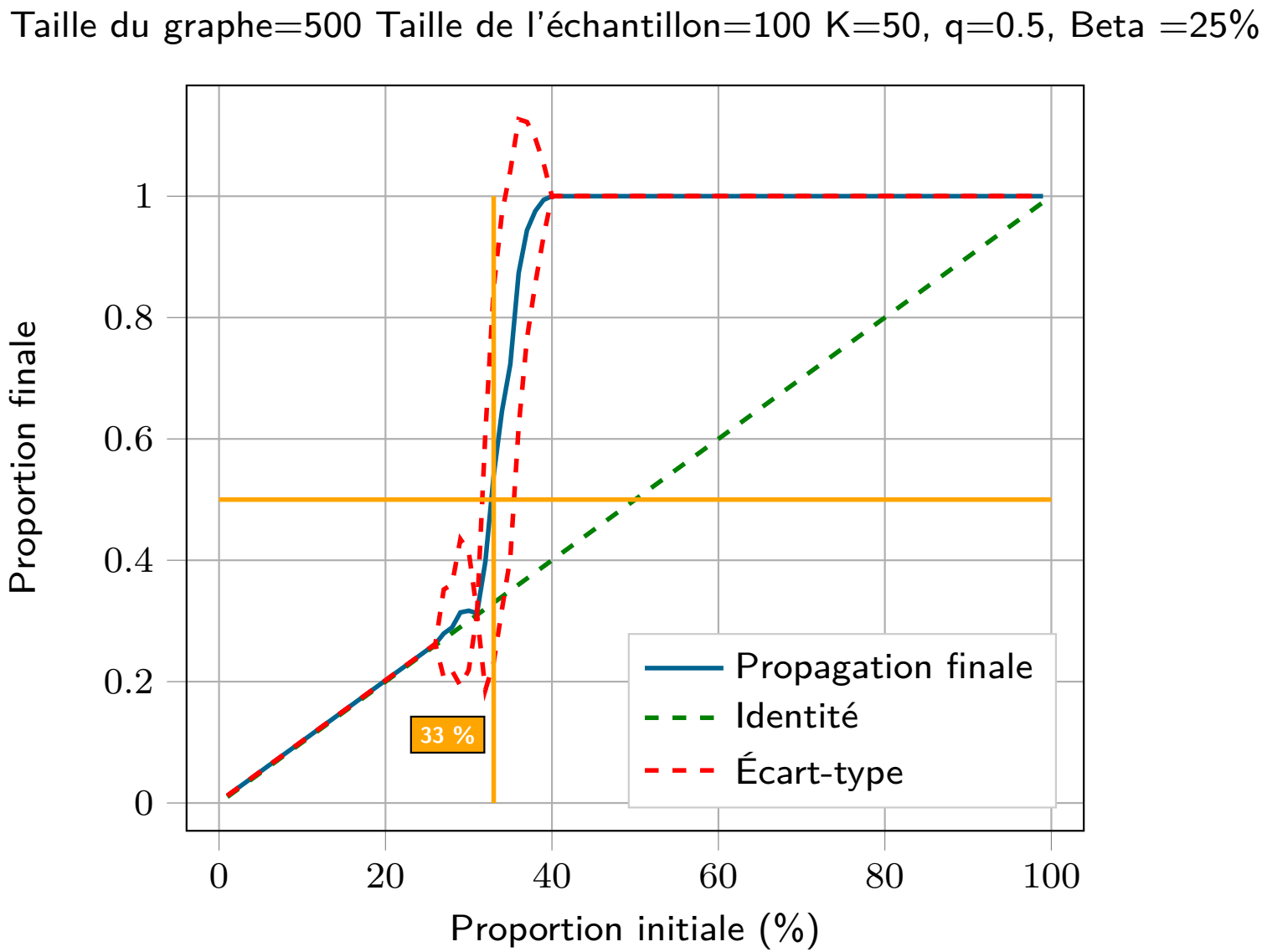
Courbe type


Taille du graphe=500 Taille de l'échantillon=100 K=50, q=0.5, Beta =50%
Proportion finale
1
0.8
0.6
0.4
0.2
0
0
20
40
60
80
100
Proportion initiale (%)
35 %
Propagation finale
Identité
Écart-type

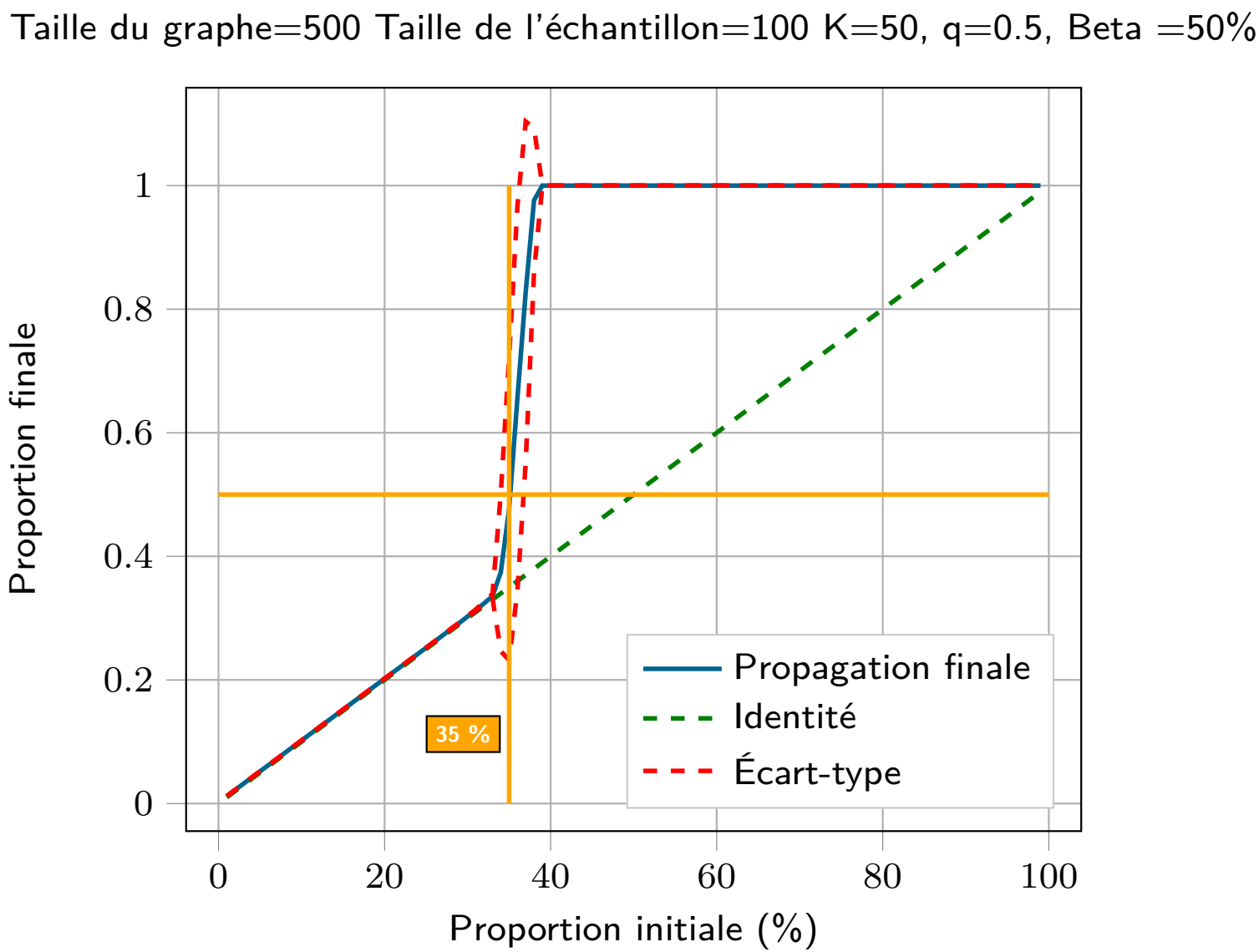
Faible influence du paramètre β



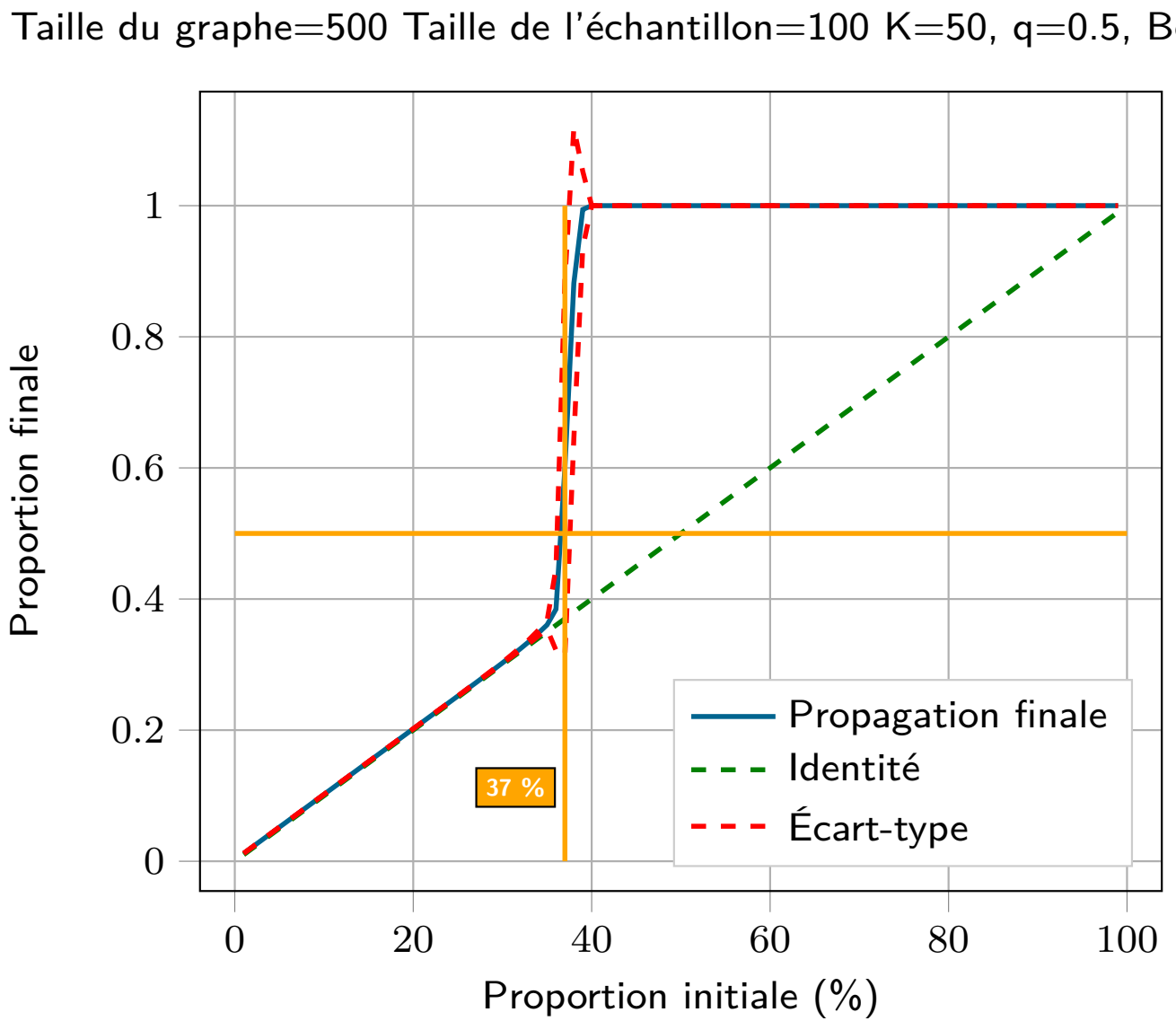
Faible influence du paramètre β



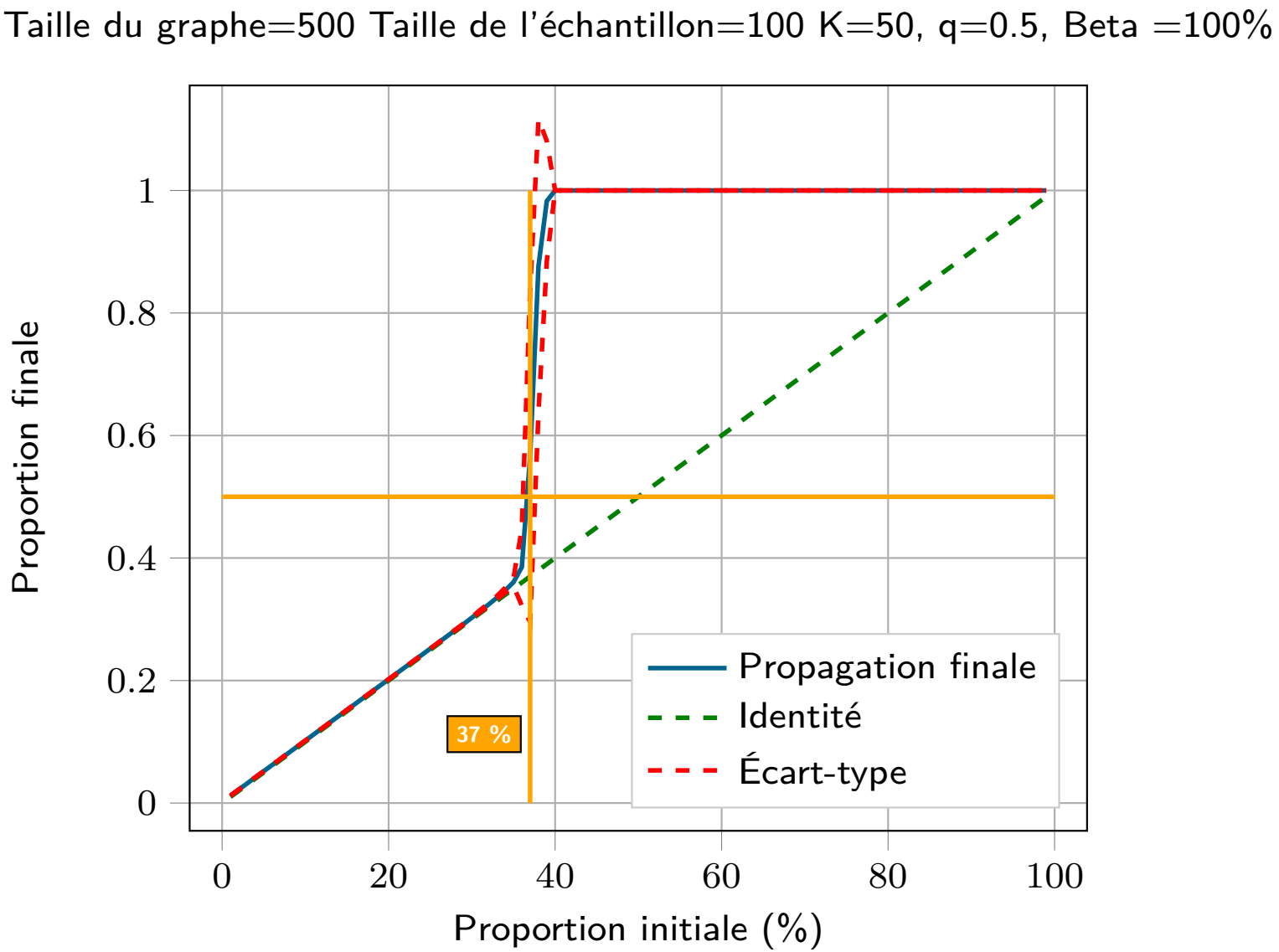
Faible influence du paramètre β



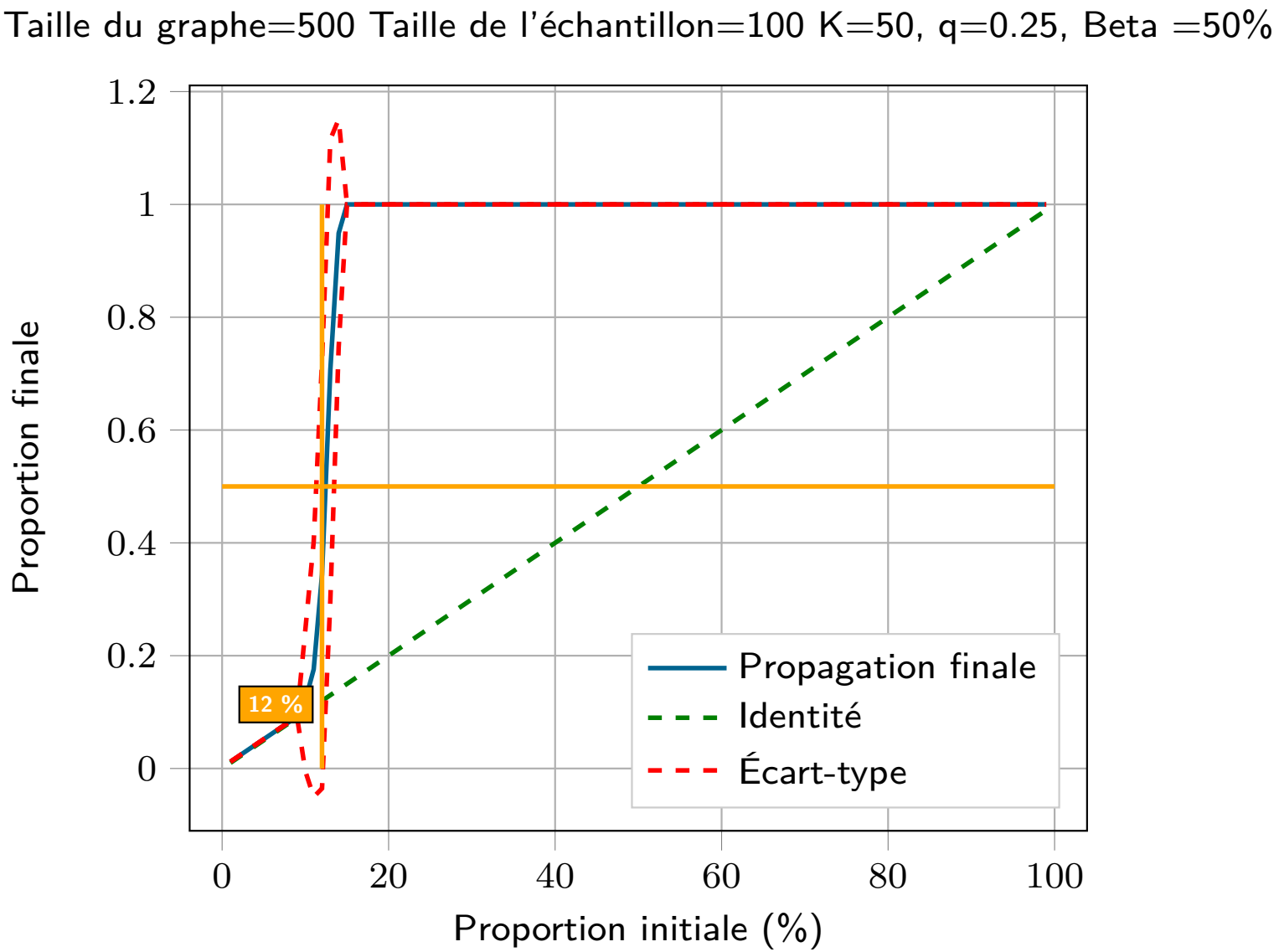
Faible influence du paramètre β



Faible influence du paramètre β

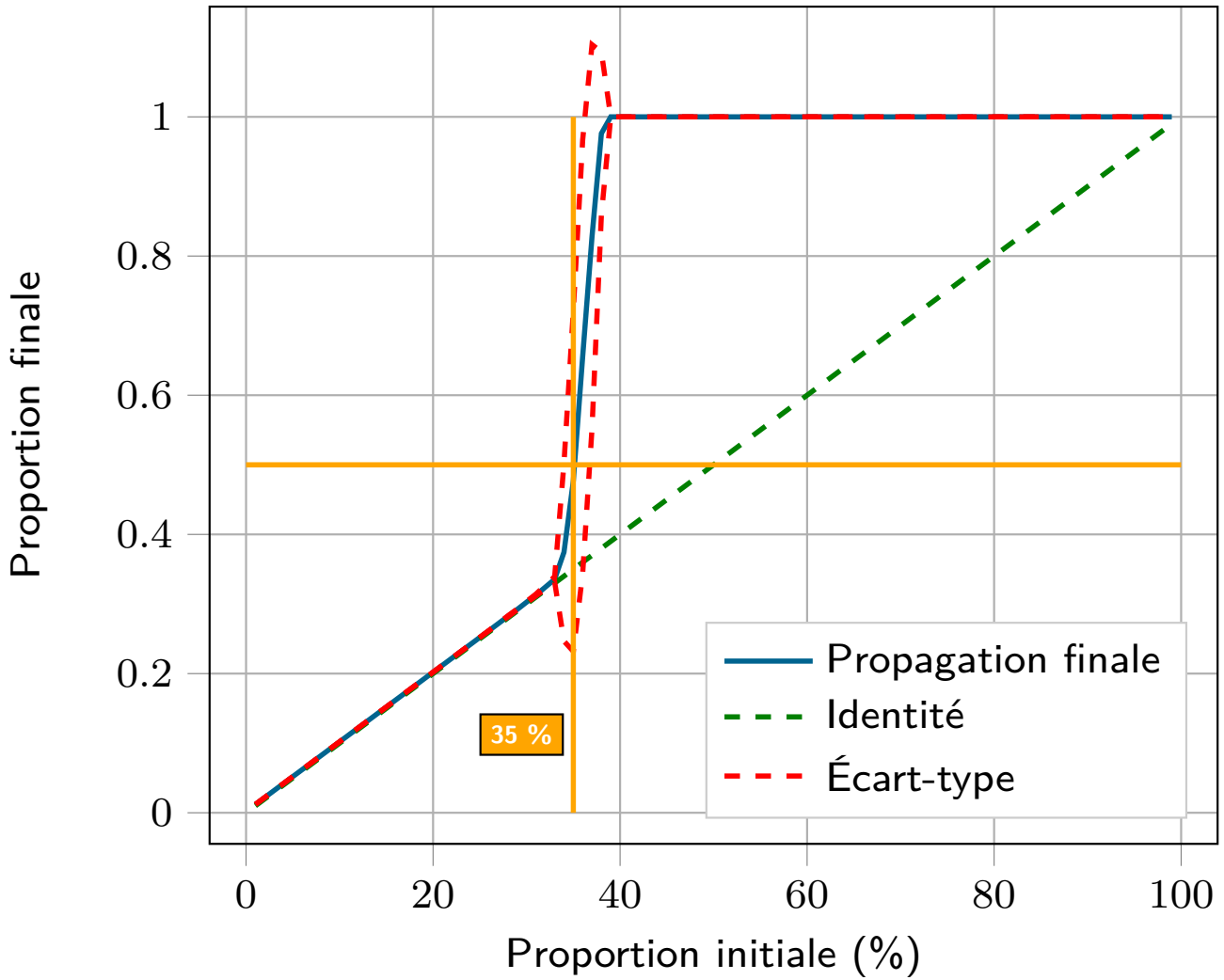


Influence de q



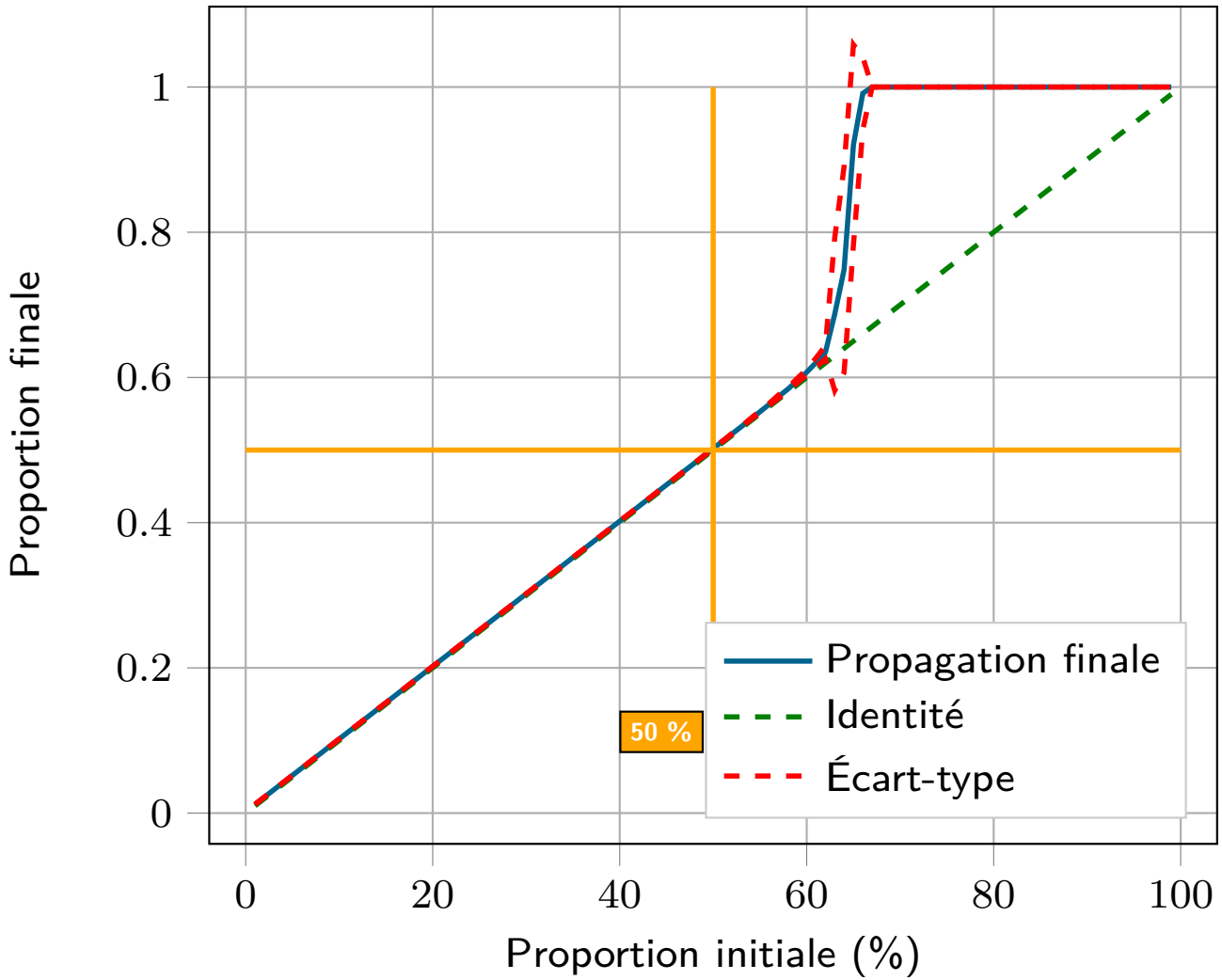
Influence de q

Taille du graphe=500 Taille de l'échantillon=100 K=50, q=0.5, Beta =50%



Influence de q

Taille du graphe=500 Taille de l'échantillon=100 K=50, q=0.75, Beta =50%



Protocole expérimental

- On fixe $K = 50$;
- $\beta \in \{0, \frac{1}{4}, \frac{1}{2}, 1\}$;
- $q \in \{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}\}$;
- proportion initiale de 1% à 99% ;

$\rightarrow$ 100 expériences de propagation en choisissant les éléments initiaux possédant les plus grands degrés et stocker la propagation à chaque étape de la simulation.

Remarques sur les résultats

- Le résultat pour $\beta = 0$ est inexploitable ;
- On retrouve les mêmes effets qualitatifs de β et q.

Protocole expérimental

- On fixe $K = 50$;
- $\beta \in \{0, \frac{1}{4}, \frac{1}{2}, 1\}$;
- $q \in \{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}\}$;
- proportion initiale de 1% à 99% ;

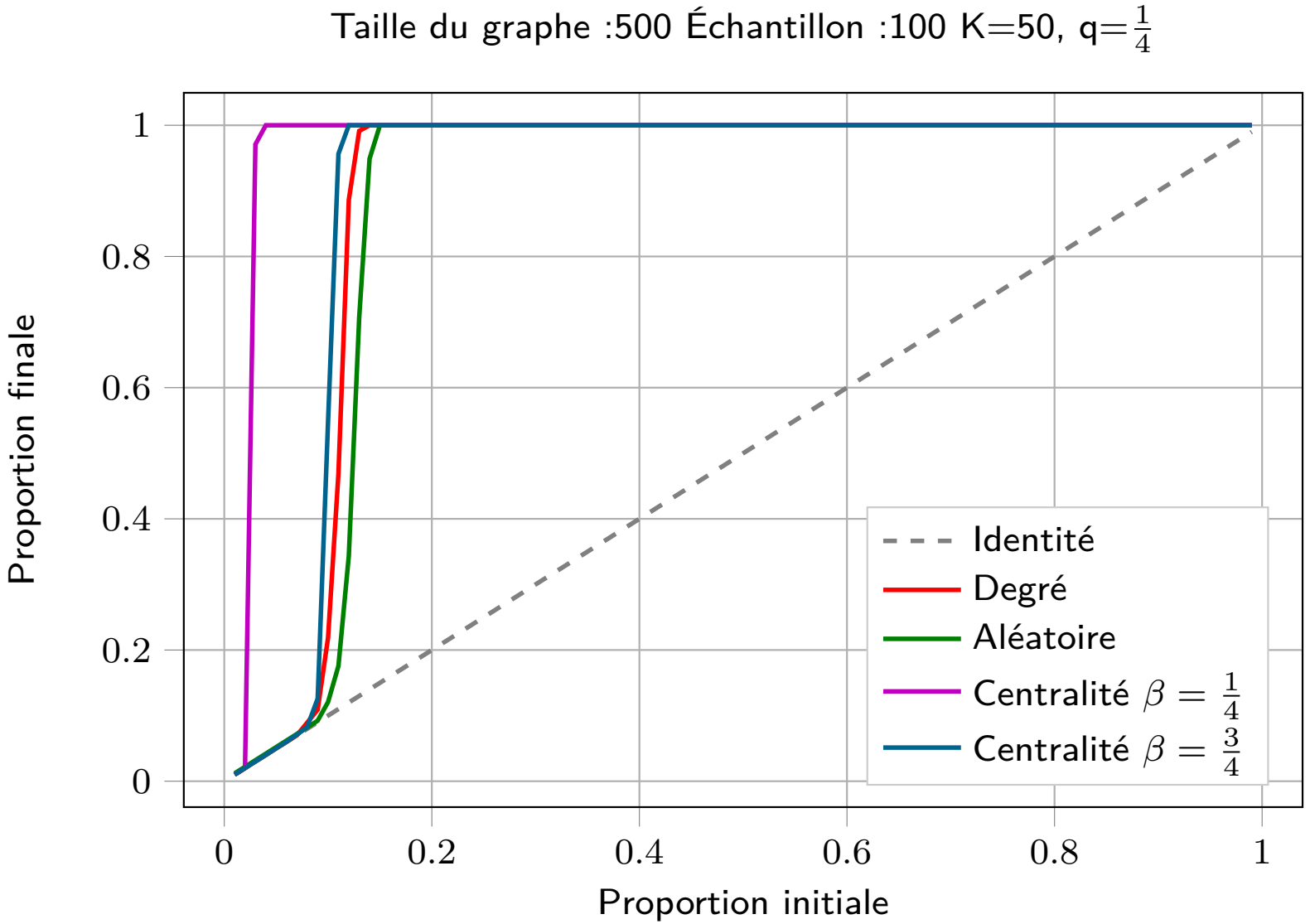
$\rightarrow$ 100 expériences de propagation en choisissant les éléments initiaux possédant possédant les plus grandes centralités (proportion de plus courts chemins passants par un nœud, algorithme de Ulrik Brandes) et stocker la propagation à chaque étape de la simulation.

Remarques sur les résultats

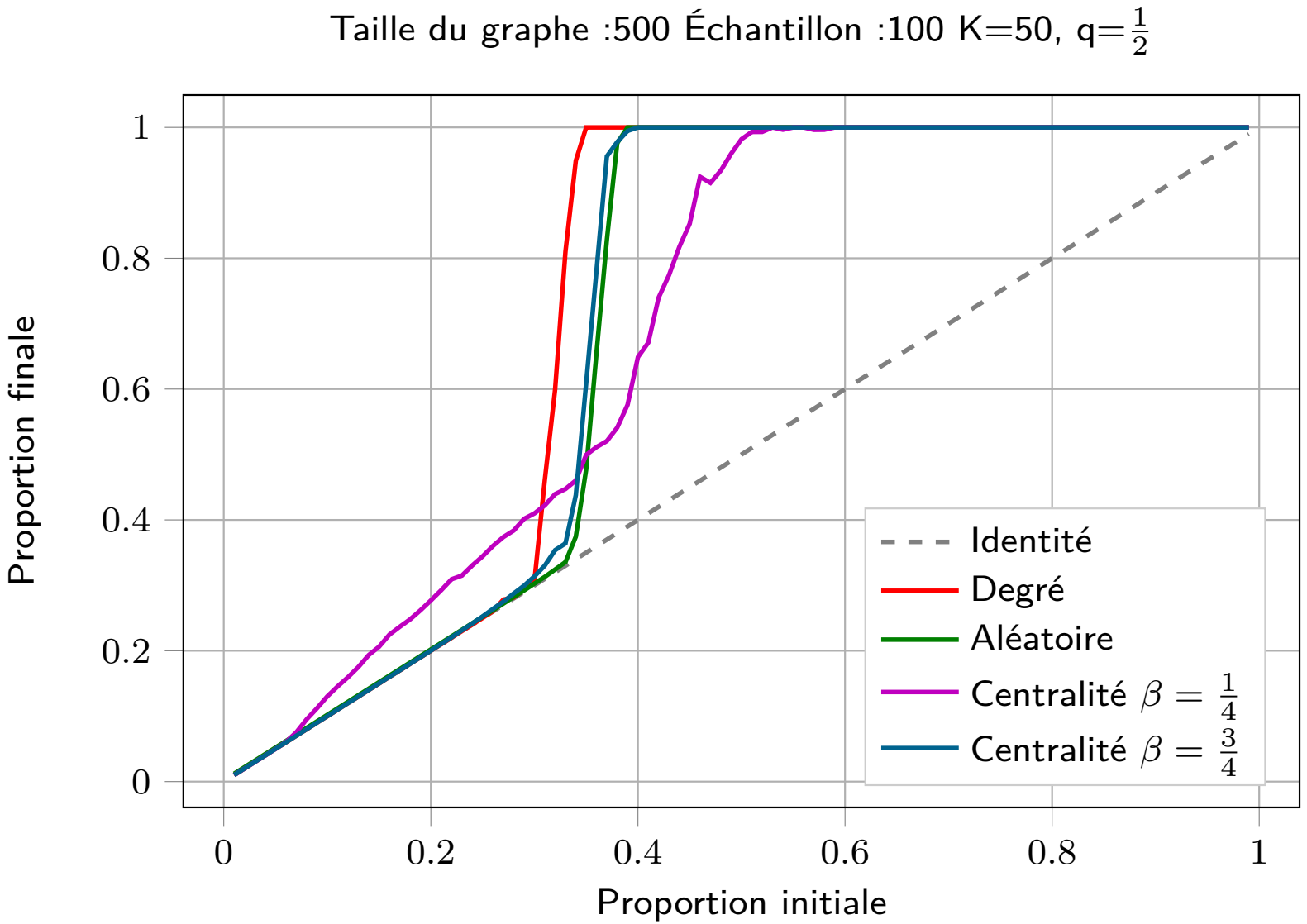
- Le résultat pour $\beta = 0$ est inexploitable ;
- On retrouve les mêmes effets qualitatifs de q.

Plan

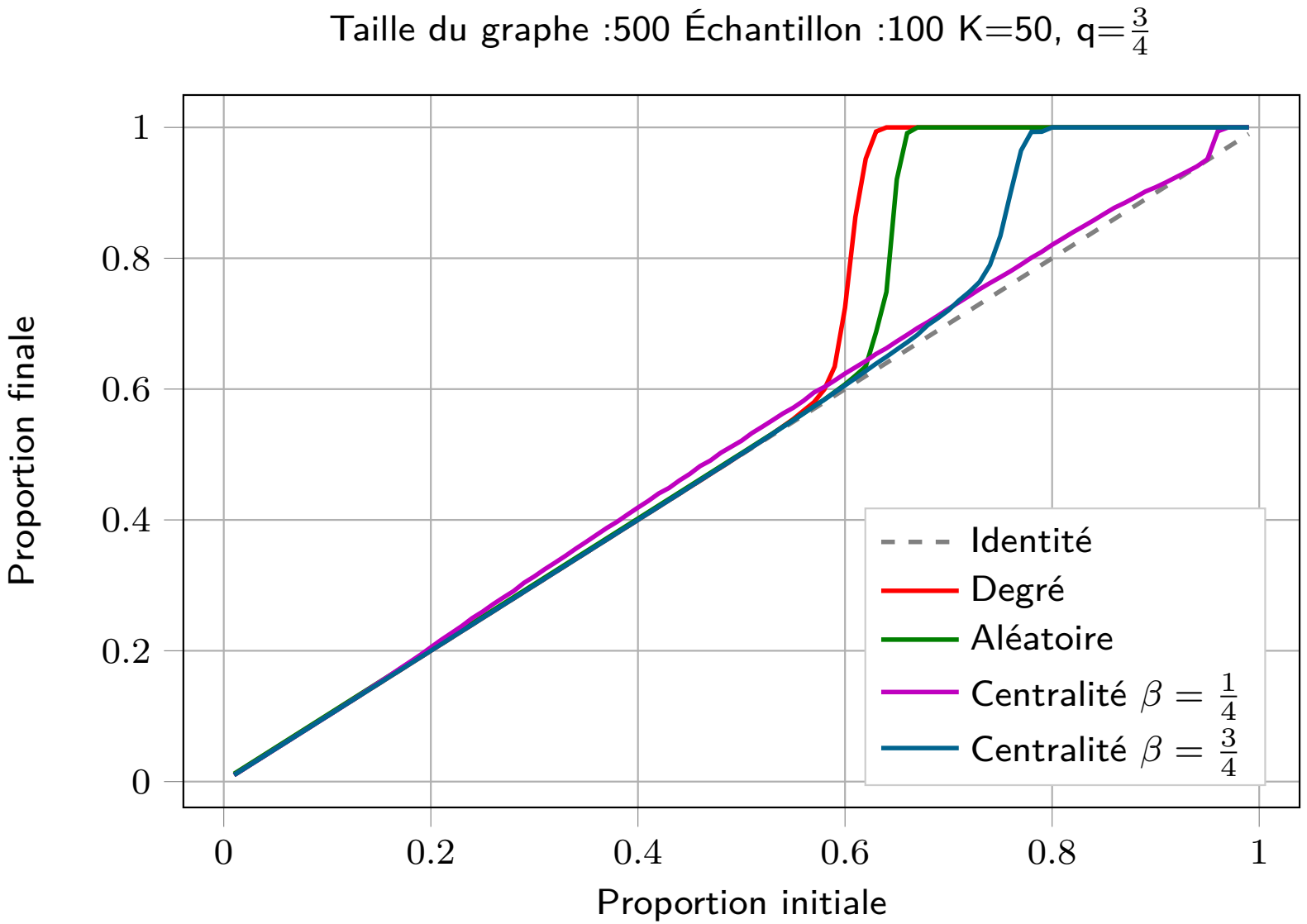
$q = \frac{1}{4}$


Taille du graphe :500 Échantillon :100 K=50, q=1/4
Proportion finale
Proportion initiale
0
0.2
0.4
0.6
0.8
1
Identité
Degré
Aléatoire
Centralité β = 1/4
Centralité β = 3/4

$q = \frac{1}{2}$


Taille du graphe :500 Échantillon :100 K=50, q=1/2
Proportion finale
Proportion initiale
0
0.2
0.4
0.6
0.8
1
Identité
Degré
Aléatoire
Centralité β = 1/4
Centralité β = 3/4

$q = \frac{3}{4}$


Taille du graphe :500 Échantillon :100 K=50, q=3/4
Proportion finale
Proportion initiale
0
0.2
0.4
0.6
0.8
1
Identité
Degré
Aléatoire
Centralité β = 1/4
Centralité β = 3/4

Conséquences

On peut choisir les éléments initialement propagateurs en connaissant q.

Pour $\beta = \frac{1}{4}$,

- Si $q \geq \frac{1}{2} \rightarrow \approx q$ des éléments de plus haut degré ;
- Si $q \leq \frac{1}{4} \rightarrow \approx 5\%$ des éléments de plus grande centralité ;
- Une étude plus quantitative serait nécessaire pour $\frac{1}{4} < q < \frac{1}{2}$.

Plan

Conclusion

- On a un premier critère de choix des éléments initiaux
  - Nécessite d'être affiné
  - Problème : longueur des calculs
- Est-ce le meilleur critère ? (vérification difficile à cause de la longueur des calculs)
- Certains choix de modélisation sont discutables (Non "retour en arrière" de la rumeur)
- La méthode de génération des graphes est également problématique : degré des nœuds, choix de β ?

Plan

Démonstration : Les clusters sont les seuls obstacles aux rumeurs

On pose $n = |V|$, q la note de la rumeur. S'il existe un p-cluster C avec $p > q$, alors tout nœud de C possède au moins une proportion p de voisins non informés. Ceci valant pour tous les nœuds de C, aucun nœud de C ne sera informé au bout de n étapes.

Démonstration : Les clusters sont les seuls obstacles aux rumeurs

On pose $n = |V|$, q la note de la rumeur. S'il existe un nœud i tel qu'au bout de n étapes i ne soit pas dans l'état informé, alors la proportion p de voisins de i dans l'état informé vérifie $p \leq q$ ou encore $(1 - p) > q \leq 0$. Il existe donc des voisins de i vérifiant cette propriété, on a un z-cluster avec $z > q$.

Plan

Algorithme de Watts-Strogatz

Données : $N \in \mathbb{N}, K \in [\![1, \lfloor \frac{N}{2} \rfloor]\!] (N \gg K \gg \ln N), \beta \in [0, 1]$
Résultat : Matrice d'adjacence d'un graphe aléatoire.
$M \leftarrow$ matrice avec pour $i \in [\![0, N-1]\!]$, $j \in [\![1, K]\!]$,
$M_{i,i+j[N]} = M_{i,i-j[N]} =$ Vrai, Faux pour les autres ;
pour $i \in [\![0, N-1]\!]$ **faire**
  pour $j \in [\![1, K]\!]$ **faire**
    $r \leftarrow$ Nombre aléatoire sur $[0, 1]$;
    si $r < \beta$ **alors**
      $M_{i,i+j[N]} \leftarrow$ Faux ;
      $M_{i+j[N],i} \leftarrow$ Faux ;
      Choisir au hasard k tel que $M_{i,k} =$ Faux ;
      $M_{i,k} \leftarrow$ Vrai ;
      $M_{k,i} \leftarrow$ Vrai ;
    fin
  fin
fin
retourner M

Plan

```
open Core.Std;;

(* Builds a random graph with the Watts and Strogatz method.
*)

Random.self_init ();;
let wattsStrogatzMatrix n k beta =
  let l = Array.make_matrix n n false in
  let rec wire i j = if i < 0 then wire (n+i) j
    else if i >= n then wire (i-n) j
    else if j < 0 then wire i (n+j)
    else if j >= n then wire i (j-n)
    else (l.(i).(j) <- true;l.(j).(i) <- true)
  in
  let rec unwire i j = if i < 0 then unwire (n+i) j
    else if i >= n then unwire (i-n) j
    else if j < 0 then unwire i (n+j)
    else if j >= n then unwire i (j-n)
    else (l.(i).(j) <- false;l.(j).(i) <- false)
  in
```

```
let rec wired i j = if i < 0 then wired (n+i) j
  else if i >= n then wired (i-n) j
  else if j < 0 then wired i (n+j)
  else if j >= n then wired i (j-n)
  else l.(i).(j)
in
for i=0 to n-1 do
  for j = i-k/2 to i+k/2 do
    if j != i then wire i j
  done
done;
for i = 0 to n-1 do
  for j = i+1 to (i+k/2) do
    let r = Random.float 1.0 in
    if r < beta then begin
      unwire i j;
      let k = ref (Random.int n) in
      while (wired i !k) || (!k = i) do
        k := Random.int n
      done;
```

```
        wire i !k
      end
    done;
  done;
  l
;;

(* Betweenness centrality of a graph via its adjacency matrix*)
let betweenness g =
  let n = Array.length g in
  let cB = Array.create n (0.0) in
  for s = 0 to n-1 do
    let stack = Stack.create() in
    let p = Array.create n ([]) in
    let sigma = Array.create n ( 0.0) in
    sigma.(s) <- 1.0;
    let d = Array.create n ((-1)) in
    d.(s) <- 0;
    let q = Queue.create() in
    Queue.enqueue q s;
```

```
while not (Queue.is_empty q) do
  let v = Queue.dequeue_exn q in
  Stack.push stack v;
  for w = 0 to (n-1) do
    let iw = g.(v).(w) in
    if iw then
      if d.(w) < 0 then begin
        Queue.enqueue q w;
        d.(w) <- d.(v) + 1;
      end;
      if d.(w) = (d.(v) + 1) then begin
        sigma.(w) <- sigma.(w) +. sigma.(v);
        p.(w) <- v::p.(w);
      end;
  done;
done;
let delta = Array.create n (0.0) in
while not (Stack.is_empty stack) do
  let w = Stack.pop_exn stack in
  List.iter ~f:(fun v -> delta.(v) <-
```

```
          delta.(v) +. sigma.(v) /. sigma.(w) *. (1.0 +. delta.(w))) p.(w)
        if w != s then begin cB.(w) <- cB.(w) +. delta.(w); end;
      done;
    done;
    cB
;;

let degree g i =
  let n = Array.length g in
  let r = ref 0 in
  for j = 0 to (n-1) do
     if g.(i).(j) then incr r
  done;
  !r
;;

let maxDegree g n =
  let deg = degree g in
  let size = Array.length g in
  let rec loop i r = if i >= size then r else
```

```
    loop (i+1) ((deg i, i)::r)
  in
  SortFirst.sortFirst n (loop 0 [])
;;

let maxBetweenness g n =
  let a = betweenness g in
  let size = Array.length g in
  let rec loop i r = if i >= size then r else
    loop (i+1) ((a.(i), i)::r)
  in
  SortFirst.sortFirst n (loop 0 [])
;;
```

Plan

```
(* Structure de la table experiments:
CREATE TABLE "experiments"
("name" TEXT,"last_id" INT,"infos" TEXT DEFAULT (null))
*)

let silent = true;;


let print_return m r = if silent then () else
  Printf.printf ("%s %s") m (Sqlite3.Rc.to_string r)
;;

let load_db () =
  Sqlite3.db_open "experiments.sqlite"
;;

let close_db db =
  Sqlite3.db_close db
;;
```

```
let cleaner target row =
  let l = Array.length row in
  for i = 0 to l-1 do
    target := Some(row.(i))
  done;
;;

let get_exp_last_id db name =
  let last_id = ref None in
  let get_last_id () =
    print_return "Recherche du dernier identifiant."
    (Sqlite3.exec_not_null_no_headers db ~cb:(cleaner last_id)
      ("SELECT last_id FROM experiments WHERE name=\""^name^"\";"))
  in
  get_last_id ();
  ! last_id
;;

let get_experiment db name =
  let last_id = get_exp_last_id db name in
```

```
  let last_result = ref None in
  let create_table () =
    print_return ("Création de la table " ^ name)
    (Sqlite3.exec db ("CREATE TABLE "^name^" (id INT, value TEXT);"));
    print_return ("Enregistrement dans 'experiments' de la table "
    ^ name) (Sqlite3.exec db
    ("INSERT INTO experiments VALUES (\""^name^"\", 0, \"\");"
    ))
  in
  let get_last_result id =
    print_return "Recherche du dernier résultat."
    (Sqlite3.exec_not_null_no_headers db ~cb:(cleaner last_result)
      ("SELECT value FROM "^name^" WHERE id=\""^id^"\";"))
  in
  let get_last_step () = match last_id with
  | None -> create_table (); !last_result
  | Some s -> get_last_result s; !last_result
  in
  get_last_step ()
;;
```

```
let add_step_id db exp id str =
  print_return (Printf.sprintf ("Ajout de l'étape %d à %s") id exp)
  (Sqlite3.exec db
    ("INSERT INTO "^exp
      ^" VALUES ("^(string_of_int id)^",\""^str^"\");"))
;;

let change_last_id db exp id =
  print_return (Printf.sprintf
    ("Mise à jour du dernier identifiant (%d) de %s") id exp)
  (Sqlite3.exec db
    ("UPDATE experiments SET last_id = "
      ^(string_of_int id)^" WHERE name=\""^exp^"\";"))
;;
let add_step db exp str = let id =
  match get_exp_last_id db exp with
  | None -> 0
  | Some(s) ->  (1 + (int_of_string s))
  in
```

Plan

```
open Core.Std;;

let sortFirst n l =
  let rec sep p l left right len_l = match l with
  | [] -> left, right, len_l
  | (a,b)::tl when a > p -> sep p tl ((a,b)::left) right (len_l + 1)
  | (a,b)::tl -> sep p tl left ((a,b)::right) len_l
  in
  let rec loop n l = match l with
  | [] -> []
  | (a,b)::tl ->
    let left, right, len_l = sep a tl [] [] 0 in
    if len_l >= n then (loop n left)
    else
    (loop n left)@[(a,b)]@(loop (n-len_l-1) right)
  in
  List.map (loop n l) ~f:(fun (a,b) -> b)
;;
```

Plan

```
open Core.Std;;


(*
  Spread the rumor and return the amount of nodes aware of it.
*)
let step_p graph a b s =
  let n = Array.length graph in
  let p_lim = b /. (a +. b) in
  let nb = ref 0 in
  for i = 0 to n-1 do
    if not s.(i) then begin
      let aware = ref 0 in
      let d = ref 0 in
      for j = 0 to n-1 do
        if graph.(i).(j) then(
          incr d;
          if s.(j) then incr aware;
        )
      done;
```

```
      let p = (float_of_int !aware) /. (float_of_int !d) in
      if p > p_lim then (s.(i) <- true; incr nb)
    end
    else incr nb
  done;
  !nb
;;
```

Plan

```
open Core.Std;;
open Yojson.Basic.Util;;


type exp_stat = {
  graph_no:int;
  prop_spread:float array;
};;


let json_of_exp_stat e =
  `Assoc [
  ("graph_no", `Int e.graph_no);
  ("prop_spread",
    `List (List.map (Array.to_list e.prop_spread)
      ~f:(fun x -> `Float x)))
];;

let exp_stat_of_json json =
  {
```

```
    graph_no = json |> member "graph_no" |> to_int;
    prop_spread = json |> member "prop_spread" |> to_list
    |> List.map ~f:(fun x -> x |> to_float) |> Array.of_list
  }
;;

let escape_double_quotes s =
  let exp = Str.regexp "\"" in
  Str.global_replace exp "\"\"" s
;;

let save_step db e exp_name=
  json_of_exp_stat e
  |> Yojson.Basic.to_string
  |> escape_double_quotes
  |> Experiment.add_step_id db exp_name e.graph_no;
  Experiment.change_last_id db exp_name e.graph_no
;;

let process ?(max=99) db graph_size nb_gen
```

```
k beta max_spread_step a b choose_spread name=
  let step i init=
    let g = Graph.wattsStrogatzMatrix graph_size k beta in
    let prop_spread = Array.create max_spread_step 0.0 in
    let spread = choose_spread g init in
    let j = ref 0 in
    let p = ref (-1.0) in
    while !j <= (max_spread_step-1) && prop_spread.(!j) != !p do
      prop_spread.(!j) <-
      (float_of_int (Spread.step_p g a b spread))
        /. (float_of_int graph_size);
      if !j > 0 then (p := prop_spread.(!j-1));
      incr j;
    done;
    {graph_no=i; prop_spread=prop_spread}
  in
  let experiment init db=
    let exp_name = Printf.sprintf
      ("r_spreading_%s_%d_%d_%d_%d_%d_%d_%d_%d")
      name graph_size (int_of_float (beta*.100.0))
```

```
      k nb_gen init max_spread_step (int_of_float a)
        (int_of_float b) in
    print_endline ("Nom de l'expérience : "^exp_name);
    let cur_step = ref (match Experiment.get_experiment db exp_name
    with
    | None -> {graph_no= (-1);prop_spread=[||]}
    | Some(s) -> s |> Yojson.Basic.from_string |> exp_stat_of_json
    ) in
    let beg = !cur_step.graph_no + 1 in
    for i = beg to nb_gen - 1 do
      cur_step := step i ((float_of_int init)/.100.0);
      save_step db !cur_step exp_name
    done;
  in
  for i = 1 to max do
    experiment i db;
  done
;;
```

Plan

```
open Core.Std;;


let graph_size = 500;;
let nb_gen = 100;;
let max_spread_step = 500;;

let choose_spread _ init =
  let s = Array.create graph_size false in
  let k = ref 0 in
  let n = int_of_float ((float_of_int graph_size) *. init) in
  while !k <= n do
    let i = Random.int graph_size in
    if not s.(i) then (incr k; s.(i) <- true)
  done;
  s
;;

let () =
  print_endline "Initialisation de Random.";
```

```
Random.self_init ();
print_endline "Ouverture de la base de données.";
let db = Experiment.load_db () in
for b = 0 to 4 do
  ExperimentSpreading.process db graph_size nb_gen 50
    ((float_of_int b) *. 0.25) max_spread_step 1.0 1.0
    choose_spread "random";
  ExperimentSpreading.process db graph_size nb_gen 50
    ((float_of_int b) *. 0.25) max_spread_step 3.0 1.0
    choose_spread "random";
  ExperimentSpreading.process db graph_size nb_gen 50
    ((float_of_int b) *. 0.25) max_spread_step 1.0 3.0
    choose_spread "random";
done;
print_endline "Fermeture de la base de données.";
if (Experiment.close_db db) then
  print_endline "Fermeture réussie."
else
  print_endline "Echec de la fermeture."
```

Plan

```
import matplotlib.pyplot as pl
from matplotlib2tikz import save as tikz_save
import numpy as np
import sqlite3
import json


graph_size = 500
nb_gen = 100
k = 50
max_spread_step = 500
conn = sqlite3.connect('experiments.sqlite')
for a,b in [(1,1), (1,3), (3,1)]:
  for beta in [0,25,50,75,100]:
    X = np.arange(1,100)
    pl.close('all')
    fig = pl.figure()
    ax = fig.add_subplot(111)
    ax.set_title('Taille du graphe={} Taille de l\'échantillon={}'
      +' K={}, q={}, Beta ={}\\%'
```

```
  .format(graph_size,nb_gen,k,b/(a+b),beta))
ax.set_ylabel("Proportion finale")
ax.set_xlabel("Proportion initiale (\\%)")


Y = np.zeros(100)
Y_high = np.zeros(100)
Y_low = np.zeros(100)


x_50 = 0

for x in X:
  cursor = conn.execute(
    "SELECT value FROM r_spreading_random_"+
    "{graph_size}_{beta}_{k}_{nb_gen}_{x}_{max_spread_step}_{a}_{b}"
    .format(**locals()))
  rows = cursor.fetchall()
  v = 0
  n = len(rows)
```

```
    for row in rows:
      v += json.loads(row[0])['prop_spread'][-1]
    Y[x] = v/n
    if abs(0.5-Y[x]) < abs(Y[x_50]-0.5):
      x_50 = x
    v = 0
    for row in rows:
      v += (Y[x] - json.loads(row[0])['prop_spread'][-1])**2
    v = (v/n)**(1/2)
    Y_high[x] = Y[x] + v
    Y_low[x] = Y[x] - v

pl.grid()

pl.plot(X,Y[1:], 'b', label="Propagation finale", lw=2.5)
pl.plot(X,X/100, 'g--', label="Identité", lw=2.5)
pl.plot(X, Y_high[1:], 'r--', label="Écart-type", lw=2.5)
pl.plot(X, Y_low[1:], 'r--', lw=2.5)
pl.axvline(x_50, c='orange', lw=2.5)
pl.axhline(0.5, xmin=0, xmax=100, c='orange', lw=2.5)
```

```
    pl.text(x_50-10, 0.1, str(x_50)+" \\%",
      bbox=dict(facecolor='orange', alpha=0.95),
      size="large",
      color="white",
      fontweight="bold")

    pl.legend(loc="lower right")
    tikz_save("resultats/random_finale_f_initiale_q{}_Beta{}_ec.tex"
     .format(int(b/(a+b)*100), beta))
conn.close()
```