A la chasse aux bugs: la vérification des programmes

Collège de France, 22 février 2008

Cours : Gérard Berry

Chaire d'innovation technologique Liliane Bettencourt

Gerard.Berry@college-de-france.fr

Séminaires : Patrick Cousot (ENS),
Gilles Dowek (X & INRIA)

Patrick.Cousot@ens.fr

Gilles.Dowek@polytechnique.org

Réservez votre vendredi 23 mai!

Colloque informatique et bio-informatique au Collège de France

Martin Abadi (U.C. Santa Cruz and Microsoft Research)

Sécurité informatique et résistance aux attaques et intrusions

Alberto Sangiovanni-Vincentelli (UC Berkeley & Parades Roma)

Embedded Everywhere: the Physical Web

Philippe Kourilsky (Collège de France)

Le système immunitaire, un grand système d'information

Alexandre Pouget (Collège de France et U. Rochester)

L'approche calculatoire des neurosciences

François Fages (INRIA)

Machines abstraites, vérification formelle et biochimie cellulaire

Que veut dire vérifier ?

- Assurer qu'un circuit ou programme
  1. fait ce qu'il doit faire
  2. ne fait pas ce qu'il ne doit pas faire
- Toujours relativement à un critère d'observation
  – une spécification complète de la fonctionnalité
  – ou un jeu d'objectifs à atteindre
  – ou un jeu de propriétés à respecter
- En supposant des propriétés de l'environnement

  exemple : pour aller du 1er au 3eme, l'ascenseur passe toujours par le 2eme

Trois techniques fondamentales

1. Le test

   boîte noire, grise, ou blanche

2. La preuve totale

   logiques diverses

   avec ou sans extraction du programme

3. La preuve partielle

   interprétation abstraite

   model-checking

Trois techniques fondamentales

1. Le test

   boîte noire, grise, ou blanche

2. La preuve totale

   logiques diverses

   avec ou sans extraction du programme

3. La preuve partielle

   interprétation abstraite

   model-checking

Toutes outillées et utilisées industriellement

Le test

- Passer des scénarios de test
  - construits manuellement
  - fabriqués automatiquement (aléatoires)
- Vérifier les résultats
  - manuellement
  - automatiquement
  - à l'aide d'observateurs et assertions
- Effectuer des tests de non-régression
  - rejouer automatiquement les tests archivés

Cf. démonstration

Programmes = données = nombres (Turing, Gödel)

- Programme = texte = nombre

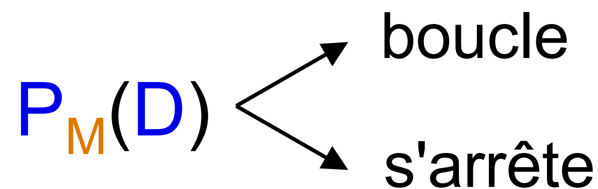
  codes numériques : ASCII = 256, ISO = 65536

- Donnée = texte = nombre

- $P_M(D)$: appliquer le programme P à une donnée D avec une machine M

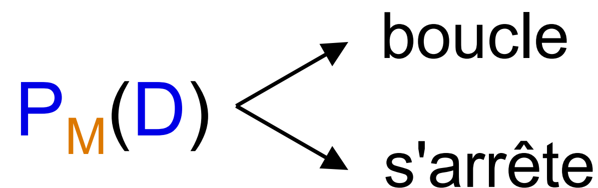
Programmes = données = nombres (Turing, Gödel)

- Programme = texte = nombre
  codes numériques : ASCII = 256, ISO = 65536
- Donnée = texte = nombre
- $P_M(D)$: appliquer le programme P à une donnée D
  avec une machine M



Programmes = données = nombres (Turing, Gödel)

- Programme = texte = nombre
  codes numériques : ASCII = 256, ISO = 65536
- Donnée = texte = nombre
- $P_M(D)$: appliquer le programme P à une donnée D
  avec une machine M



Thèse de Church : toutes les machines M suffisamment puissantes se valent

Peut-on vérifier automatiquement par un programme H qu'un programme P va s'arrêter pour la donnée D?

Peut-on vérifier automatiquement par un programme H qu'un programme P va s'arrêter pour la donnée D?

$$\text{Supposons } H(P,D) = \begin{cases} 1 \text{ si } P \text{ s'arrête sur } D \\ 0 \text{ sinon} \end{cases}$$

Peut-on vérifier automatiquement par un programme H qu'un programme P va s'arrêter pour la donnée D?

Supposons $H(P,D) = \begin{cases} 1 \text{ si } P \text{ s'arrête sur } D \\ 0 \text{ sinon} \end{cases}$

Définissons $M(P) = \text{if } H(P,P) = 1 \text{ then } M(P) \text{ else } 0$

Peut-on vérifier automatiquement par un programme H qu'un programme P va s'arrêter pour la donnée D?

Supposons H(P,D) = | 1 si P s'arrête sur D
| 0 sinon

Définissons M(P) = if H(P,P) = 1 then M(P) else 0

1. Si M(M) s'arrête, alors H(M,M) = 1, donc M(M) boucle !
2. Si M(M) boucle, alors H(M,M) = 0, donc M(M) s'arrête !

Peut-on vérifier automatiquement par un programme H qu'un programme P va s'arrêter pour la donnée D?

Supposons H(P,D) = | 1 si P s'arrête sur D
| 0 sinon

Définissons M(P) = if H(P,P) = 1 then M(P) else 0

1. Si M(M) s'arrête, alors H(M,M) = 1, donc M(M) boucle !
2. Si M(M) boucle, alors H(M,M) = 0, donc M(M) s'arrête !

Théorème (Turing) : H n'existe pas, l'arrêt est indécidable

Corollaire : beaucoup d'autres problèmes le sont aussi...

A essayer!

```
let rec f n = if n = 1 then 1
                          else if pair(n) then f (n/2)
                                              else f (3*n+1)
```

7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

Conjecture : s'arrête pout tout n

Conjecture de Goldbach (1743)

Tout nombre pair plus grand que 3
est la somme de deux nombres premiers

```
i = 4;
Next : loop
   for j < i, k < i do
      if premier(j) and premier(k) and i = j + k then
         continue Next  // passer au i suivant
      end if
   end for;
   break Next // arrêt si contre-exemple
end loop
```

Vraie si et seulement si ce programme boucle!

Vérification totale

On sait spécifier ce que doit faire le programme indépendamment de son écriture

- Tri (L) : définition formelle du tri

permutation π : bijection de [0, n] sur lui-même

1 2 3 4 5 $\rightarrow$ 3 5 2 1 4

Définition : L' est le tri de L de longueur n ssi

$$\forall\, i < n-1.\ L'[i] \leq L'[i+1]$$

$$\exists \pi.\ \forall\, i \leq n.\ L'[\pi(i)] = L[i]$$

- Fact (n) : définition de la fonction factorielle

Tri : preuve par récurrence

12 17 10 23 33 77 83 11 39 45 14 18 15 31 91 24

Tri : preuve par récurrence

12 17 10 23 33 77 83 11 39 45 14 18 15 31 91 24

Tri : preuve par récurrence

12 17 10 23 33 77 83 11 39 45 14 18 15 31 91 24

Tri : preuve par récurrence

12 17 10 23 33 77 83 11 39 45 14 18 15 31 91 24

---

12 17 10 23 33 77 83 11

39 45 14 18 15 31 91 24

Tri : preuve par récurrence

12 17 10 23 33 77 83 11 39 45 14 18 15 31 91 24

---

12 17 10 23 33 77 83 11 → 10 11 12 17 23 33 77 83

39 45 14 18 15 31 91 24 → 14 15 18 24 31 39 45 91

Tri : preuve par récurrence

12 17 10 23 33 77 83 11 39 45 14 18 15 31 91 24

---

12 17 10 23 33 77 83 11 → 10 11 12 17 23 33 77 83

39 45 14 18 15 31 91 24 → 14 15 18 24 31 39 45 91

Tri : preuve par récurrence

12 17 10 23 33 77 83 11 39 45 14 18 15 31 91 24

---

12 17 10 23 33 77 83 11 → ~~10~~ 11 12 17 23 33 77 83

39 45 14 18 15 31 91 24 → 14 15 18 24 31 39 45 91

10

Tri : preuve par récurrence

12 17 10 23 33 77 83 11 39 45 14 18 15 31 91 24

---

12 17 10 23 33 77 83 11 → ~~10~~ ~~11~~ 12 17 23 33 77 83

39 45 14 18 15 31 91 24 → 14 15 18 24 31 39 45 91

---

10 11

Tri : preuve par récurrence

12 17 10 23 33 77 83 11 39 45 14 18 15 31 91 24

---

12 17 10 23 33 77 83 11 ⟶ ~~10~~ ~~11~~ ~~12~~ 17 23 33 77 83

39 45 14 18 15 31 91 24 ⟶ 14 15 18 24 31 39 45 91

---

10 11 12

Tri : preuve par récurrence

12 17 10 23 33 77 83 11 39 45 14 18 15 31 91 24

---

12 17 10 23 33 77 83 11 → ~~10~~ ~~11~~ ~~12~~ 17 23 33 77 83

39 45 14 18 15 31 91 24 → ~~14~~ 15 18 24 31 39 45 91

---

10 11 12 14

Tri : preuve par récurrence

12 17 10 23 33 77 83 11 39 45 14 18 15 31 91 24

---

12 17 10 23 33 77 83 11 ⟶ ~~10~~ ~~11~~ ~~12~~ ~~17~~ ~~23~~ ~~33~~ ~~77~~ ~~83~~

39 45 14 18 15 31 91 24 ⟶ ~~14~~ ~~15~~ ~~18~~ ~~24~~ ~~31~~ ~~39~~ ~~45~~ ~~91~~

---

10 11 12 14 15 17 18 23 24 31 33 39 45 77 83 91

Tri : preuve par récurrence

12 17 10 23 33 77 83 11 39 45 14 18 15 31 91 24

---

12 17 10 23 33 77 83 11 → ~~10~~ ~~11~~ ~~12~~ ~~17~~ ~~23~~ ~~33~~ ~~77~~ ~~83~~

39 45 14 18 15 31 91 24 → ~~14~~ ~~15~~ ~~18~~ ~~24~~ ~~31~~ ~~39~~ ~~45~~ ~~91~~

---

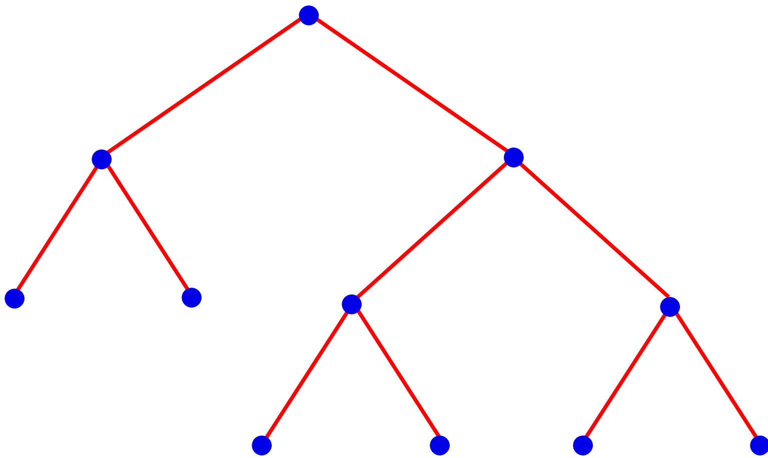
10 11 12 14 15 17 18 23 24 31 33 39 45 77 83 91

Récurrence 1 : les deux sous-listes sont triées

Récurrence 2 : le mélange de deux listes triées est trié

Une mauvaise hypothèse de récurrence

Théorème : un arbre binaire à n sommets a au plus n arêtes

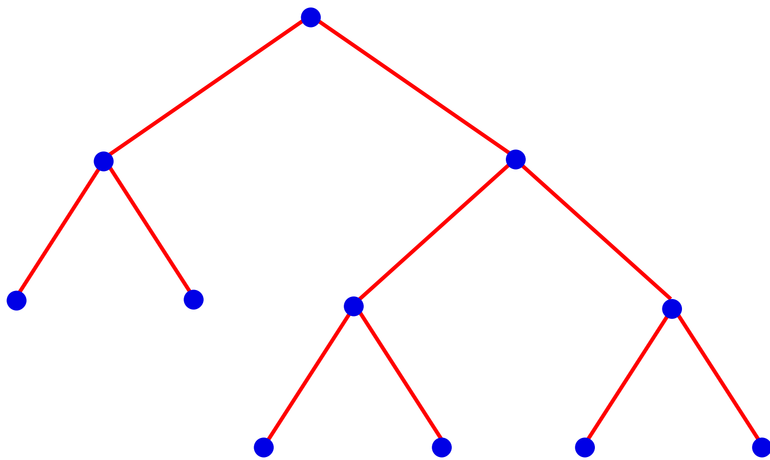


11 sommets

10 arêtes

Une mauvaise hypothèse de récurrence

Théorème : un arbre binaire à n sommets a au plus n arêtes



cas de base

1 sommet

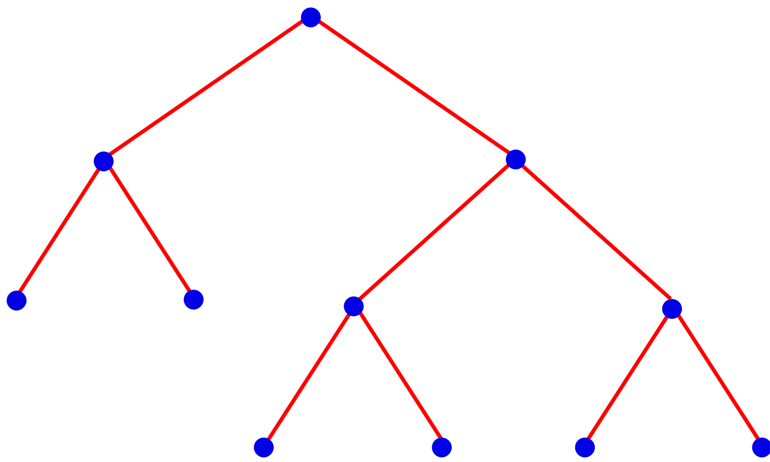
0 = 1-1 arêtes

11 sommets

10 arêtes

Une mauvaise hypothèse de récurrence

<u>Théorème</u> : un arbre binaire à n sommets a au plus n arêtes



11 sommets

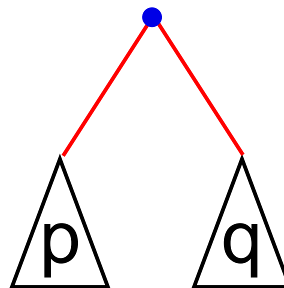
10 arêtes

cas de base

1 sommet

0 = 1-1 arêtes

récurrence

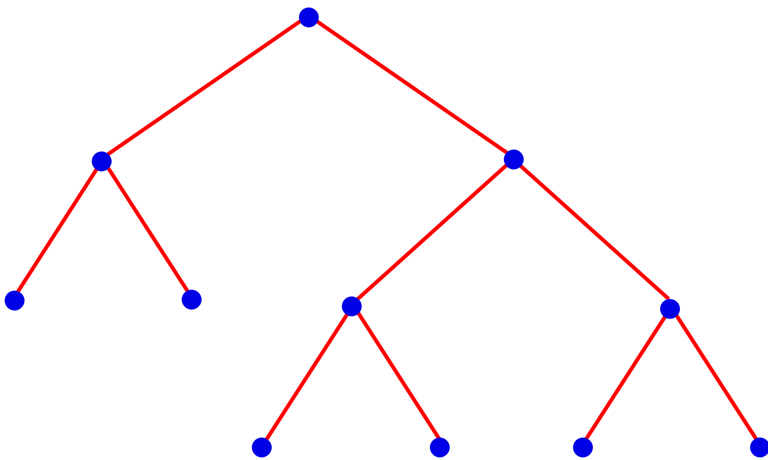


p+q+1 sommets

p+q+2 arêtes

Une mauvaise hypothèse de récurrence

Théorème : un arbre binaire à n sommets a au plus n arêtes



11 sommets

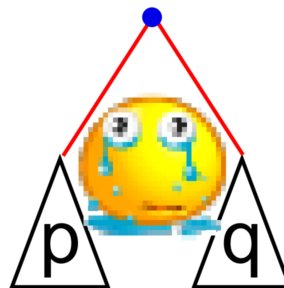
10 arêtes

cas de base

1 sommet

0 = 1-1 arêtes

récurrence

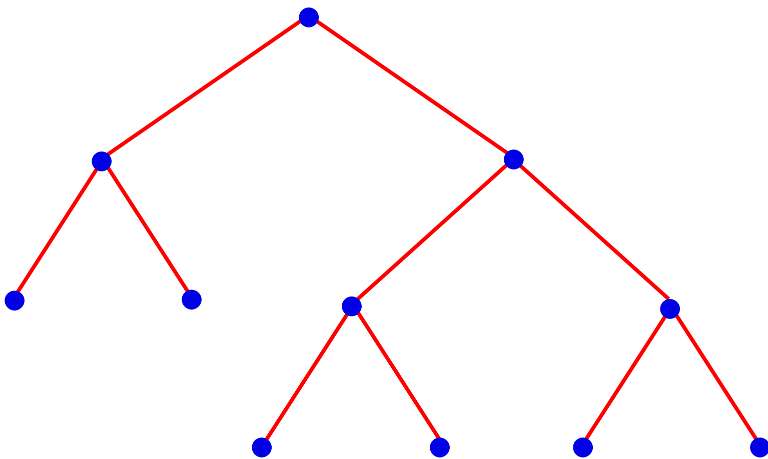


p+q+1 sommets

p+q+2 arêtes

Une bonne hypothèse de récurrence

Théorème : un arbre binaire à n sommets a exactement n-1 arêtes



cas de base

1 sommet

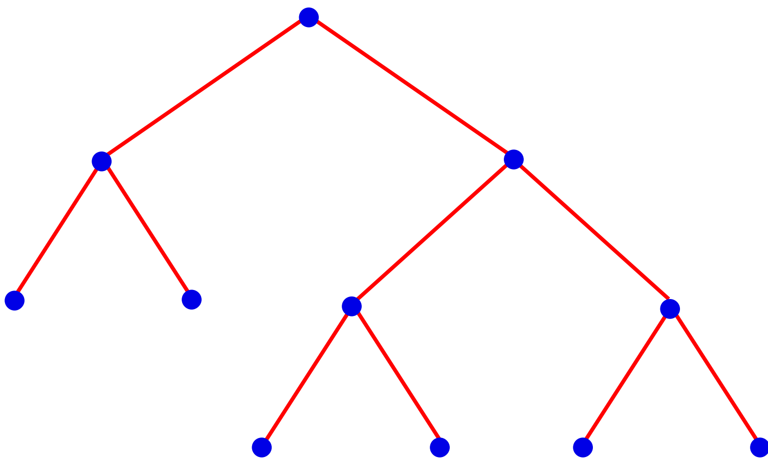
0 = 1-1 arêtes

11 sommets

10 arêtes

Une bonne hypothèse de récurrence

Théorème : un arbre binaire à n sommets a exactement n-1 arêtes



11 sommets

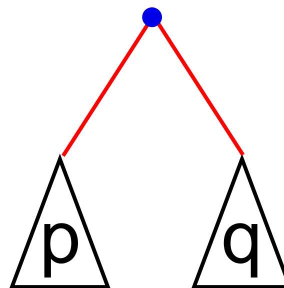
10 arêtes

cas de base

1 sommet

0 = 1-1 arêtes

récurrence



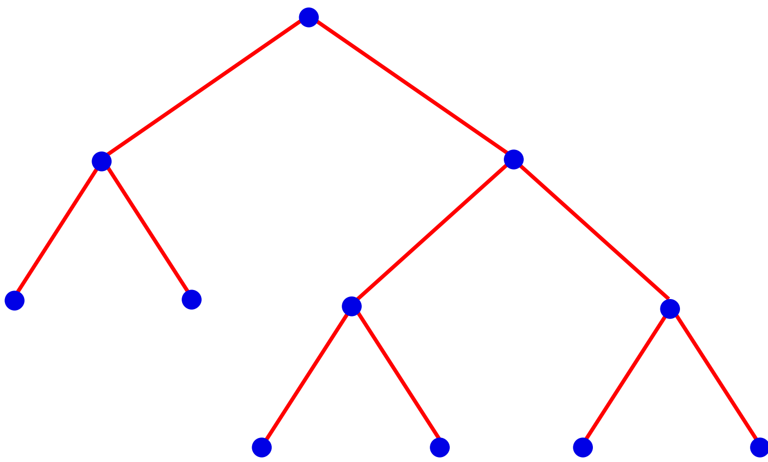
p+q+1 sommets

(p-1)+(q-1)+2 = p+q

= (p+q+1)-1 arêtes

Une bonne hypothèse de récurrence

Théorème : un arbre binaire à n sommets a exactement n-1 arêtes



11 sommets

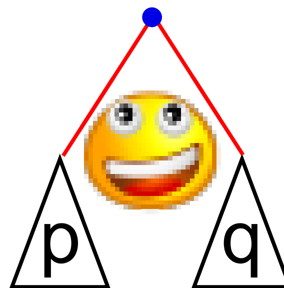
10 arêtes

cas de base

1 sommet

0 = 1-1 arêtes

récurrence



p+q+1 sommets

(p-1)+(q-1)+2 = p+q

= (p+q+1)-1 arêtes

Modèle impératif

```
int fact (int n) {
   int r = 1, i;
   for (i = 2; i <= n; i++) {
      r = r*i;
   }
   return r;
}
```

fact(4) : n ← 4; r ← 1;
i ← 2; r ← 1*2 = 2;
i ← 3; r ← 2*3 = 6;
i ← 4; r ← 6*4 = 24;
i ← 5;
return 24;

Modèle impératif

```
int fact (int n) {
   int r = 1, i;
   for (i = 2; i <= n; i++) {
      r = r*i;
   }
   return r;
}
```

fact(4) : n ← 4; r ← 1;
i ← 2; r ← 1*2 = 2;
i ← 3; r ← 2*3 = 6;
i ← 4; r ← 6*4 = 24;
i ← 5;
return 24;

Comment prouver que le résultat est le bon ?

Logique de Floyd-Hoare

```
int fact (int n) {

   int r = 1, i = 1;

   for (i = 2; i <= n; i++) {

      r = r*i;

   }

   return r;
}
```

Logique de Floyd-Hoare

```
int fact (int n) {
  { n ≥ 1 }  // hypothèse
  int r = 1, i = 1;

  for (i = 2; i <= n; i++) {

    r = r*i;

  }

  return r;
}
```

Logique de Floyd-Hoare

```
int fact (int n) {
  { n ≥ 1 }  // hypothèse
  int r = 1, i = 1;
  { r = i! }
  for (i = 2; i <= n; i++) {

     r = r*i;

  }

  return r;
}
```

Logique de Floyd-Hoare

```
int fact (int n) {
   { n ≥ 1 }  // hypothèse
   int r = 1, i = 1;
   { r = i! }
   for (i = 2; i <= n; i++) {
      { r = (i–1)! }
      r = r*i;

   }

   return r;
}
```

Logique de Floyd-Hoare

```
int fact (int n) {
  { n ≥ 1 }  // hypothèse
  int r = 1, i = 1;
  { r = i! }
  for (i = 2; i <= n; i++) {
    { r = (i–1)! }
    r = r*i;
    { r = i! }
  }

  return r;
}
```

Logique de Floyd-Hoare

```
int fact (int n) {
  { n ≥ 1 }  // hypothèse
  int r = 1, i = 1;
  { r = i! }
  for (i = 2; i <= n; i++) {
    { r = (i–1)! }
    r = r*i;
    { r = i! }
  }

  return r;
}
```

invariant de boucle

Logique de Floyd-Hoare

```
int fact (int n) {
  { n ≥ 1 }  // hypothèse
  int r = 1, i = 1;
  { r = i! }
  for (i = 2; i <= n; i++) {
    { r = (i–1)! }
    r = r*i;
    { r = i! }
  }
  { r = i!,  i = n }
  return r;
}
```

invariant de boucle

Logique de Floyd-Hoare

```
int fact (int n) {
  { n ≥ 1 }  // hypothèse
  int r = 1, i = 1;
  { r = i ! }
  for (i = 2; i <= n; i++) {
    { r = (i–1) ! }
    r = r*i;
    { r = i ! }
  }
  { r = i !,  i = n }
  return r; { fact(n) = n ! }
}
```

invariant de boucle

Assistants de preuves

- Logiques classique : prédicats, Scott, Hoare

  B (Abrial) : métro et RER, PVS : NASA

  HOL (Cambridge), HOL Light (Intel) : calculs flottants Intel

  ACL-2 (MIT) : calculs flottants AMD, ...

- Logiques constructives (CoQ)
  - Un algorithme est une preuve de consistance de la spécification.
  - On développe en même temps l'algorithme et la preuve, puis on extrait automatiquement le programme.

  compilateur C vérifié (Leroy et. al), protocoles bancaires, théorème des 4 couleurs (Gonthier)

La vérification de modèles (Model-Checking)

- Un programme définit un système de transitions

$S = \{s_i \mid i \in N\}$

s_0 : état initial

$s_i \rightarrow s_j$: transition

Clarke, Emerson, Sifakis : Prix Turing 2007

La vérification de modèles (Model-Checking)

- Un programme définit un système de transitions

$S = \{s_i \mid i \in N\}$

s_0 : état initial

$s_i \rightarrow s_j$: transition

- Propriété P = prédicat sur les états potentiels

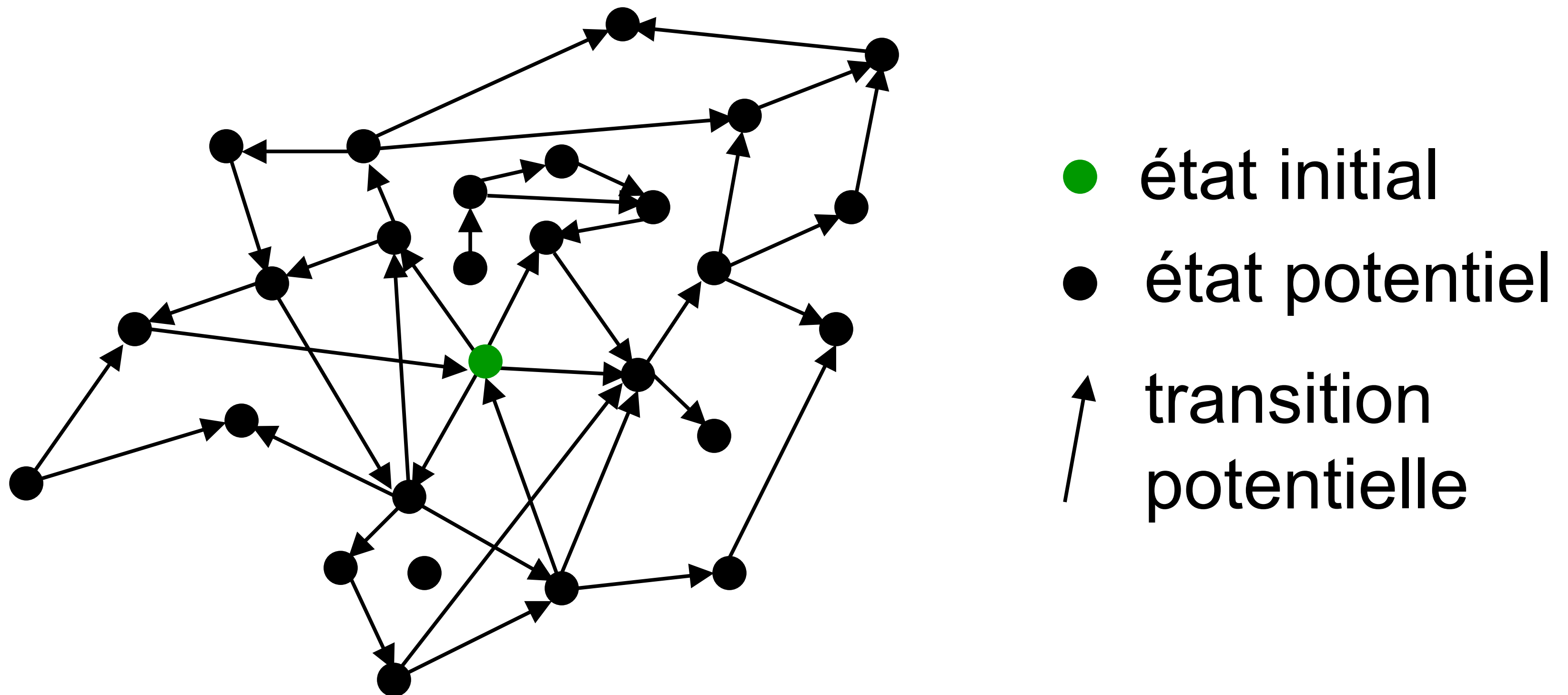
- Questions sur les états accessibles depuis l'initial

  P peut-elle être vraie (ou fausse)?

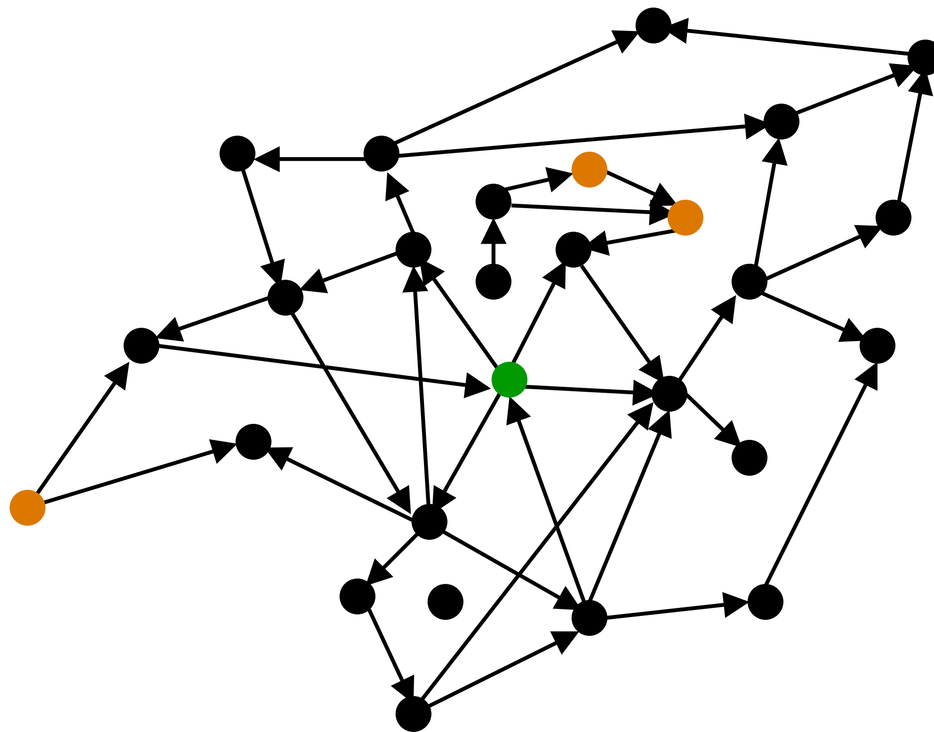
  P est-elle inévitablement vraie sur toute trajectoire infinie?

Clarke, Emerson, Sifakis : Prix Turing 2007

Graphe des états / transitions potentiels (extrait du programme)

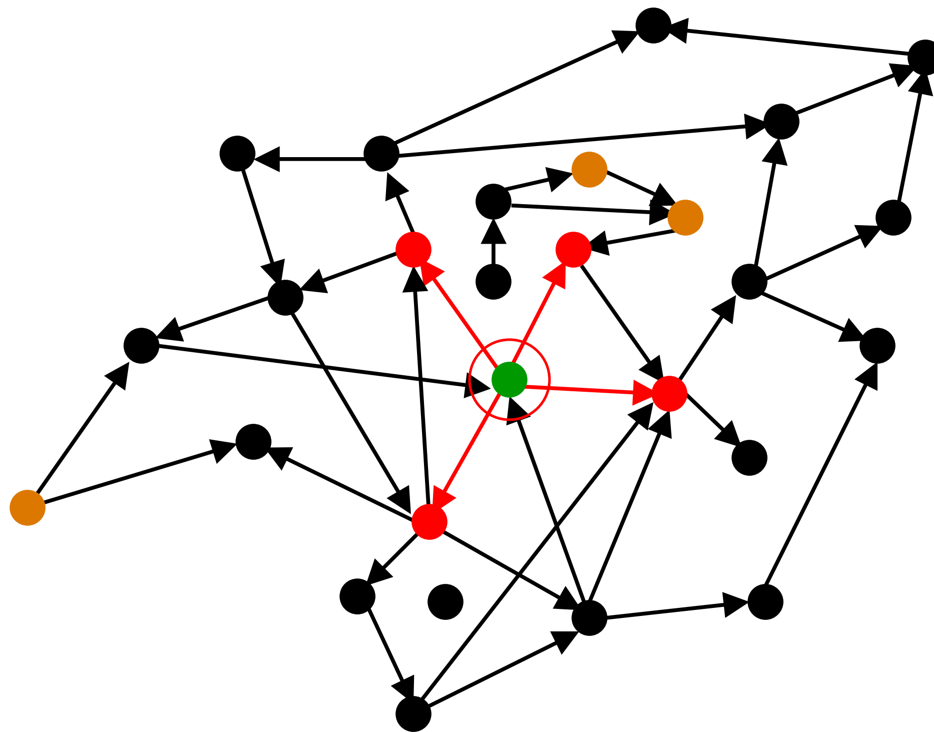


Propriété P = ensemble d'états



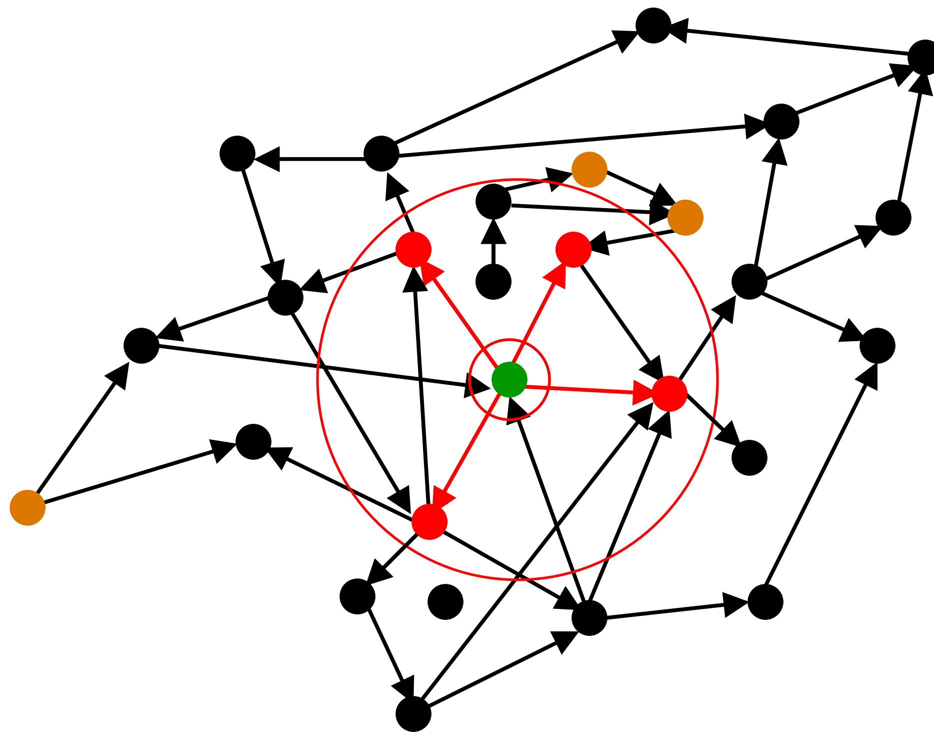
- état initial
- état potentiel
- transition potentielle
- P fausse

Analyse en avant
Etats atteignables en 1 transition



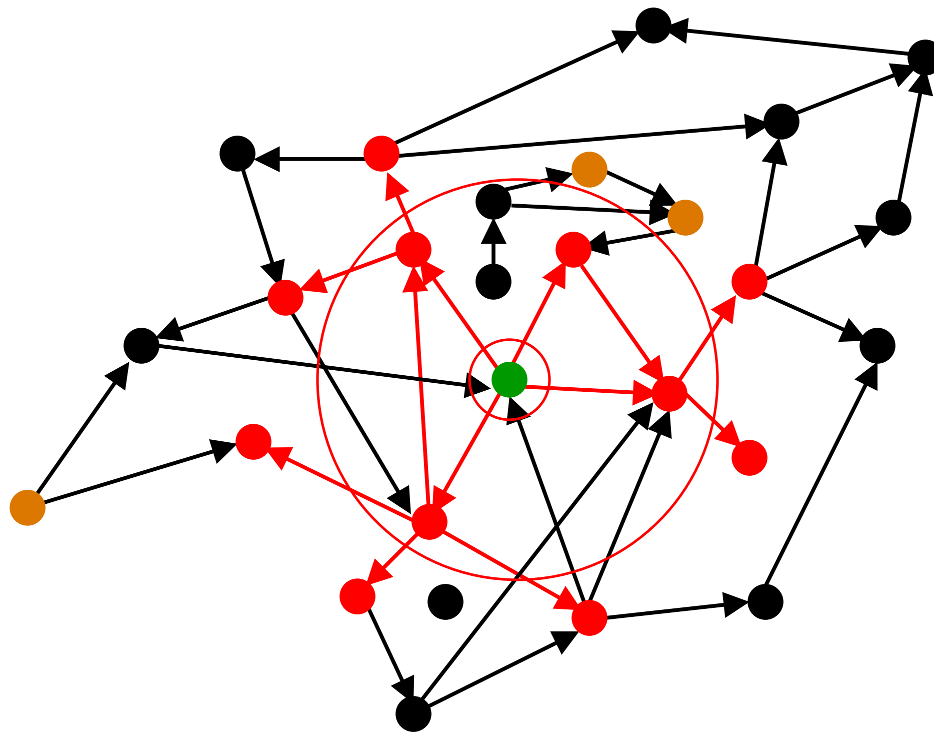
- état initial
- état atteignable
- transition atteignable

Analyse en avant
Etats atteignables en 1 transition



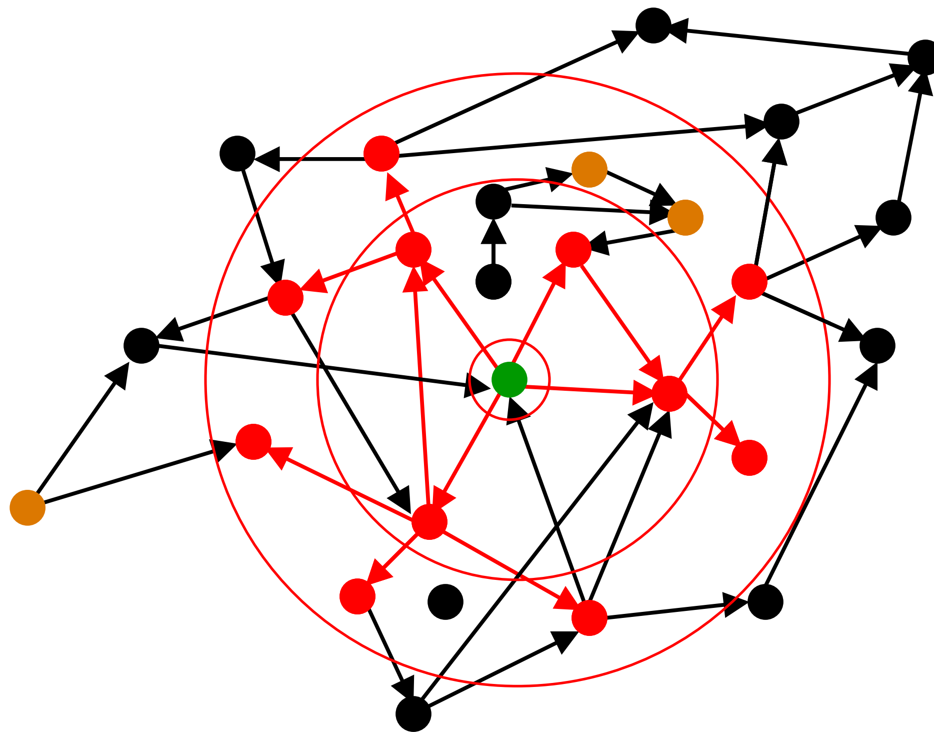
- état initial
- état atteignable
- transition atteignable

Etats atteignables en 2 transitions



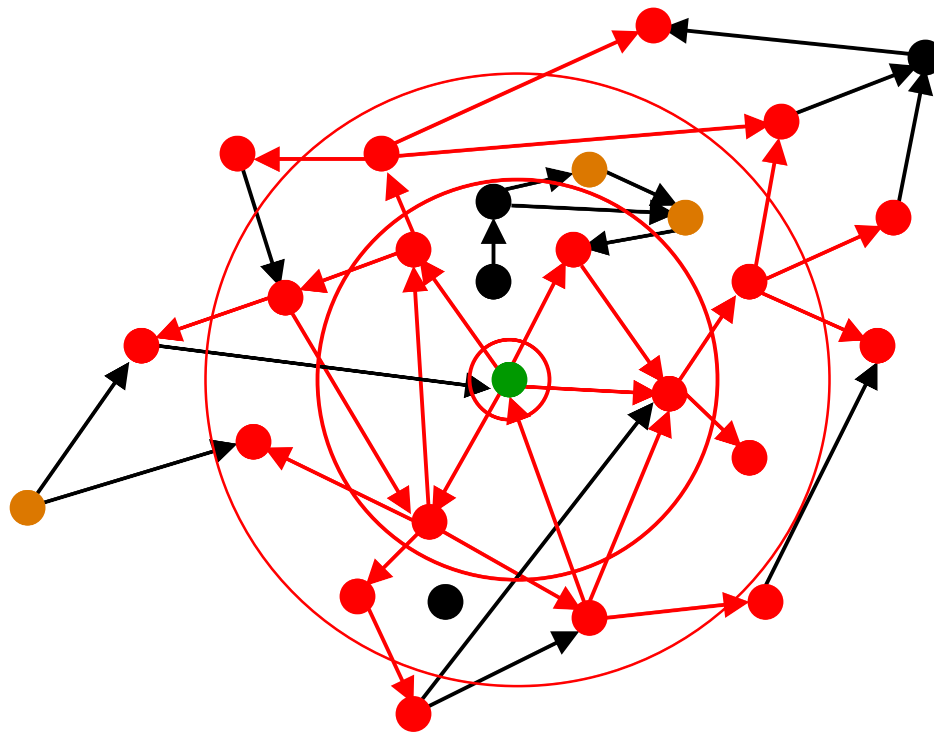
- état initial
- état atteignable
- transition atteignable

Etats atteignables en 2 transitions



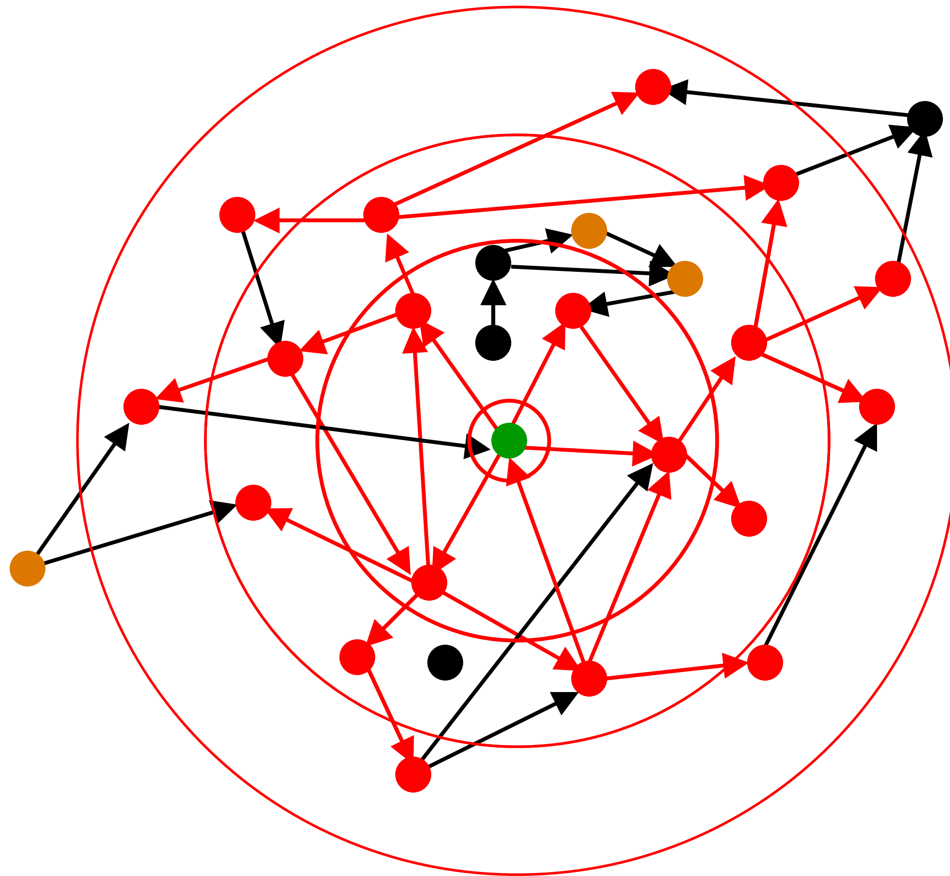
- état initial
- état atteignable
- transition atteignable

Etats atteignables en 3 transitions



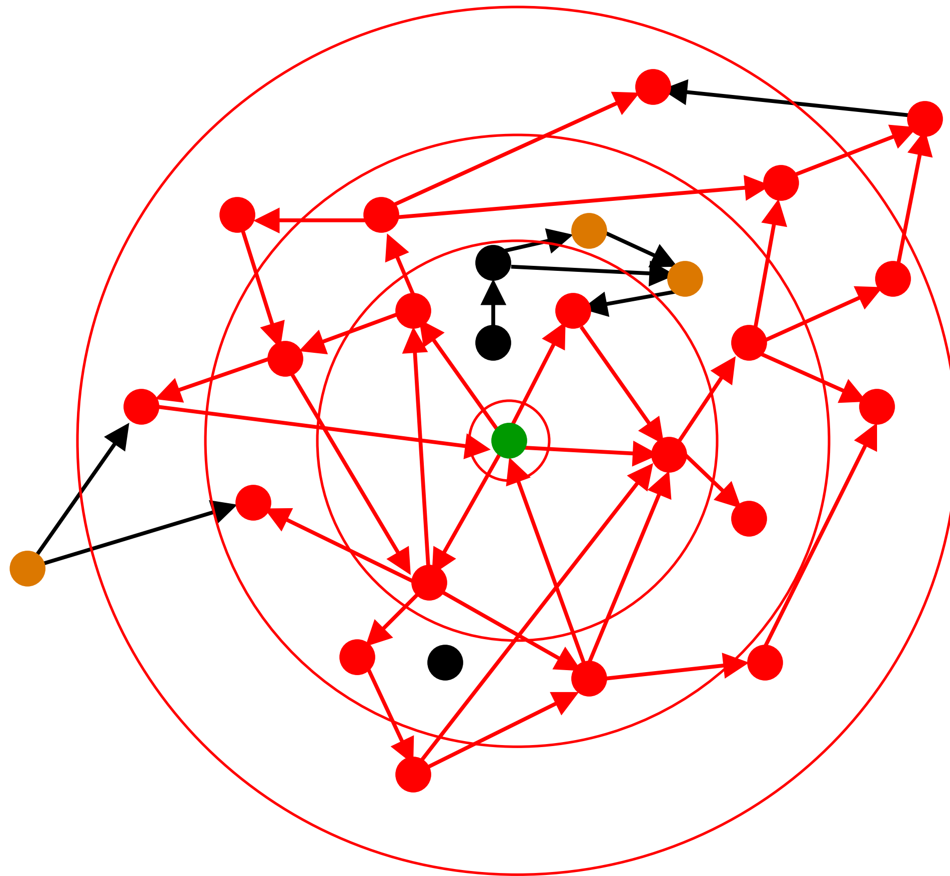
- état initial
- état atteignable
- transition atteignable

Etats atteignables en 3 transitions



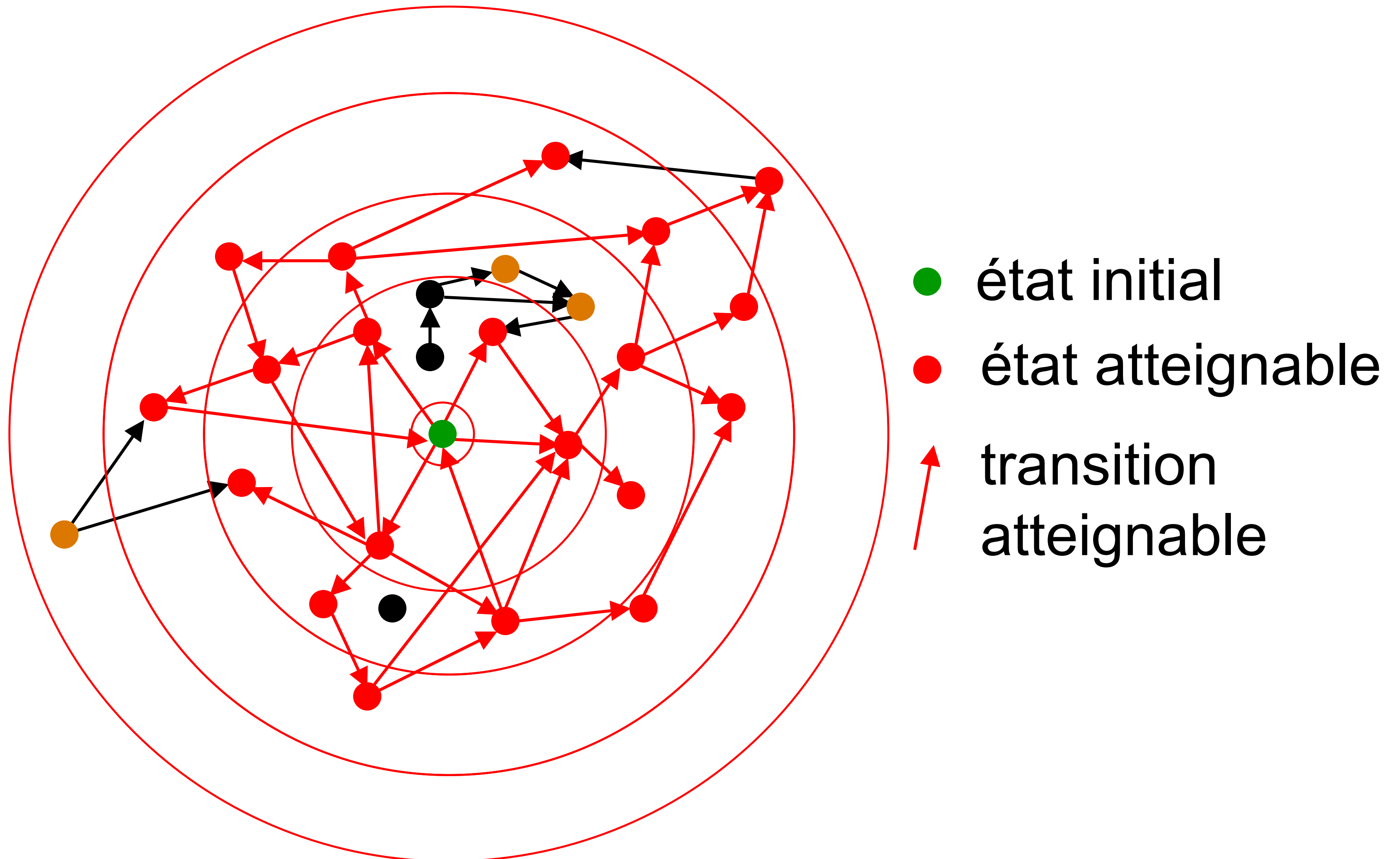
- état initial
- état atteignable
- transition atteignable

Etats atteignables en 4 transitions

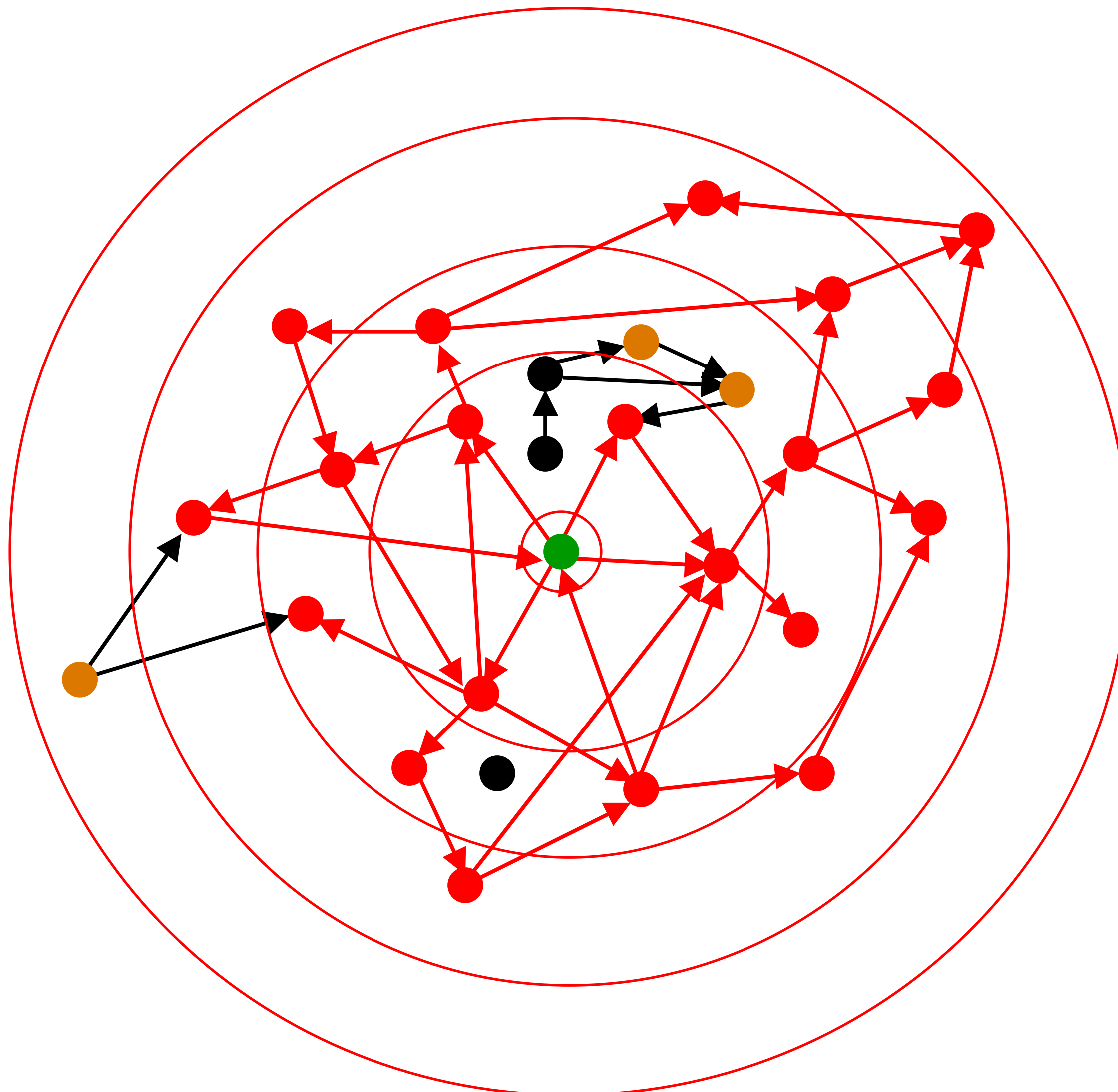


- état initial
- état atteignable
- transition atteignable

Etats atteignables en 4 transitions

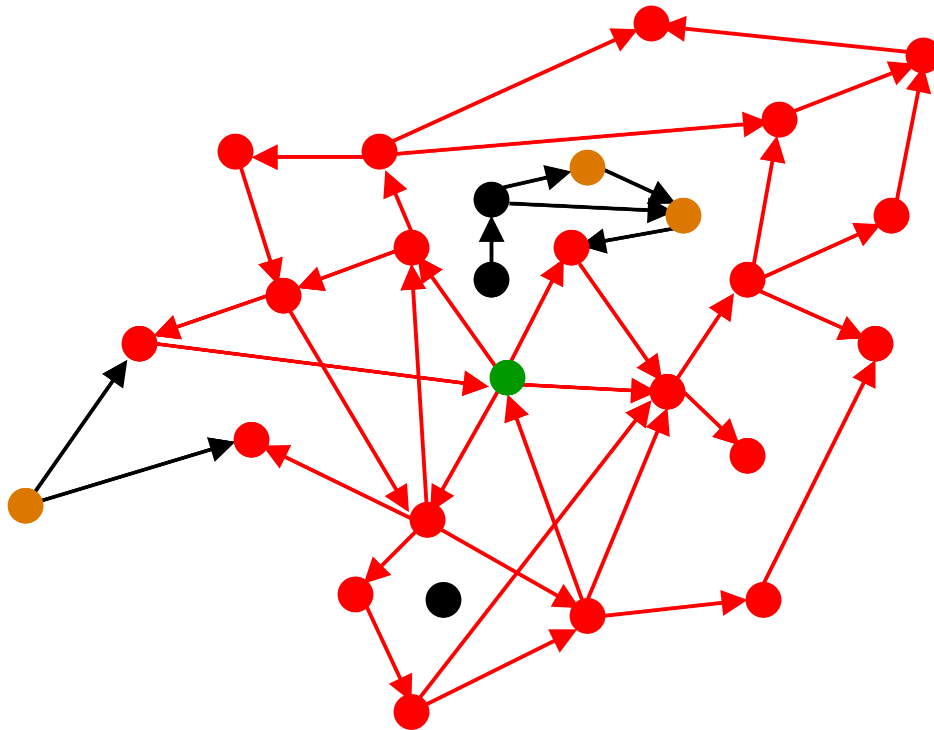


Denier marquage,
Pas de nouveaux états => point fixe

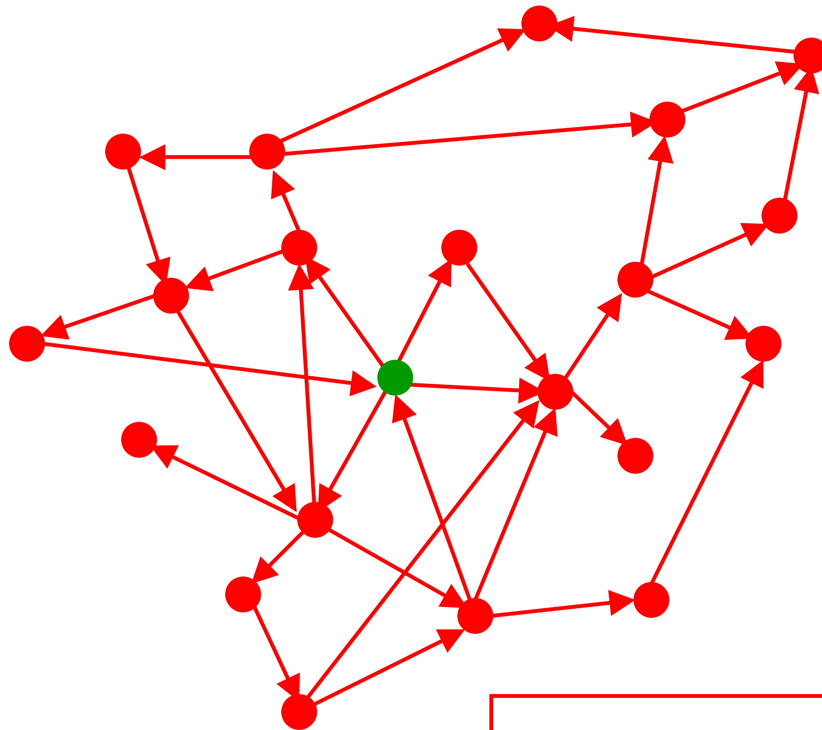


diamètre = 4

Ignorer les états non atteignables

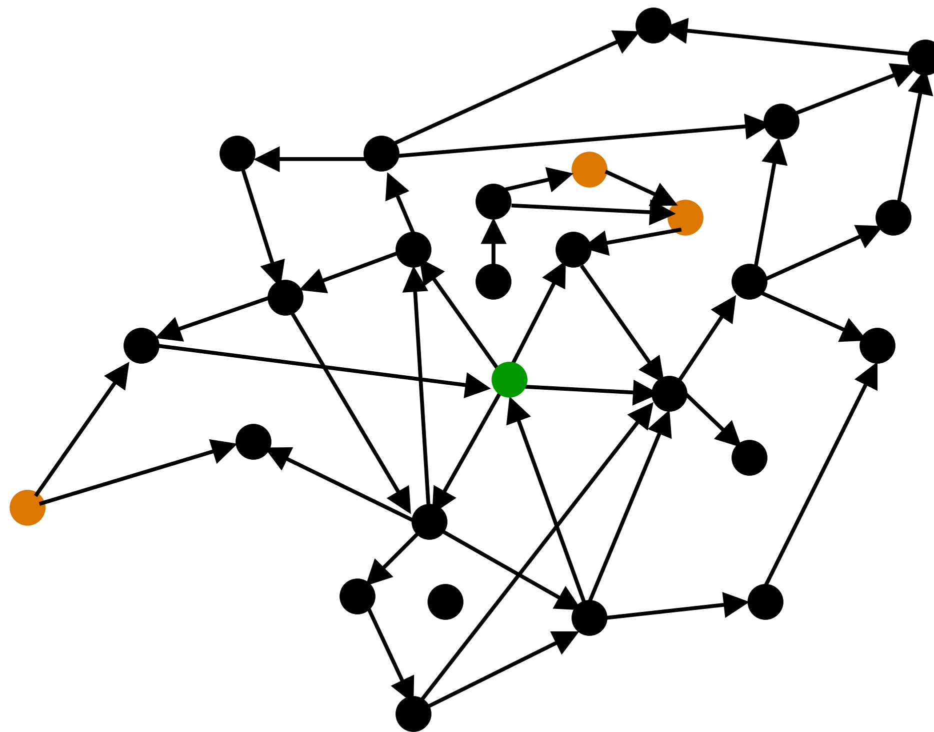


Ignorer les états non atteignables



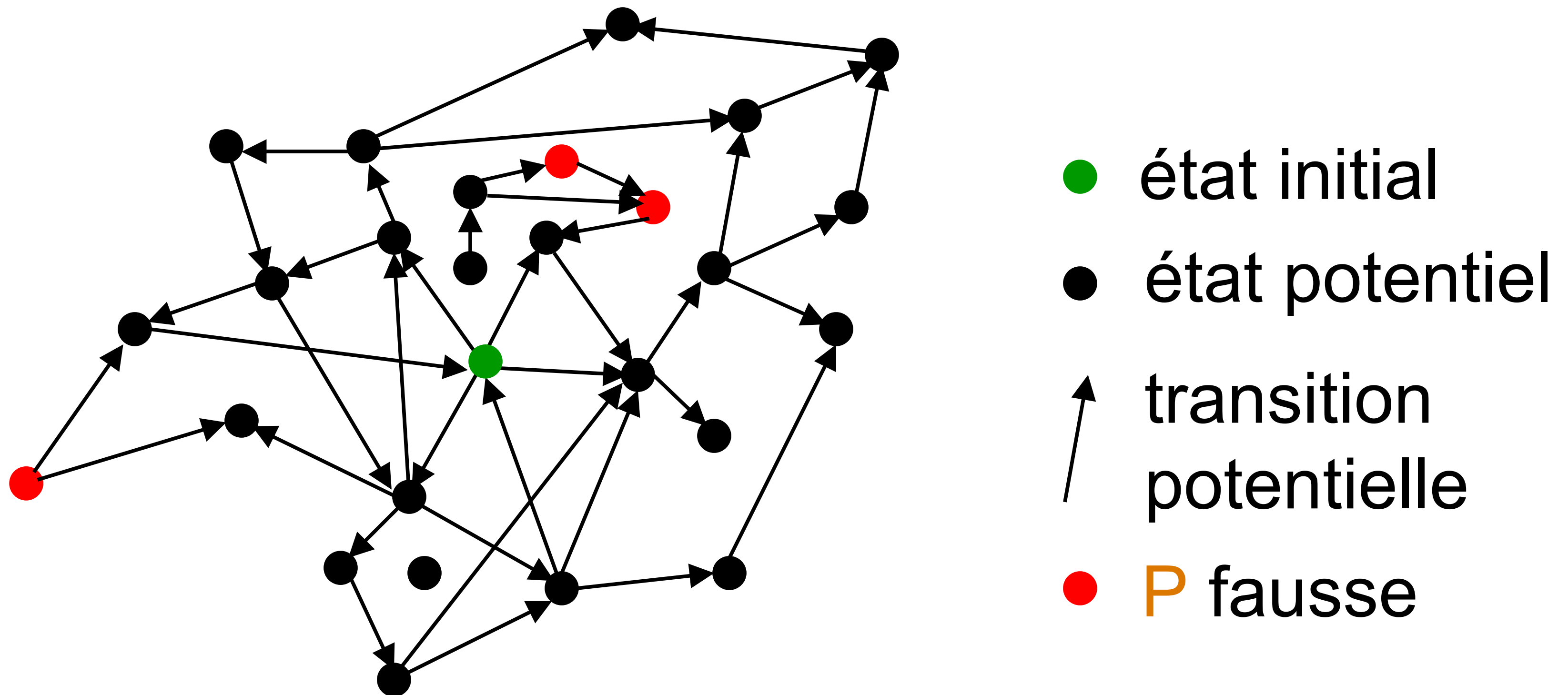
Pas d'états où P est fausse
=> P est vraie pour le système

Analyse en arrière

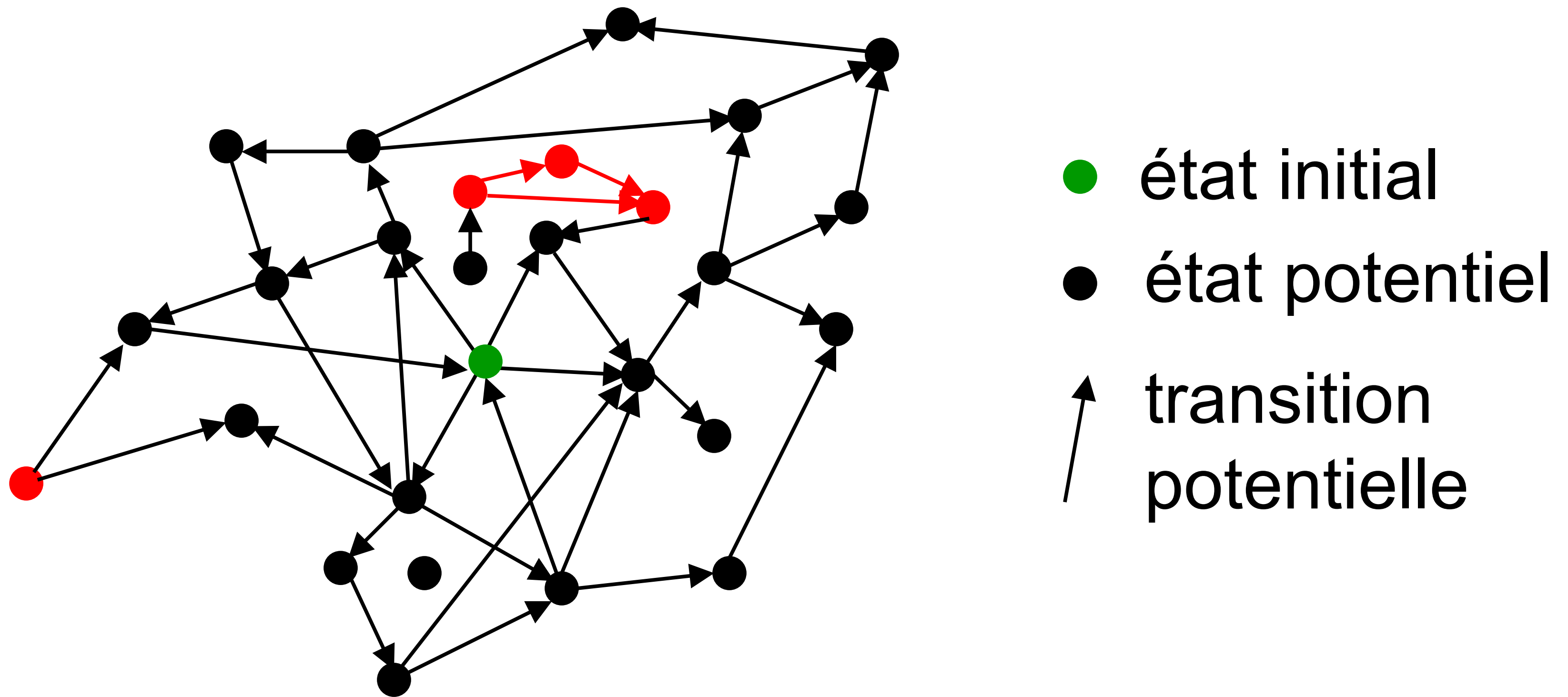


- état initial
- état potentiel
- transition potentielle
- P fausse

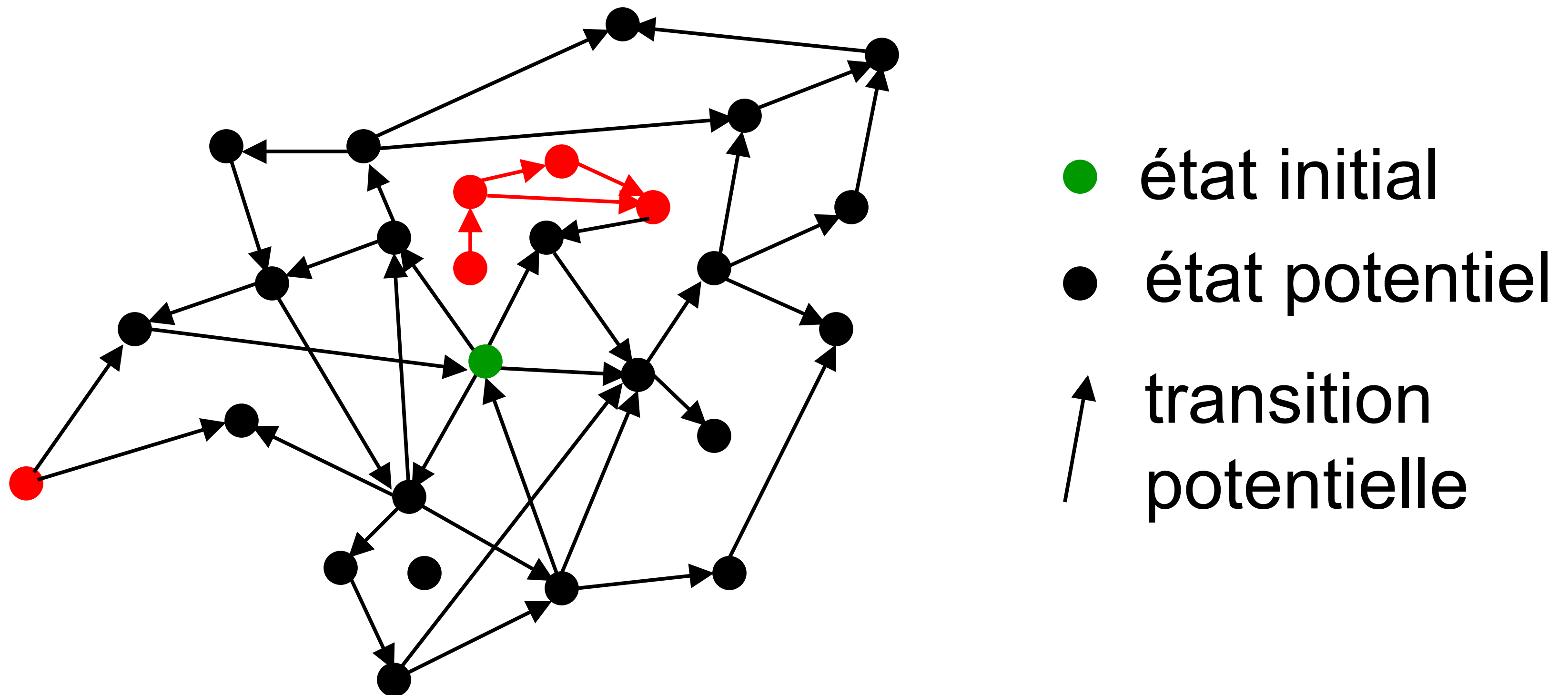
Marquer les états où P est fausse



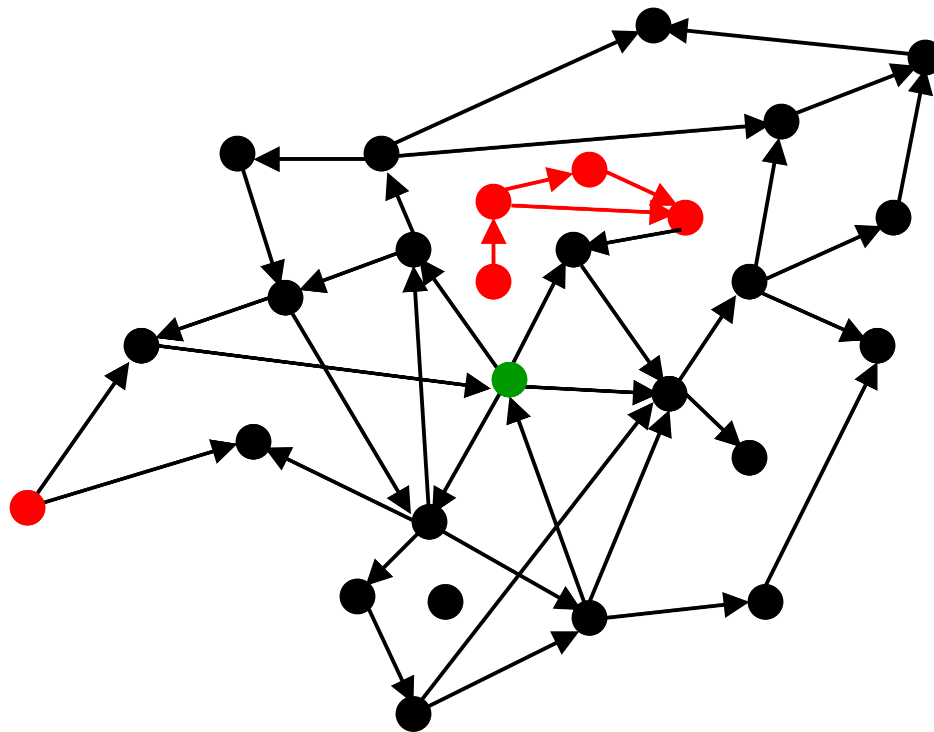
Marquer itérativement leurs prédécesseurs



Marquer itérativement leurs prédécesseurs

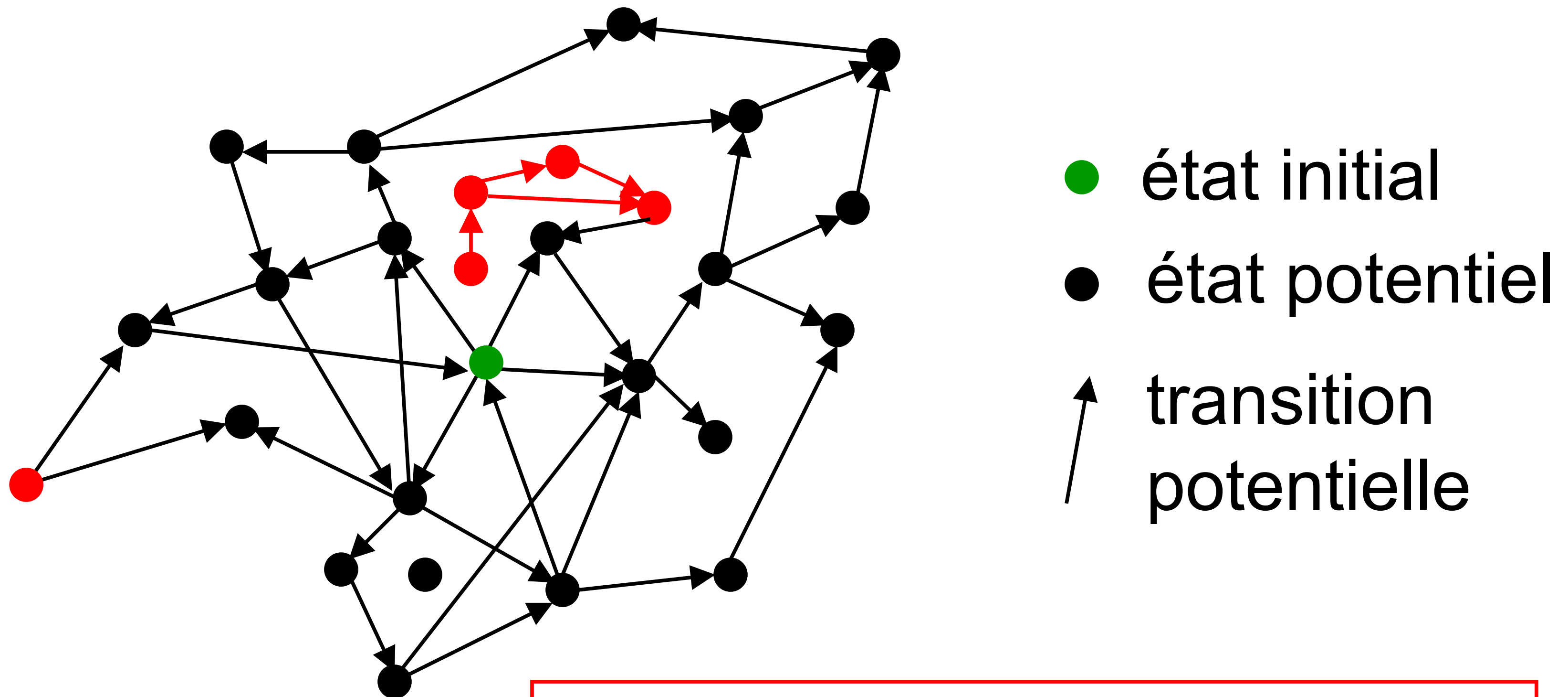


Fin du marquage



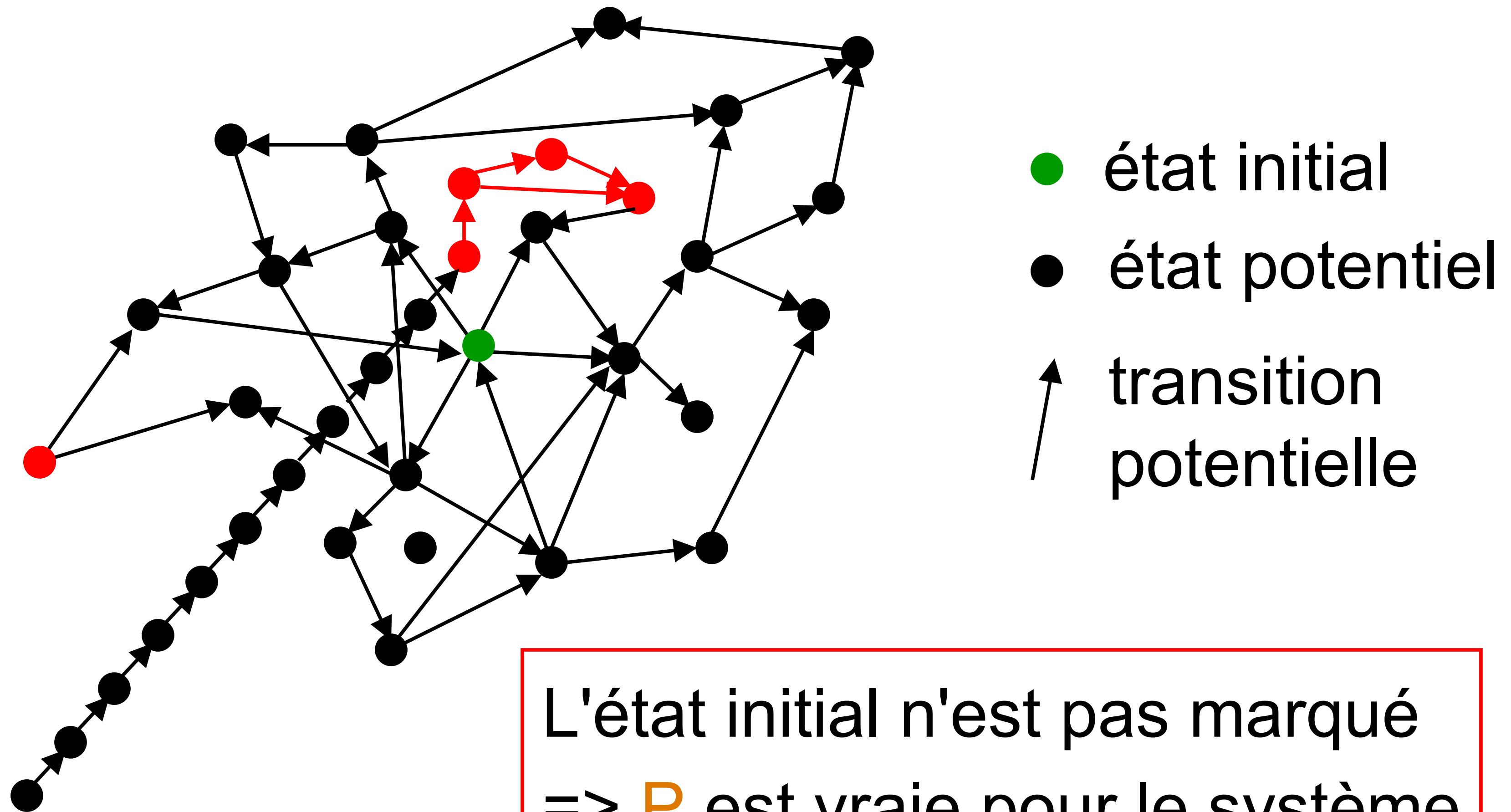
- état initial
- état potentiel
- transition potentielle

Fin du marquage

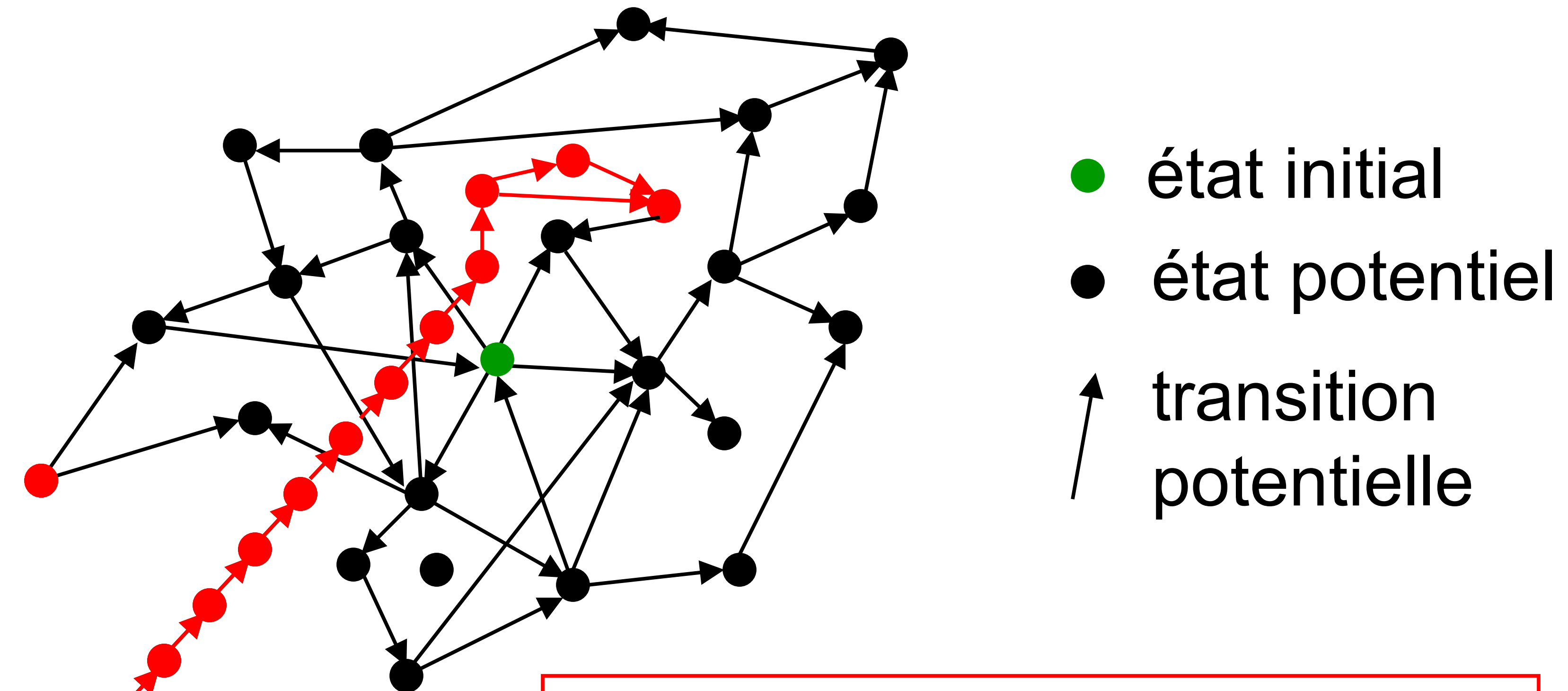


L'état initial n'est pas marqué
=> P est vraie pour le système

Fin du marquage



Fin du marquage



L'état initial n'est pas marqué
=> P est vraie pour le système

Eviter l'explosion de la taille

- Exploration explicite très rapide, randomisation

  SPIN : protocoles, caches, etc.

- Exploration symbolique par calcul sur formules

  fonctions caractéristiques : BDDs

  dépliement symbolique et récurrence : SAT

  couplage de moteurs Booléens et numériques

- Interprétation abstraite

  cf. séminaire

Eviter l'explosion de la taille

- Exploration explicite très rapide, randomisation

  SPIN : protocoles, caches, etc.

- Exploration symbolique par calcul sur formules

  fonctions caractéristiques : BDDs

  dépliement symbolique et récurrence : SAT

  couplage de moteurs Booléens et numériques

- Interprétation abstraite

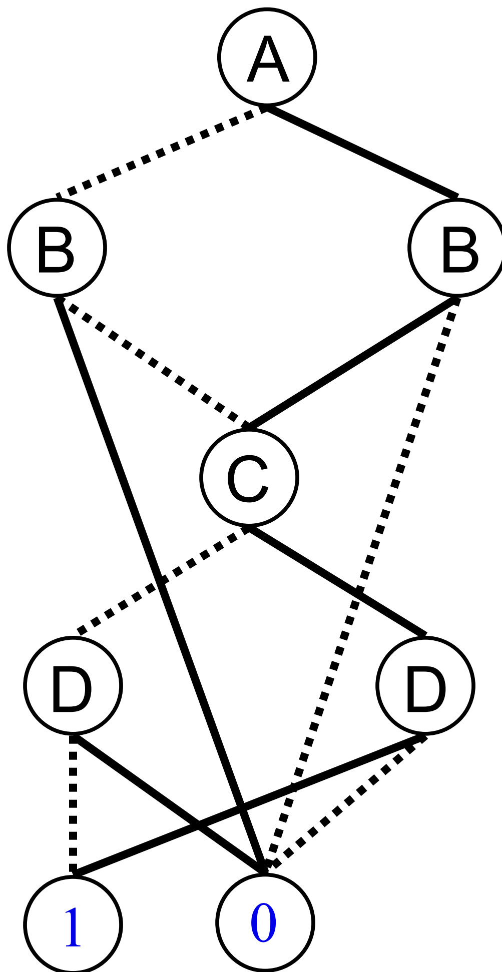
  cf. séminaire

Nombreux systèmes académiques et industriels

SMV, Prover, Magellan, Formal Check, SLAM, Yices,...

BDD = Binary Decision Diagram

$$(A \Leftrightarrow B) \wedge (C \Leftrightarrow D)$$



Ensemble de tous les états
où A=B et C=D
~1/4 du nombre d'états total !

Un domaine en pleine évolution

Un domaine en pleine évolution

- Passer des cas précis aux cas génériques

  des ascenseurs à 4 étages aux ascenseurs à n étages

Un domaine en pleine évolution

- Passer des cas précis aux cas génériques

  des ascenseurs à 4 étages aux ascenseurs à n étages

- Coupler vérification assistée + moteurs logiques

  + moteurs numériques

  se ramener à des sous problèmes finis

  ou traitables automatiquement

Un domaine en pleine évolution

- Passer des cas précis aux cas génériques

  des ascenseurs à 4 étages aux ascenseurs à n étages

- Coupler vérification assistée + moteurs logiques

  + moteurs numériques

  se ramener à des sous problèmes finis

  ou traitables automatiquement

- Profiter des machines multi-coeurs et réseaux

  lancer 10 algos différents et prendre le 1er qui gagne

  mettre un algorithme sur multi-coeurs ?

Un domaine en pleine évolution

- Passer des cas précis aux cas génériques
  des ascenseurs à 4 étages aux ascenseurs à n étages
- Coupler vérification assistée + moteurs logiques
  + moteurs numériques
  se ramener à des sous problèmes finis
  ou traitables automatiquement
- Profiter des machines multi-coeurs et réseaux
  lancer 10 algos différents et prendre le 1er qui gagne
  mettre un algorithme sur multi-coeurs ?

Comment prévoir le temps de vérification ?
Approche géométrique des systèmes de transition...

Ce dont nous n'avons pas parlé

- Les calculs et logiques de l'informatique
  - lambda-calcul, domaines de Scott
  - logiques constructives, théorie des constructions
  - automates, logiques temporelles
- La génération automatique de tests
  - randomisés, dirigés par un prouveur, etc.
- Les preuves de sécurité
  - protocoles bancaires, sécurité anti-intrusions, etc.

Ce dont nous n'avons pas parlé

- Les calculs et logiques de l'informatique
  - lambda-calcul, domaines de Scott
  - logiques constructives, théorie des constructions
  - automates, logiques temporelles
- La génération automatique de tests
  - randomisés, dirigés par un prouveur, etc.
- Les preuves de sécurité
  - protocoles bancaires, sécurité anti-intrusions, etc.
- Et le Sudoku qui est un jeu d'enfant pour SAT...

Conclusion

- La vérification est une activité clef

  pourquoi programmer plus vite si vérifier est plus long ?

- La programmation doit s'adapter à la vérification

  et non l'inverse (cf ponts et résistance des matériaux)

- Le test reste indispensable et dominant

- Les méthodes formelles commencent à se répandre

  elles trouvent des bugs introuvables par tests

  et permettent des preuves infaisables à la main

  (actuellement, surtout utilisées pour les circuits)