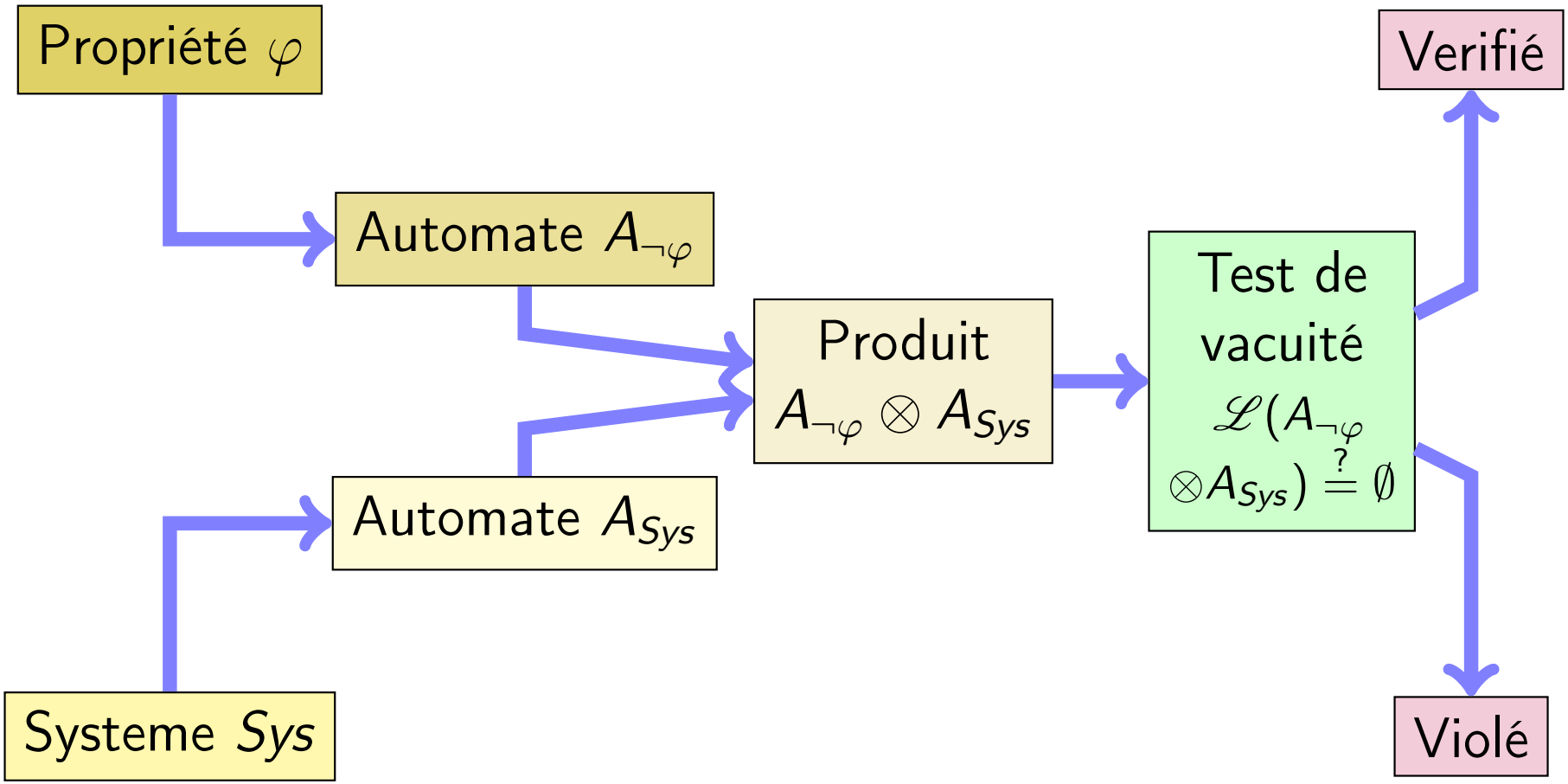
Tests de vacuité et réduction de la complexité

Etienne Renault

Avril 2015

https://www.lrde.epita.fr/~renault/teaching/imc/

L'approche automate pour le model-checking



Tests de vacuité (aka emptiness checks)

Objectif

$$\text{Calculer si } \mathscr{L}(A_{\neg\varphi} \otimes A_{Sys}) \stackrel{?}{=} \emptyset$$

Un automate est non-vide si :

1. Il existe un chemin vers un état acceptant,
2. Il existe un cycle autour de cet état

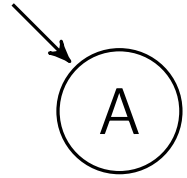
Le model-checking LTL peut être réduit à
un simple problème de graphe !

Rappels sur les parcours de graphes

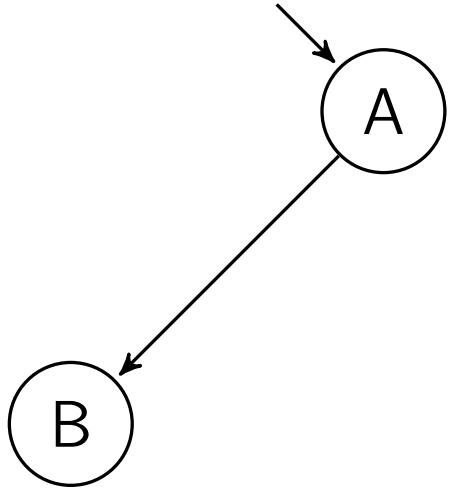
Deux types de parcours existent :

- Depth-first search (DFS) : parcours en profondeur avec l'utilisation d'une pile
- Bread-first search (BFS) : parcours en largeur avec l'utilisation d'une file

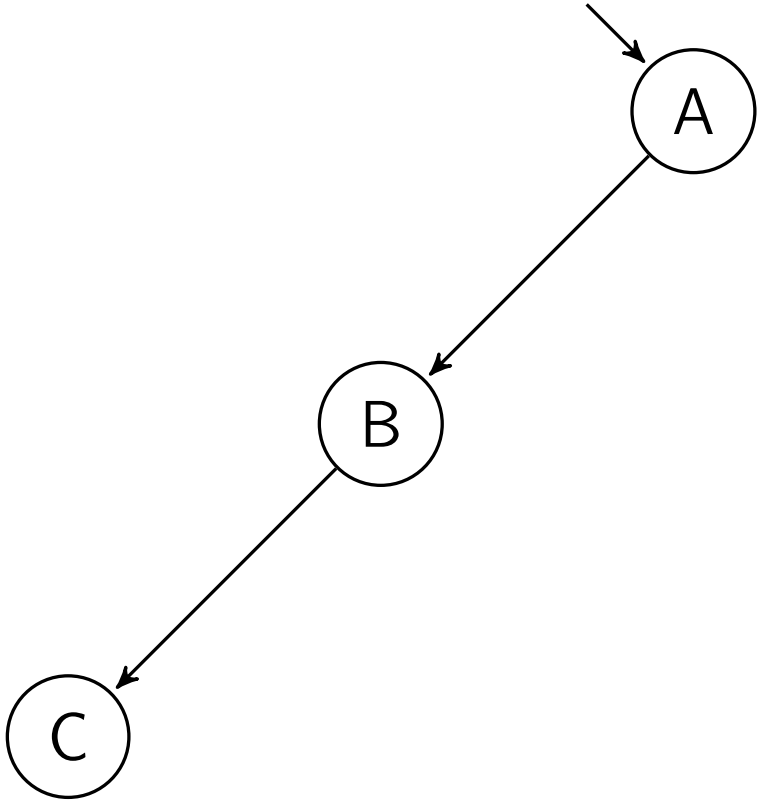
Depth-first search



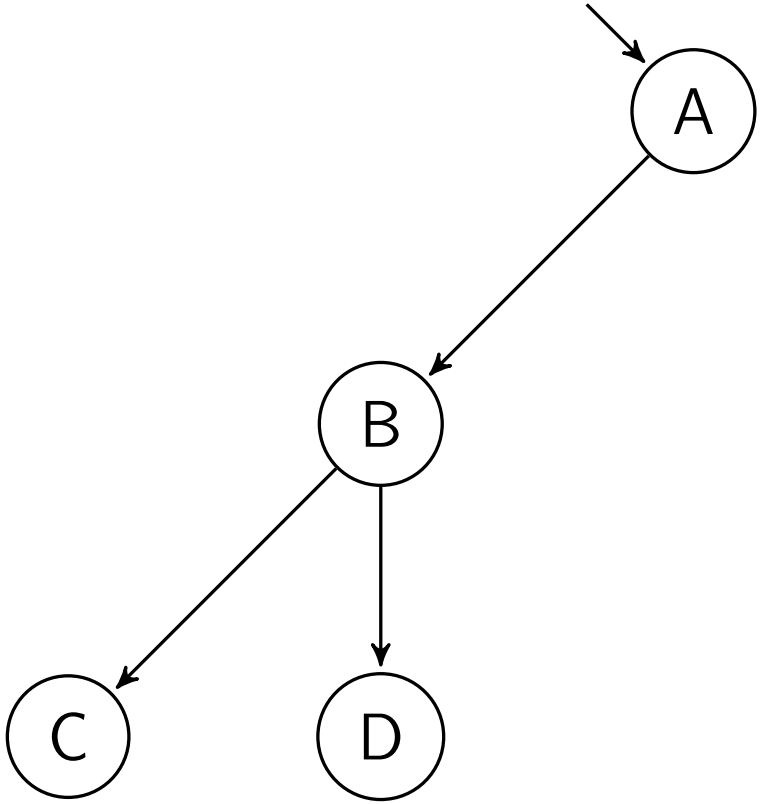
Depth-first search



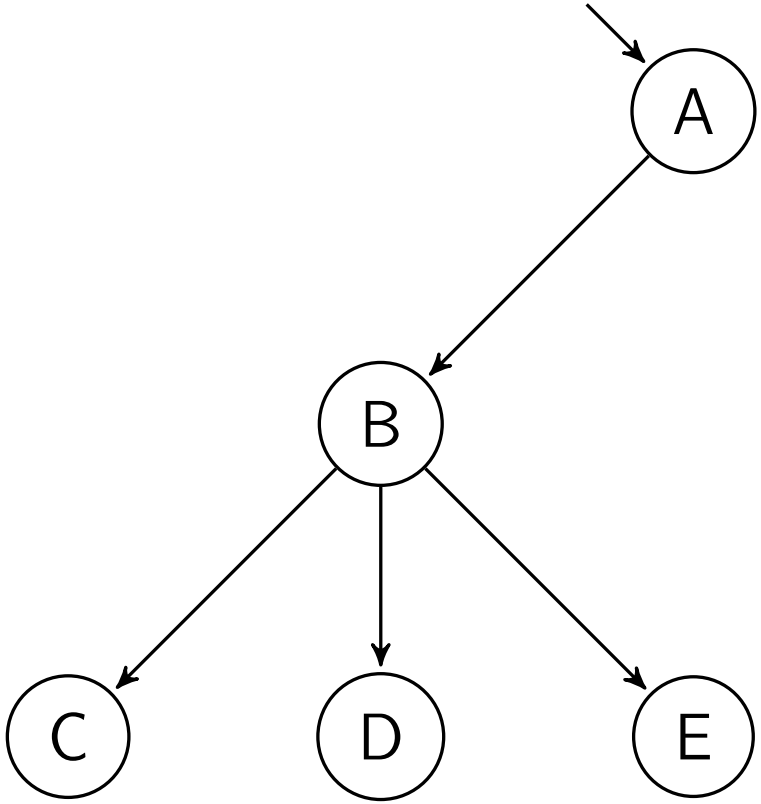
Depth-first search



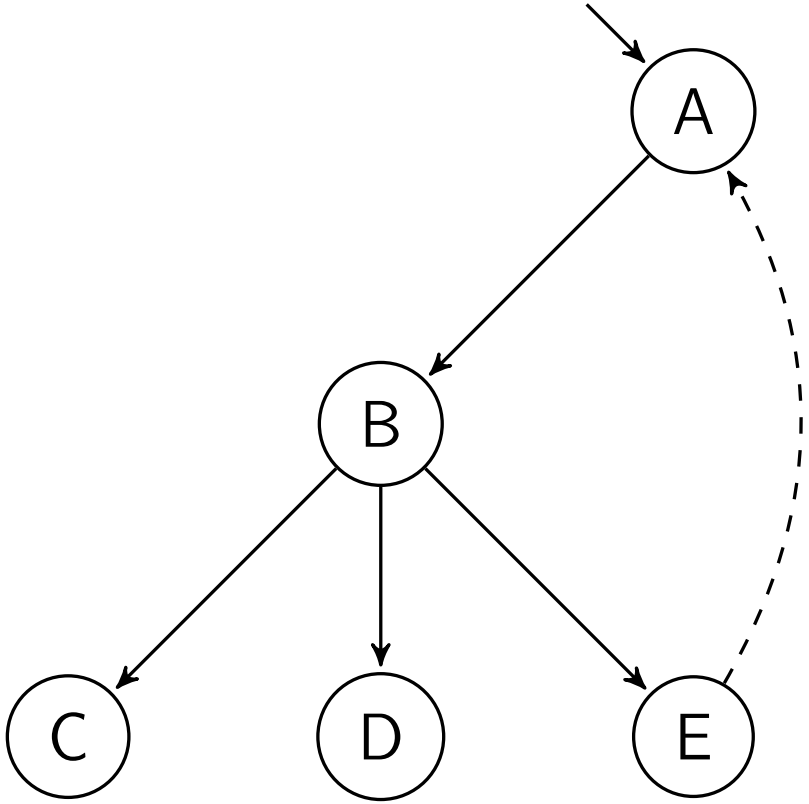
Depth-first search



Depth-first search

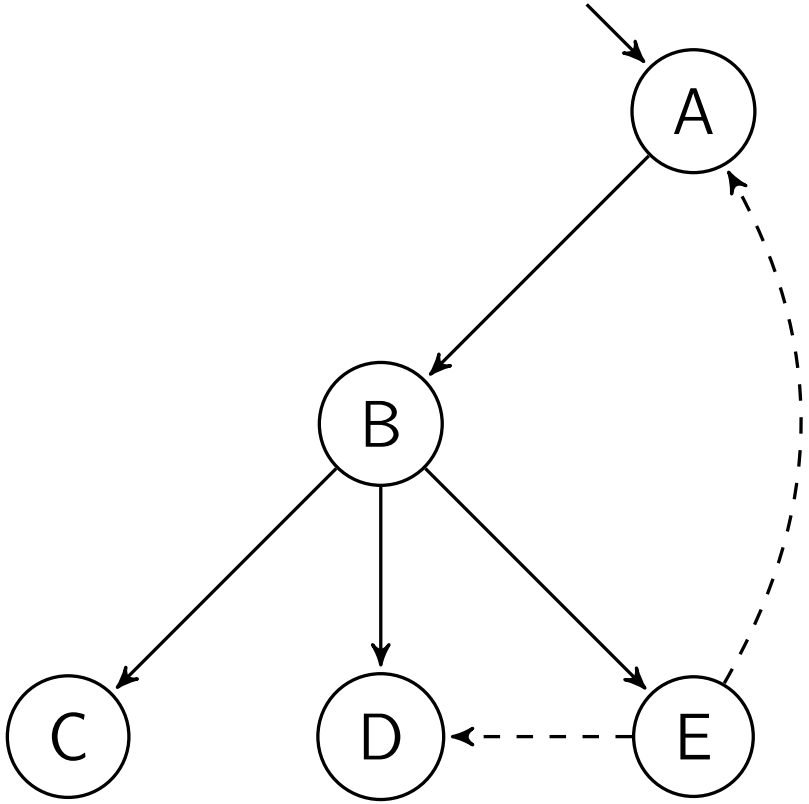


Depth-first search



La transition (E,A) est un *closing-edge*.
Les états A,B,E constituent une composante fortement connexe.

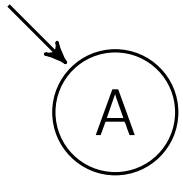
Depth-first search



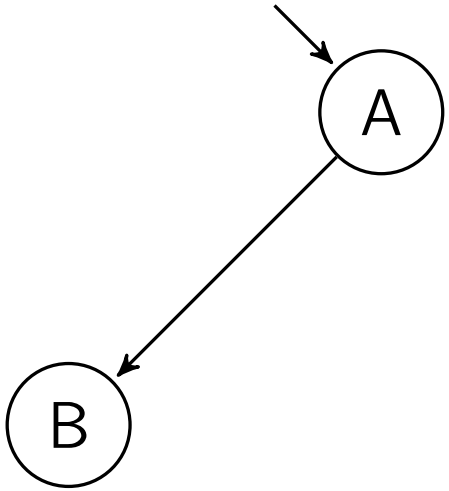
La transition (E,A) est un *closing-edge*.
Les états A,B,E constituent une composante fortement connexe.

La transition (E,D) est un *back-edge*.

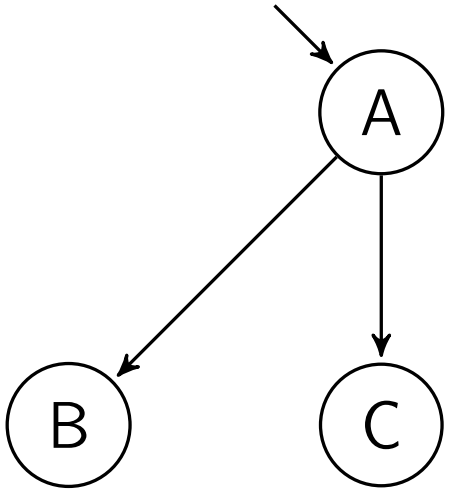
Bread-first search



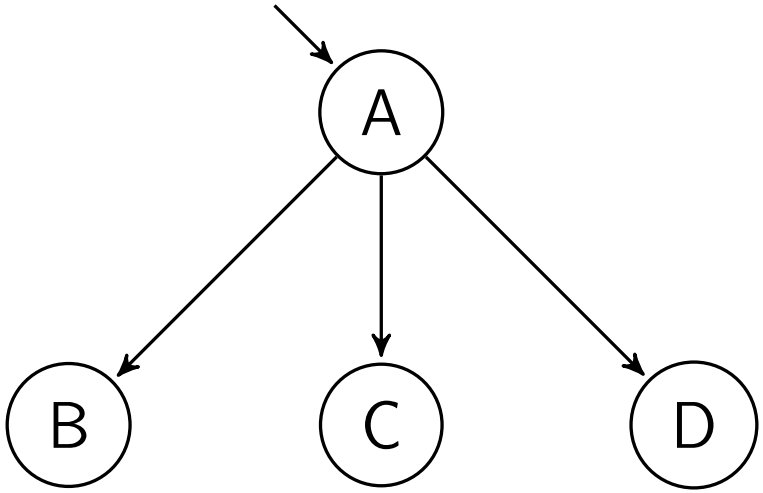
Bread-first search



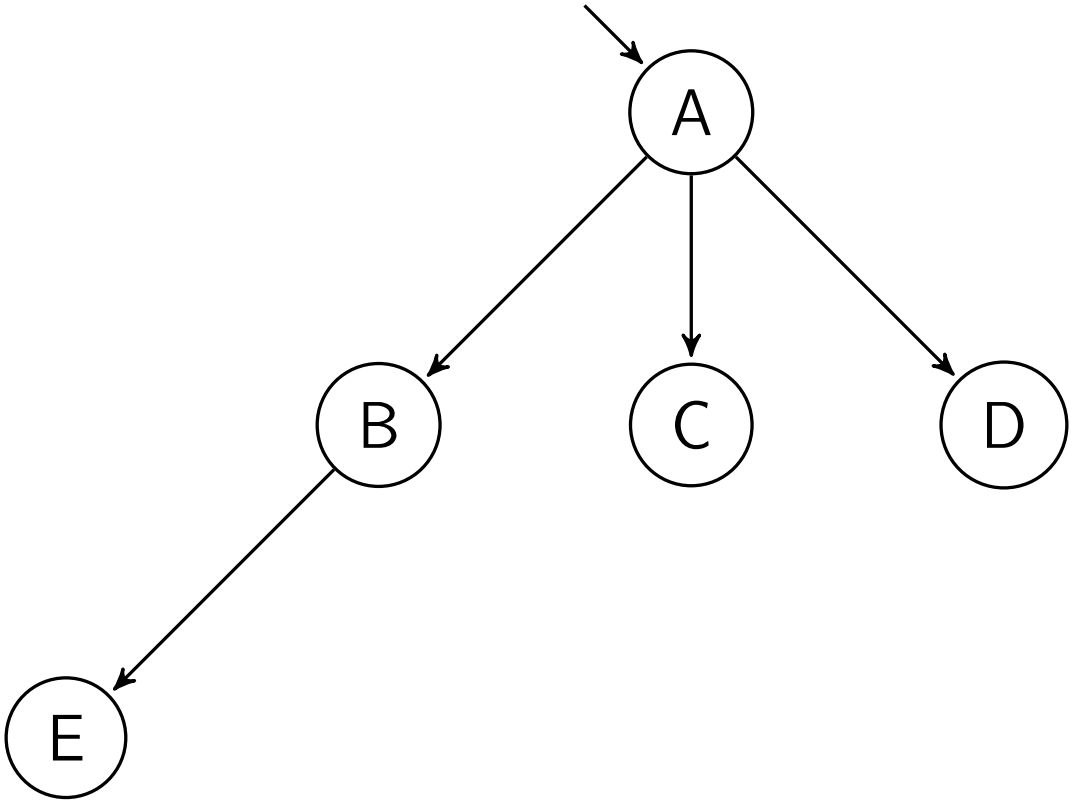
Bread-first search



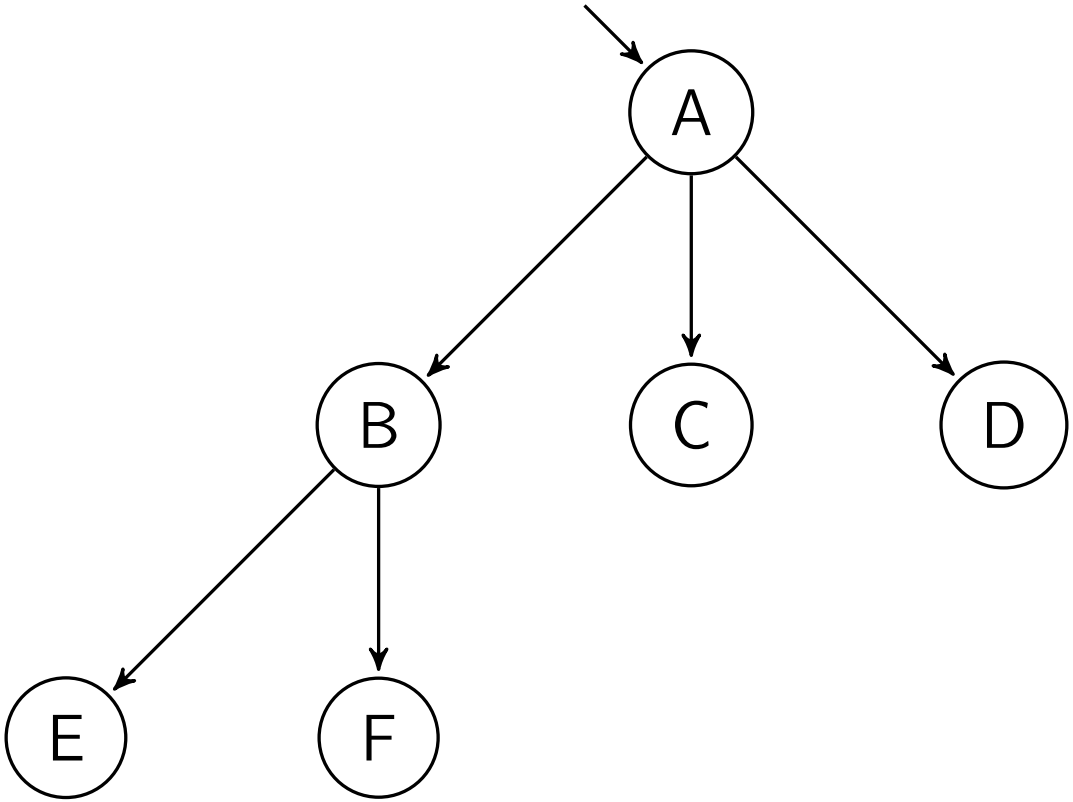
Bread-first search



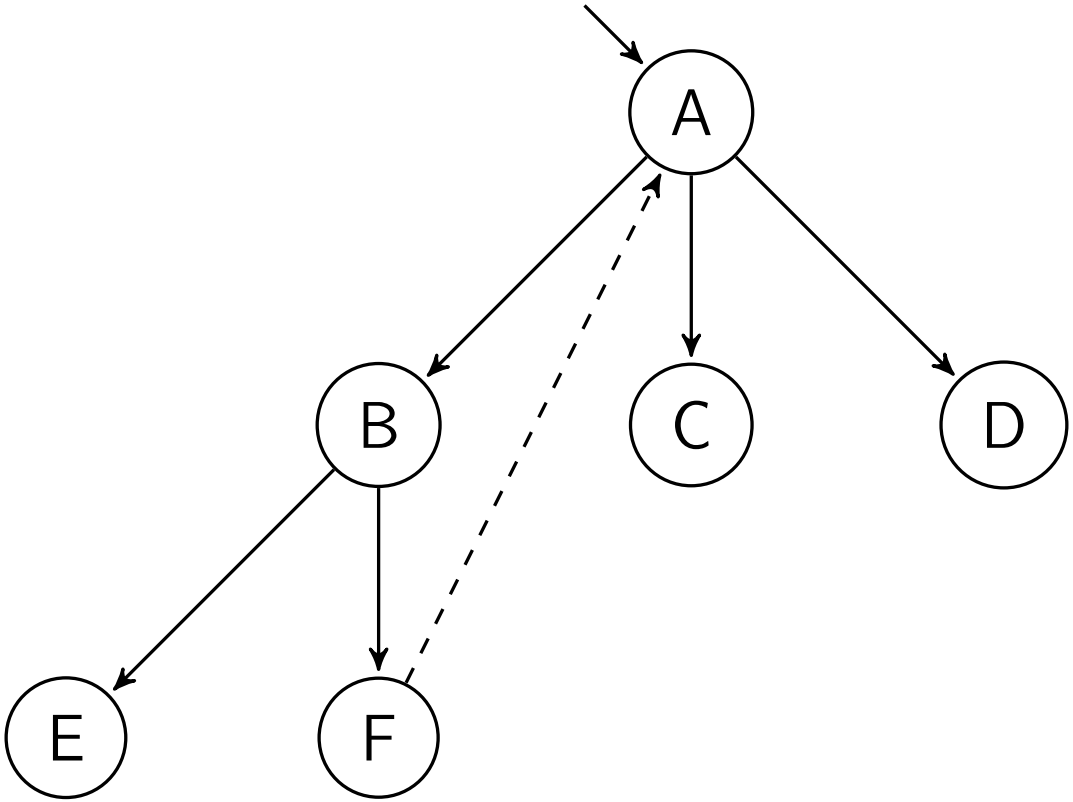
Bread-first search



Bread-first search



Bread-first search



Les parcours BFS ne permettent pas de différencier *backedge* et *closing-edge*.

Comparaison des parcours

Ces deux parcours ont la même complexité théorique . . . mais dans la pratique les DFS sont mieux adaptés au model checking LTL car :

- L'espace d'état de $A_{\neg\varphi} \otimes A_{Sys}$ est généralement plus large que haut !
- Les parcours BFS ne permettent pas de détecter facilement les closing-edges
- Les parcours BFS ne permettent pas d'extraire facilement les contre-exemples (mais ils en produisent de plus petits)

Two Steps [Courcoubetis 1990]

Idée : Algorithme en deux phases

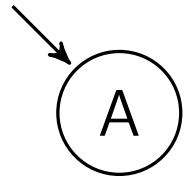
1. trouver tous les états acceptants accessibles en utilisant un simple DFS
2. chercher un cycle autour de ces états en utilisant un simple DFS

\+ On évite l'énumération systématique de tous les cycles (algorithme de Johnson, complexité $O((|states| + |edges|) \times (cycles + 1)))$

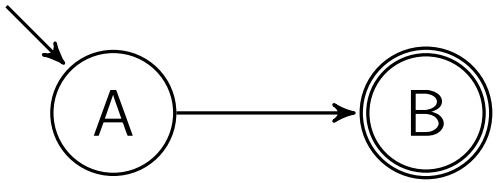
– Certains états peuvent être reparcourus plusieurs fois

Comment trouver les contre-exemples ?

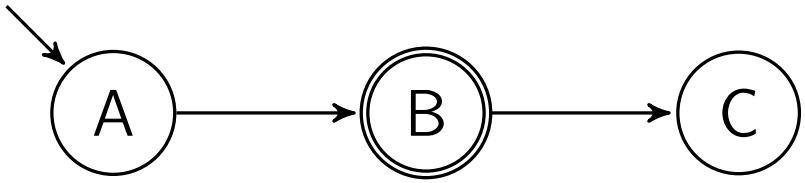
Two Steps – Exemple



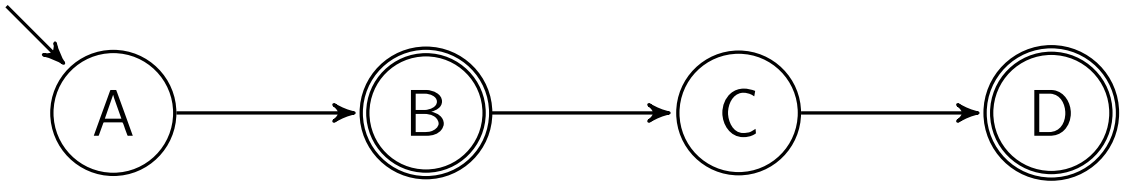
Two Steps – Exemple



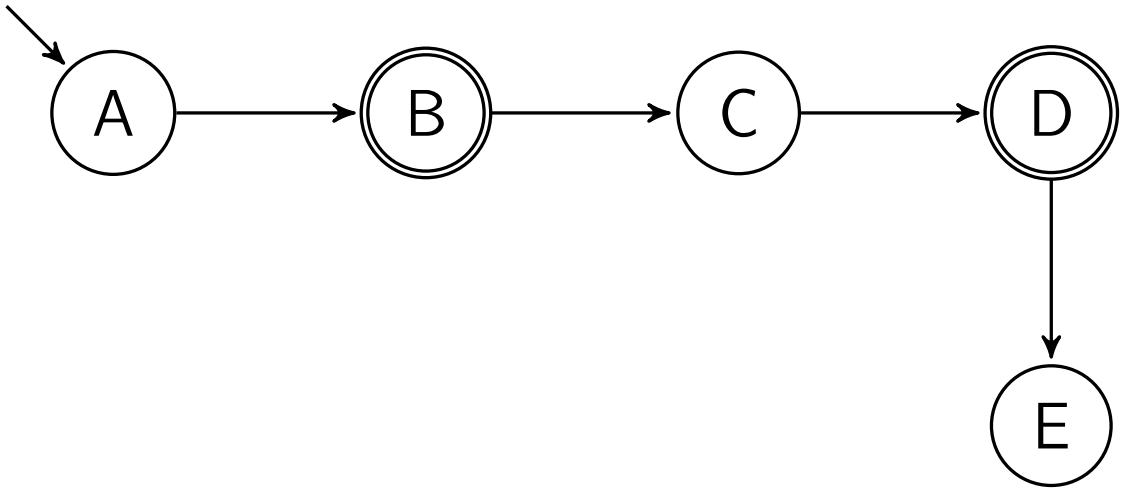
Two Steps – Exemple



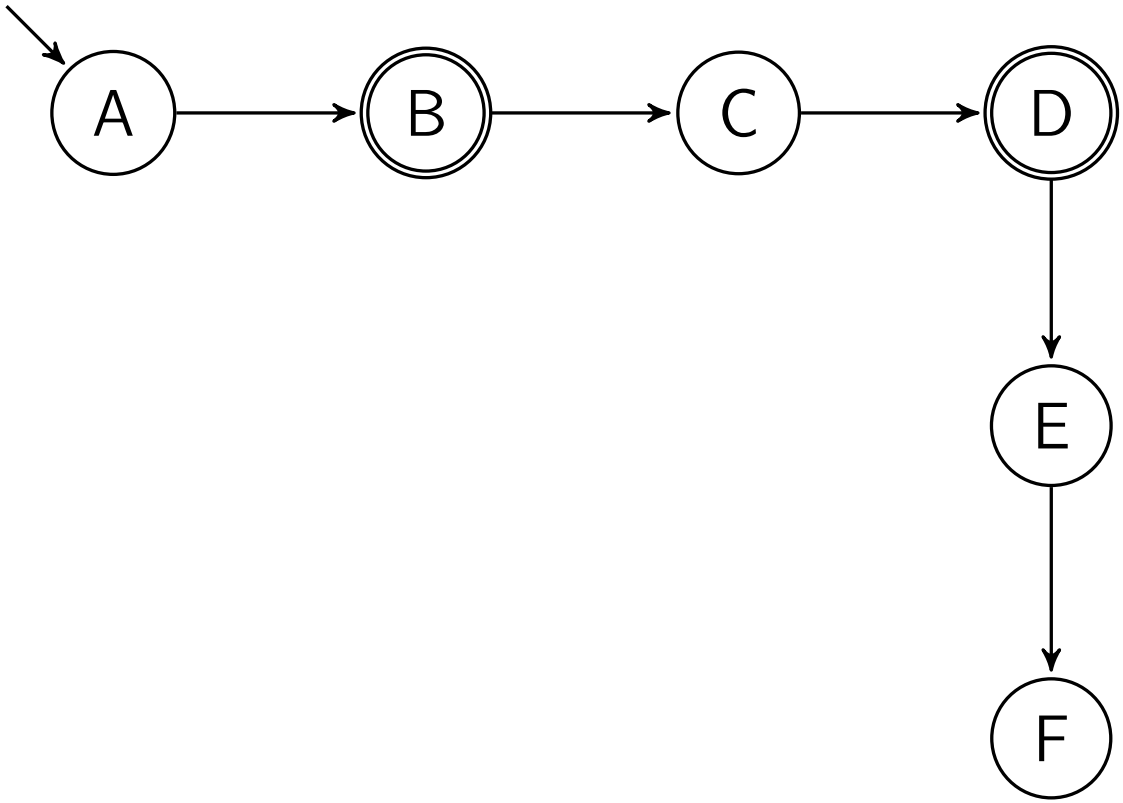
Two Steps – Exemple



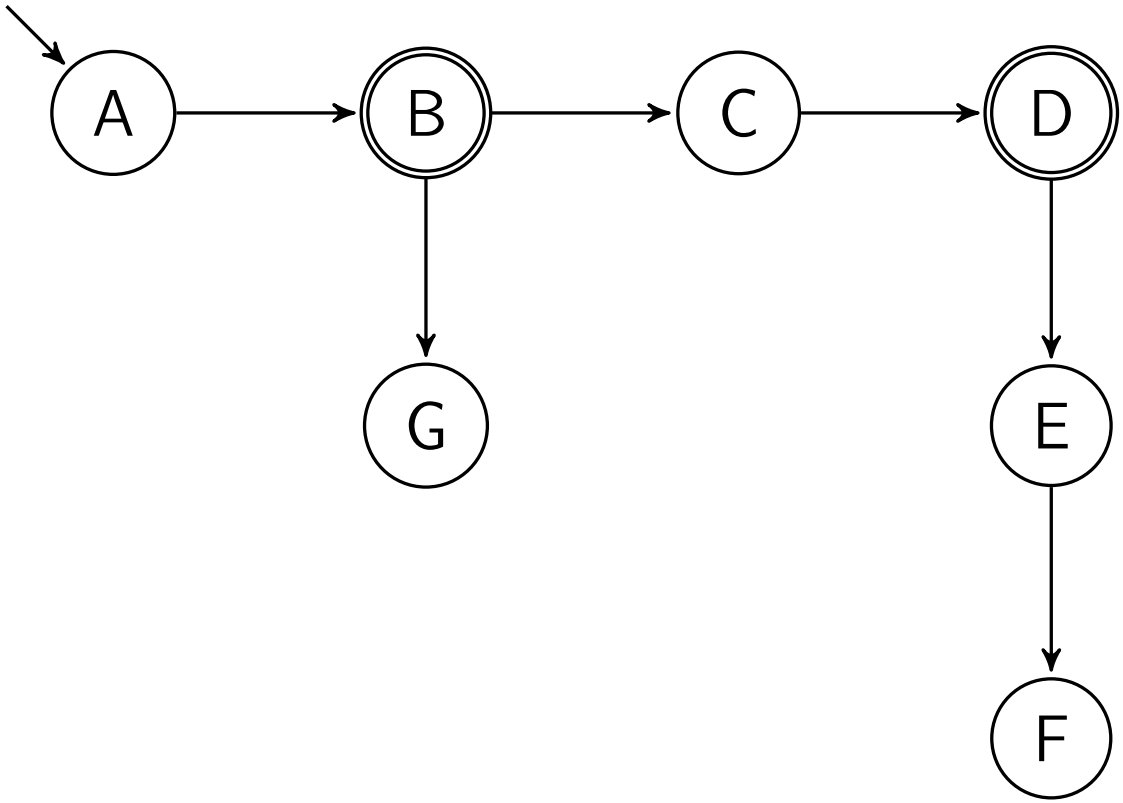
Two Steps – Exemple



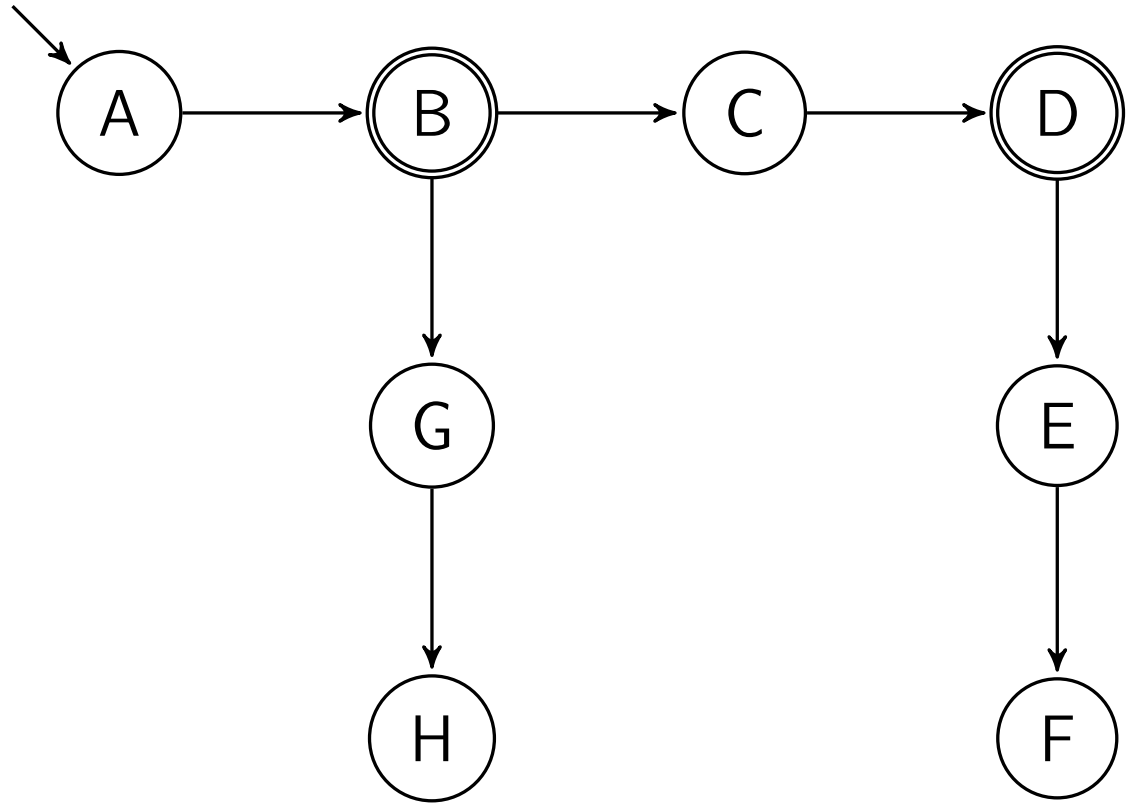
Two Steps – Exemple



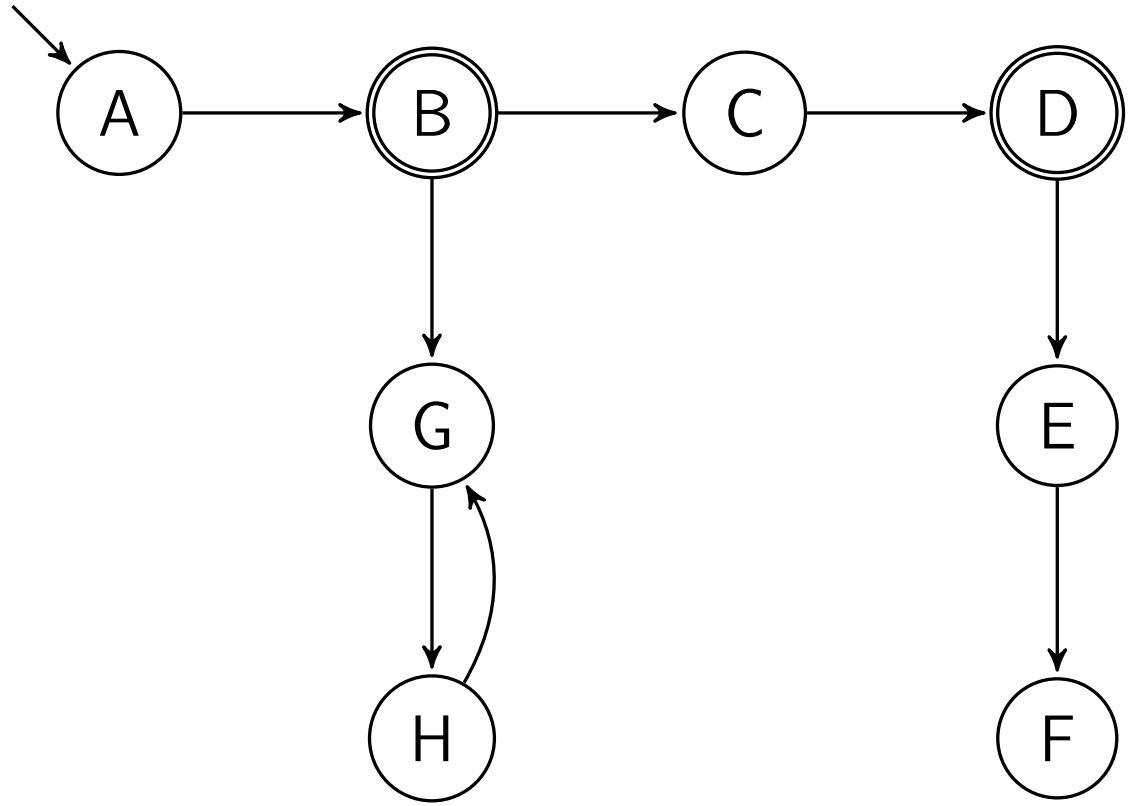
Two Steps – Exemple



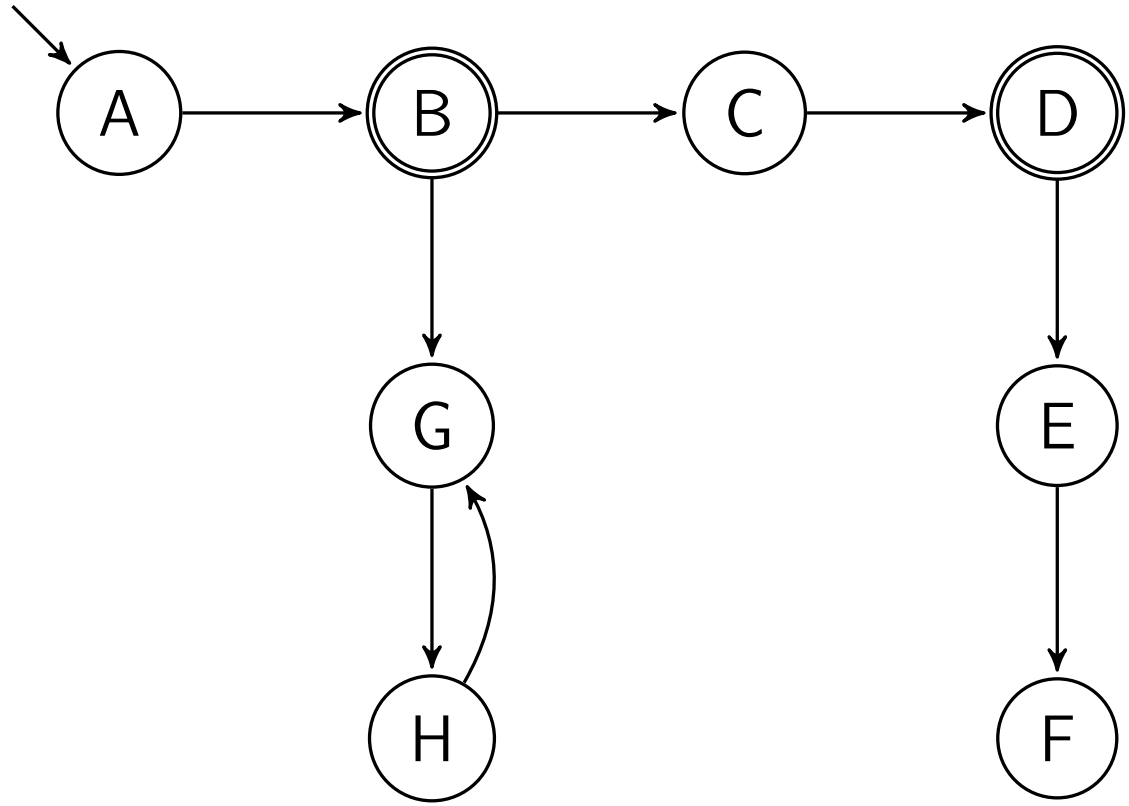
Two Steps – Exemple



Two Steps – Exemple

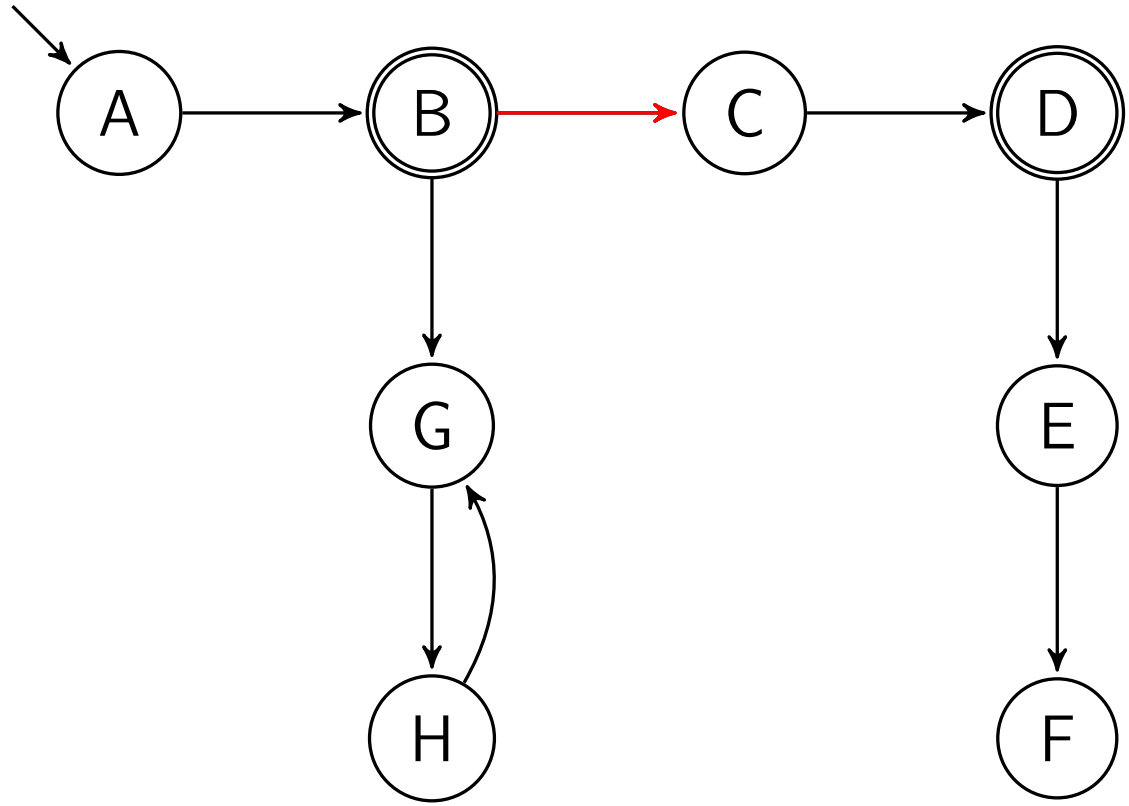


Two Steps – Exemple



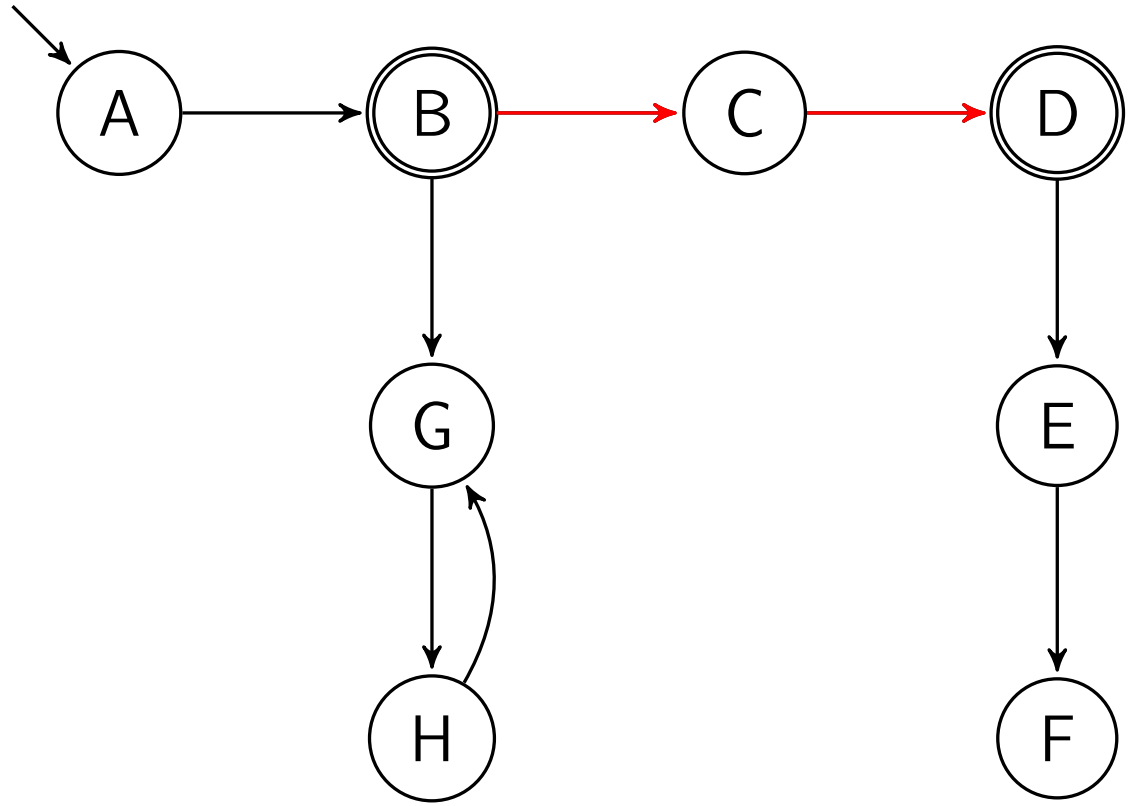
A la fin du premier parcours les état B et D ont été mis de coté.

Two Steps – Exemple



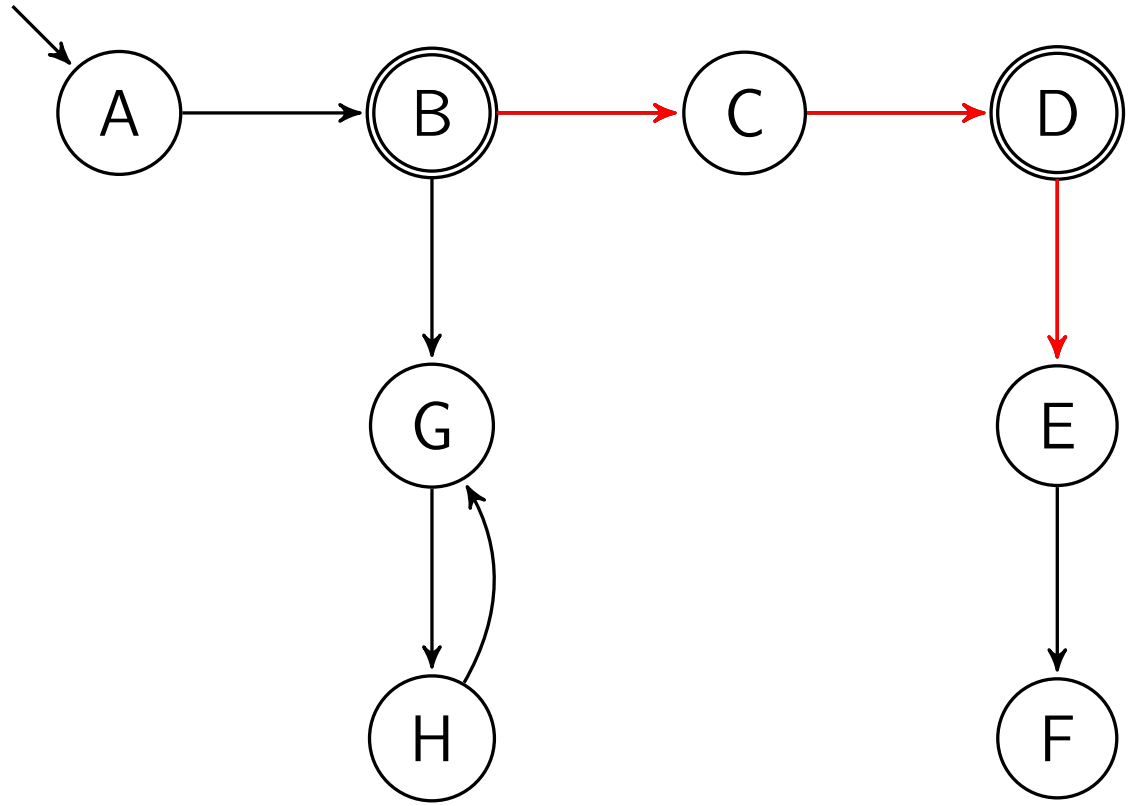
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...

Two Steps – Exemple



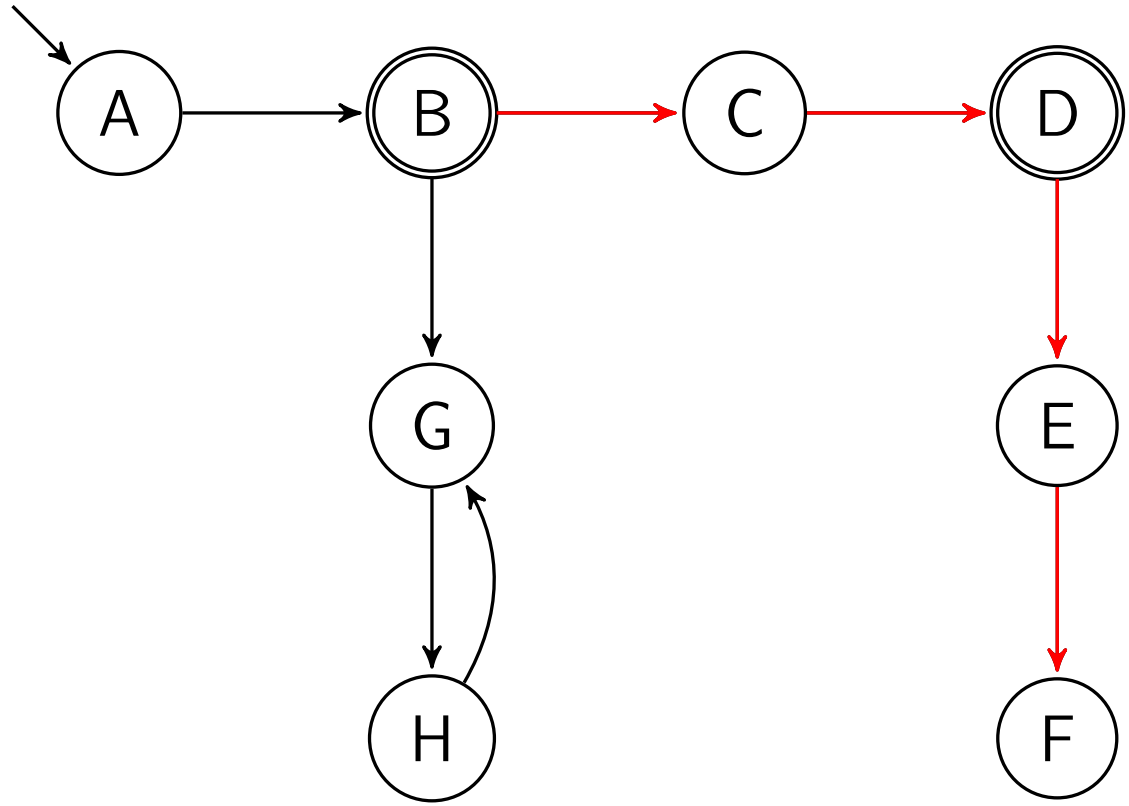
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...

Two Steps – Exemple



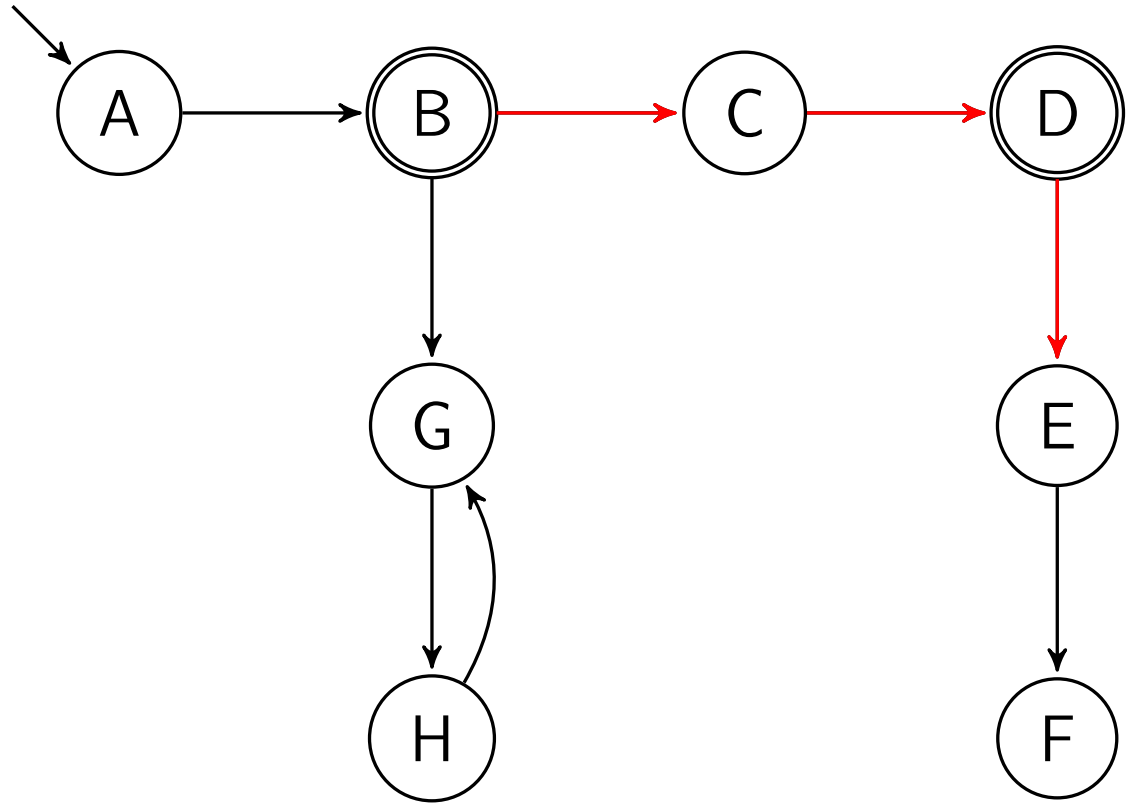
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...

Two Steps – Exemple



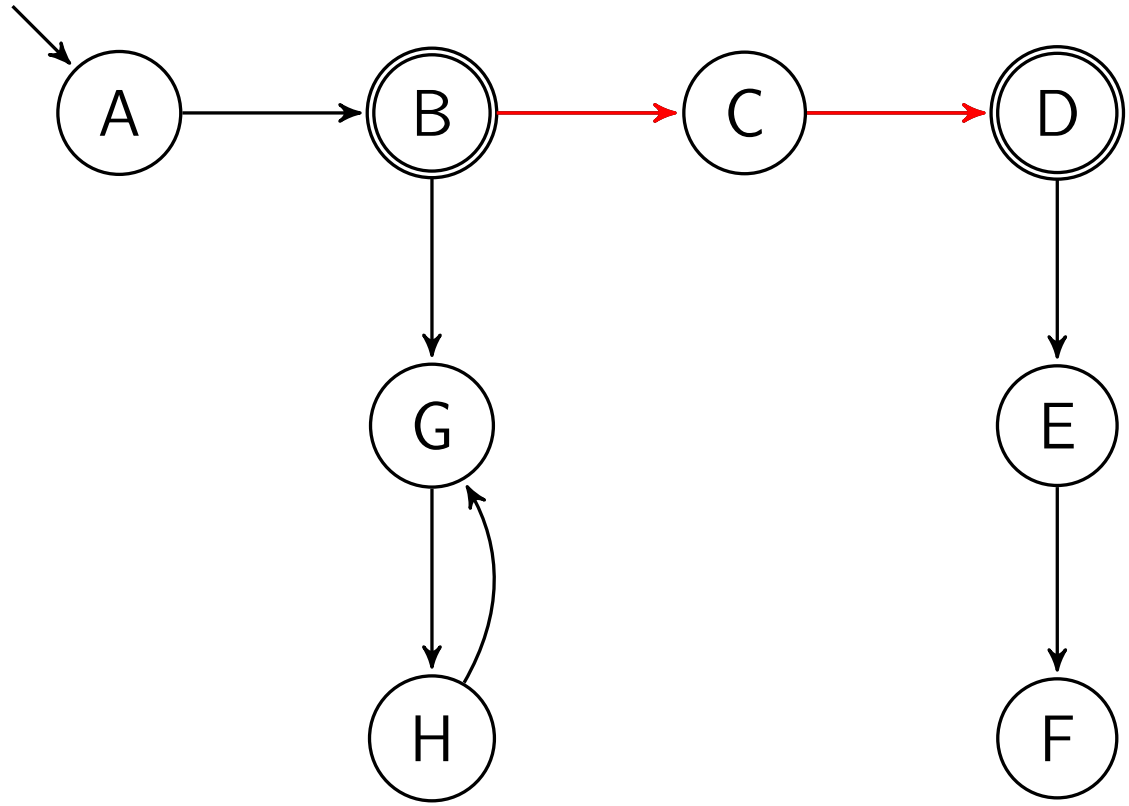
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...

Two Steps – Exemple



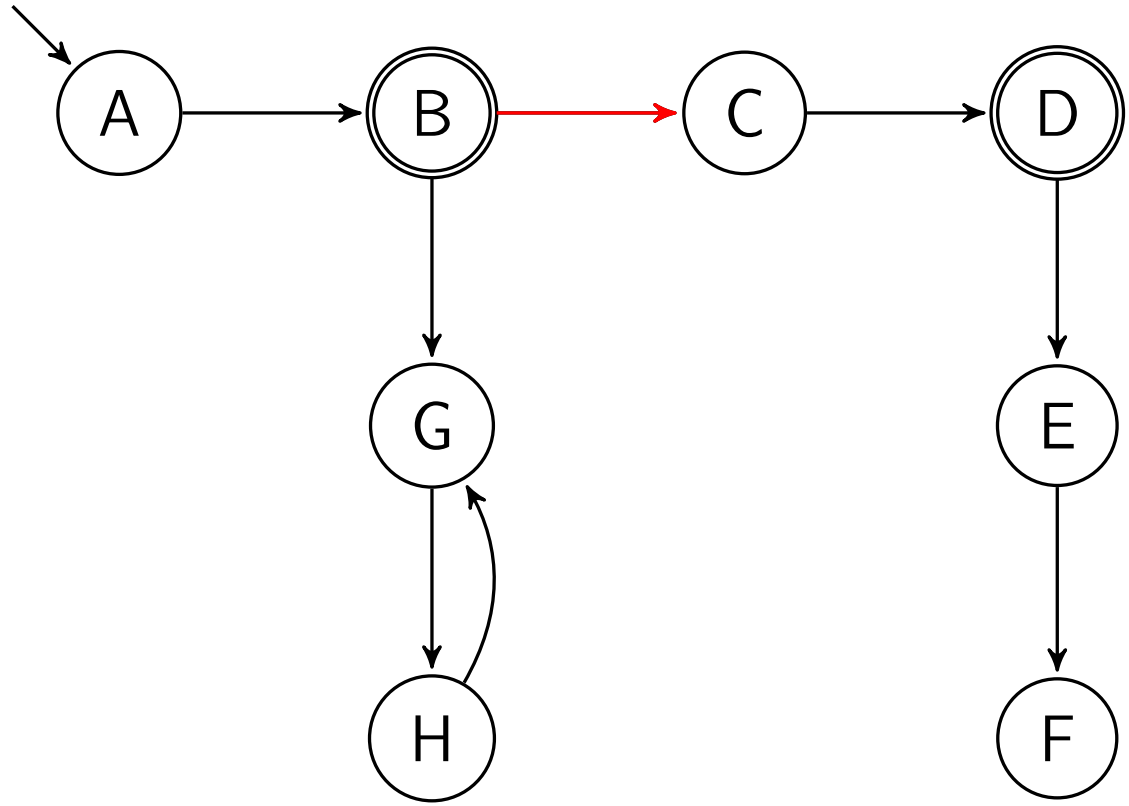
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...

Two Steps – Exemple



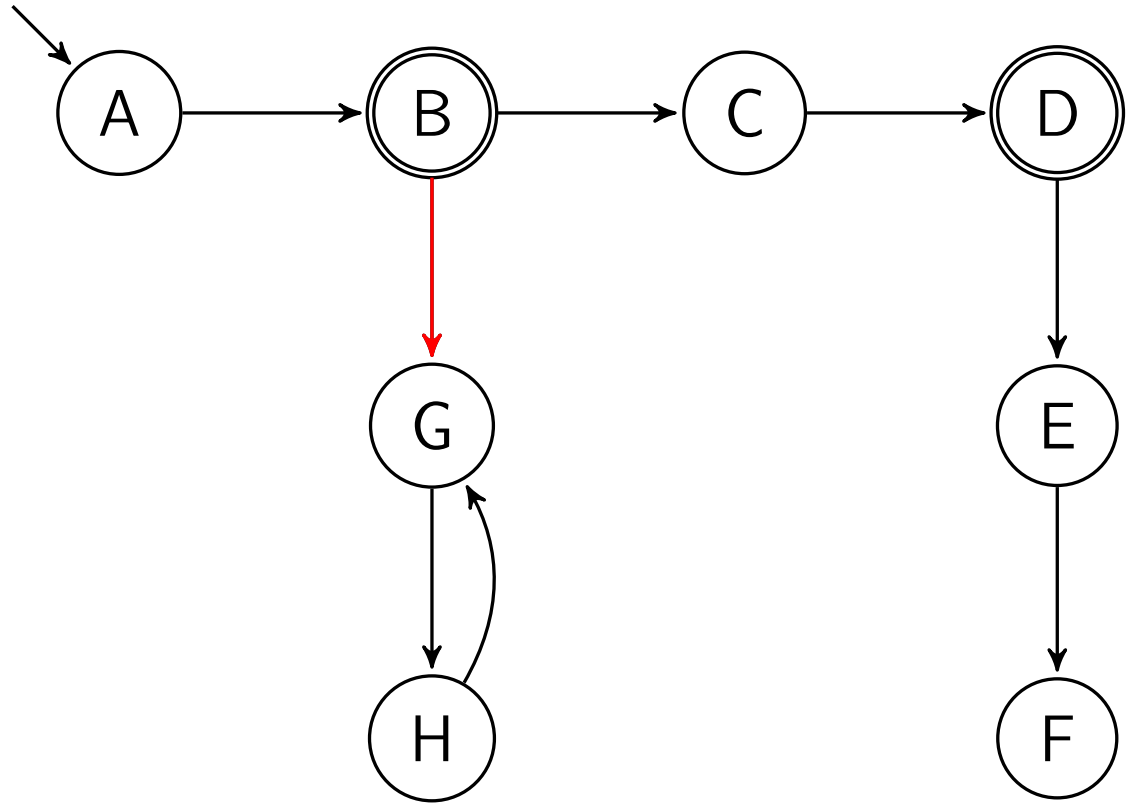
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...

Two Steps – Exemple



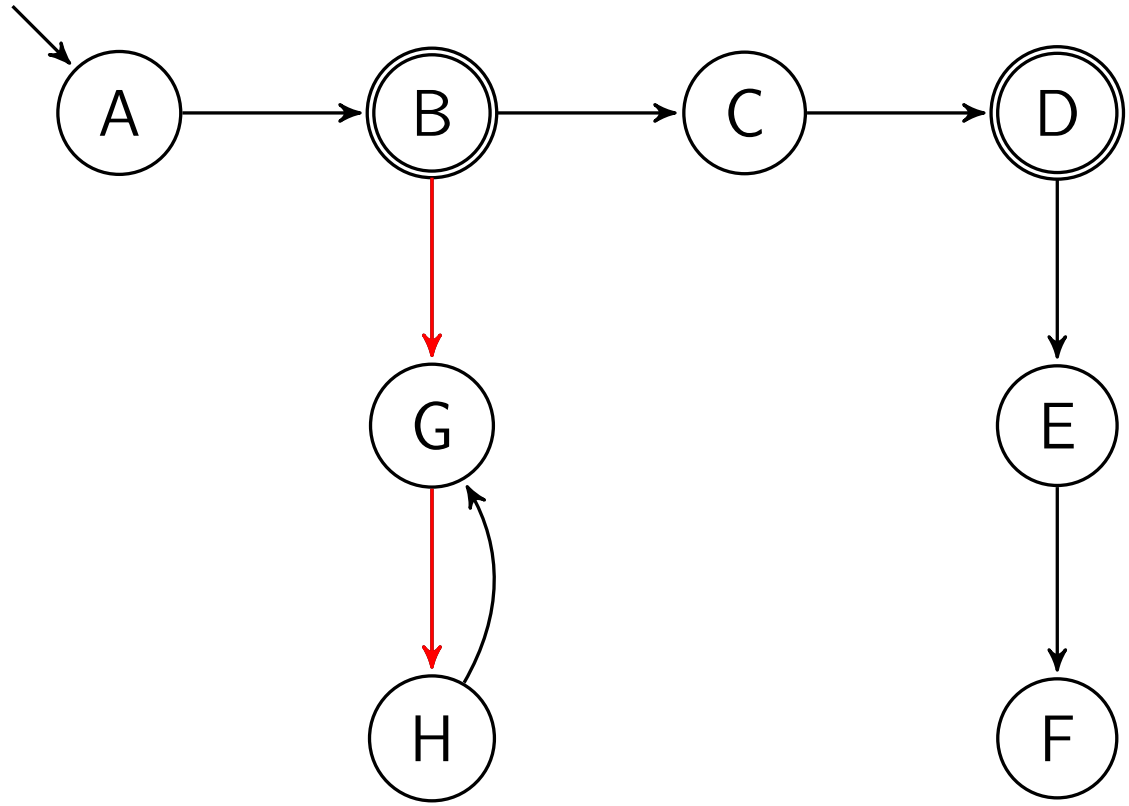
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...

Two Steps – Exemple



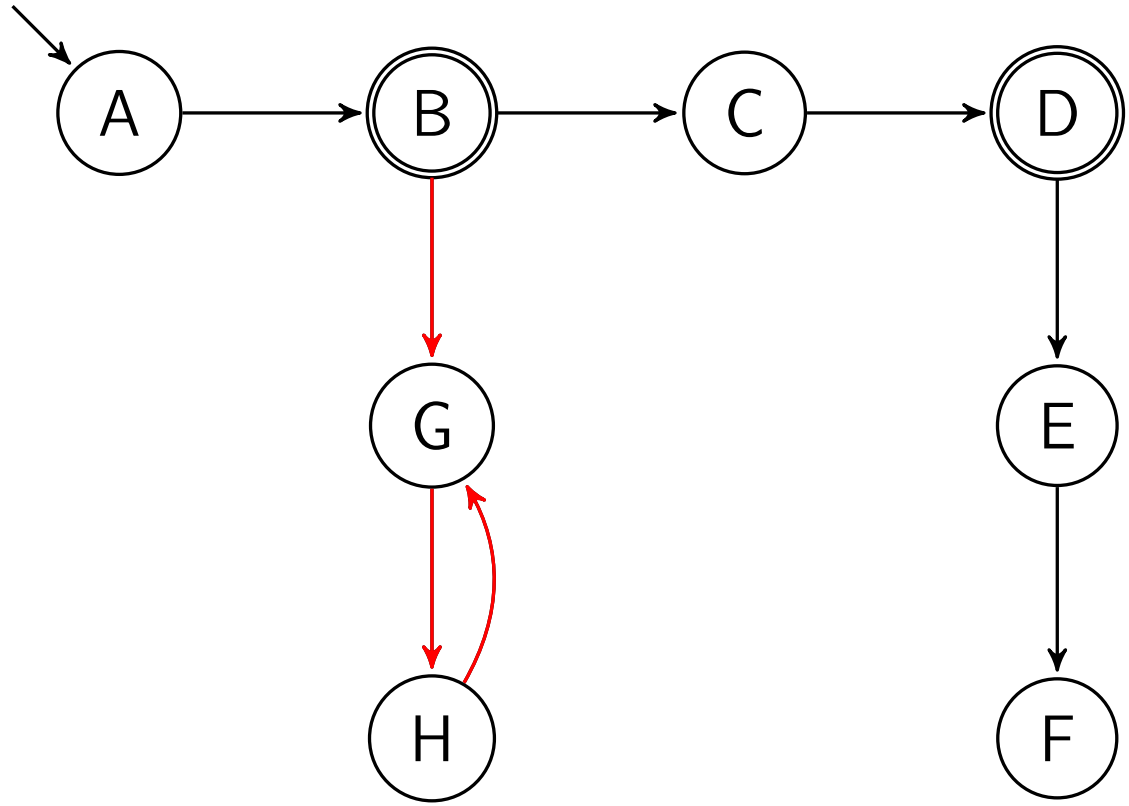
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...

Two Steps – Exemple



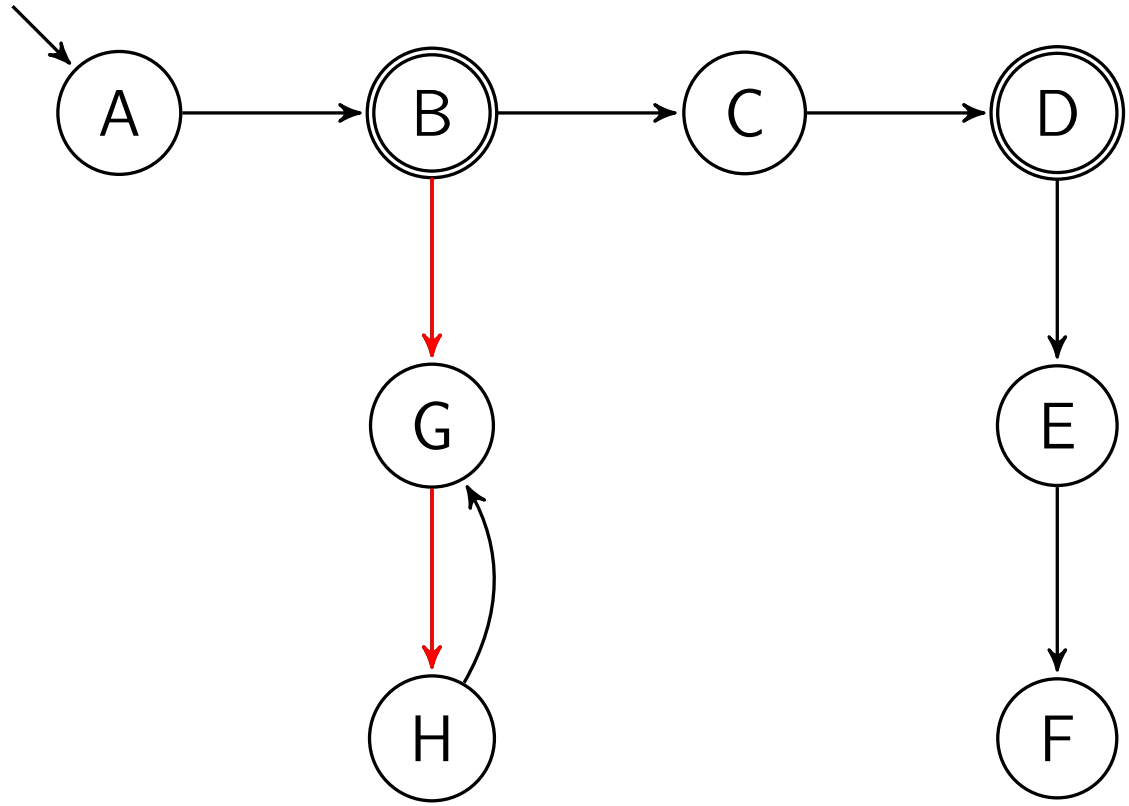
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...

Two Steps – Exemple



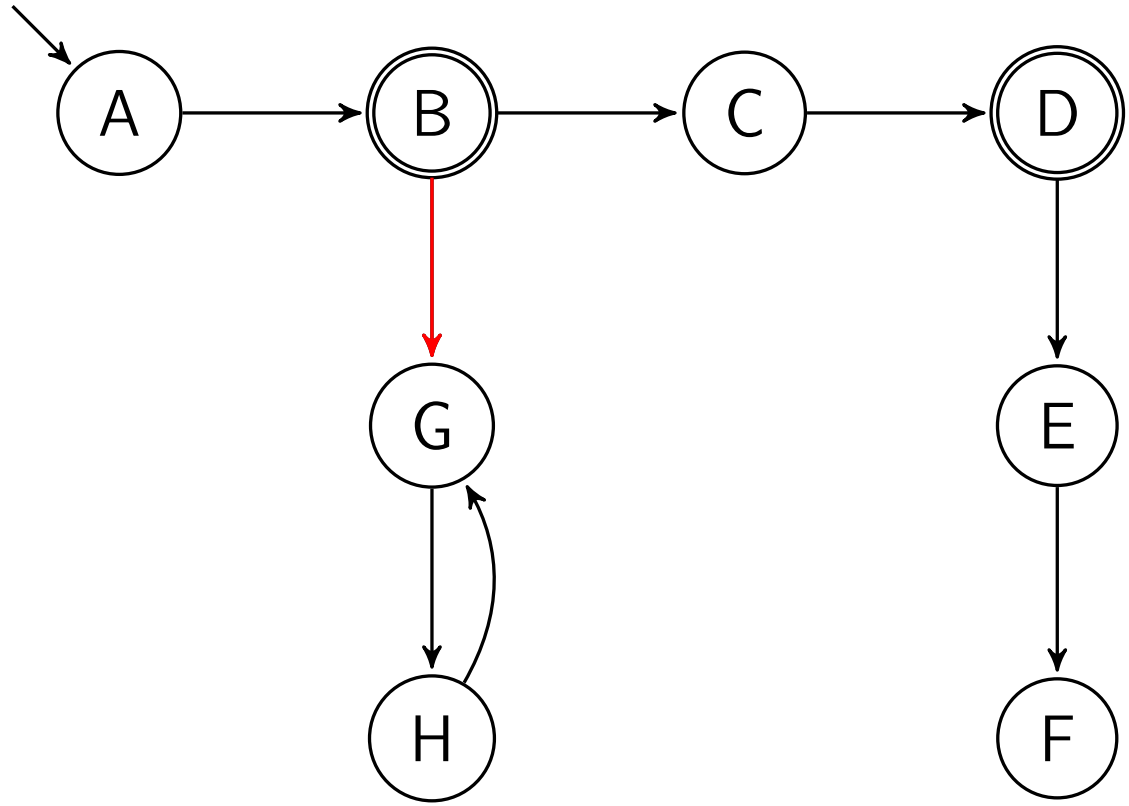
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...

Two Steps – Exemple



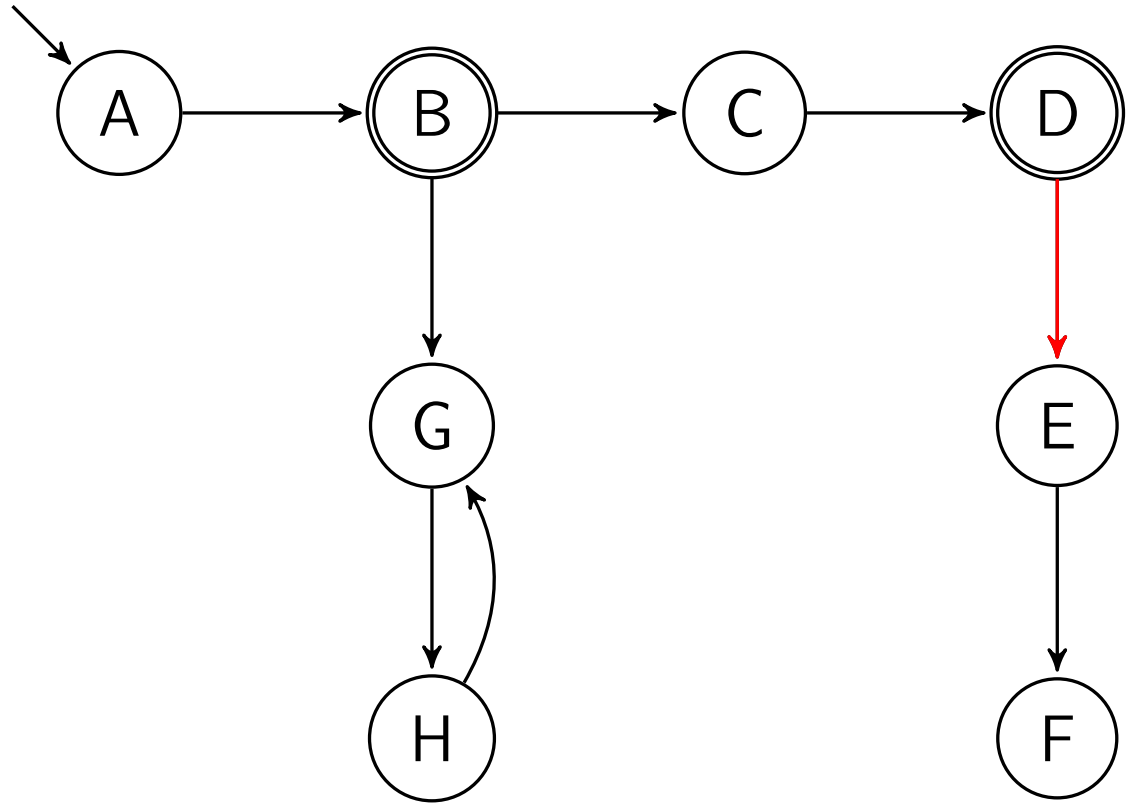
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...

Two Steps – Exemple



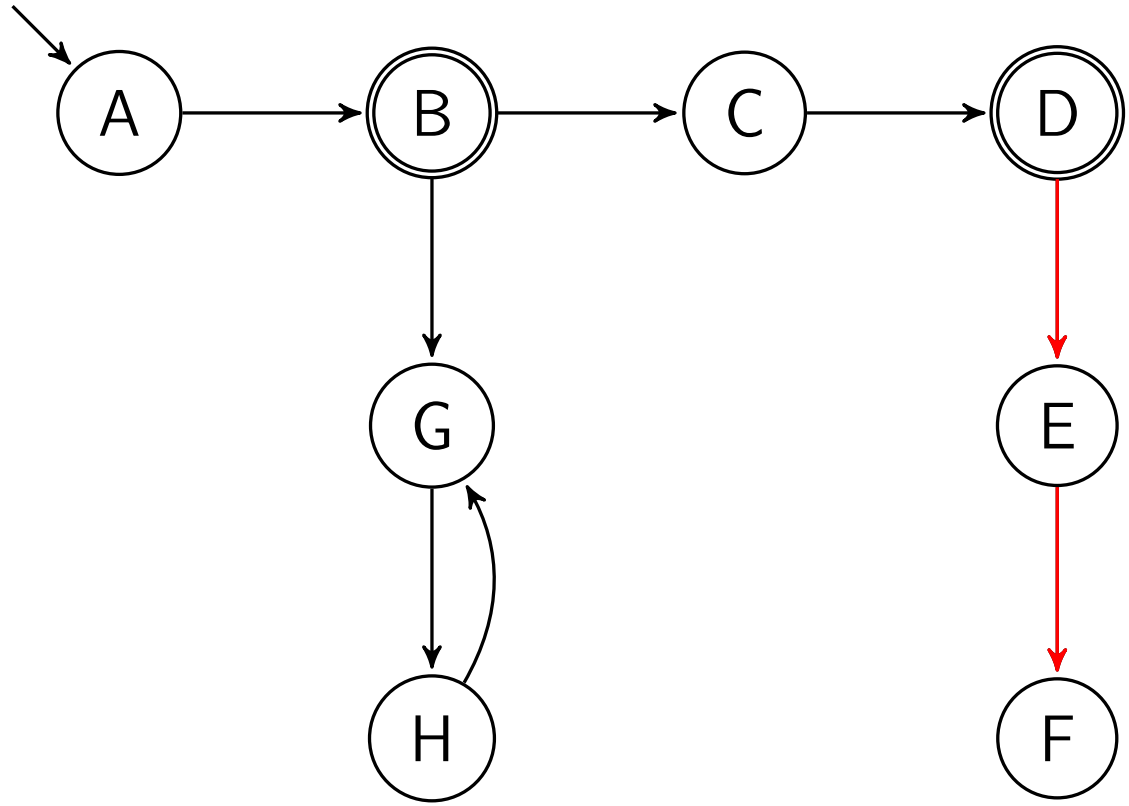
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...

Two Steps – Exemple



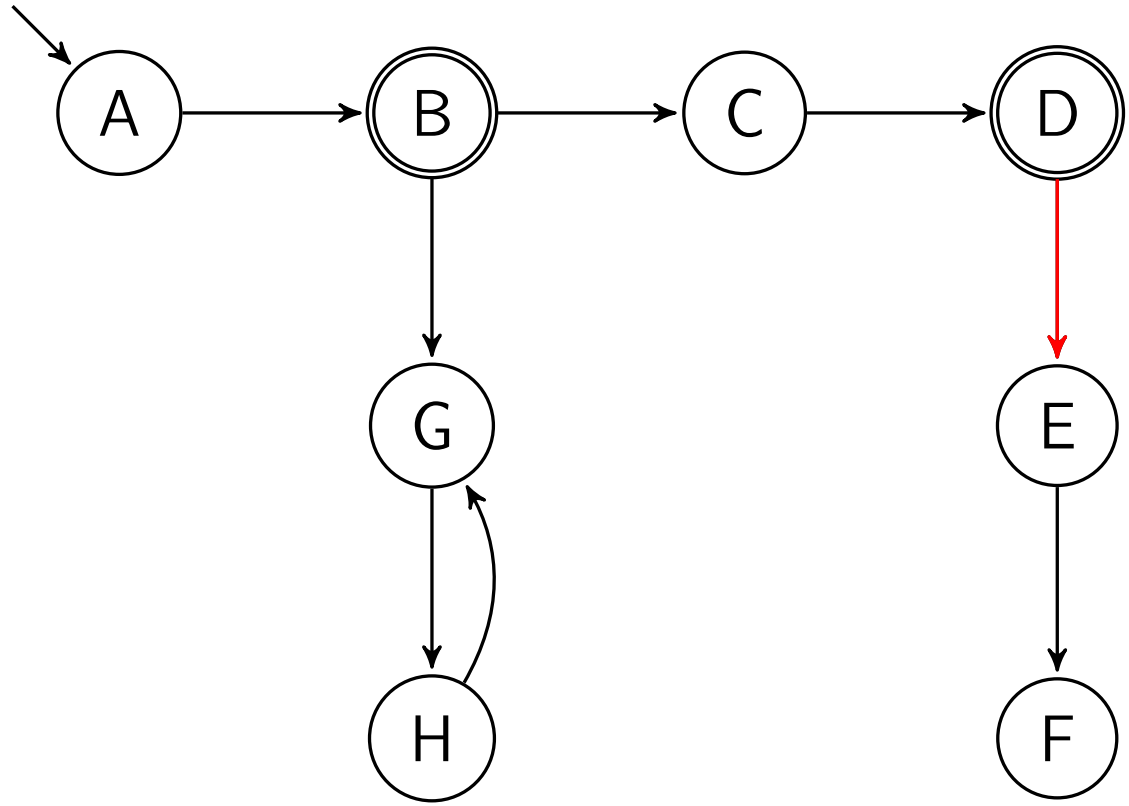
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...
On commence un parcours depuis D ...

Two Steps – Exemple



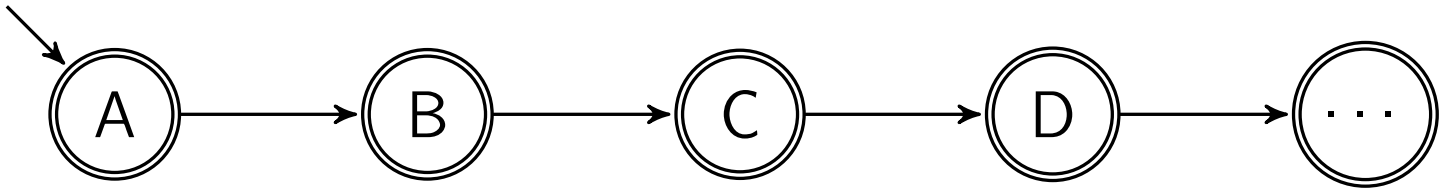
A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...
On commence un parcours depuis D ...

Two Steps – Exemple



A la fin du premier parcours les état B et D ont été mis de coté.
On commence un parcours depuis B ...
On commence un parcours depuis D ...

Two Steps – cas problématique !



Le nombre d'exploration du graphe des états accessibles dépend du nombre d'états acceptants accessibles !

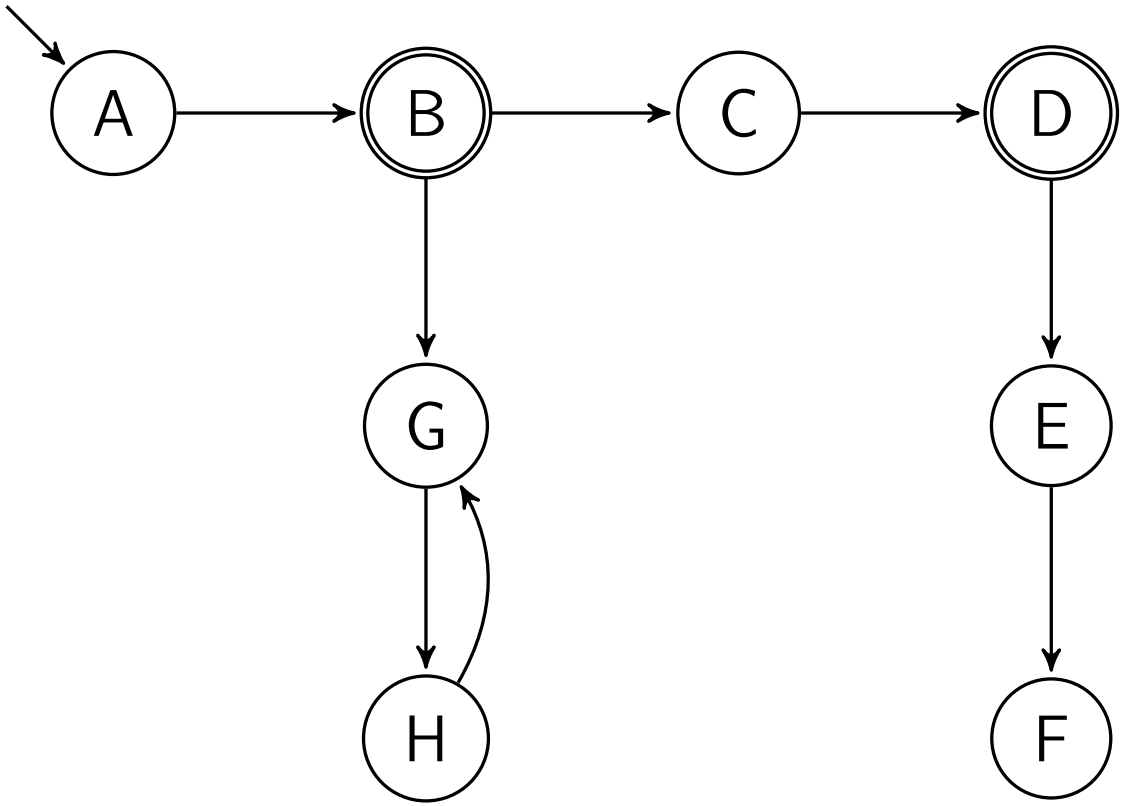
Double DFS

Soit n_0 et n_1 deux états acceptants tels que le DFS number de n_1 soit supérieur au DFS number de n_0. Autrement dit n_0 est vu plus tôt que n_1 par le DFS.

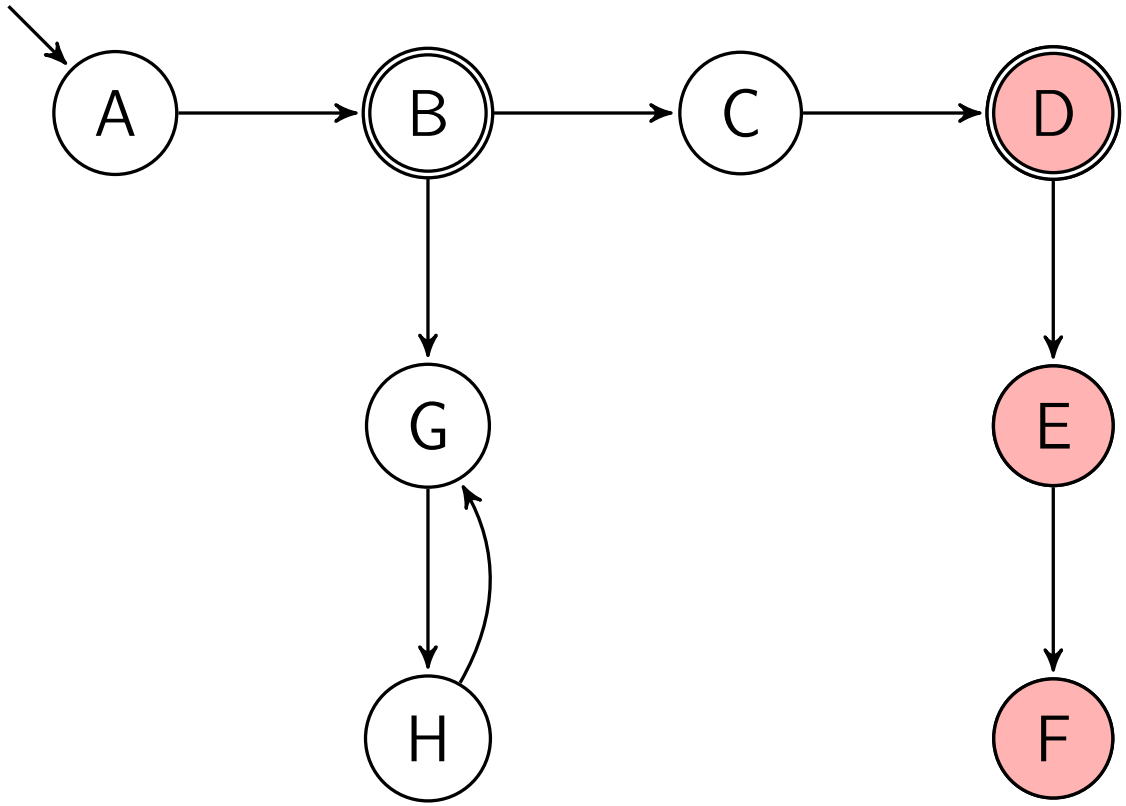
Si il n'existe pas de cycle acceptant autour de n_1, alors il n'existe pas de cycle acceptant autour de n_0 qui contienne un successeur (direct ou indirect) de n_1.

On va maintenant lancer les second parcours DFS dans l'ordre postfixe !

Explication Double DFS

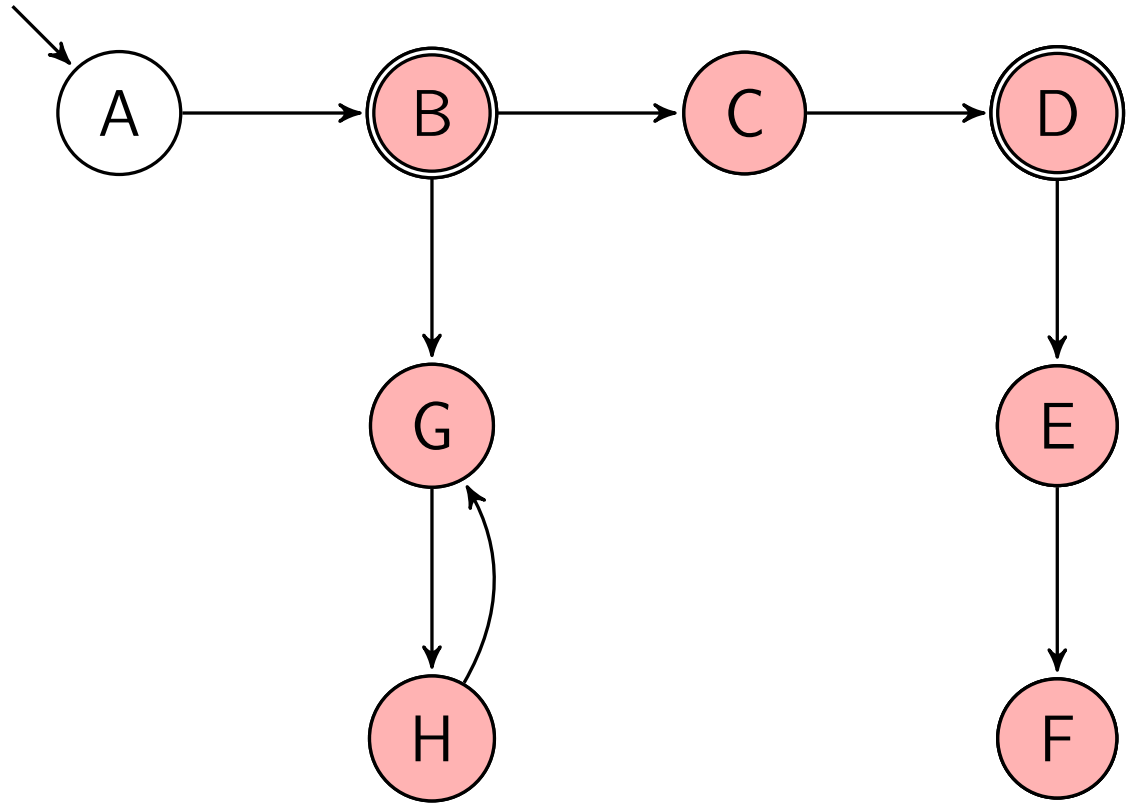


Explication Double DFS



On commence par lancer le DFS à partir de D, et on marque tous les états accessibles depuis D ...

Explication Double DFS



On commence par lancer le DFS à partir de D, et on marque tous les états accessibles depuis D . . . On lance ensuite le DFS à partir de B, et on marque tous les états accessibles depuis D et les états rouges sont ignorés

Double DFS – analyse

Le problème est maintenant linéaire en temps et en mémoire !

Comment reporter les contre-exemples ?

Un contre exemple est composé :

- d'un préfixe
- d'un cycle acceptant

Nested-DFS (NDFS) 1/2

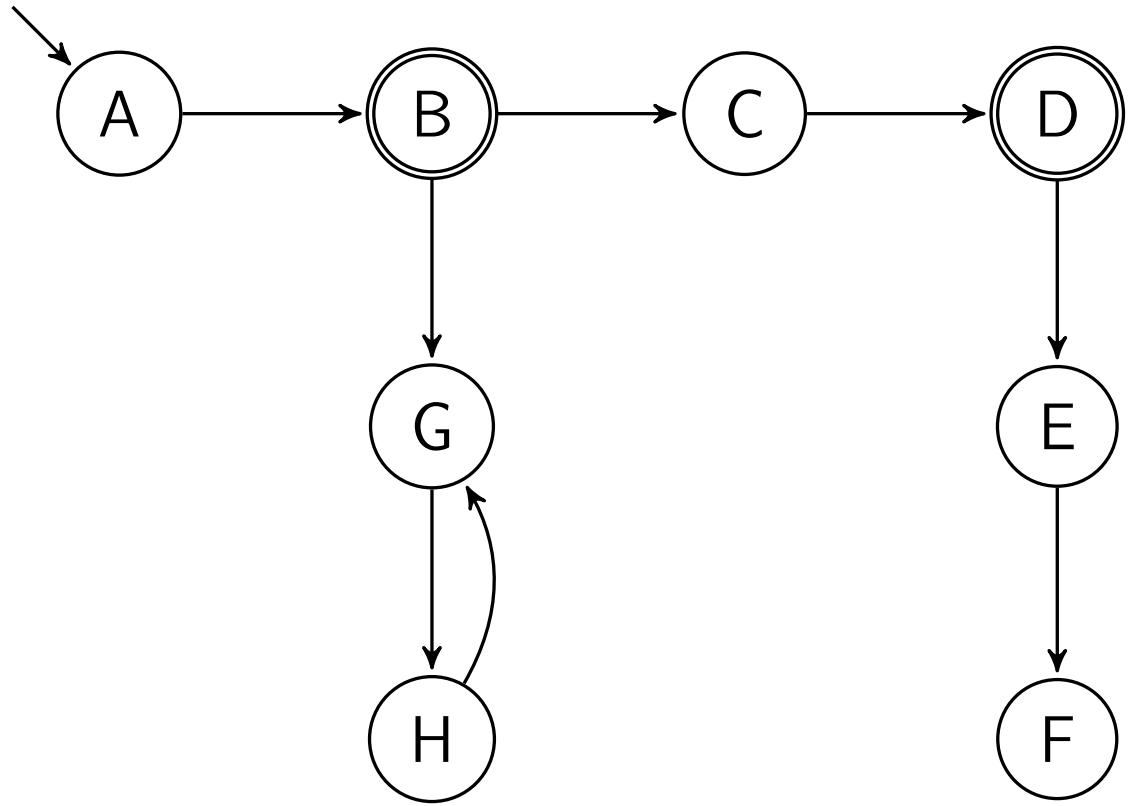
Idée :

- On commence un premier DFS (DFS bleu)
- Lorsque l'on backtrack un état acceptant, on arrête le premier DFS et on lance un second DFS (DFS rouge)
- Si l'on trouve un contre-exemple, c'est terminé !
- Sinon, on continue le premier DFS lorsque le second DFS se termine

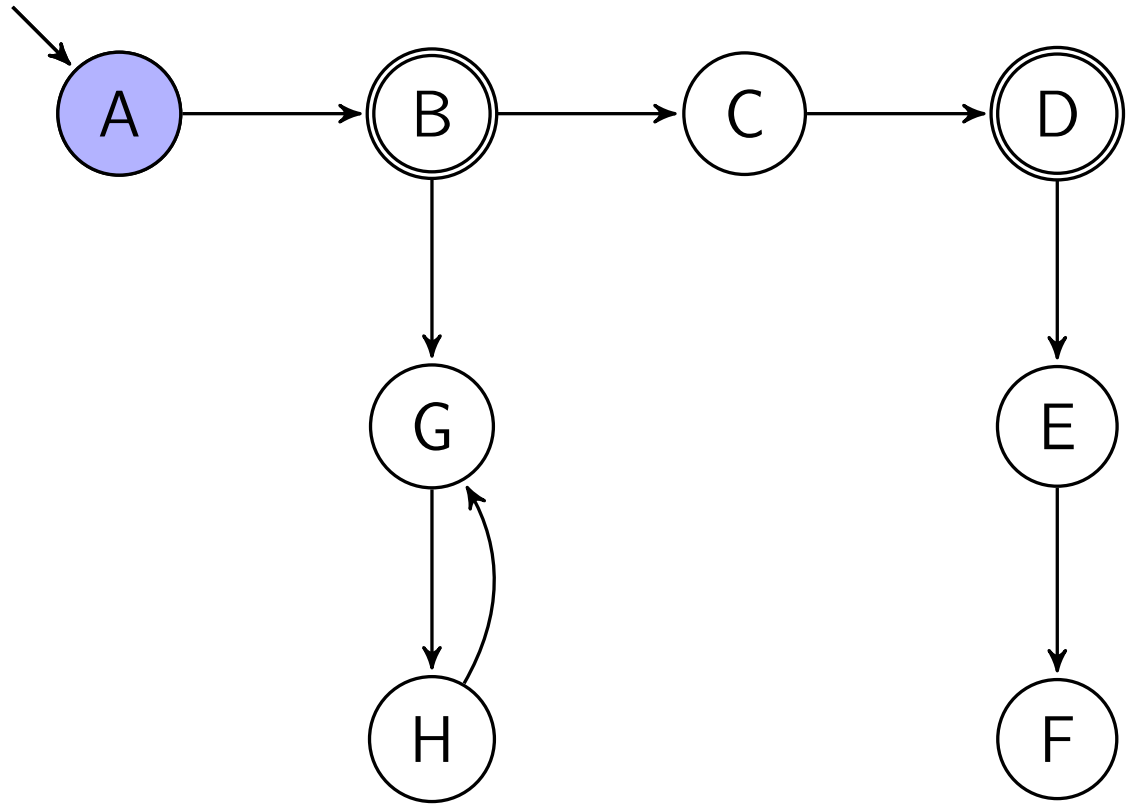
On a maintenant besoin de deux tables de hachages, une pour le premier DFS, une pour le second.

Un contre exemple est maintenant facile à trouver : le préfixe est sur la pile du premier DFS, le cycle sur la pile du second DFS.

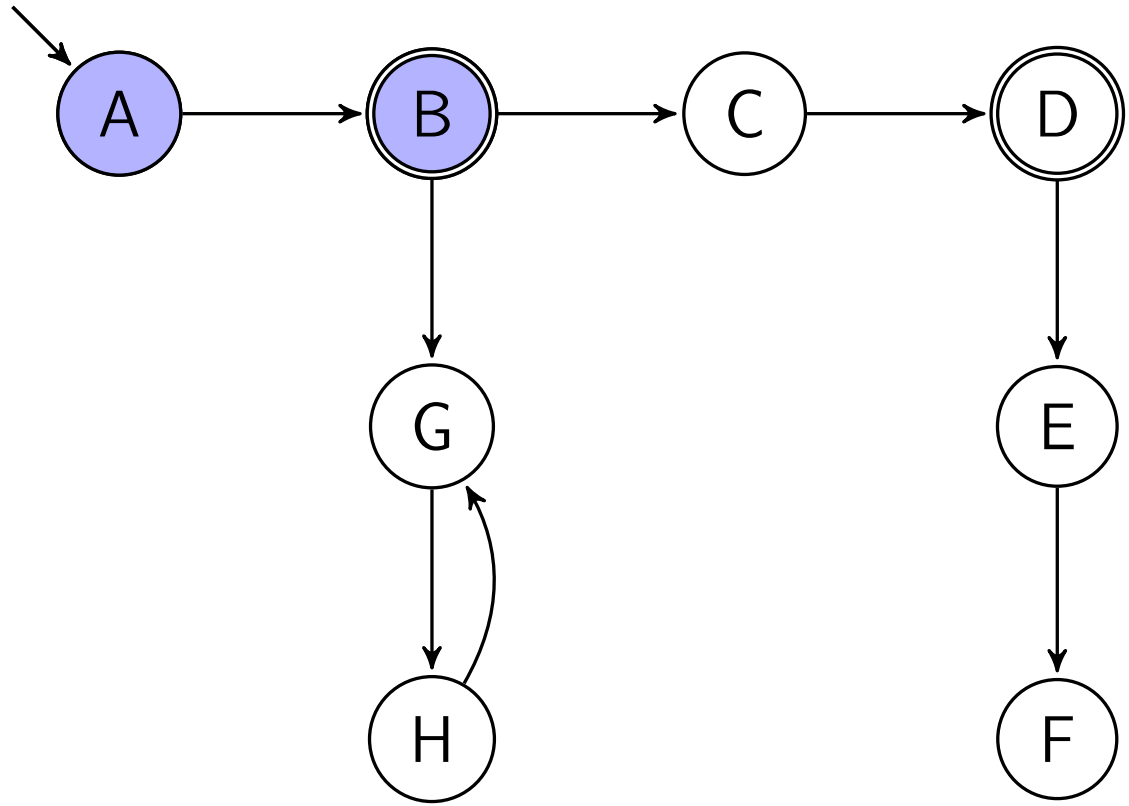
NDFS exemple



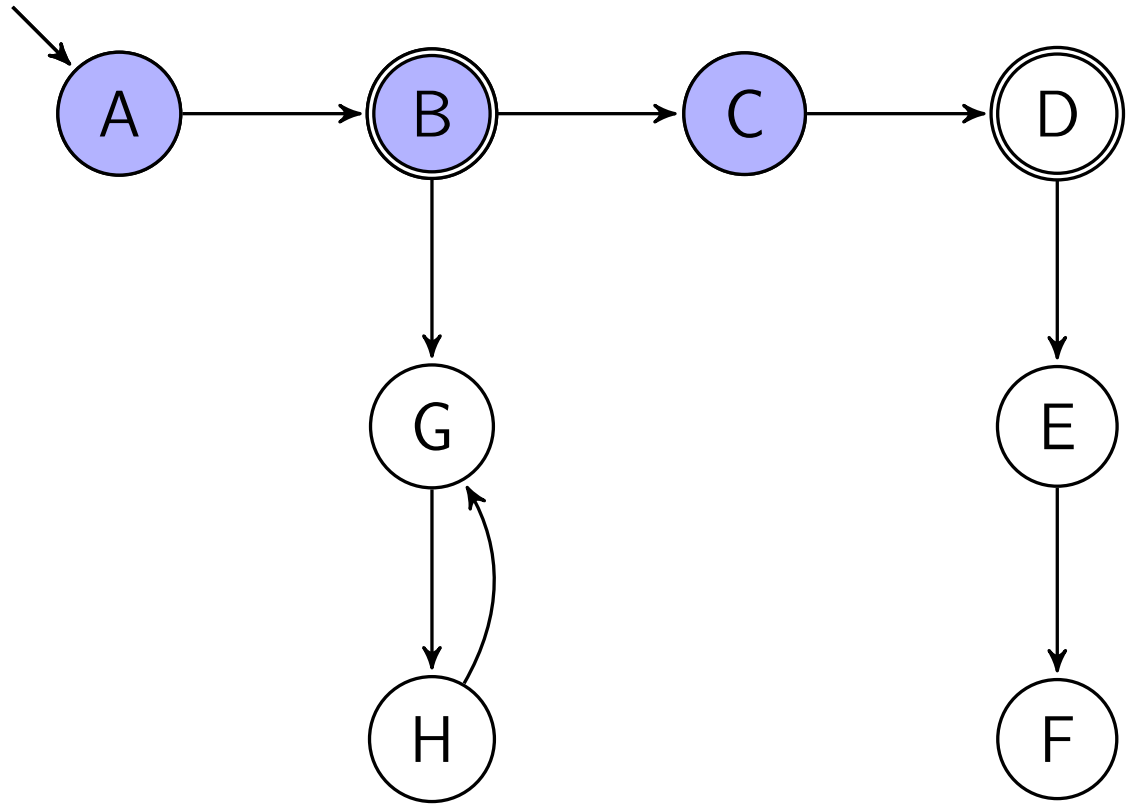
NDFS exemple



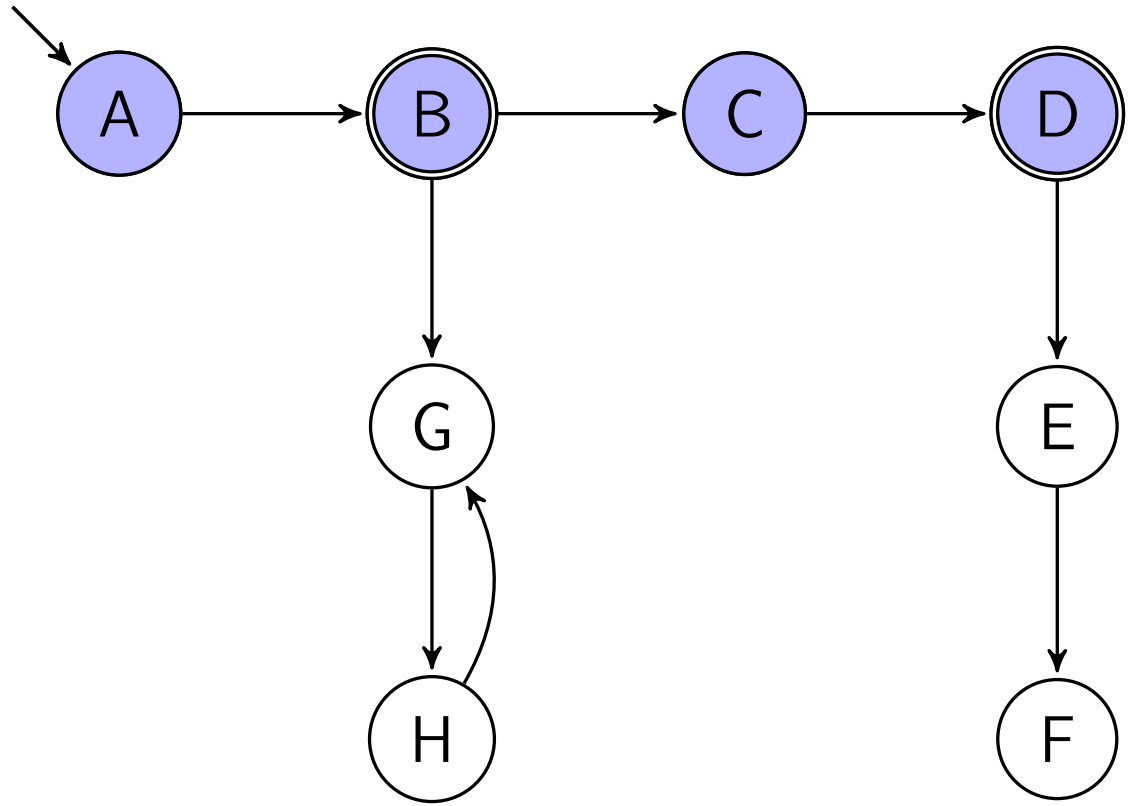
NDFS exemple



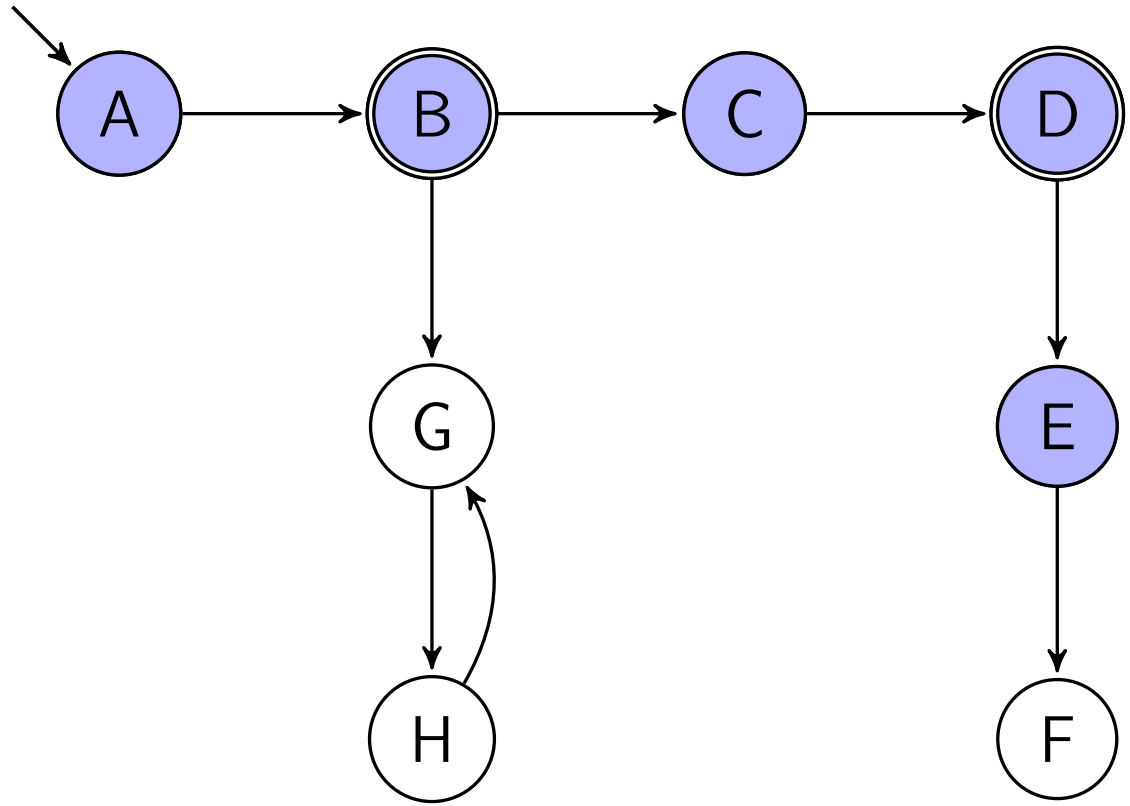
NDFS exemple



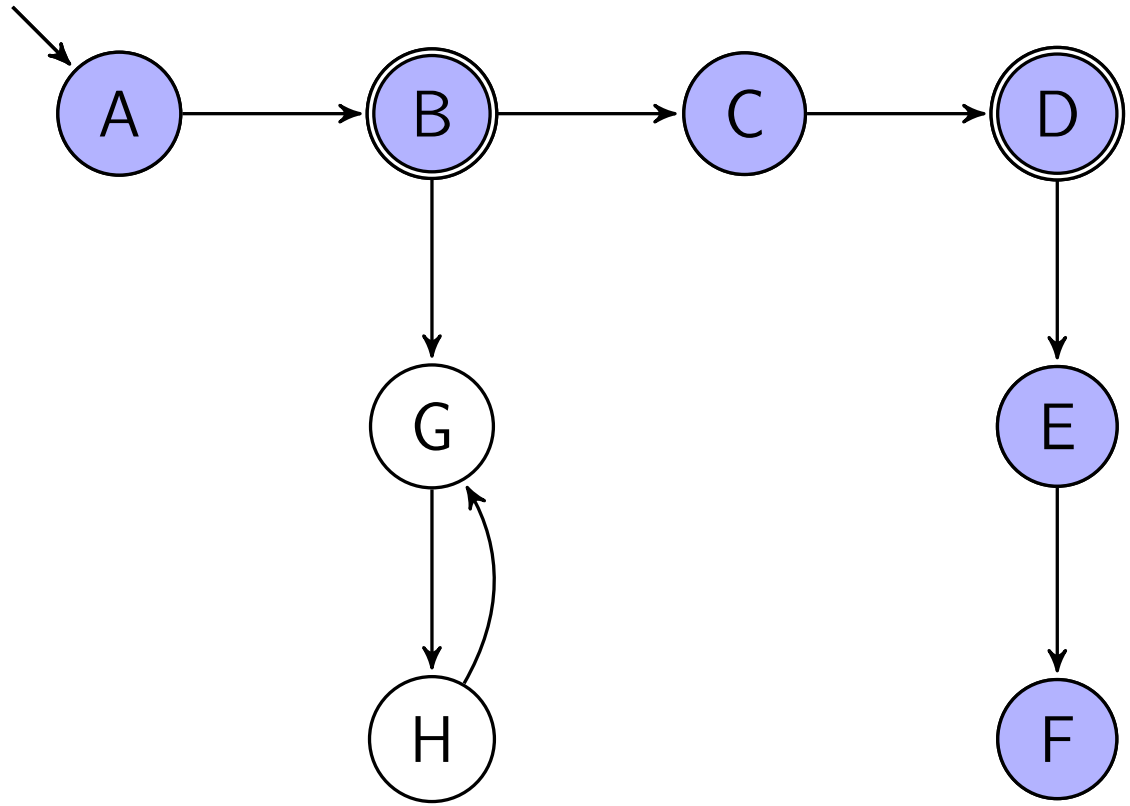
NDFS exemple



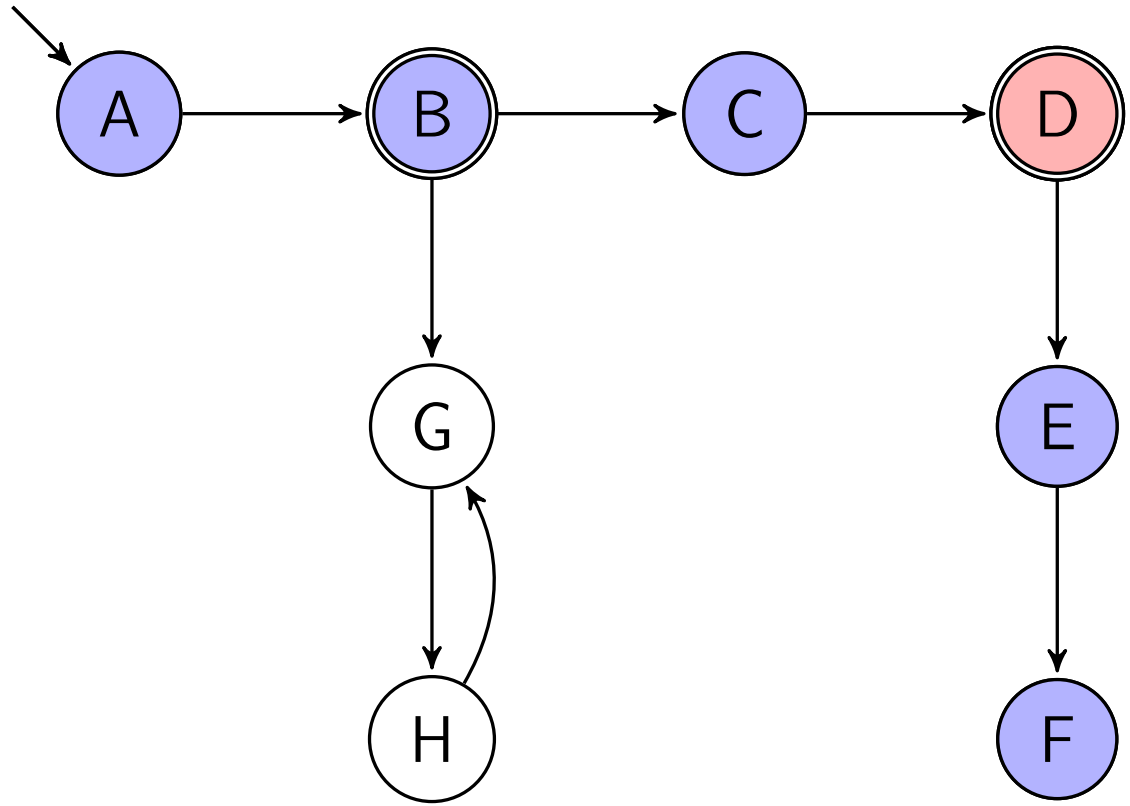
NDFS exemple



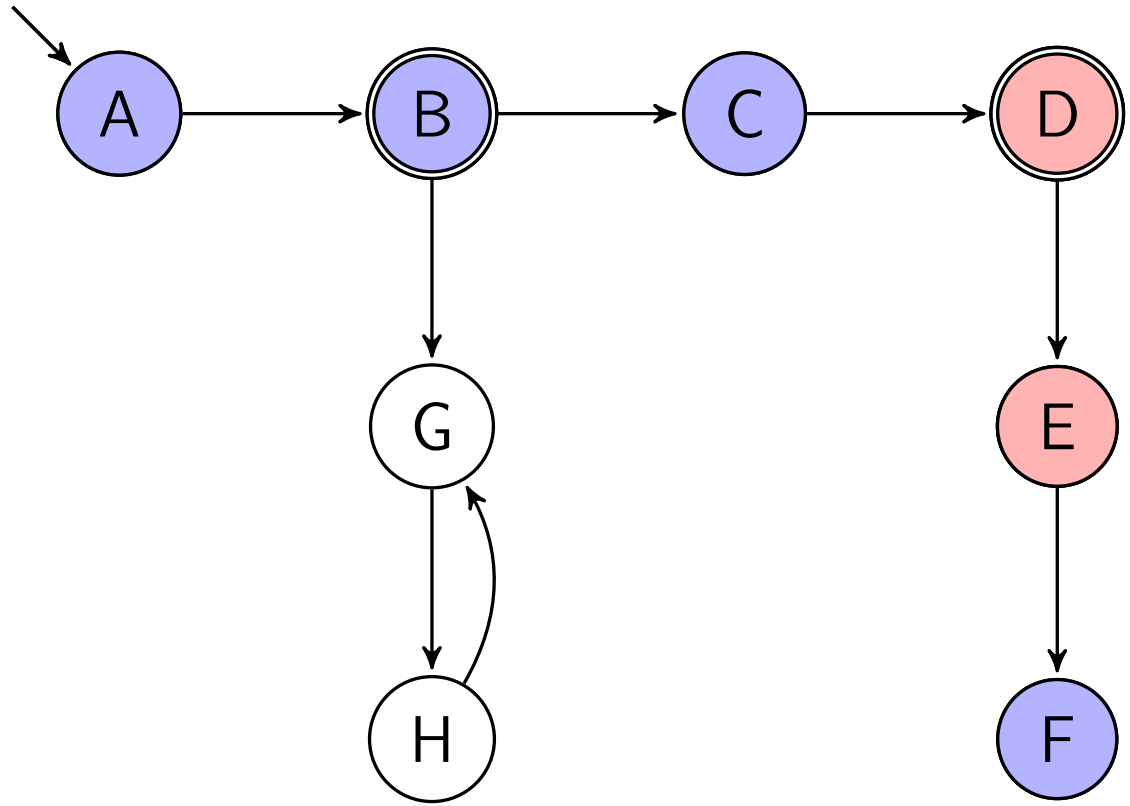
NDFS exemple



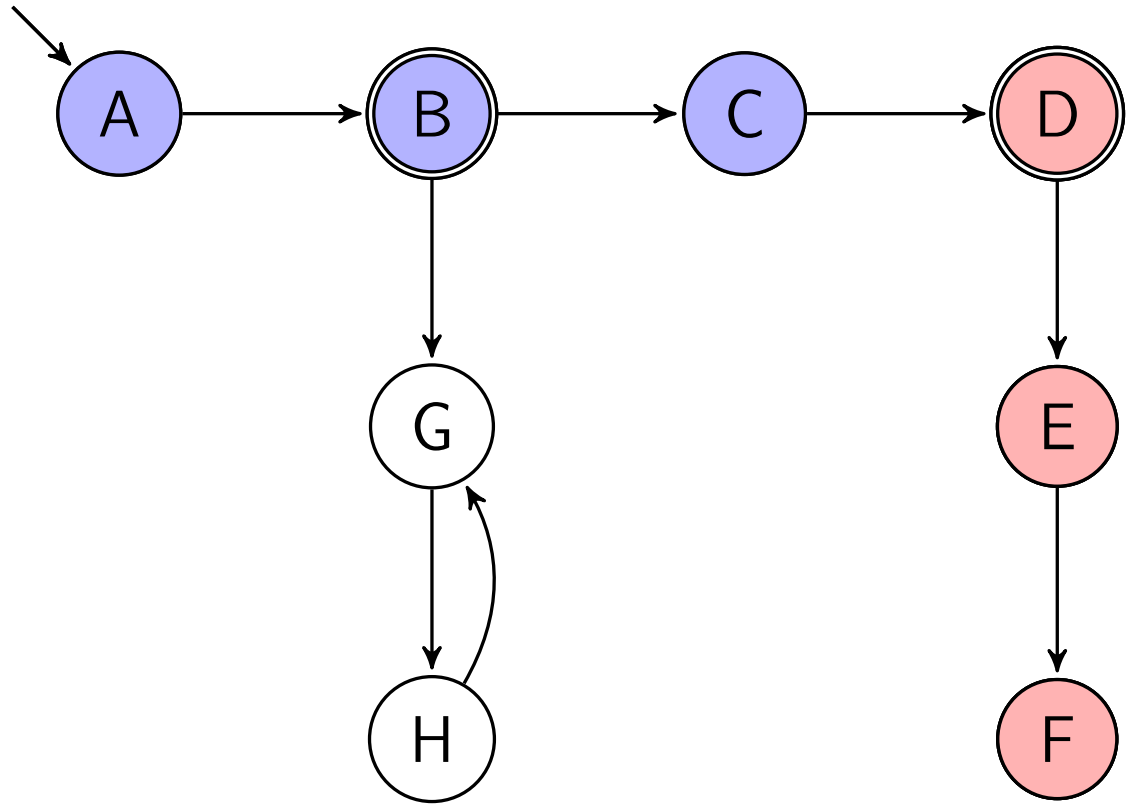
NDFS exemple



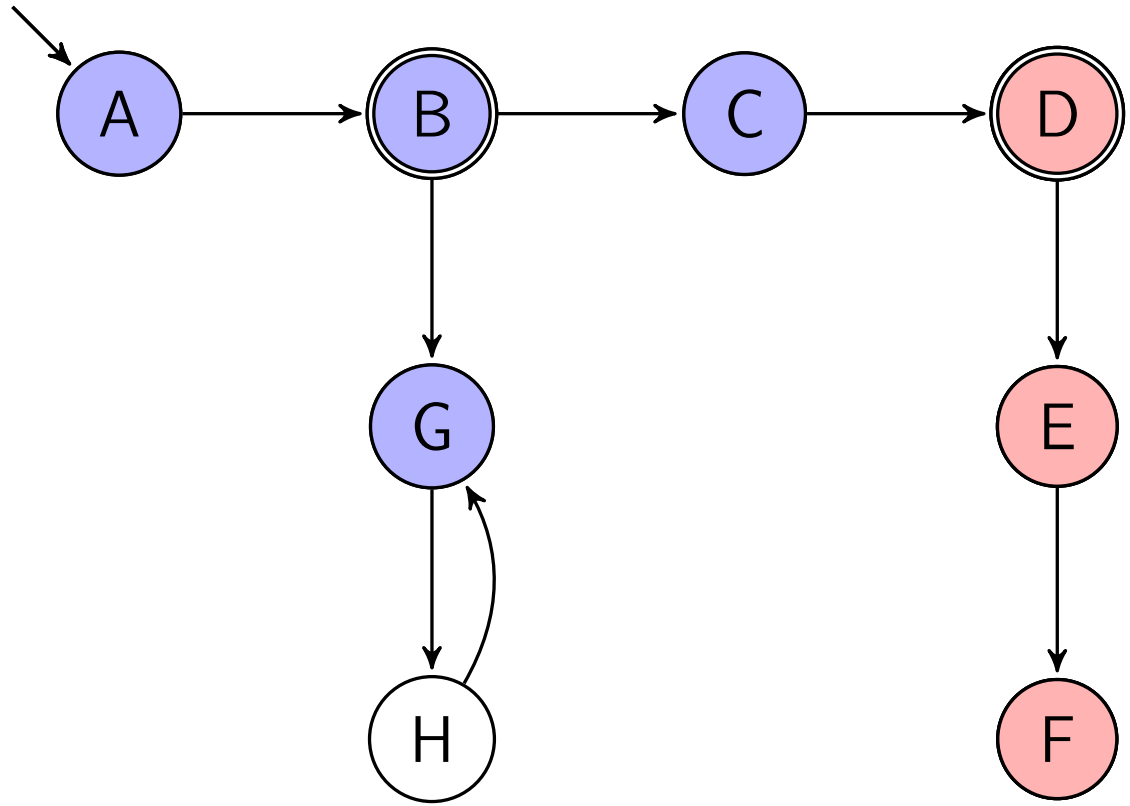
NDFS exemple



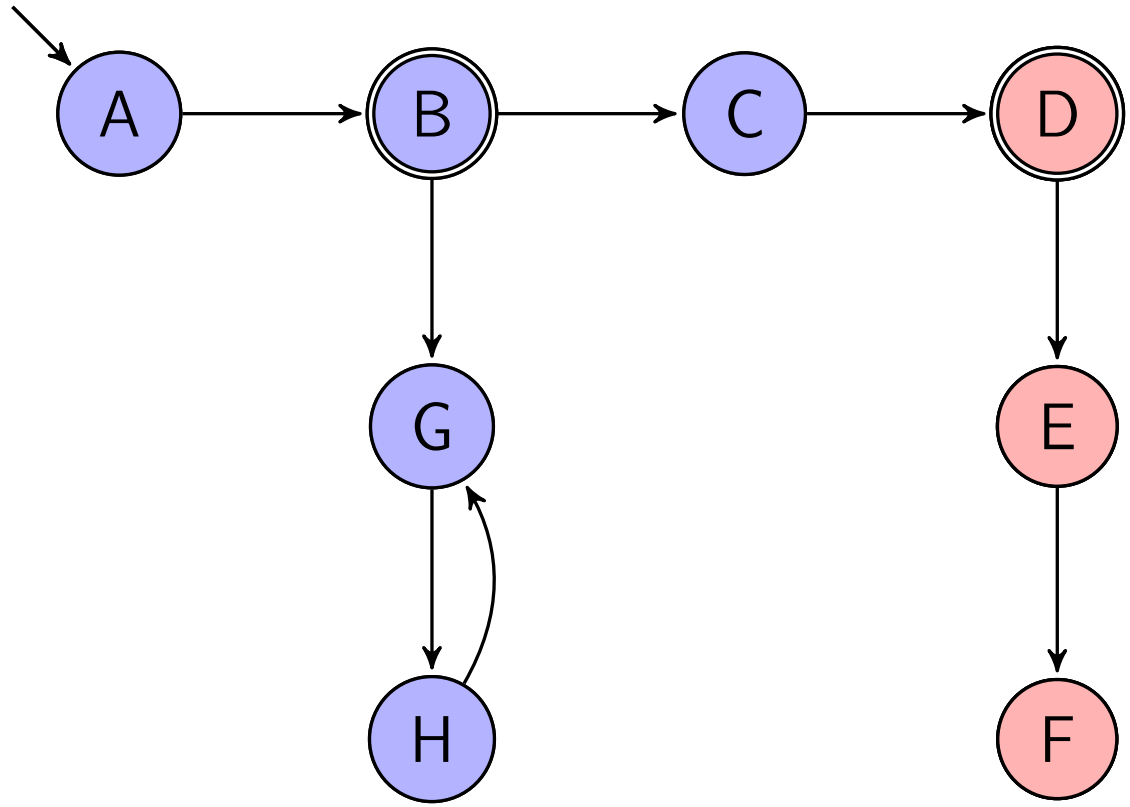
NDFS exemple



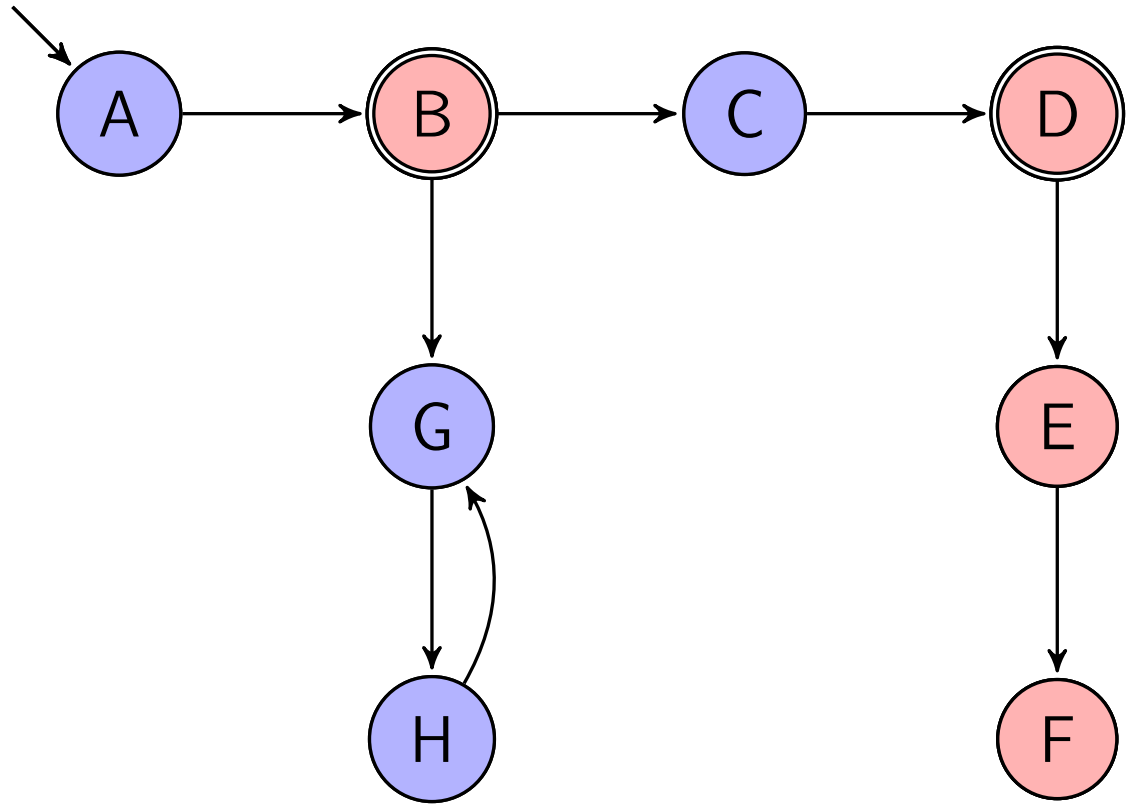
NDFS exemple



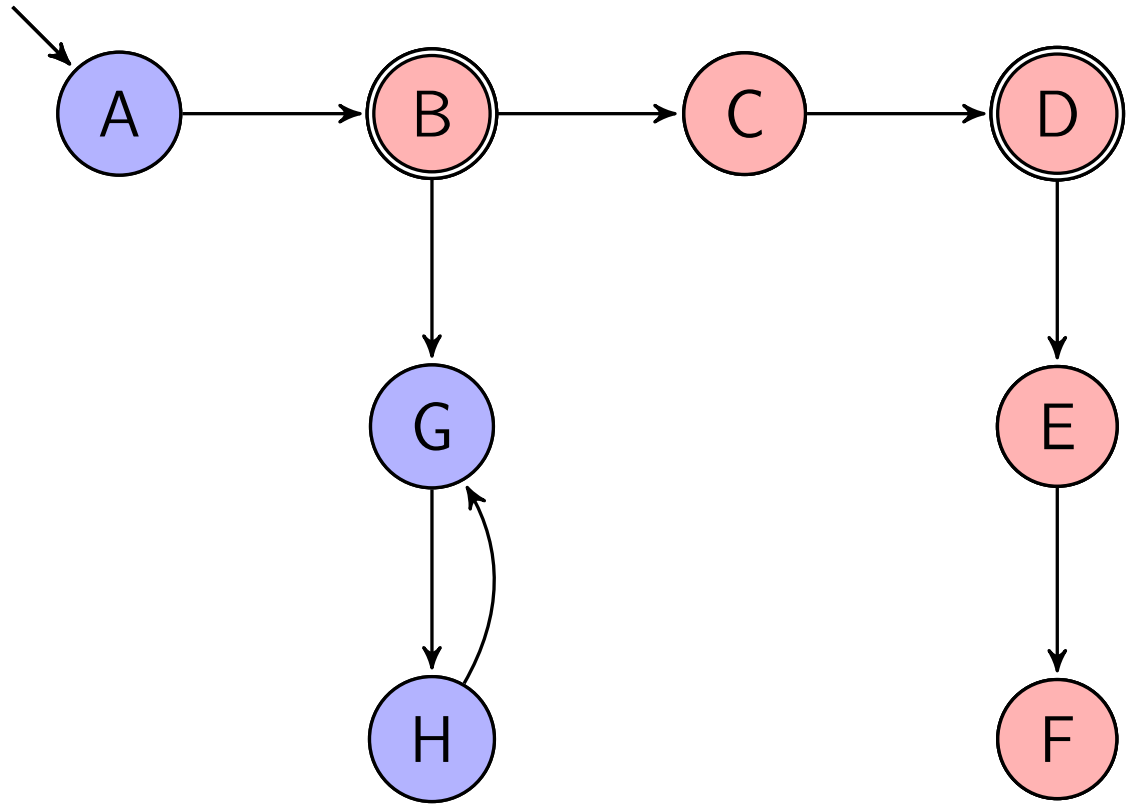
NDFS exemple



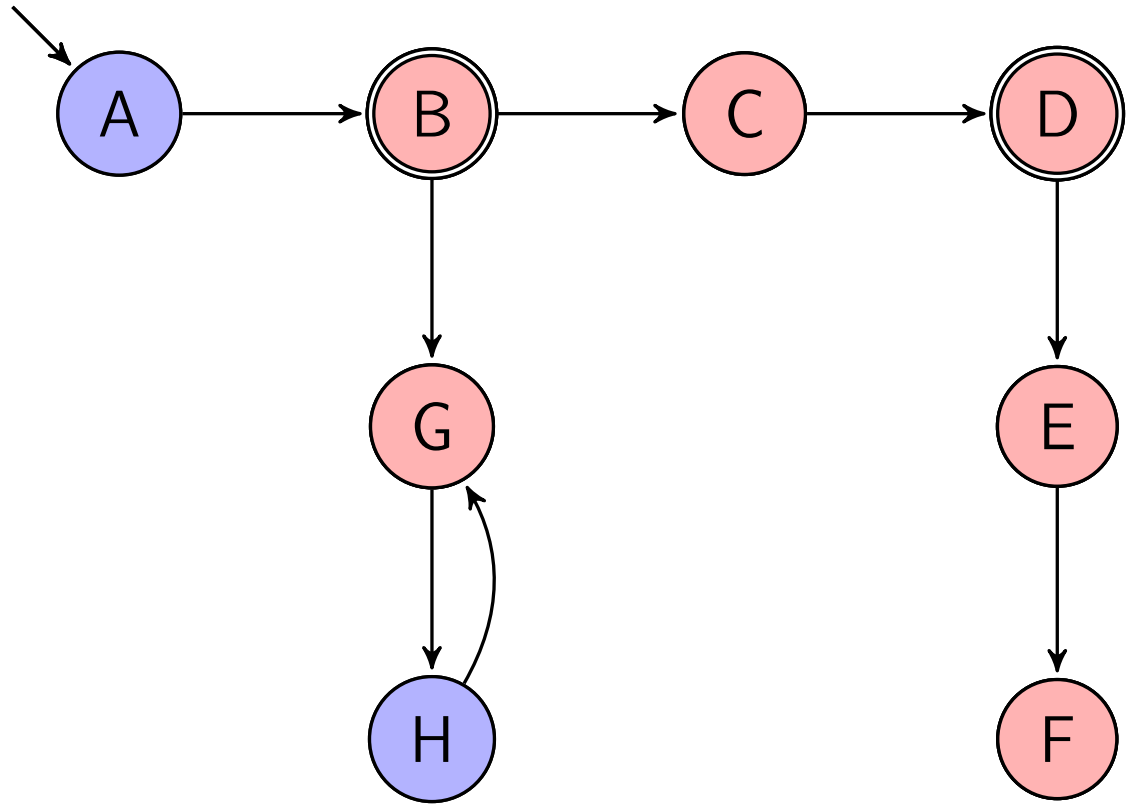
NDFS exemple



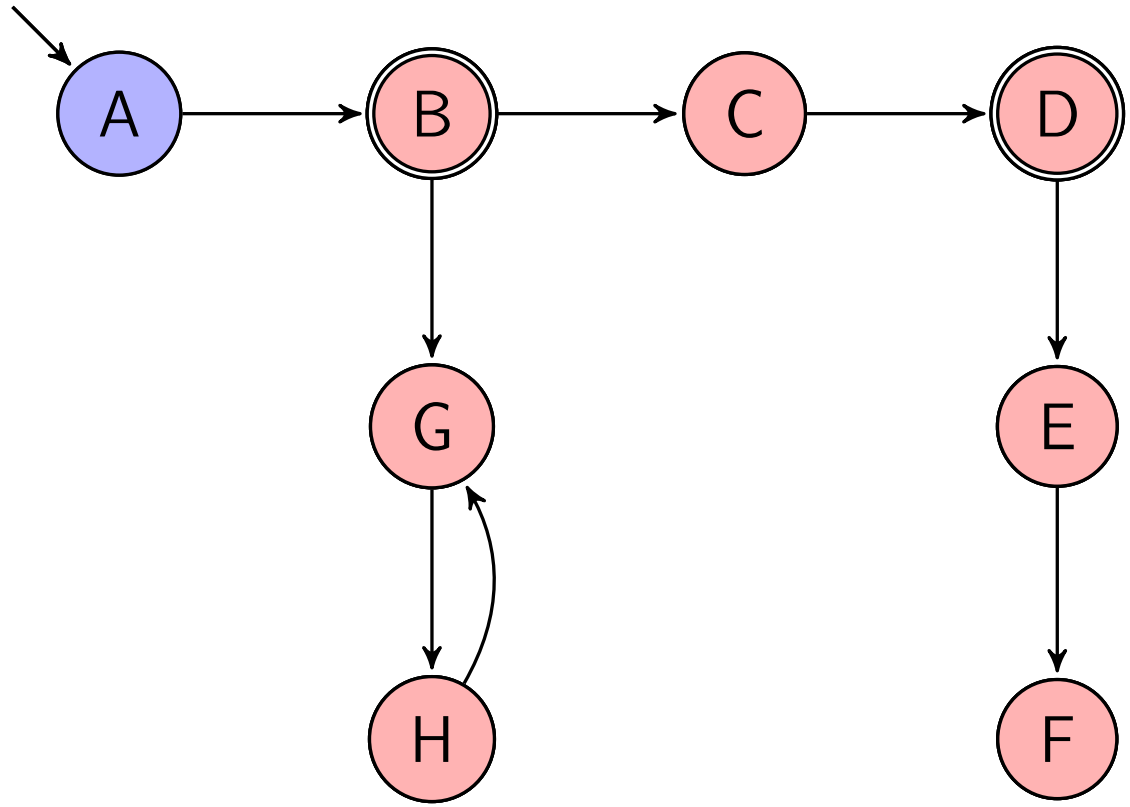
NDFS exemple



NDFS exemple



NDFS exemple



Quelques optimisations ! 1/2

Réduire la consommation mémoire

Les deux tables de hachages ne sont pas nécessaire, deux bits suffisent !

On-The-Fly

Il n'est pas nécessaire de construire l'intégralité du produit dans le cas où un contre-exemple existe

[Gastin et al, 2004][Gaiser et al, 2009]

Un état dont tous ses successeurs sont rouges peut être marqué comme rouge prématurément dans le parcours bleu.

Cela permet de limiter la portée du DFS rouge !

Quelques optimisations ! 2/2

[Gaiser et al, 2009]

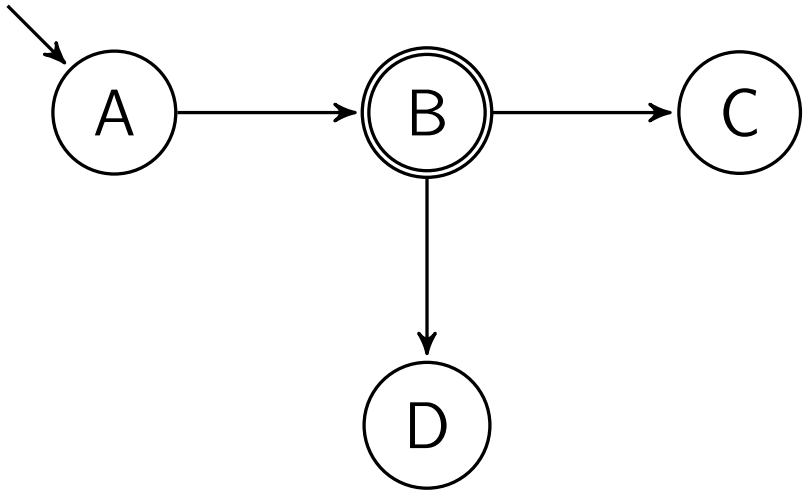
Utilisation d'une troisième couleur :

- Cyan : l'état est sur la pile du DFS
- Bleu : l'état a été visité par le premier parcours mais pas le second
- Rouge : l'état a été visité par le second DFS

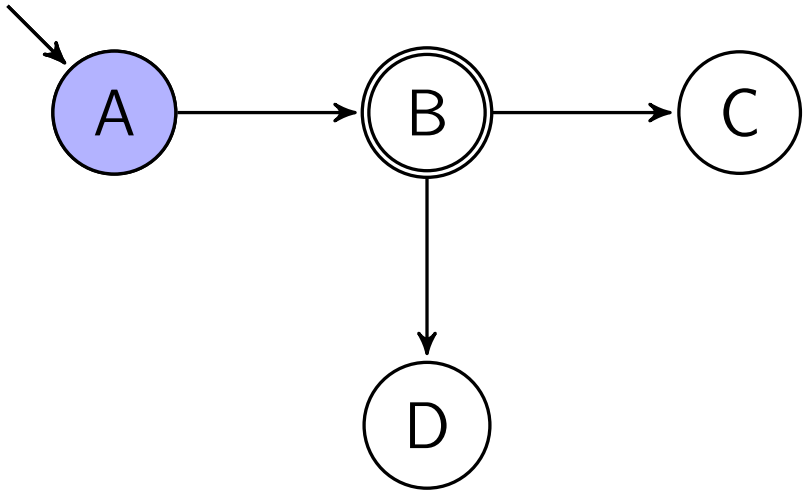
Les états peuvent seulement passer de cyan à bleu et de bleu à rouge !

Dès que l'on détecte une transition acceptante allant vers un état cyan, un contre-exemple est détecté !

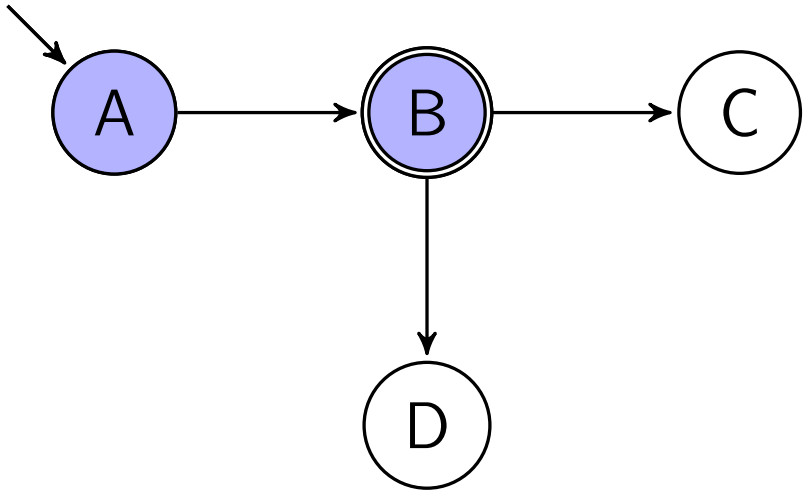
Exemple



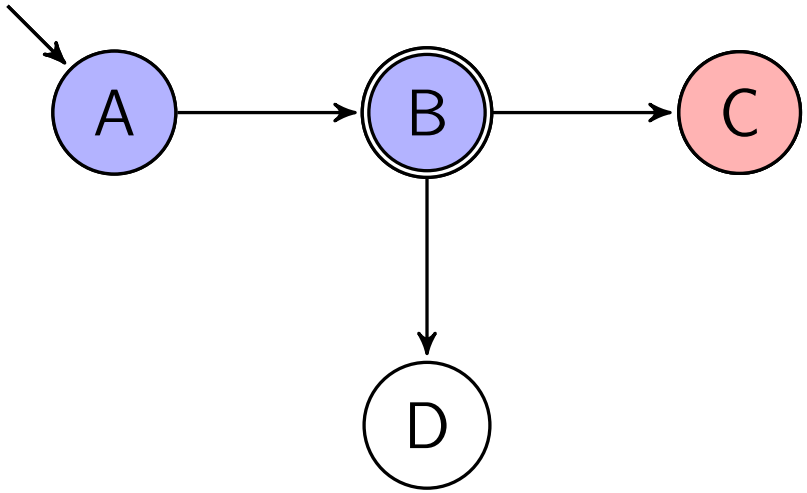
Exemple



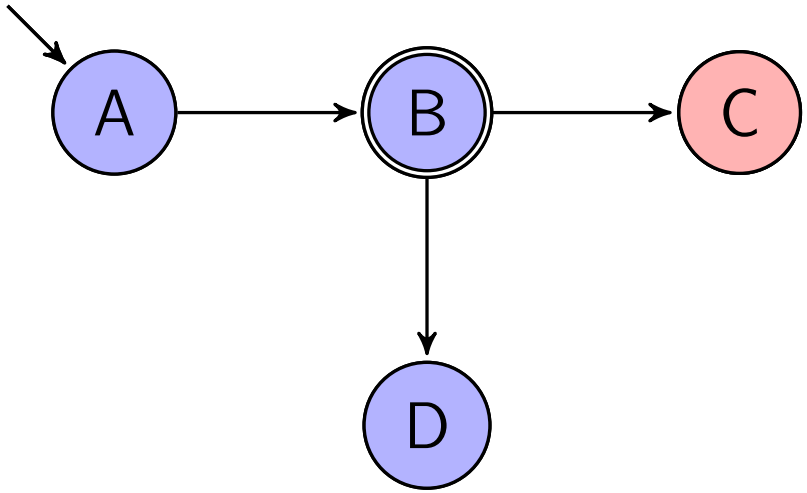
Exemple



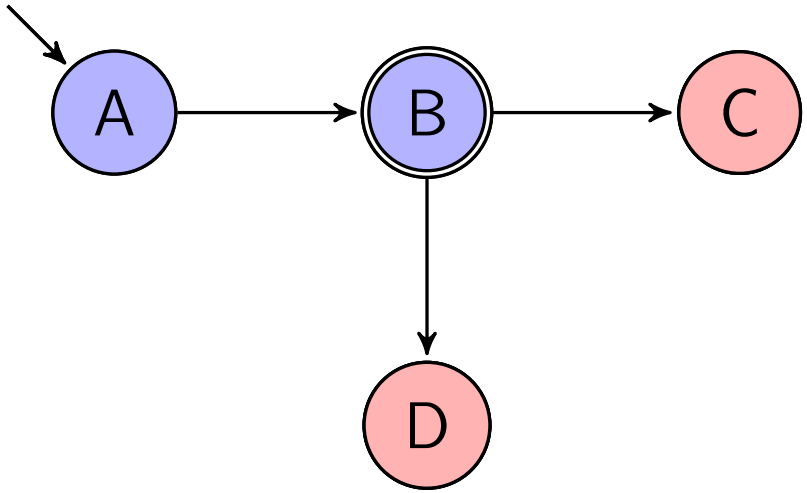
Exemple



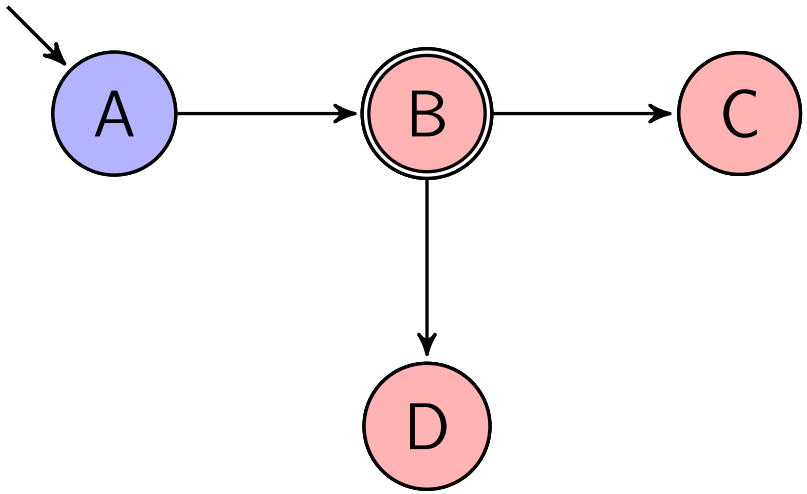
Exemple



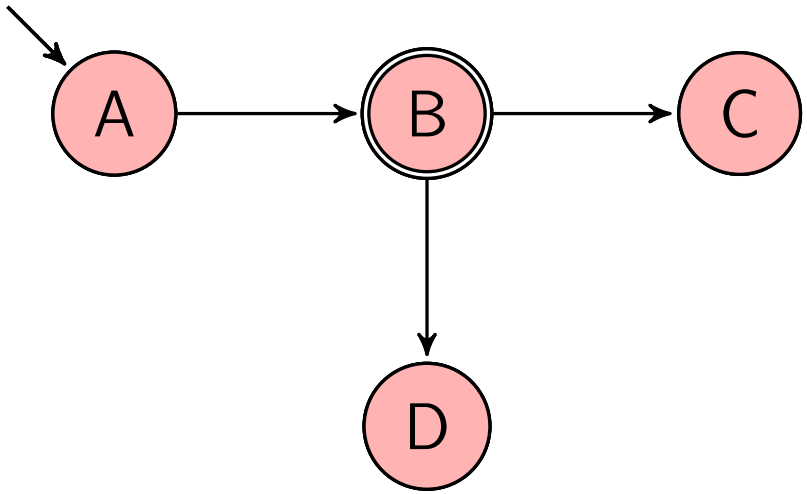
Exemple



Exemple



Exemple



Automates de Büchi généralisés

Les algorithmes précédents fonctionnent avec les automates généralisés (basés sur les états ou transitions) mais ont une complexité dépendante du nombre de conditions d'acceptations.

Dans les NDFS il y a un parcours qui est déclenché pour chaque marque rencontrée.

L'utilisation d'automates non-généralisé est toujours couteux :

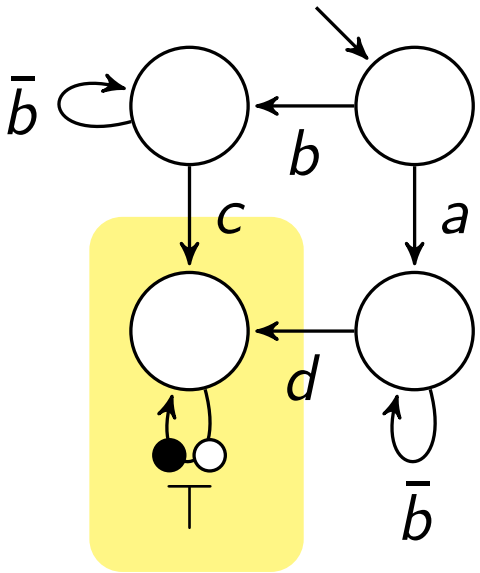
- soit via une dégénéralisation
- soit via une complexité de l'emptiness check

L'étude de l'automate de la formule peut-il permettre de réduire ce coût ?

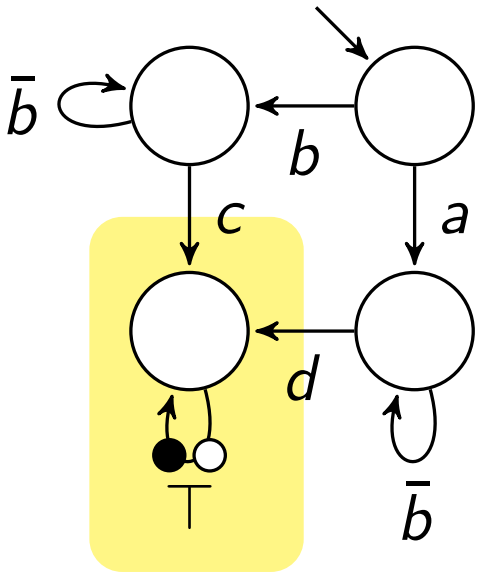
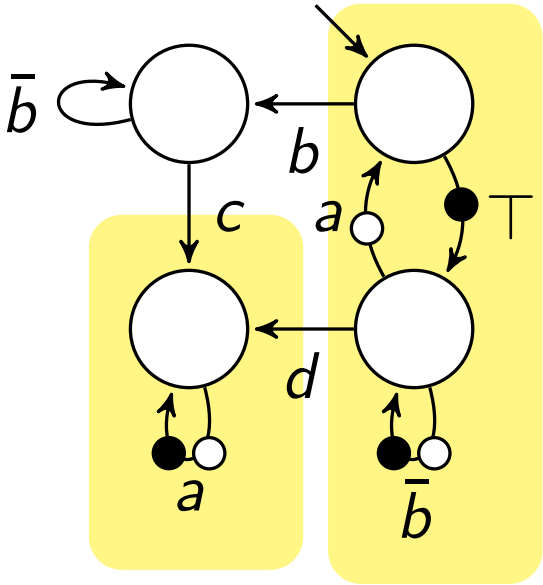
Meilleure utilisation de l'automate de la formule

Automate terminal	Automate faible	Automate fort
Les SCC acceptante sont complètes et ne contiennent que cycles acceptants	*Les SCC acceptantes ne contiennent que des cycles acceptants*	*Les SCC acceptantes peuvent mixer des des acceptants et non acceptants*
Accessibilité Si A_{Sys} n'a pas de deadlock.	Recherche simple de cycle	NDFS ou SCC

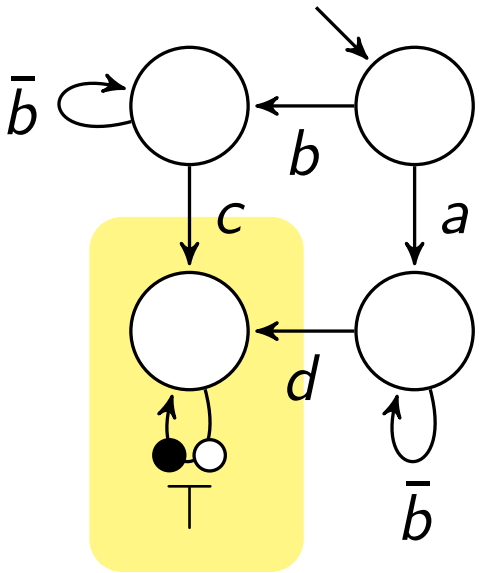
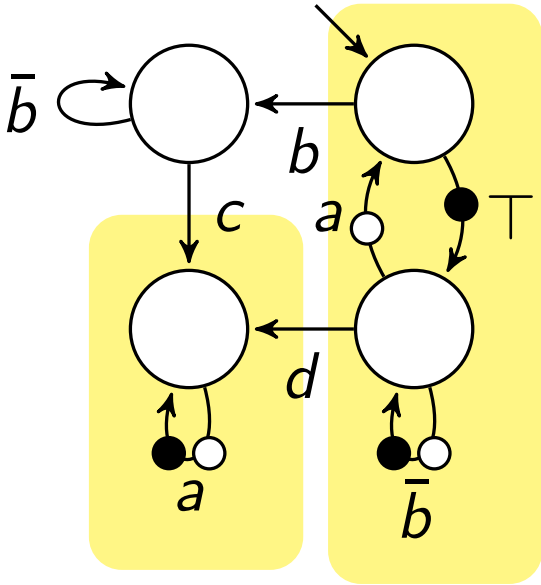
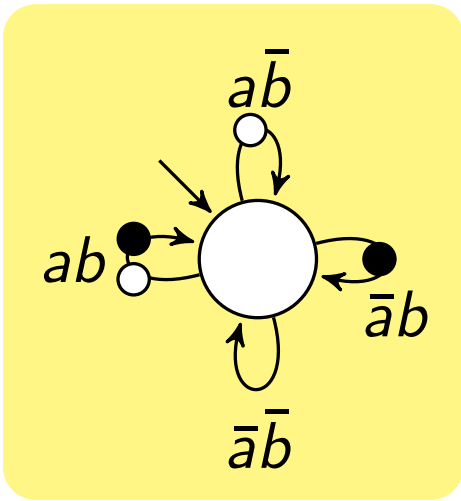
Meilleure utilisation de l'automate de la formule

Automate terminal	Automate faible	Automate fort
	*Les SCC acceptantes ne contiennent que des cycles acceptants*	*Les SCC acceptantes peuvent mixer des des acceptants et non acceptants*
Accessibilité Si A_{Sys} n'a pas de deadlock.	Recherche simple de cycle	NDFS ou SCC

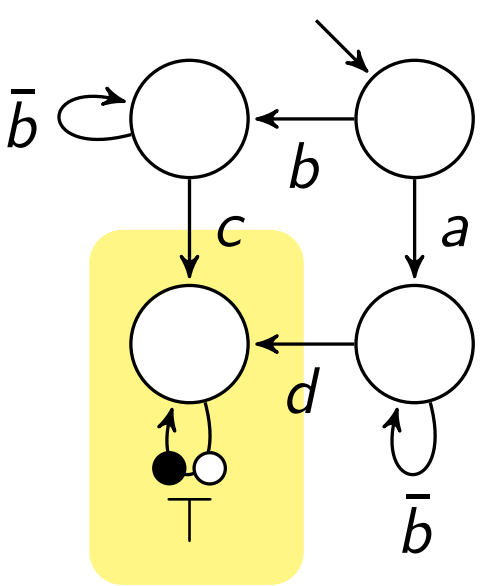
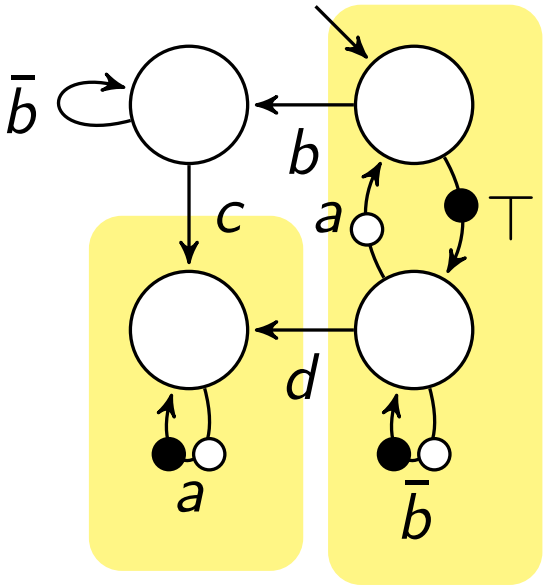
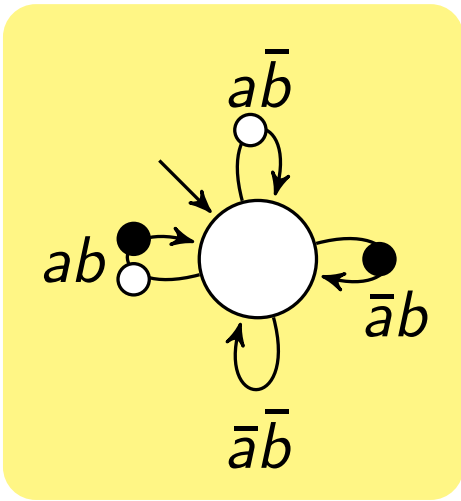
Meilleure utilisation de l'automate de la formule

Automate terminal	Automate faible	Automate fort
		*Les SCC acceptantes peuvent mixer des des acceptants et non acceptants*
Accessibilité Si A_{Sys} n'a pas de deadlock.	Recherche simple de cycle	NDFS ou SCC

Meilleure utilisation de l'automate de la formule

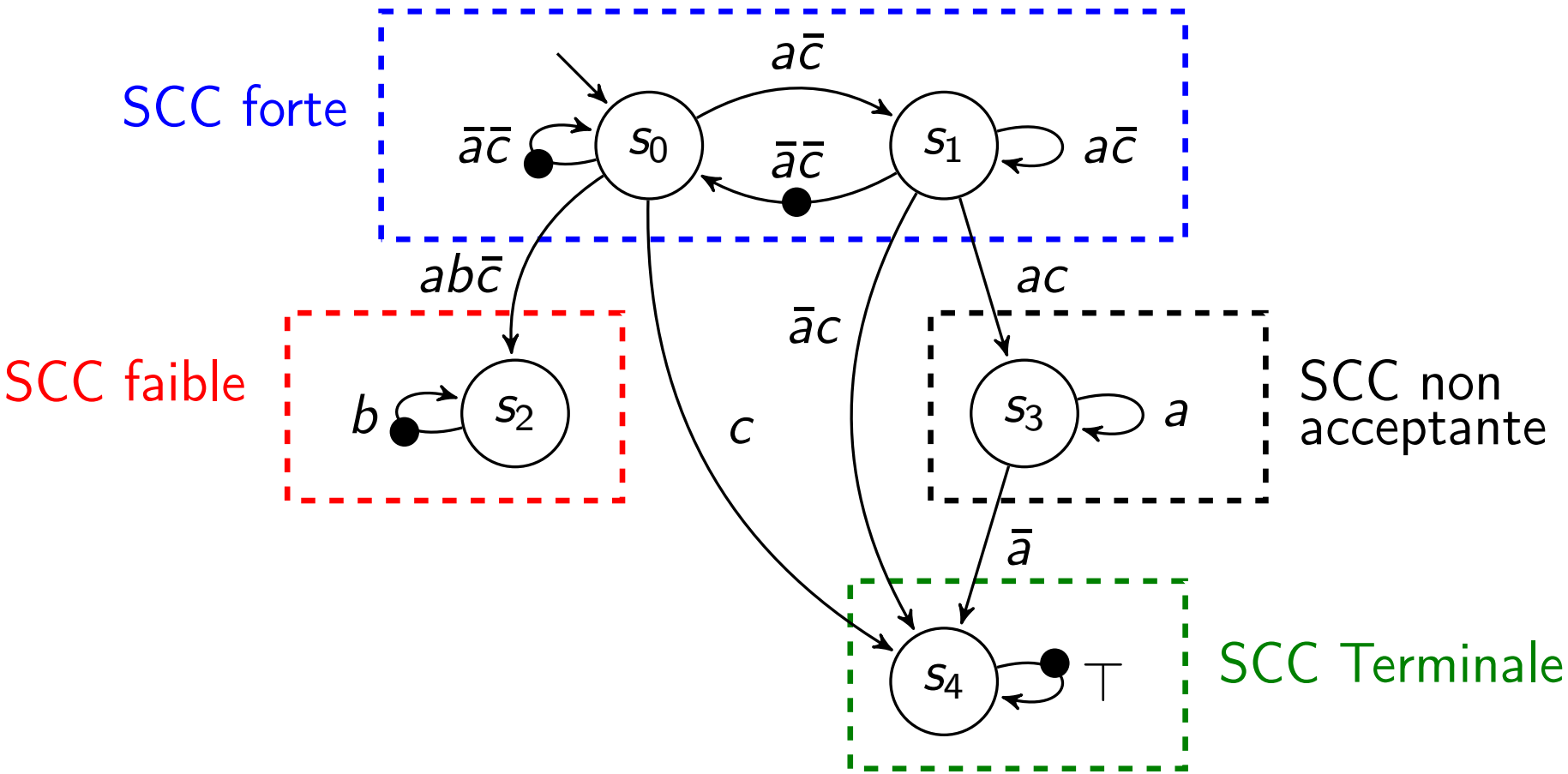
Automate terminal	Automate faible	Automate fort
		
Accessibilité Si A_{Sys} n'a pas de deadlock.	Recherche simple de cycle	NDFS ou SCC

Meilleure utilisation de l'automate de la formule

Automate terminal	$\subset$	Automate faible	$\subset$	Automate fort
				
Accessibilité Si A_{Sys} n'a pas de deadlock.	$<$	Recherche simple de cycle	$<$	NDFS ou SCC

Automates avec plusieurs forces de SCCs

[Edelkamp et al., 2004]



$A_{\neg\varphi}$ for $\neg\varphi = (\mathbf{G}\, a \rightarrow \mathbf{G}\, b)\, \mathbf{W}\, c$

NDFS optimisé [Edelkamp et al., 2004]

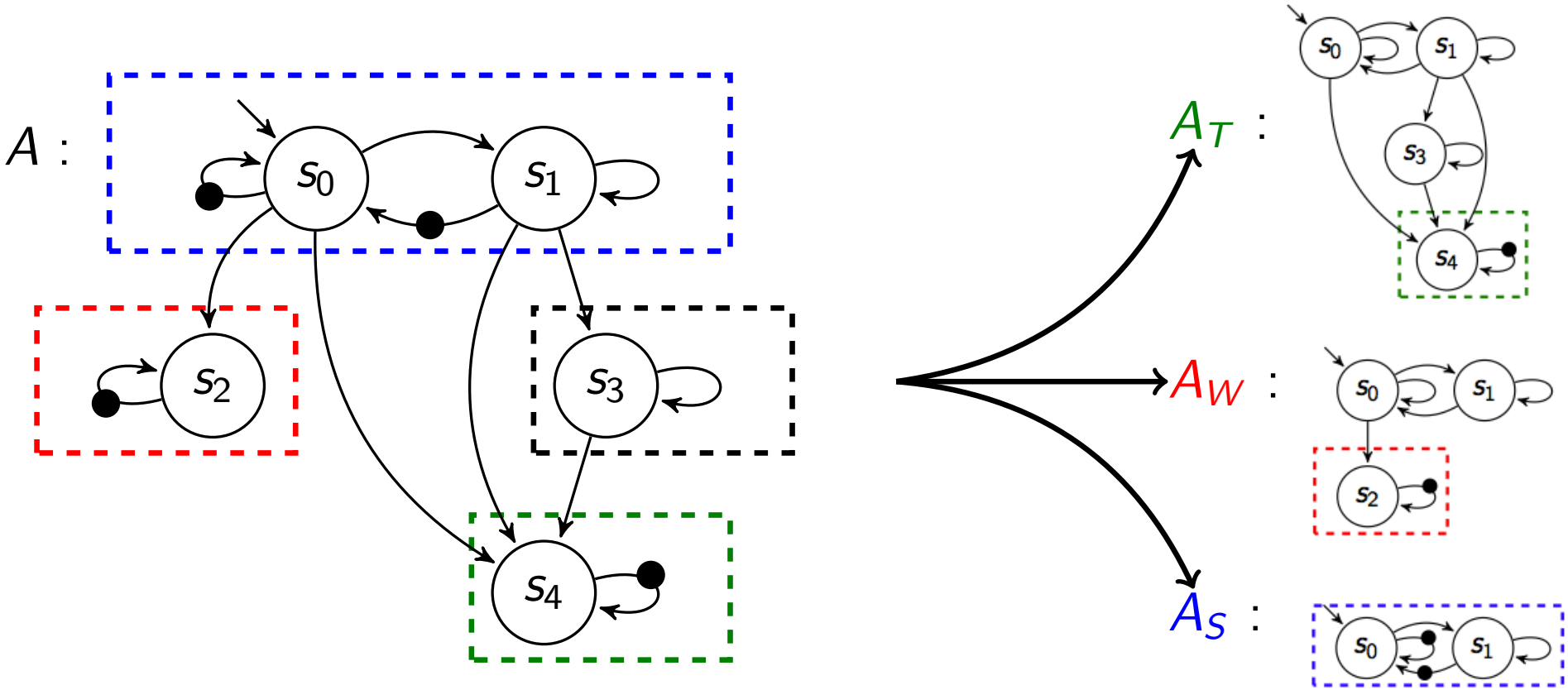
Idée :

- Lorsqu'un état acceptant s est dépilé du premier DFS, un second DFS est lancé. Ce DFS va explorer tous les états accessibles depuis s.

  On peut se restreindre aux états qui sont synchronisé avec la même SCC de l'automate de la formule

  Peut facilement être combiné avec l'optimisation qui limite la portée des DFS imbriqués.
- Pour un état donné on peut adapter l'emptiness check en fonction de la force de la SCC à laquelle appartient la projection de cet état sur l'automate de la formule.

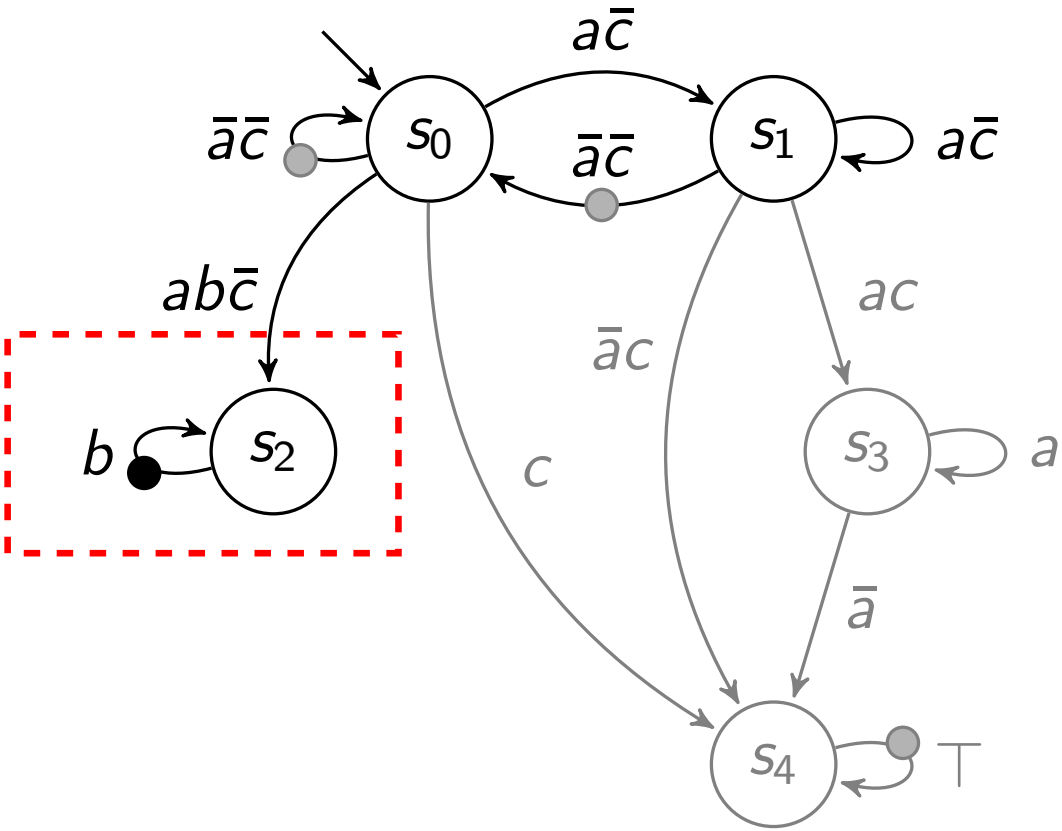
Décomposition de l'automate de la propriété



$$\mathscr{L}(A) = \mathscr{L}(A_T) \cup \mathscr{L}(A_W) \cup \mathscr{L}(A_S).$$

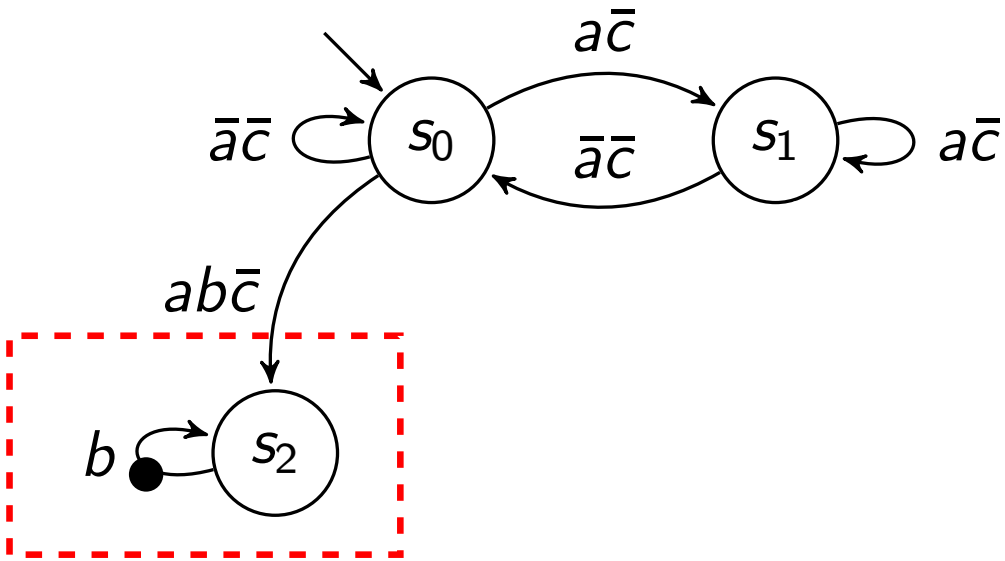
- A_T : capture les comportements terminaux de A
- A_W : capture les comportements faibles de A
- A_S : capture les comportements forts de A

Construction de A_W



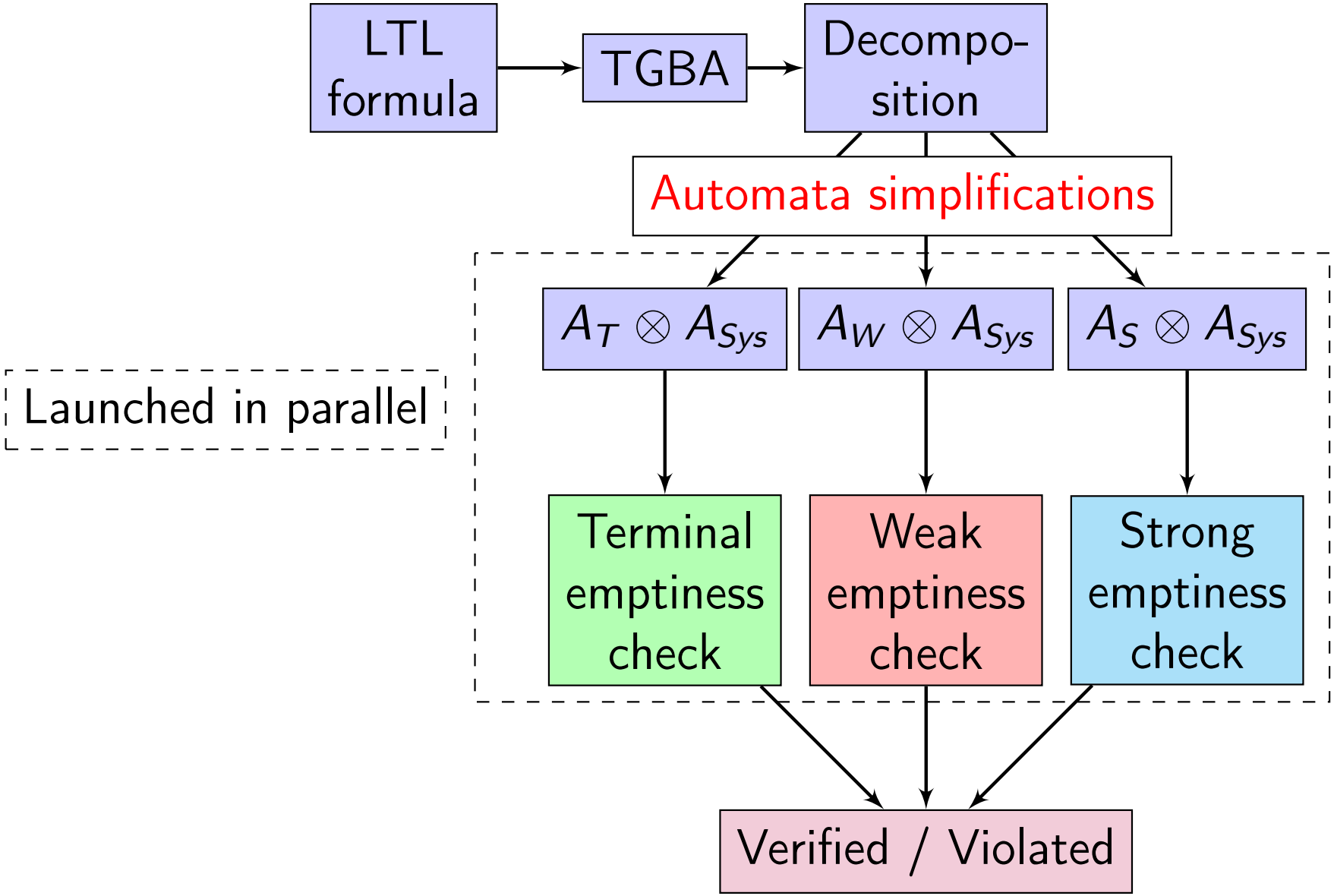
Toutes les ensembles d'acceptation sont supprimés et
un seul ensemble étiquette toutes les transitions des SCC faibles.

Construction de A_W



Toutes les ensembles d'acceptation sont supprimés et
un seul ensemble étiquette toutes les transitions des SCC faibles.

Decomposition Canevas [Renault et al., 2013]



Vers d'autres emptiness checks ...

Problème !

Pour la partie forte, la complexité reste proportionnelle au nombre d'ensembles d'acceptations

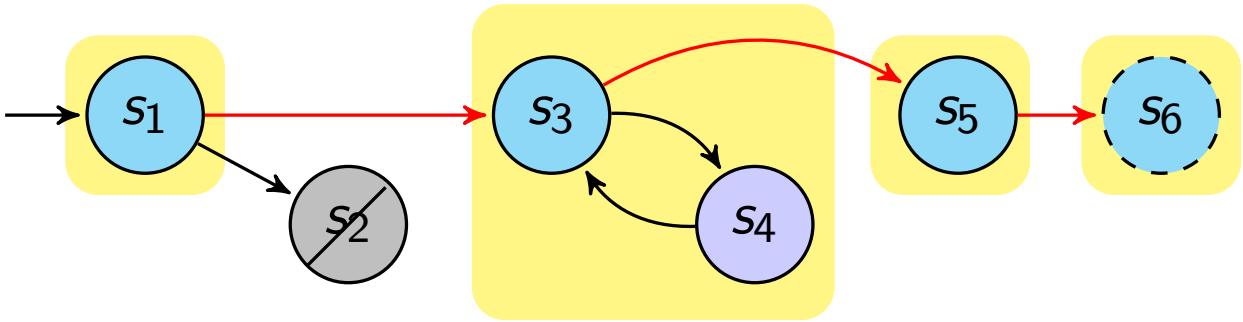
Solution

Utiliser des algorithmes basés sur le calcul des composantes fortement connexes !

- Trouver une SCC (dans l'automate du produit) qui contient tous les ensembles d'acceptation garantie la présence d'un cycle acceptant dans cette composante !

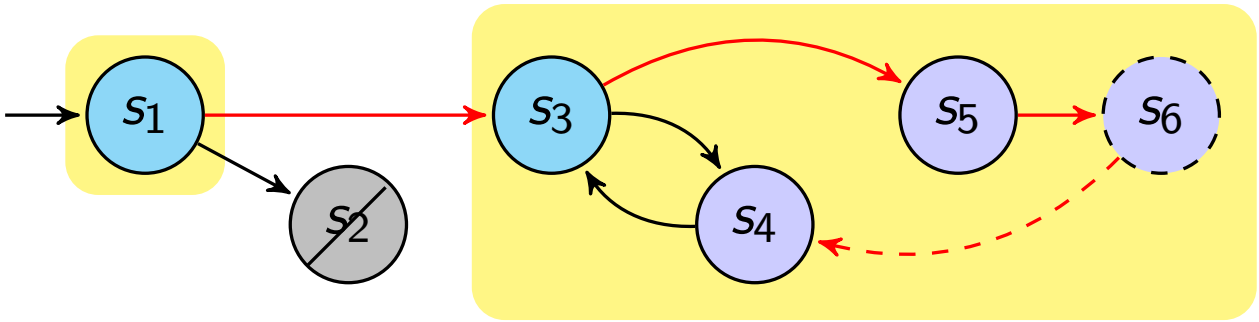
Algorithmes de calcul de SCC

- [Dijkstra, 1973] maintient le meilleur candidat à être une *racine*



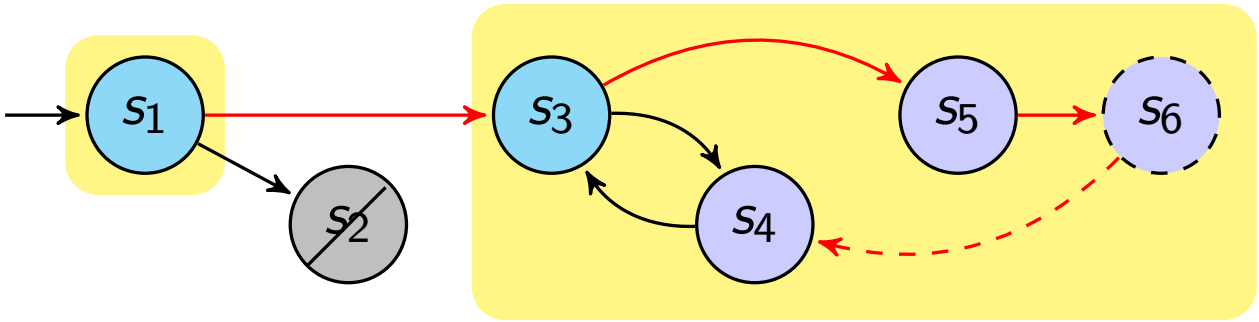
Algorithmes de calcul de SCC

- [Dijkstra, 1973] maintient le meilleur candidat à être une *racine*

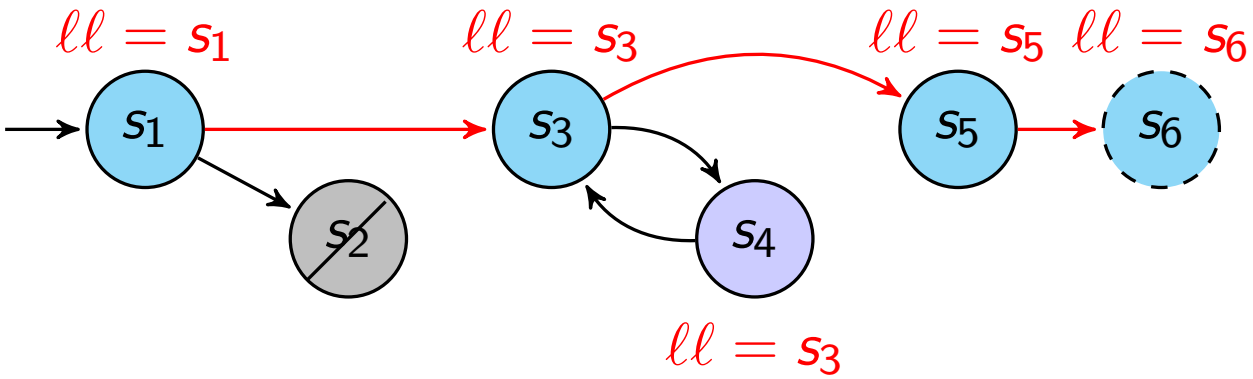


Algorithmes de calcul de SCC

- [Dijkstra, 1973] maintient le meilleur candidat à être une *racine*

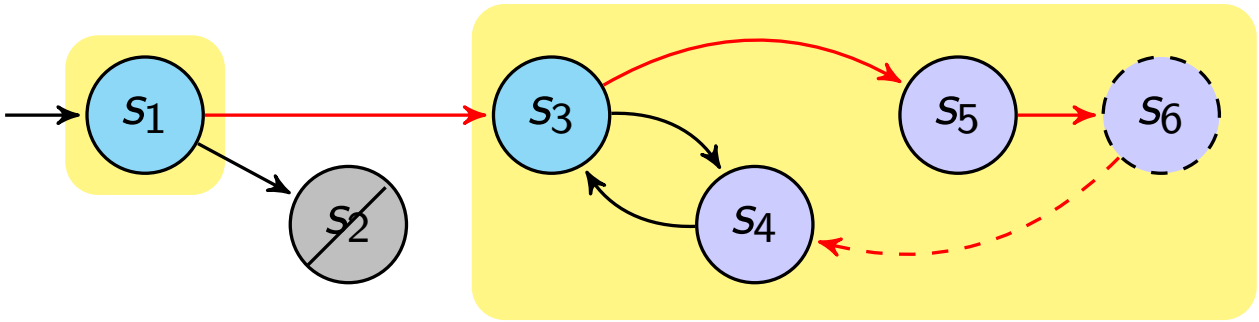


- [Tarjan, 1971] maintient des *lowlinks* pour détecter des racines

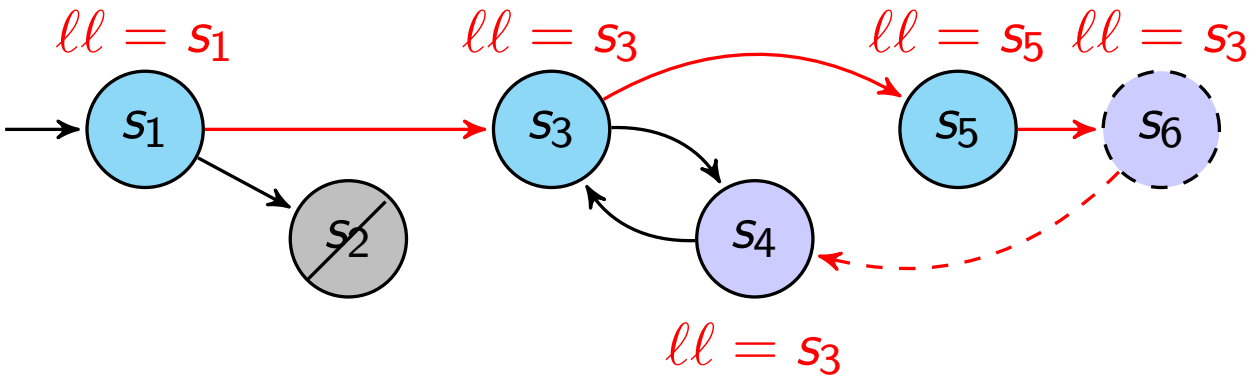


Algorithmes de calcul de SCC

- [Dijkstra, 1973] maintient le meilleur candidat à être une *racine*

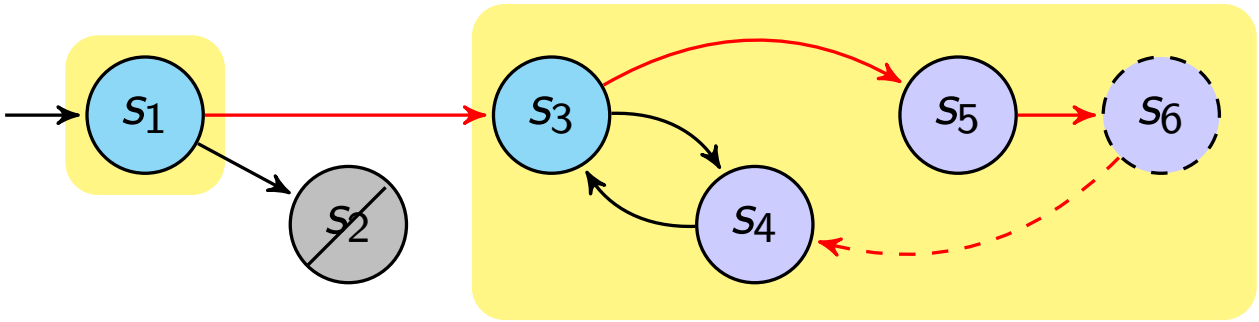


- [Tarjan, 1971] maintient des *lowlinks* pour détecter des racines

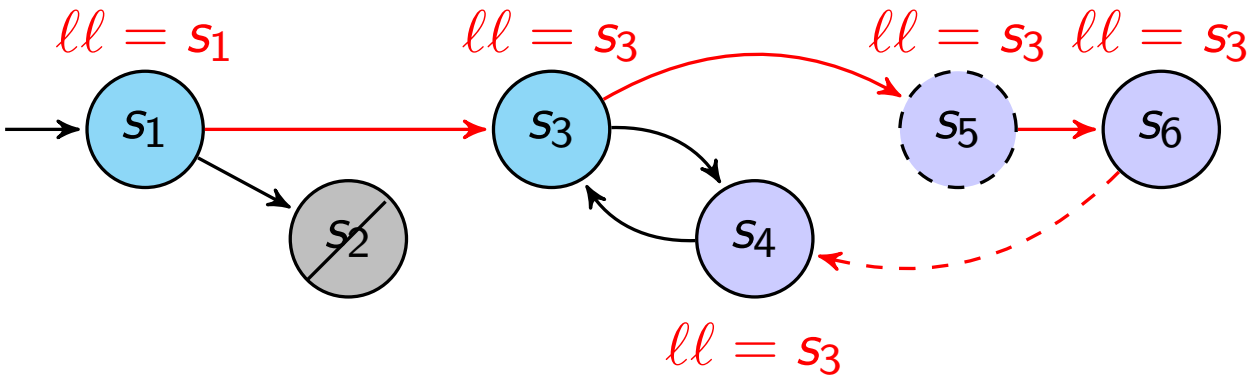


Algorithmes de calcul de SCC

- [Dijkstra, 1973] maintient le meilleur candidat à être une *racine*



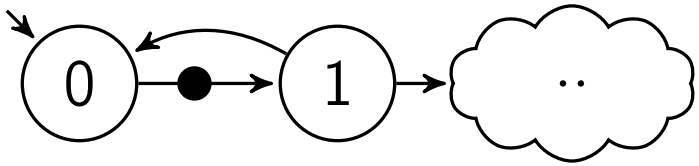
- [Tarjan, 1971] maintient des *lowlinks* pour détecter des racines



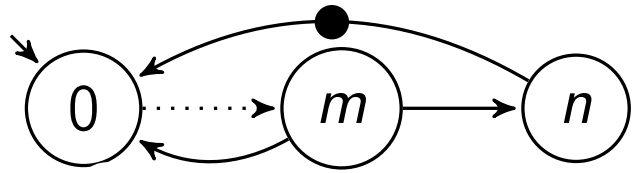
Différences entre les deux algorithmes

Les deux algorithmes peuvent servir de base à des emptiness check et ont un intérêt pour détecter les contre-exemples !

Pire cas pour Tarjan



Pire cas pour Dijkstra



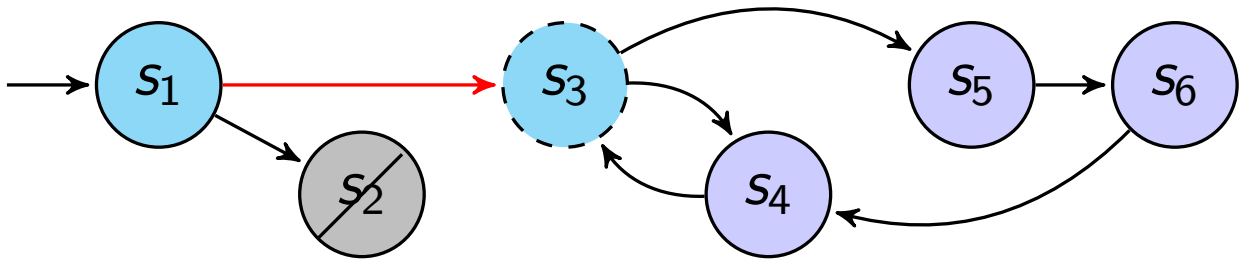
Les algorithmes basés sur le calcul des composantes fortement connexes :

- ont une complexité indépendante du nombre d'ensemble d'acceptation
- utilisent plus de mémoire (besoin d'une pile pour stocker les racines ou les lowlinks)

Union-Find pour les Emptiness Checks

Problème

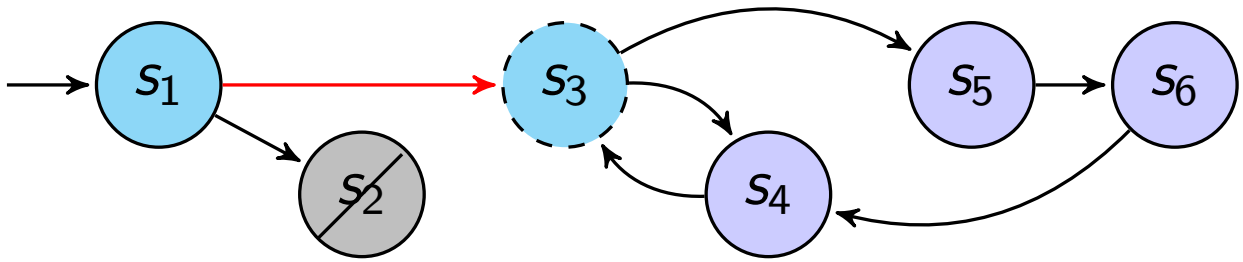
Tous les états d'une SCC doivent être markés morts un par un



Union-Find pour les Emptiness Checks

Problème

Tous les états d'une SCC doivent être markés morts un par un



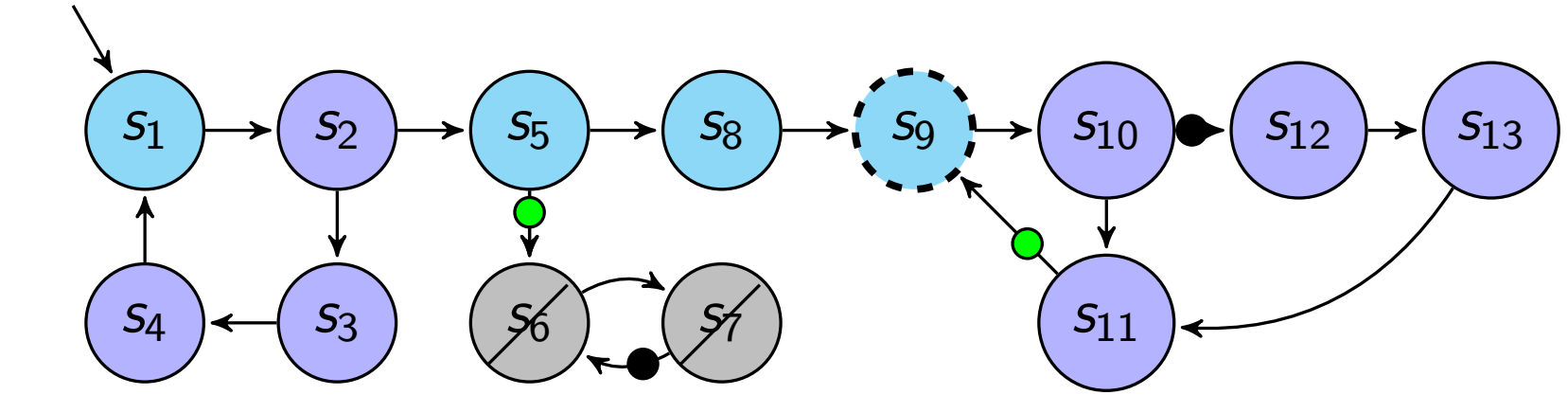
Solution : l'union-find

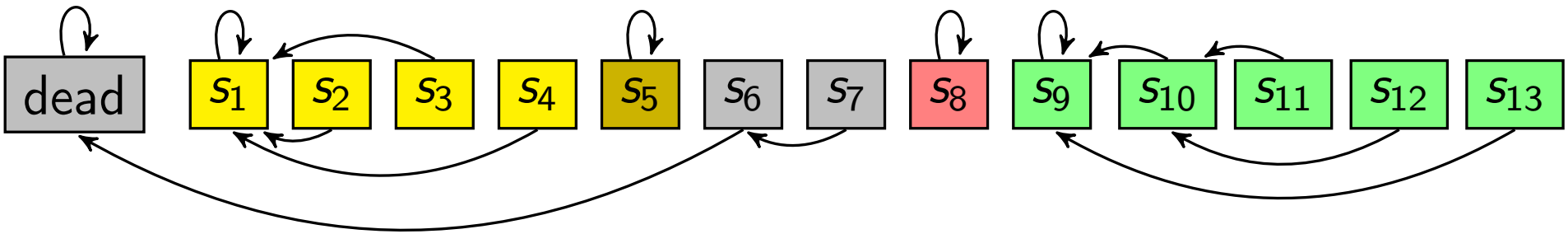
Structure de donnée appropriée :

- Avec de nombreuses optimisations
- Peut **créer** ou **unir** des partitions
- Complexité moyenne pour chaque opération **unite** : $Ack^{-1}(n)$

Union-Find en action !

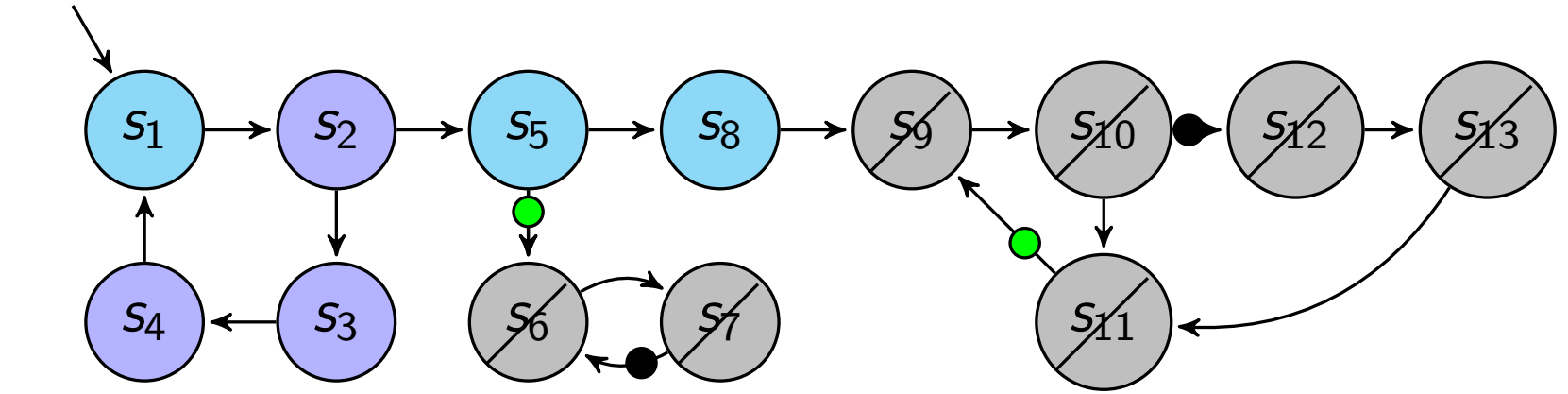
Utilisation d'une partition spéciale pour les éléments "morts".

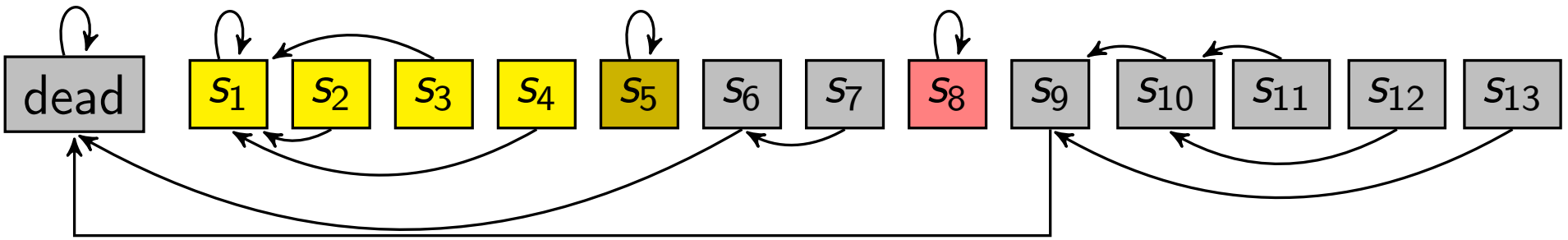




Union-Find en action !

Utilisation d'une partition spéciale pour les éléments "morts".





Peut-ont améliorer les choses ?

- Parallélisme
- Partial-Order
- Bit-state-hashing

Parallélisme

Swarming

Holzmann et al, 2007

Si l'on possède n threads, alors on peut lancer n emptiness checks en parallèle. Chaque thread possède une graine qui va lui permettre de tirer les transitions dans un ordre qui lui est propre.

Dès qu'un thread détecte un contre-exemple, tous les threads sont arrêtés !

Swarming

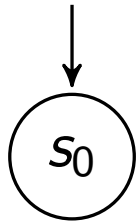
Thread 1

Thread 2

Swarming

Thread 1

s_0



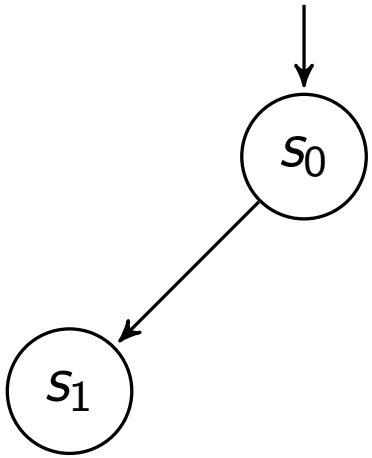
Thread 2

Swarming

Thread 1

s_0

s_1



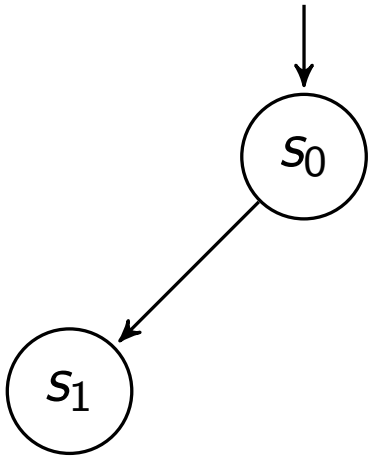
Thread 2

Swarming

Thread 1

s_0

s_1



Thread 2

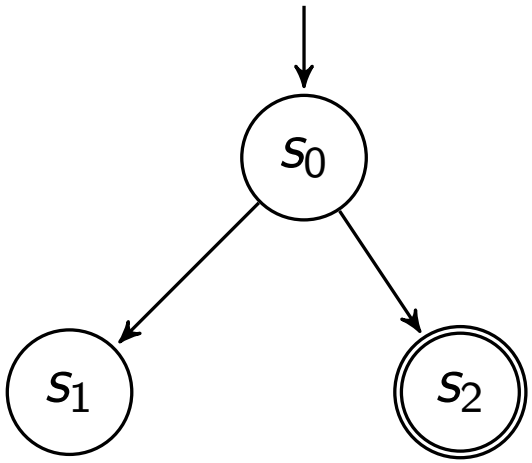
s_0

Swarming

Thread 1

s_0

s_1



Thread 2

s_0

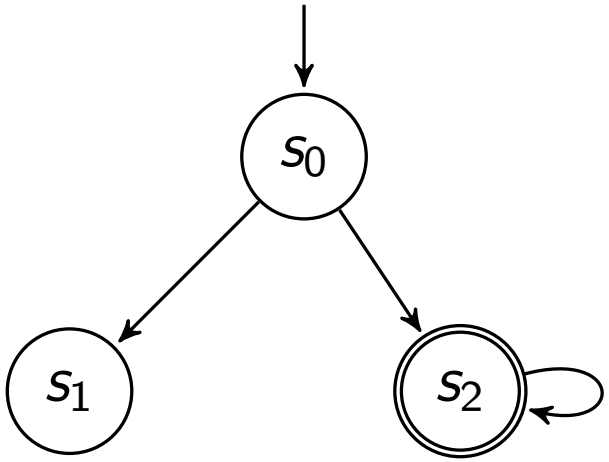
s_2

Swarming

Thread 1

s_0

s_1



Thread 2

s_0

s_2

Ctrx Found!

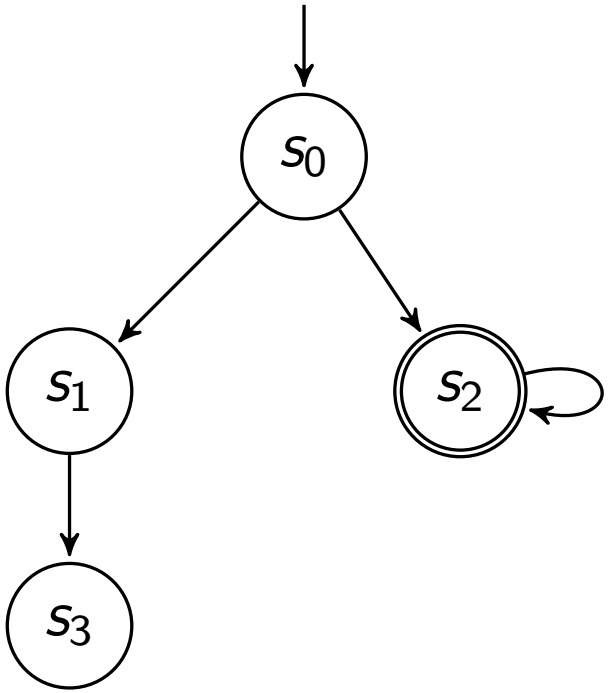
Swarming

Thread 1

s_0

s_1

s_3



Thread 2

s_0

s_2

Ctrx Found!

Swarming

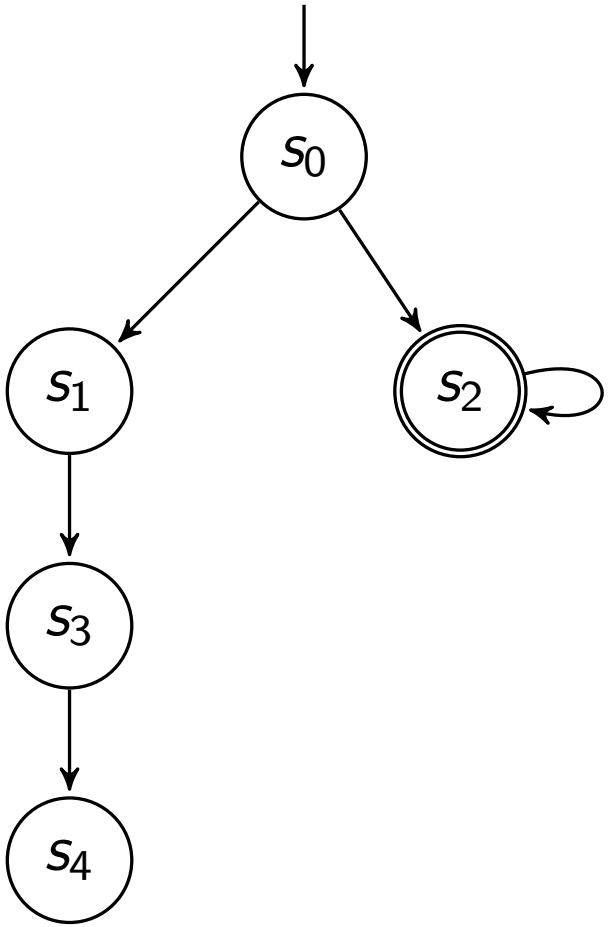
Thread 1

s_0

s_1

s_3

s_4



Thread 2

s_0

s_2

Ctrx Found!

Swarming

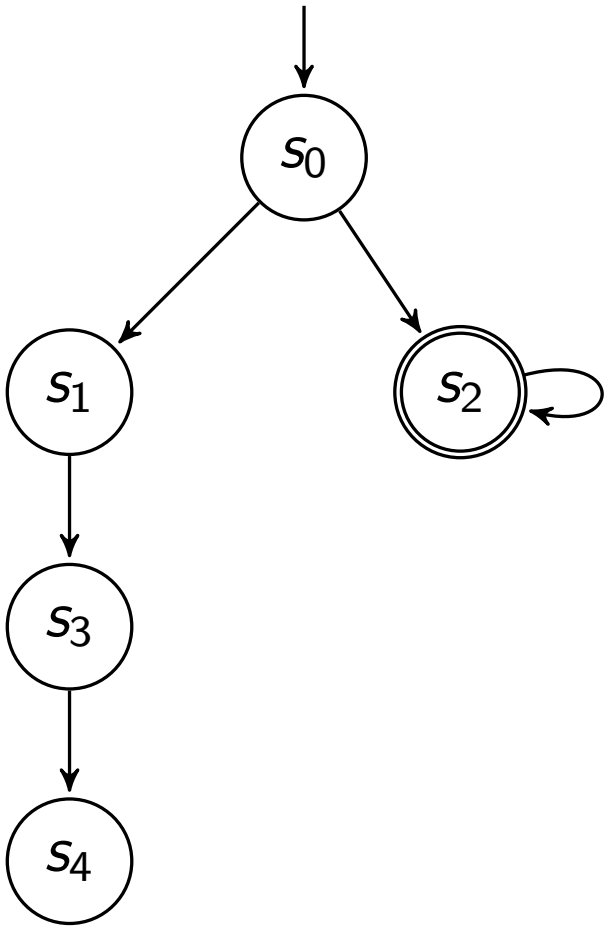
Thread 1

s_0

s_1

s_3

s_4



Thread 2

s_0

s_2

Ctrx Found!

S'il n'y avait eu que le thread 1, tous les états auraient du être visité.
Avec deux threads seuls 3 états ont été découverts.

Swarming – Analyse

Avantages

1. Facilité de mise en oeuvre
2. Permet de mixer plusieurs emptiness checks
3. Support d'automates généralisés ou non
4. Synchronisation seulement sur la détection d'un contre-exemple.

Swarming – Analyse

Avantages

1. Facilité de mise en oeuvre
2. Permet de mixer plusieurs emptiness checks
3. Support d'automates généralisés ou non
4. Synchronisation seulement sur la détection d'un contre-exemple.

Problème

Où est le partage ?

Overview of parallel emptiness checks

Non DFS-based

NDFS-based

SCC-based

Overview of parallel emptiness checks

Non DFS-based [Barnat et al., since 2003]

- \+ Theoretically scales better than DFS-based emptiness checks
- – Successors are re-computed many times
- – Late counterexample detection

NDFS-based

SCC-based

Overview of parallel emptiness checks

Non DFS-based [Barnat et al., since 2003]

- \+ Theoretically scales better than DFS-based emptiness checks
- − Successors are re-computed many times
- − Late counterexample detection

NDFS-based [Laarman et al., since 2011][Evangelista et al., since 2011]

- \+ Scales better in practice than non DFS-based emptiness checks
- \+ Faster counterexample detection (Swarming)
- − No support for generalized acceptance
- − Require synchronization points or repair procedures

SCC-based

Overview of parallel emptiness checks

Non DFS-based [Barnat et al., since 2003]

- \+ Theoretically scales better than DFS-based emptiness checks
- − Successors are re-computed many times
- − Late counterexample detection

NDFS-based [Laarman et al., since 2011][Evangelista et al., since 2011]

- \+ Scales better in practice than non DFS-based emptiness checks
- \+ Faster counterexample detection (Swarming)
- − No support for generalized acceptance
- − Require synchronization points or repair procedures

SCC-based ?

Tests de vacuité parallèle généralisé

Question [Evangelista, 2012]

Peut on construire un test de vacuité basé sur un DFS qui ne requiert ni synchronisations ni procédures de réparation ?

Tests de vacuité parallèle généralisé

Question [Evangelista, 2012]

Peut on construire un test de vacuité basé sur un DFS qui ne requiert ni synchronisations ni procédures de réparation *et qui supporte les automates de Büchi généralisé ?*

Tests de vacuité parallèle généralisé

Question [Evangelista, 2012]

Peut on construire un test de vacuité basé sur un DFS qui ne requiert ni synchronisations ni procédures de réparation *et qui supporte les automates de Büchi généralisé ?*

Suggestion

Partager de l'information structurelle entre les threads permet de construire de tels tests de vacuité.

Information Structurelle

l'information Structurelle ne dépend pas de l'ordre de parcours d'un thread :

- Deux états sont dans la même SCC
- Une marque d'acceptation est présente dans une SCC
- Un état ne peut faire parti d'un cycle acceptant

L'union-find :

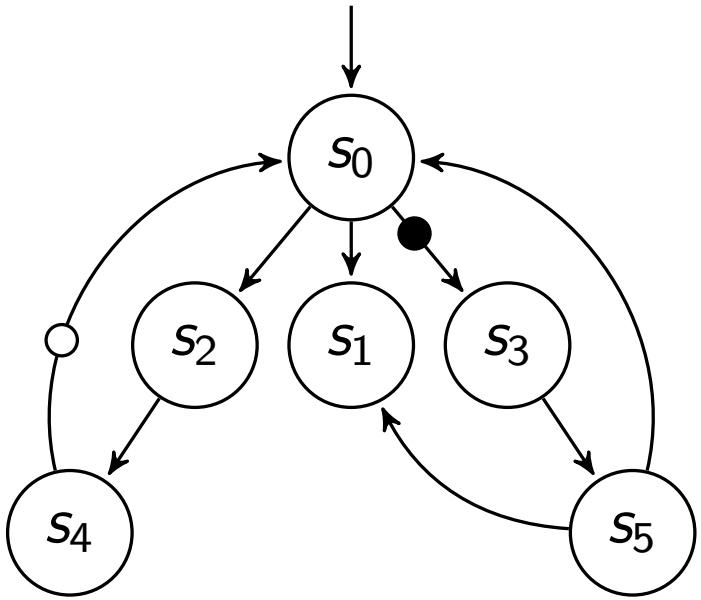
- can be extended to store acceptance sets
- est partagé entre les threads
- is lock-free since it relies on hash-tables and linked lists

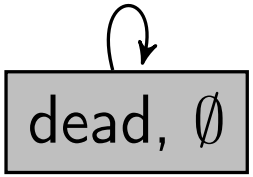
On peut méler les algorithmes basés sur le calcul des SCC vu que l'information partagée est structurelle.

Idée Générale

Thread 1
(Tarjan-based)

Thread 2
(Dijkstra-based)

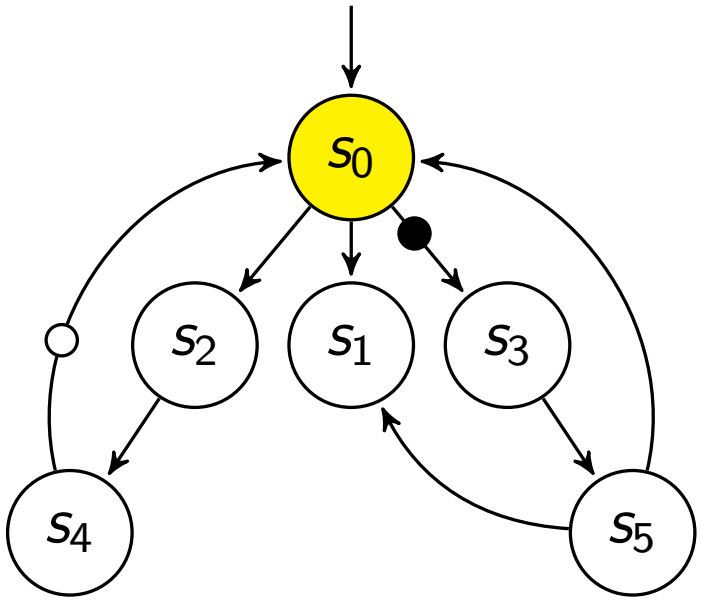




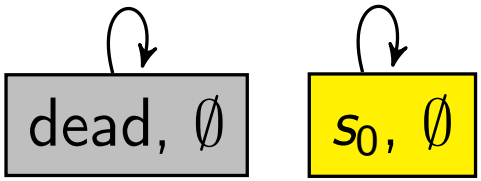
Idée Générale

Thread 1
(Tarjan-based)

s_0



Thread 2
(Dijkstra-based)

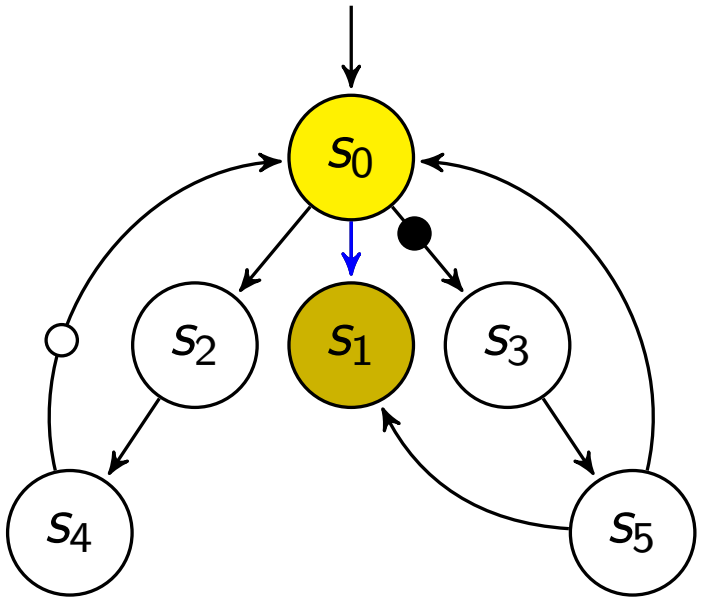


Idée Générale

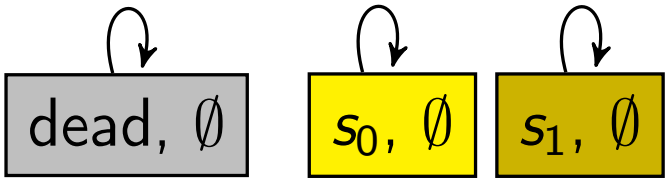
Thread 1
(Tarjan-based)

s_0

s_1



Thread 2
(Dijkstra-based)

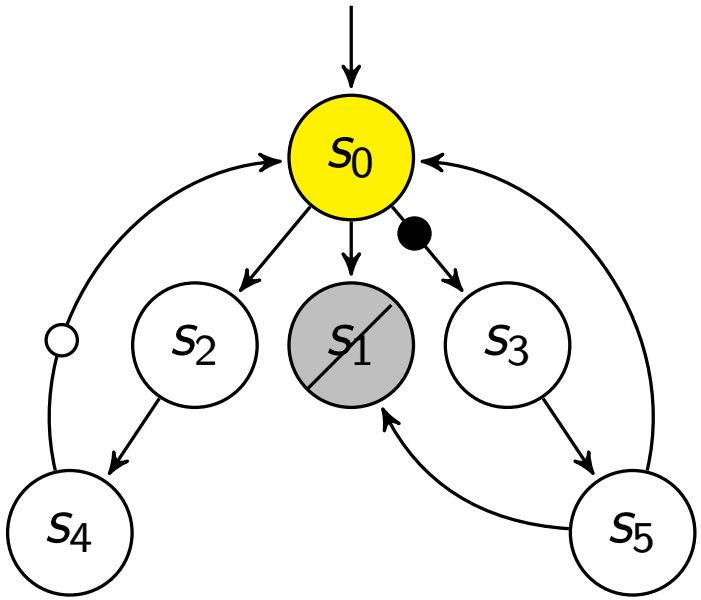


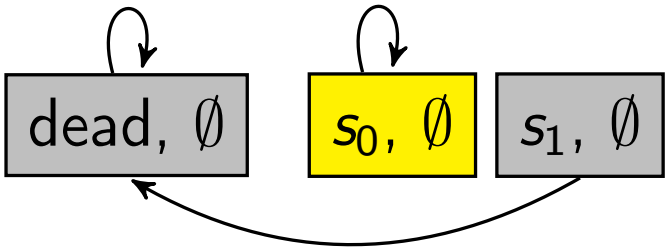
Idée Générale

Thread 1
(Tarjan-based)

s_0

Thread 2
(Dijkstra-based)





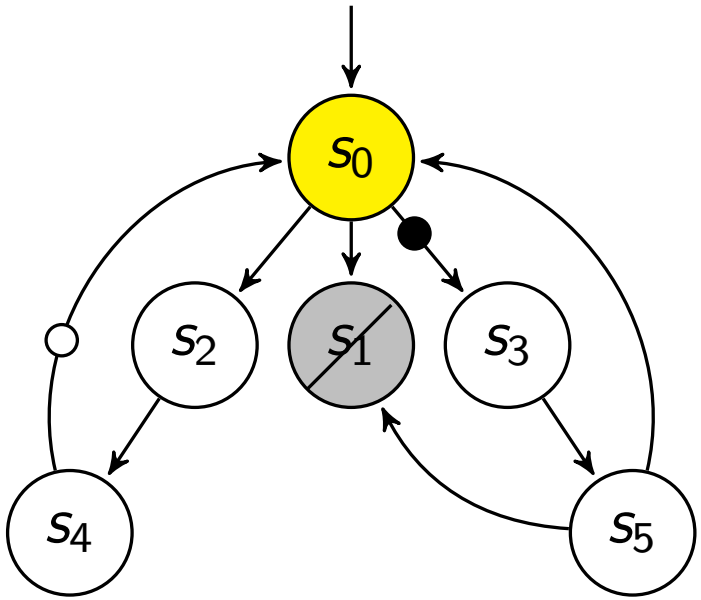
Idée Générale

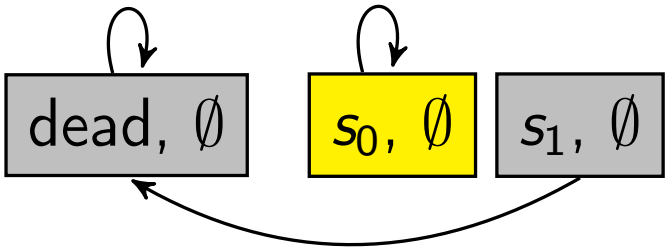
Thread 1
(Tarjan-based)

s_0

Thread 2
(Dijkstra-based)

s_0

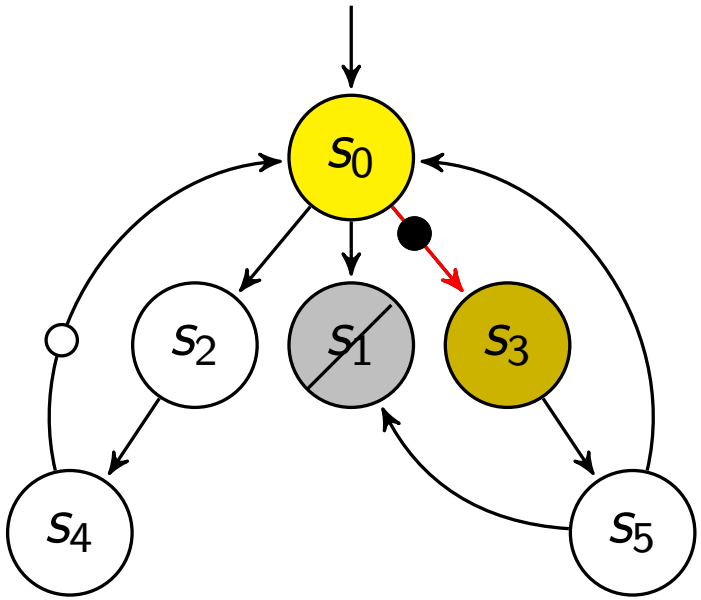




Idée Générale

Thread 1
(Tarjan-based)

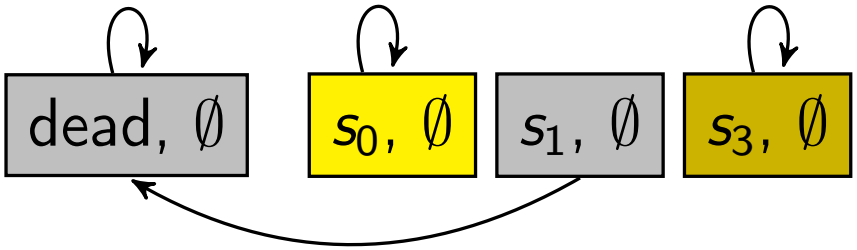
s_0



Thread 2
(Dijkstra-based)

s_0

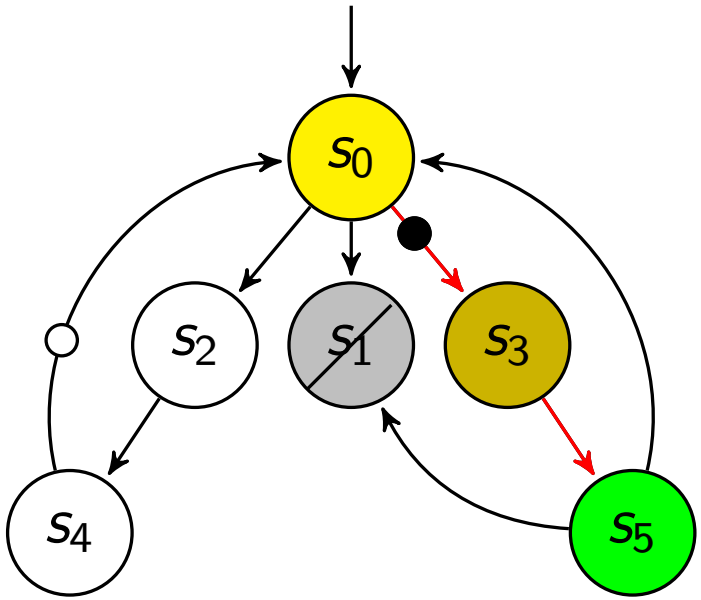
s_3



Idée Générale

Thread 1
(Tarjan-based)

s_0

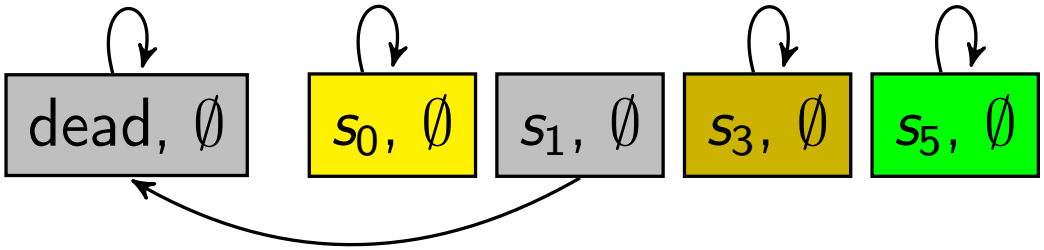


Thread 2
(Dijkstra-based)

s_0

s_3

s_5



Idée Générale

Thread 1
(Tarjan-based)

s_0

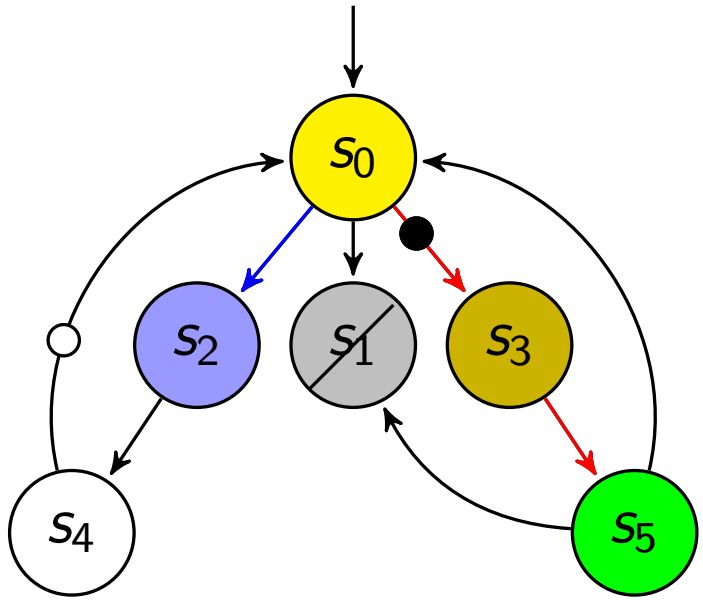
s_2

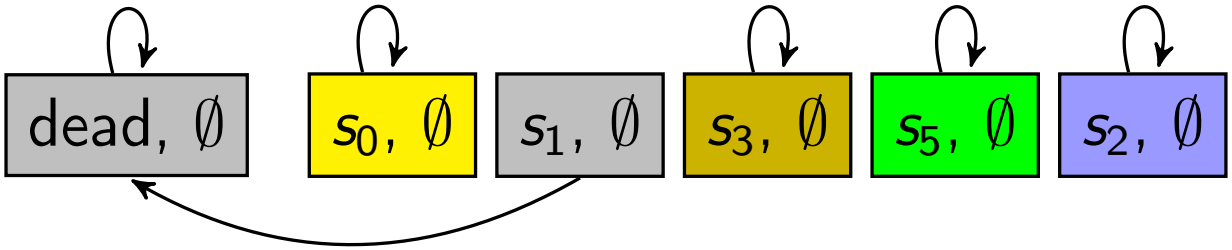
Thread 2
(Dijkstra-based)

s_0

s_3

s_5





Idée Générale

Thread 1
(Tarjan-based)

s_0

s_2

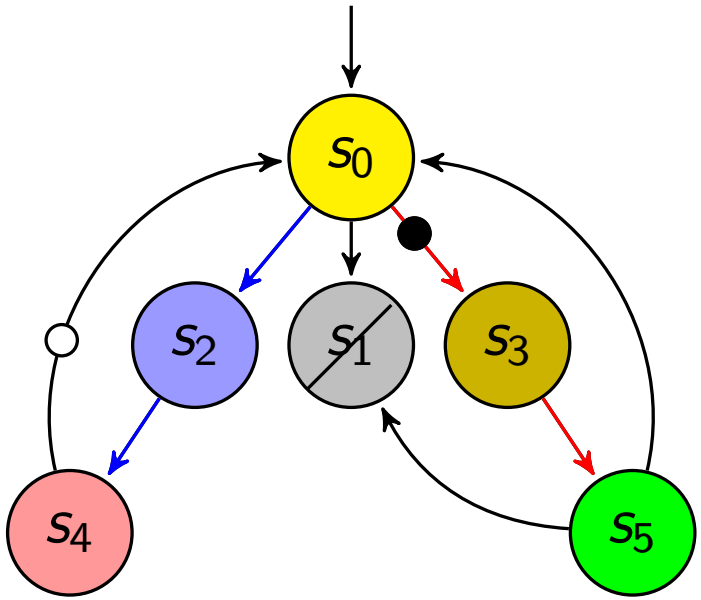
s_4

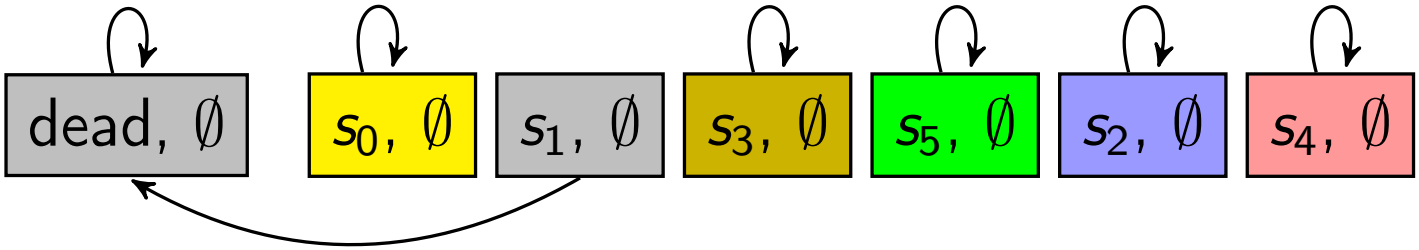
Thread 2
(Dijkstra-based)

s_0

s_3

s_5





Idée Générale

Thread 1
(Tarjan-based)

s_0

s_2

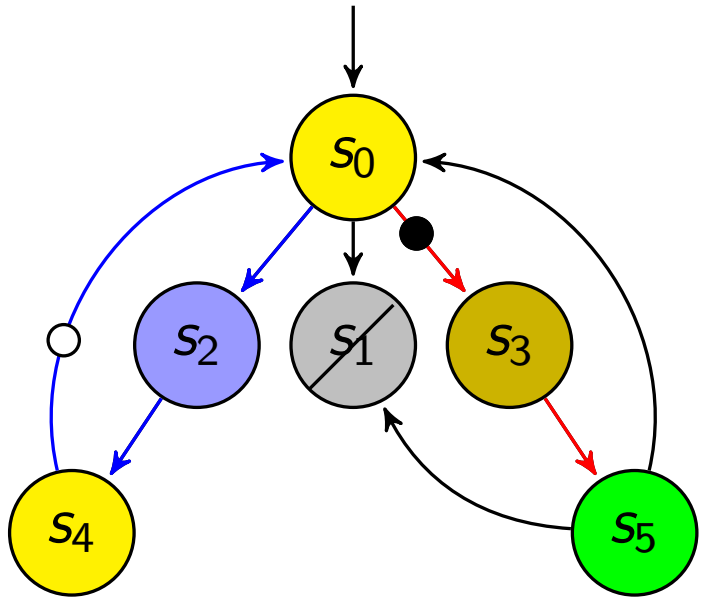
s_4

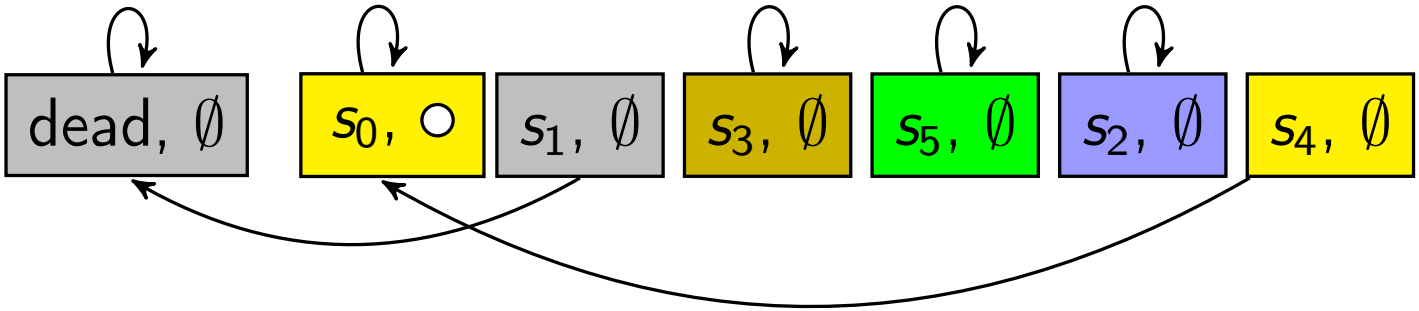
Thread 2
(Dijkstra-based)

s_0

s_3

s_5





Idée Générale

Thread 1
(Tarjan-based)

s_0

s_2

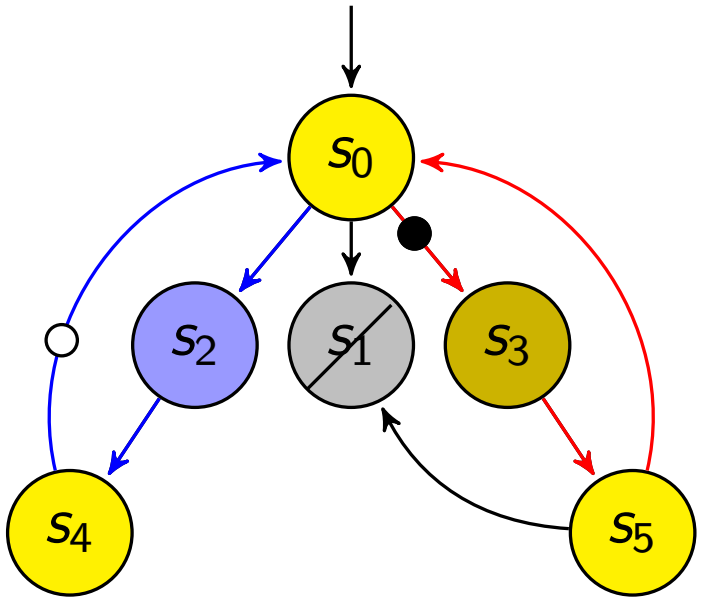
s_4

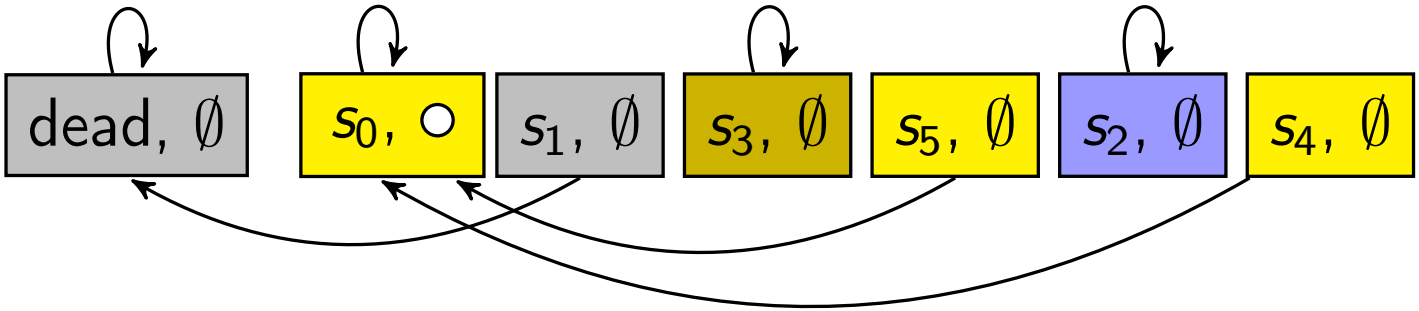
Thread 2
(Dijkstra-based)

s_0

s_3

s_5





Idée Générale

Thread 1
(Tarjan-based)

s_0

s_2

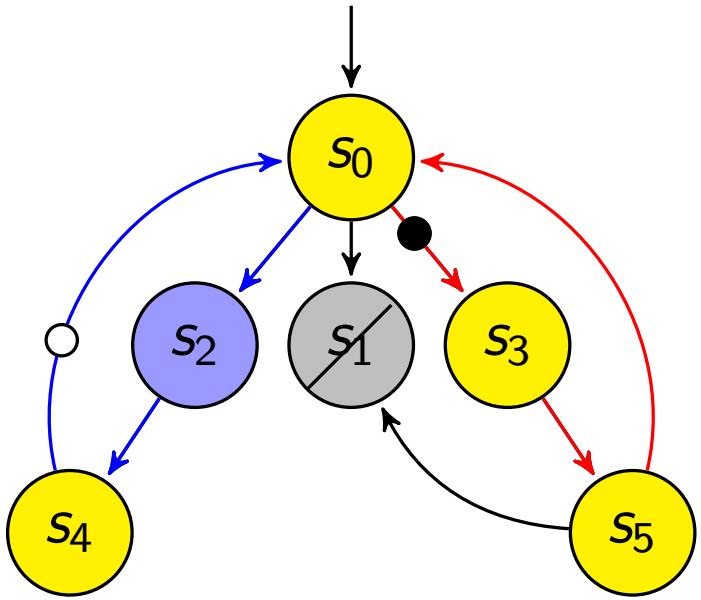
s_4

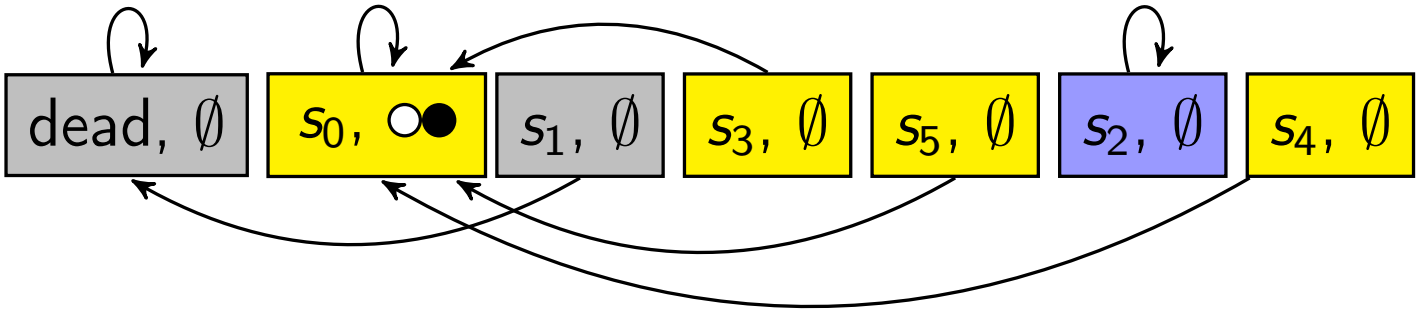
Thread 2
(Dijkstra-based)

s_0

s_3

s_5





Partial-Order Reduction

Combattre l'explosion combinatoire

Nous avons vu que les BDDs permettent de combattre l'explosion combinatoire en compressant les ensembles d'états

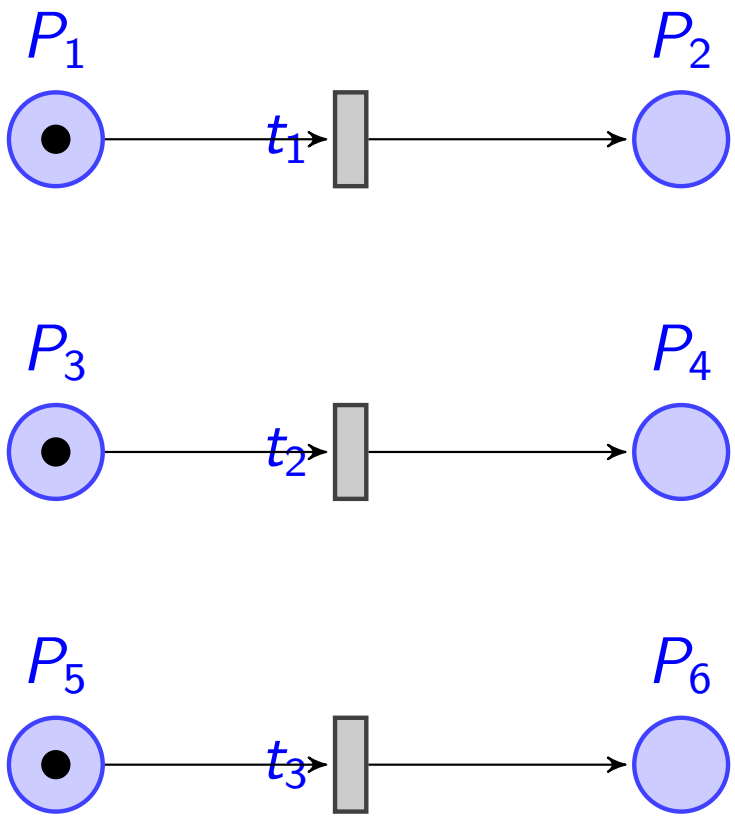
Une autre approche vise à réduire le système en un système équivalent mais plus petit

Question

Comment produire un tel système ?

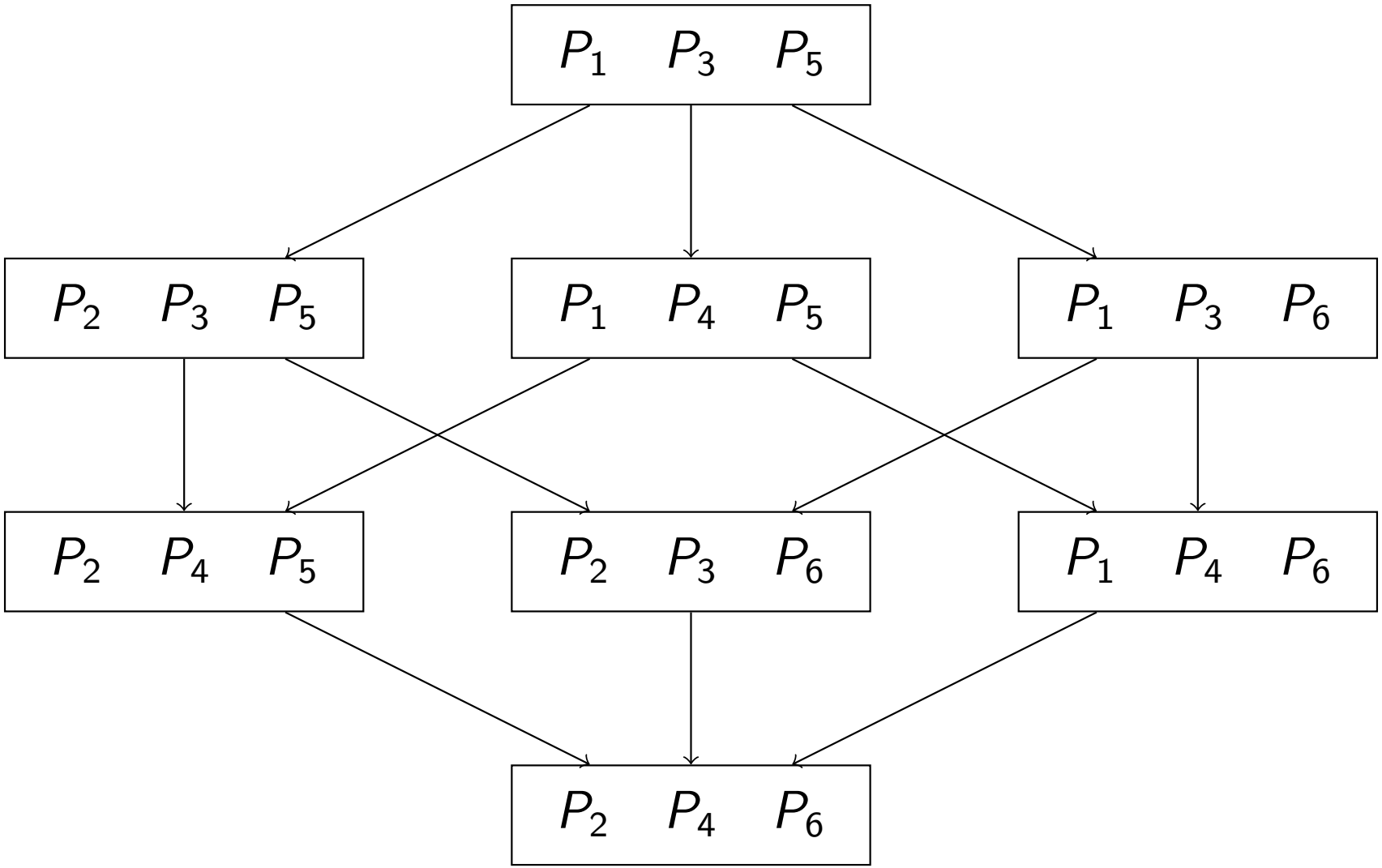
Exemple 1/2

Considérons le système suivant :

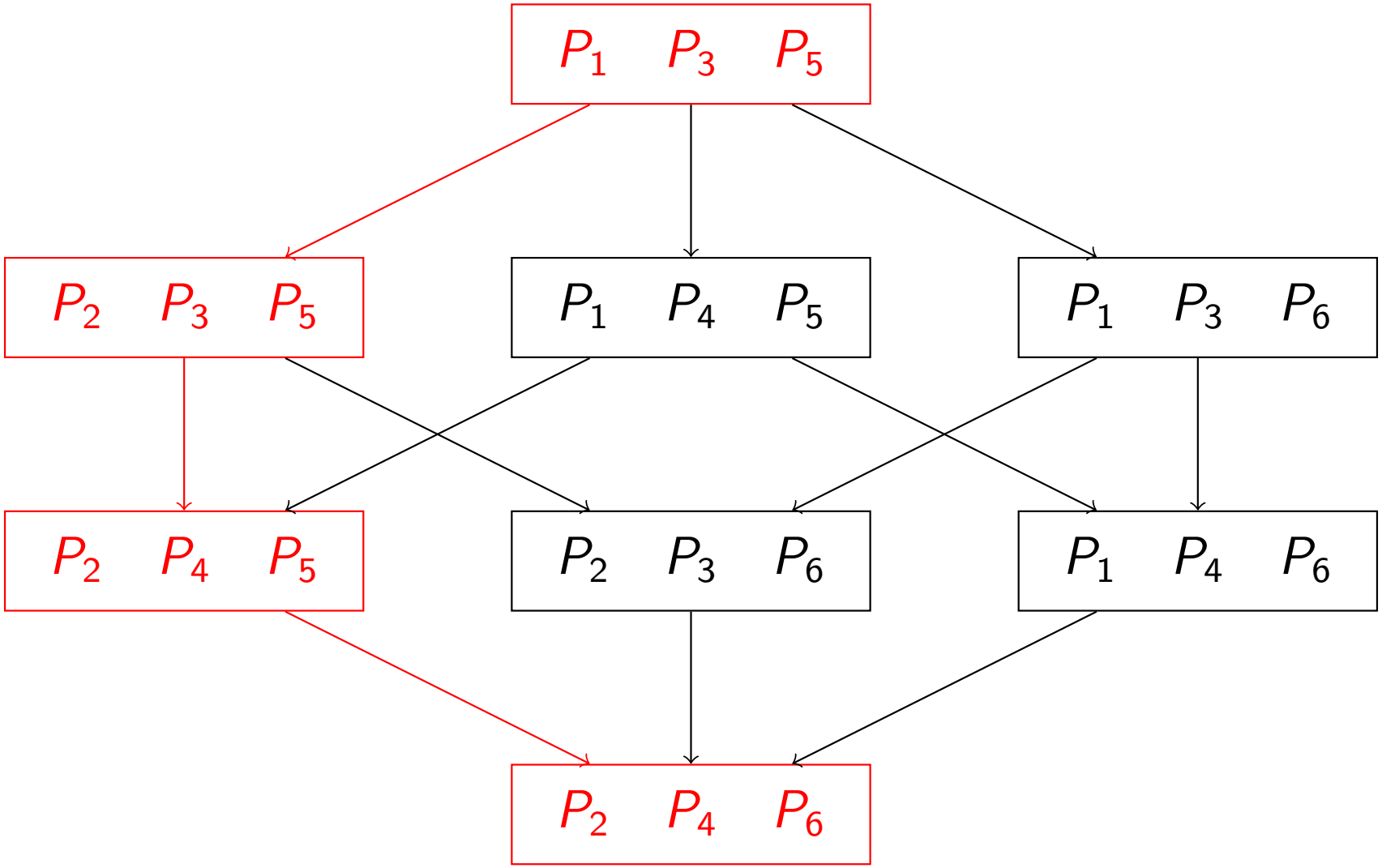


Exemple 2/2

On obtient le graphe d'accessibilité suivant :



Exploiter les entrelacement



Partial Order Reduction

Cette technique vise à réduire l'espace d'état en exploitant la concurrence entre les différents processus présents dans le système. Pour cela on essaye d'exploiter l'indépendance entre les transitions :

- l'affectation de variables qui ne sont pas interdépendantes, i.e. qui ne manipulent pas les mêmes variables
- l'envoi et la réception sur des canaux FIFO non-vides (ou non-pleins)

Remarque

Il serait absurde de construire le système tout entier pour ensuite chercher à détecter les transition indépendantes.

Choisir quelles transitions explorer doit se faire avant d'explorer les transitions d'un état dans le DFS !

Informations supplémentaires

Savoir quelles transitions explorer requiert d'avoir accès à des informations additionnelles. Ces informations dépendent des actions portées par les transitions :

- Indépendances des actions
  Est ce que deux actions sont indépendantes l'une de l'autre ?

- Visibilité des actions
  Est-ce qu'une action peut impacter la validité d'un proposition atomique ?

Définitions

Une structure de Kripke labellisée est un sextuplet $K = \langle AP, \mathcal{Q}, q^0, \delta, I, Act \rangle$ où

- AP est une ensemble fini de propositions atomiques,
- $\mathcal{Q}$ est un ensemble fini d'états (les nœuds du graphe),
- $q^0 \in \mathcal{Q}$ est l'état initial,
- $\delta \subseteq \mathcal{Q} \times Act \times \mathcal{Q}$ est une fonction indiquant les états successeurs d'un état,
- $I : \mathcal{Q} \mapsto 2^{AP}$ est une fonction indiquant l'ensemble des propositions atomiques satisfaites dans un état.

L'ensemble des actions actives pour un état s est noté $en(s)$ et défini par $en(s) = \{a | \exists s' : (s, a, s') \in \delta\}$.

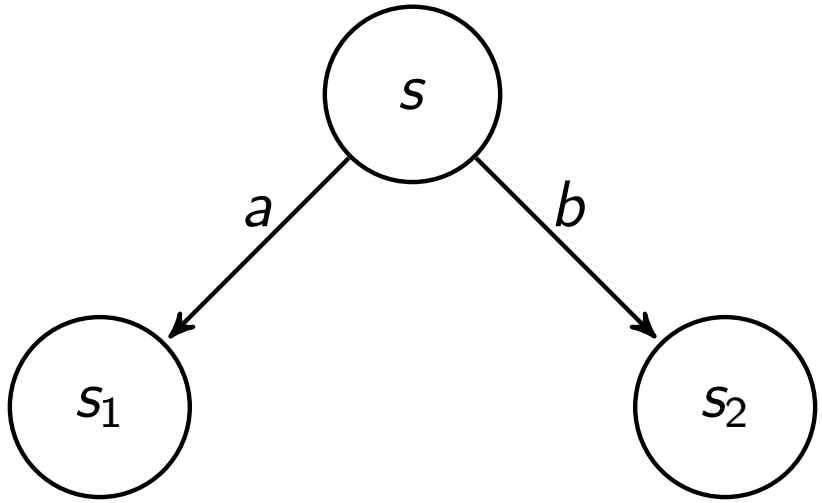
Par la suite on considérera des structures de Kripke déterministes, i.e. pour une action a il existe au plus un état s' t.q. on ait (s, a, s').

Indépendance des actions

$I \subseteq Act \times Act$ est une relation d'indépendance pour K si :

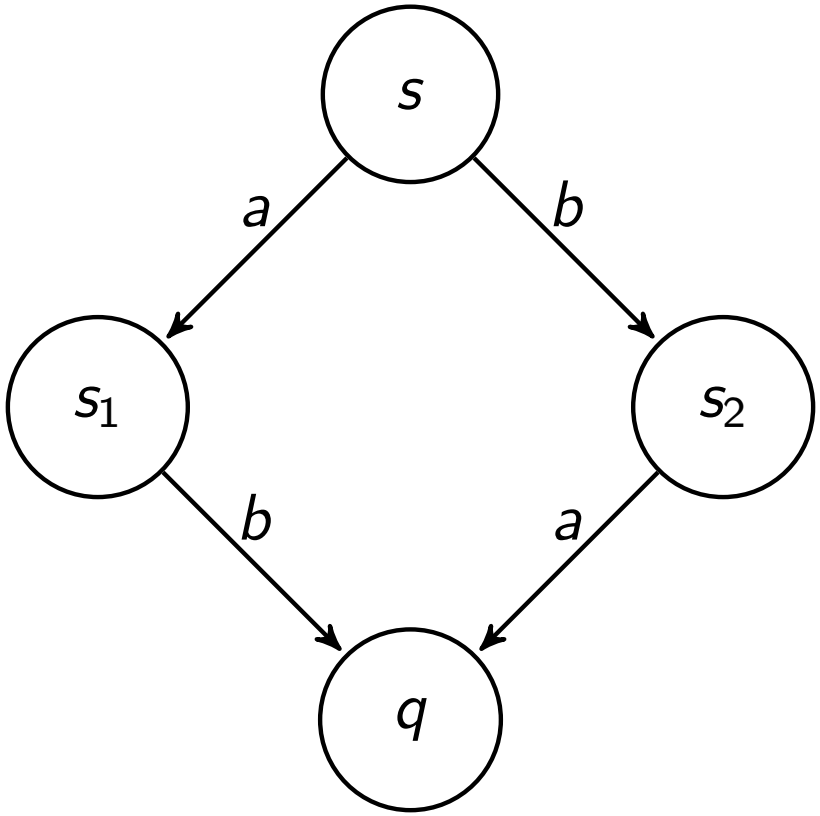
- pour tout $a \in Act$, on a $(a, a) \notin Act$ (non-reflexivité)
- pour tout $(a, b) \in I$, on a $(b, a) \in I$ (symétrie)
- pour $(a, b) \in I$ et $s \in Q$, on a :
  - **si** $a, b \in en(s)$ (i.e. $\exists s_1, s_2 \in Q$, t.q. (s, a, s_1) et (s, b, s_2))
  - **alors** $a \in en(s_2), b \in en(s_1)$ et $\exists q \in Q$ t.q. (s_1, b, q) et (s_2, a, q)

Indépendance des actions – Exemple



a et b sont indépendantes

Indépendance des actions – Exemple



a et b sont indépendantes cela signifie qu'il existe q tel que l'on puisse tirer b depuis s_1 et a depuis s_2

Invisibilité

Un ensemble $U \in Act$ est appelé un ensemble invisible si aucune action de U ne change la valeur d'une proposition atomique.

Motivations

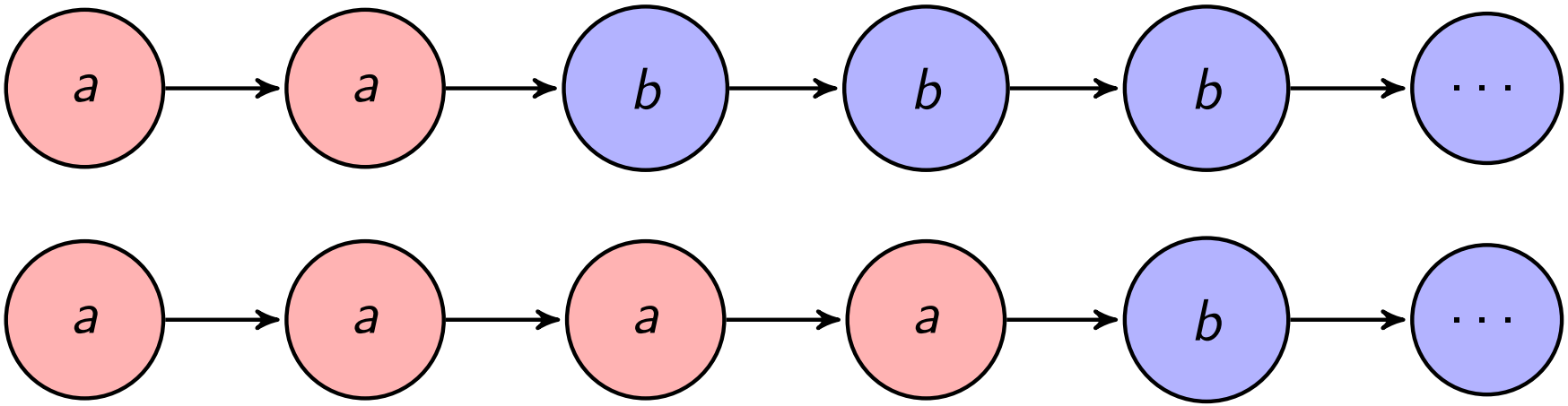
Les entrelacements entres les actions visibles ne peuvent être supprimés car ils peuvent avoir une influence sur la validité des propriétés LTL.

Exécutions équivalentes

On veut mettre en place des conditions sur la réduction qui garantissent que toutes les classes d'équivalence sont préservés dans le système réduit.

Suttering equivalence

Soit φ et ψ deux séquences sur 2^{AP}, φ et ψ sont stuttering équivalent ssi φ et ψ peuvent être partitionnées en block avec les même valuations, i.e. :



Structures de Kripke équivalente

La notion d'exécutions équivalentes peut être étendue au structures de Kripke.

Soit deux structures de Kripke K et K' sur le même ensemble de propositions AP.

K et K' sont appelés stuttering équivalents ssi pour toute séquence dans K il existe une séquence équivalente dans K'.

Autrement dit, K et K' contiennent les même classes d'équivalences d'exécutions

Stratégie

- On remplace K par une structure équivalente plus petite K'
- Pour tester si la propriété φ est vérifiée sur K on a juste besoin de tester si elle est vérifiée sur K'
- Il suffit simplement de générer K' directement depuis une exploration DFS de K (avec les informations extraites de I et U).

Plusieurs techniques existent pour réaliser cela efficacement :

- les ample set
- les sleep set
- les persistent set

Ample set

Pour chaque état s, on calcule l'ensemble $red(s) \subseteq en(s)$ qui contient l'ensemble des successeurs réduits d'un état. L'ensemble $red(s)$ doit être choisi en respectant les règles suivantes :

- $C0$: $en(s) = \emptyset$ ssi $red(s) = \emptyset$
- $C1$: chaque chemin de K commençant dans l'état s doit satisfaire : "Aucune action qui dépend d'une action de $red(s)$ se produit avant une action de $red(s)$"
- $C2$: si $red(s) \neq en(s)$ alors toutes les actions dans $red(s)$ sont invisibles
- $C3$: pour tous les cycles dans K' la règle suivante doit être respectée : "si $a \in en(s)$ pour un état s dans un cycle , alors $a \in red(s')$ pour un état s' dans le cycle"

Remarques

C0 assure qu'aucun *deadlock* ne sera introduit

C1 & C2 assurent que chaque classe de stutter équivalence est préservée

C3 assure qu'aucune action active ne sera omit pour toujours

Chaque cycle doit contenir un état complètement expansé !

Bit-State Hashing

Idée Générale

Objectif

Analyser des systèmes trop volumineux pour tenir en mémoire !

Dans le cas où la propriété est vérifiée, l'intégralité de l'espace d'état du produit synchronisé doit être exploré.

- Si un produit synchronisé possède N états de taille s et que la mémoire disponible est inférieure à $N * s$, alors le système ne peut être analysé

Idée

Compresser l'espace d'état dans un tableau T de taille m

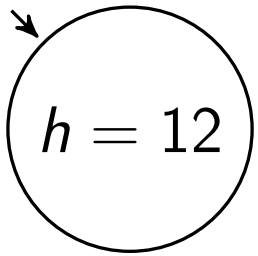
Algorithme

- Lorsqu'un état est découvert, on y applique une fonction de hachage
- Cette fonction retourne un entier i entre 0 et m
- Il suffit de regarder $T[i]$:
  - si $T[i] = \mathit{false}$ alors l'état n'a pas été visité, et l'on peut positionner $T[i]$ à $\top$
  - sinon, l'état est considéré comme ayant déjà été visité

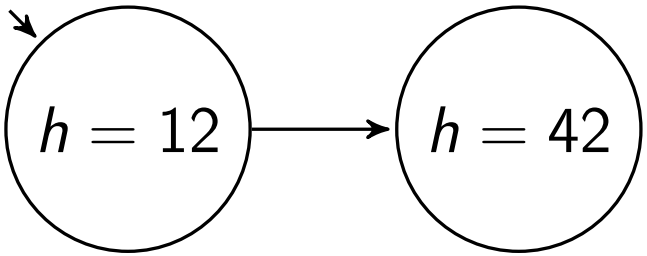
L'emptiness check devient alors une procédure de semi-décision car certains états ont pu être ignorés à cause des collisions dans T

- Si un contre-exemple est détecté, c'est un vrai contre-exemple
- Sinon, on ne peut rien dire

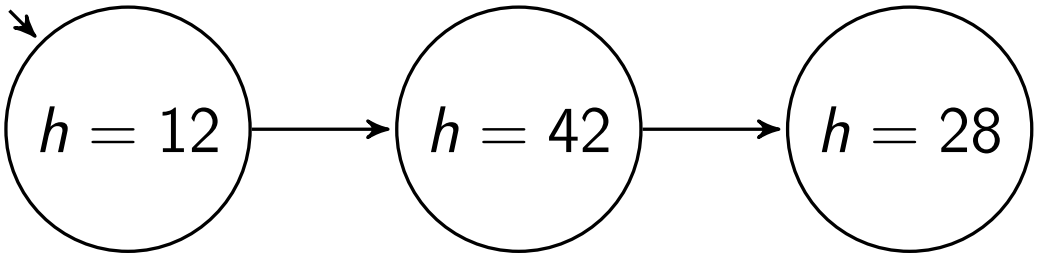
Exemple



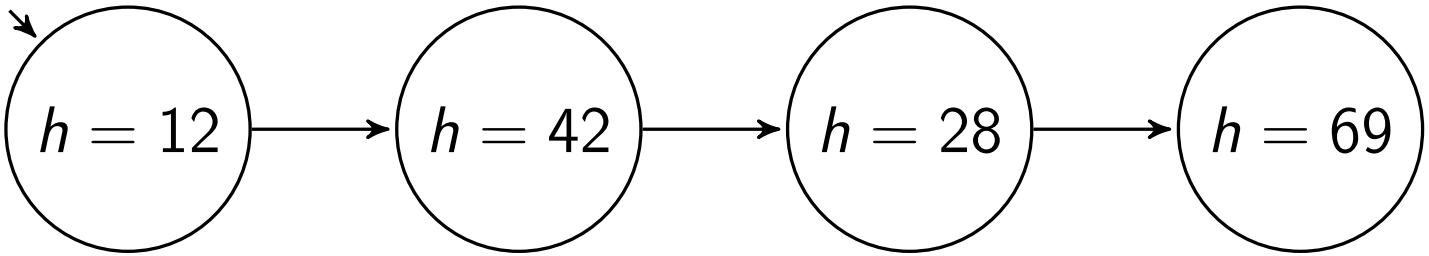
Exemple



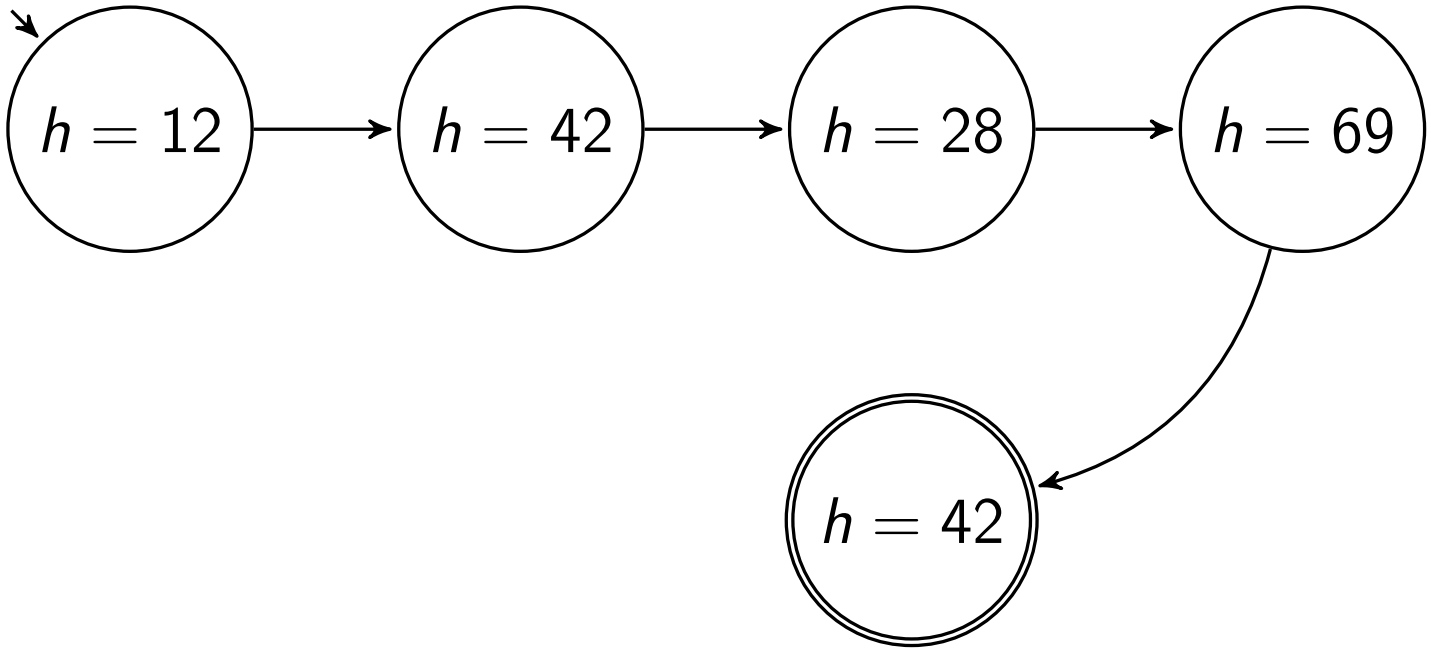
Exemple



Exemple

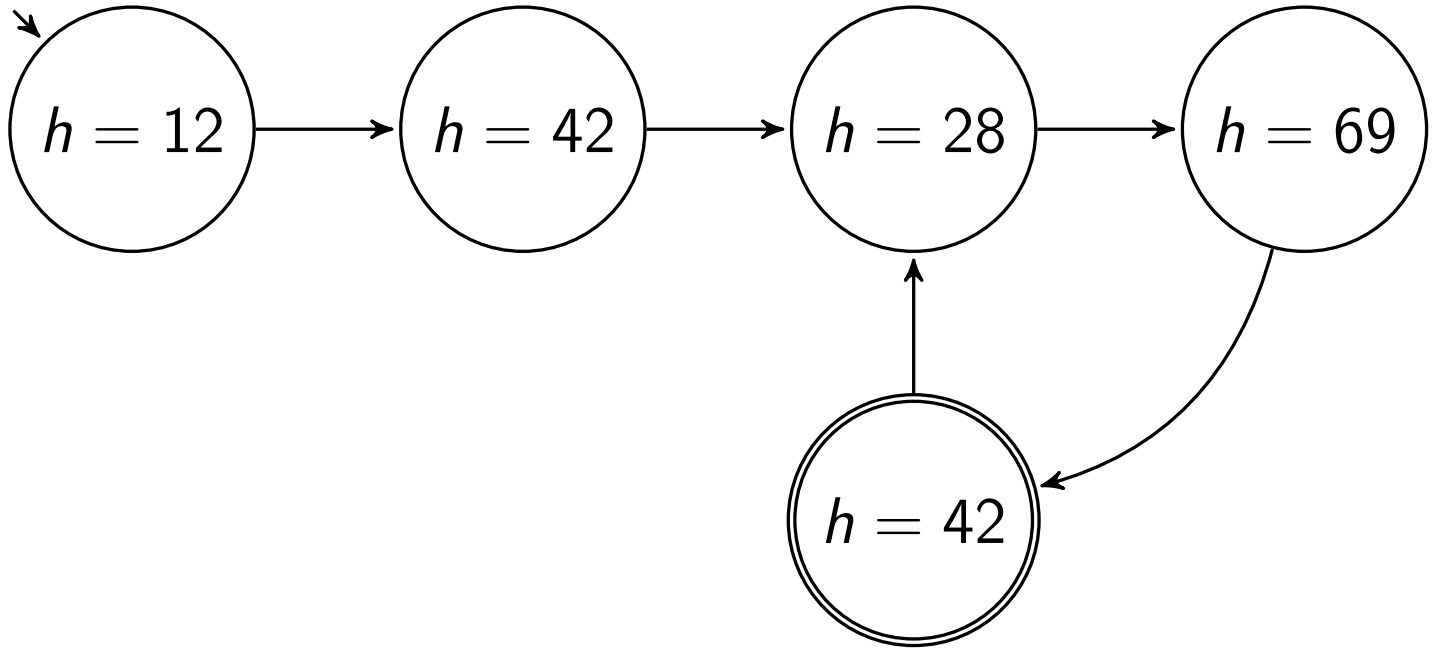


Exemple



Cet état ne sera pas exploré !

Exemple



Cet état ne sera pas exploré !

Ce cycle acceptant ne sera pas découvert !

Remarques

Le risque de collision peut entre réduit en faisant du multi-hachage

Cette technique est efficace pour la recherche de bug puisqu'il suffit d'augmenter régulièrement la taille des tables de hachages.

State Space Caching

Idée Générale

Objectifs

Fixer une borne mémoire en augmentant la redondance de travail effectué.

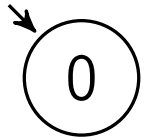
Il s'agit d'une technique exacte, mais qui peut conduire à de la redondance de travail

- Lorsque tous les successeurs d'un états ont été visités durant un DFS, cet état doit être conservé jusqu'à la fin du parcours pour éviter d'avoir à le reparcourir. Cette conservation peut avoir un impact non négligeable sur la mémoire utilisée.

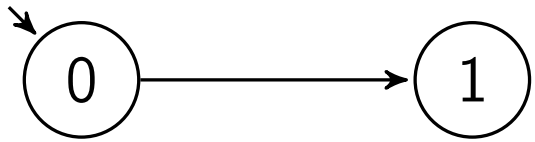
Idée

Ne conserver qu'un certain nombre d'état. Dès que la limite est atteinte les nouveaux états remplaceront les plus anciens

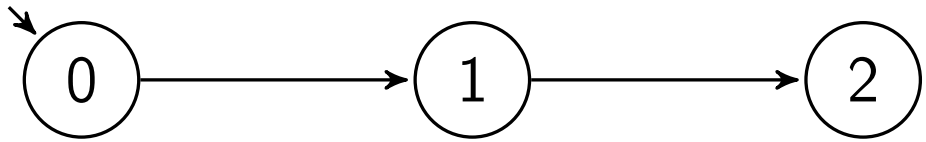
Exemple



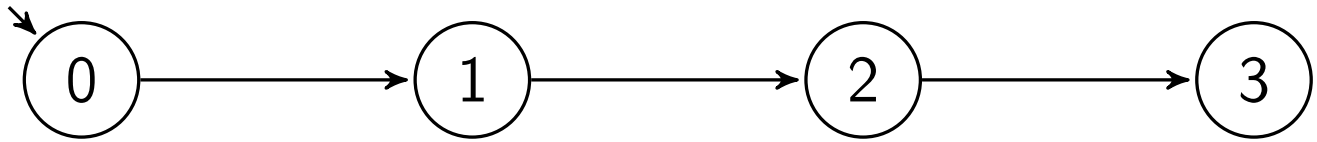
Exemple



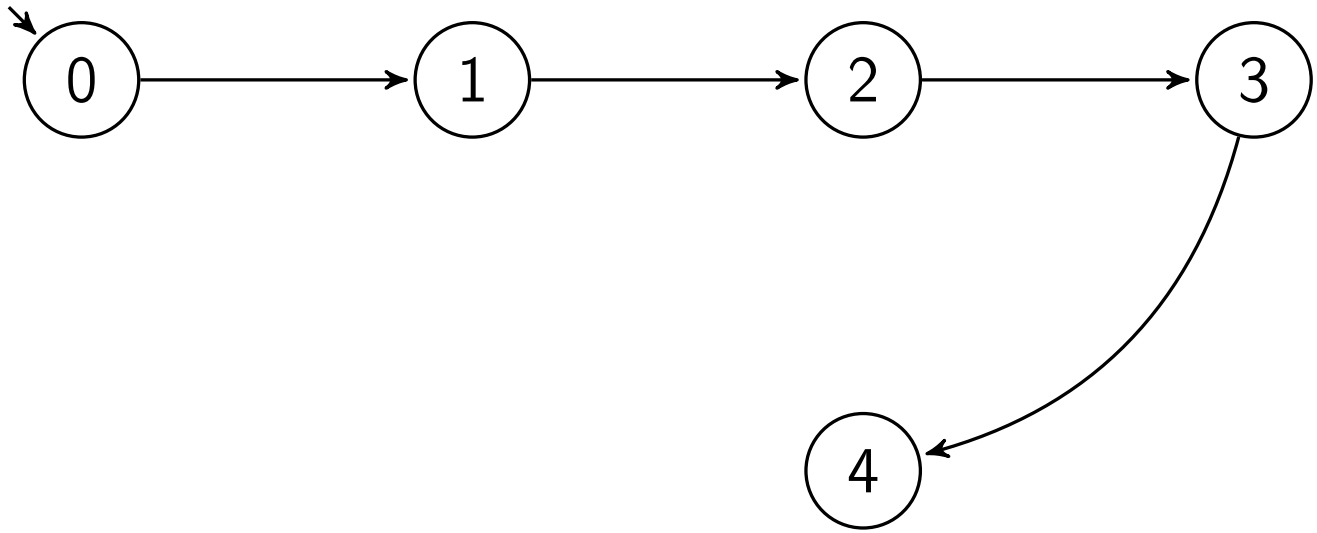
Exemple



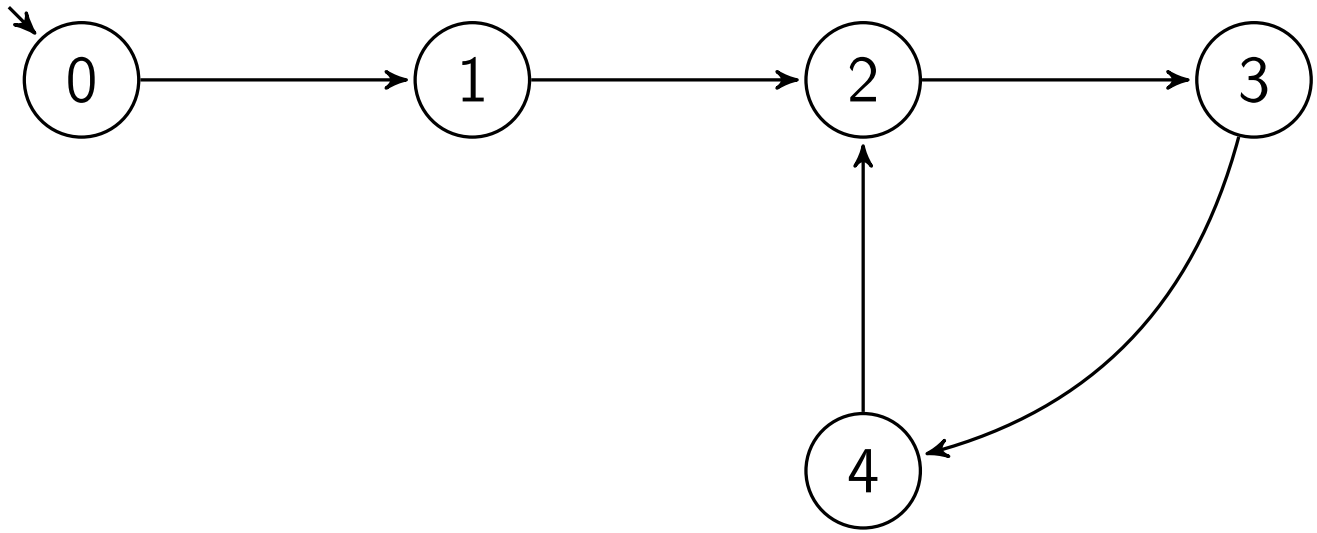
Exemple



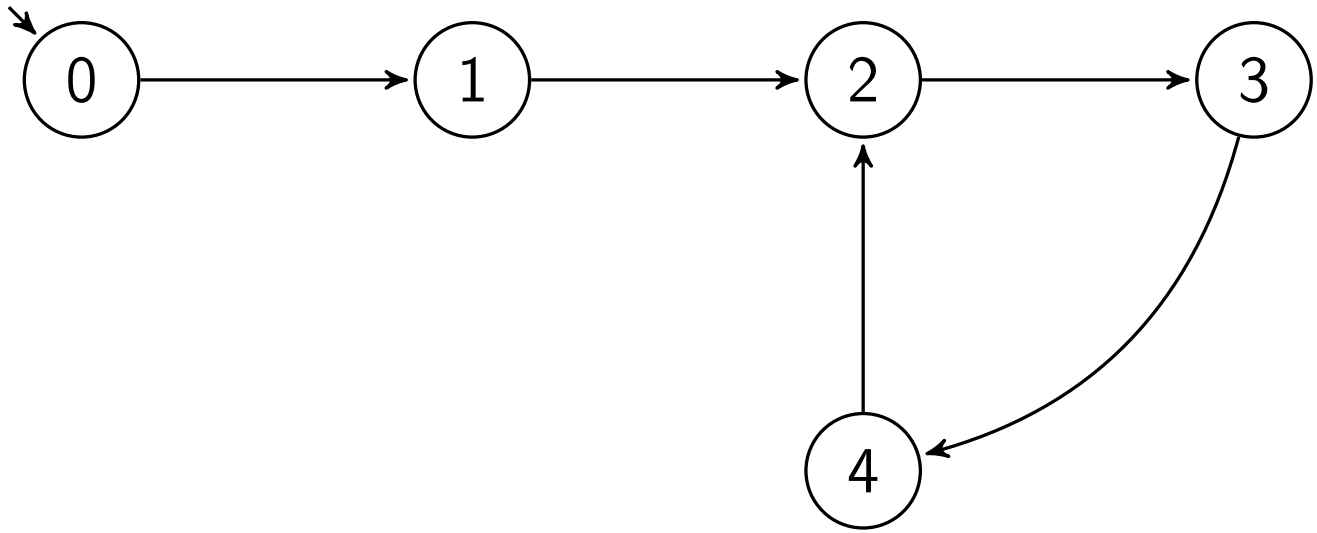
Exemple



Exemple

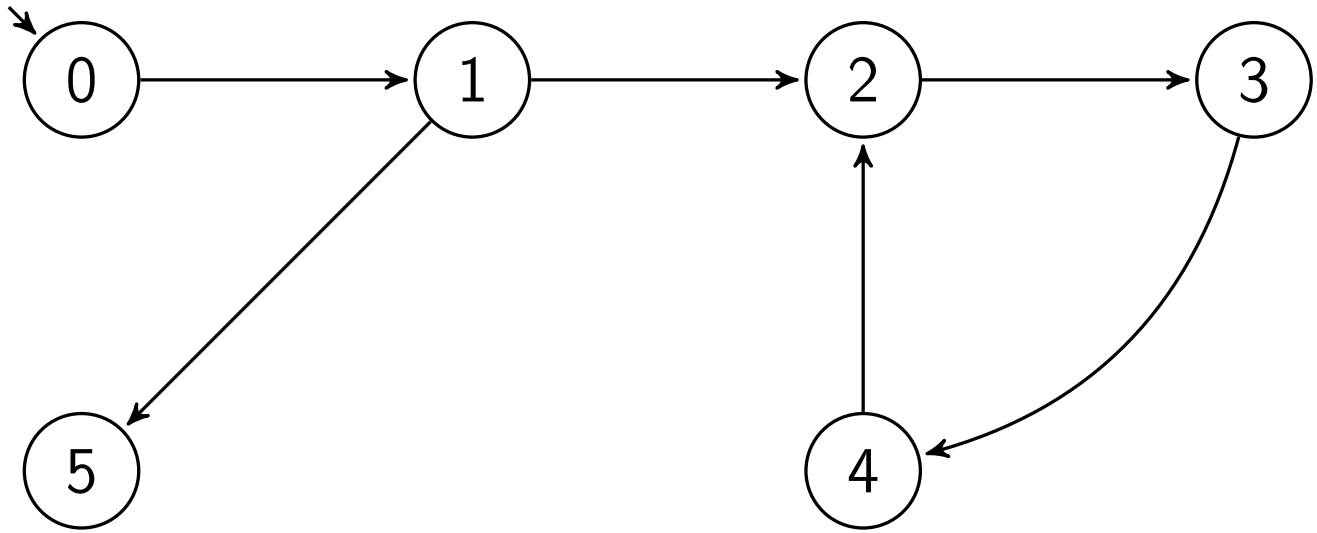


Exemple



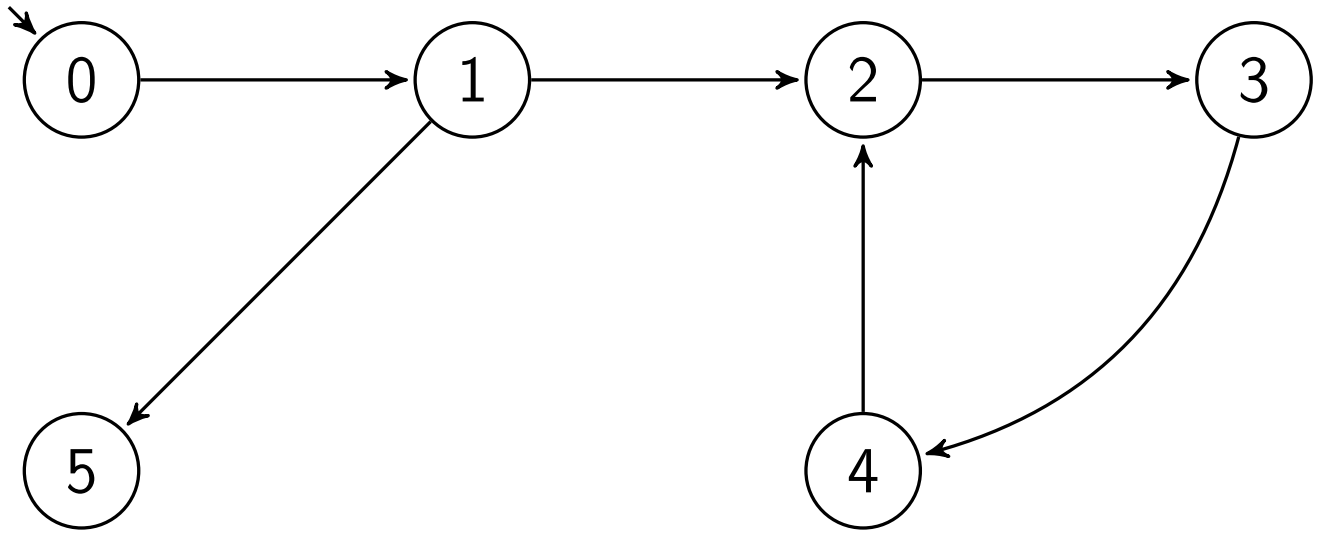
On dépile jusqu'à l'état 1. Le cache contient donc les états 4, 3, et 2 (car ils ne sont plus sur le DFS).

Exemple



On dépile jusqu'à l'état 1. Le cache contient donc les états 4, 3, et 2 (car ils ne sont plus sur le DFS).

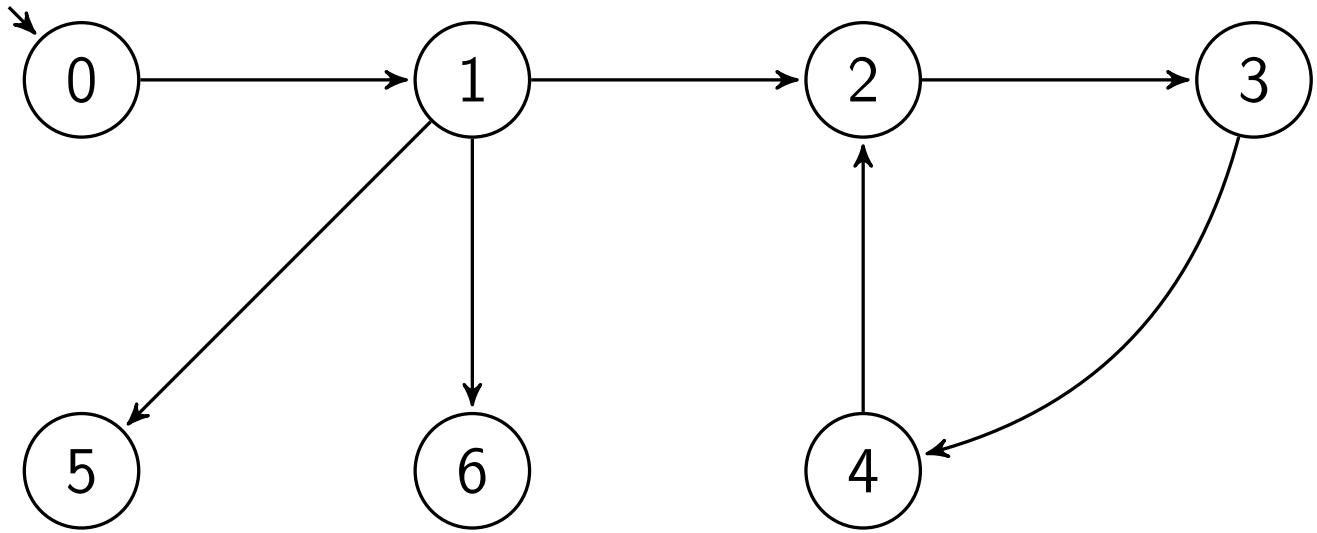
Exemple



On dépile jusqu'à l'état 1. Le cache contient donc les états 4, 3, et 2 (car ils ne sont plus sur le DFS).

Lorsque l'on dépile l'état 5, si le cache ne peut contenir que 3 états, on supprime l'état 4. Le cache contient donc les états 3, 2, 5.

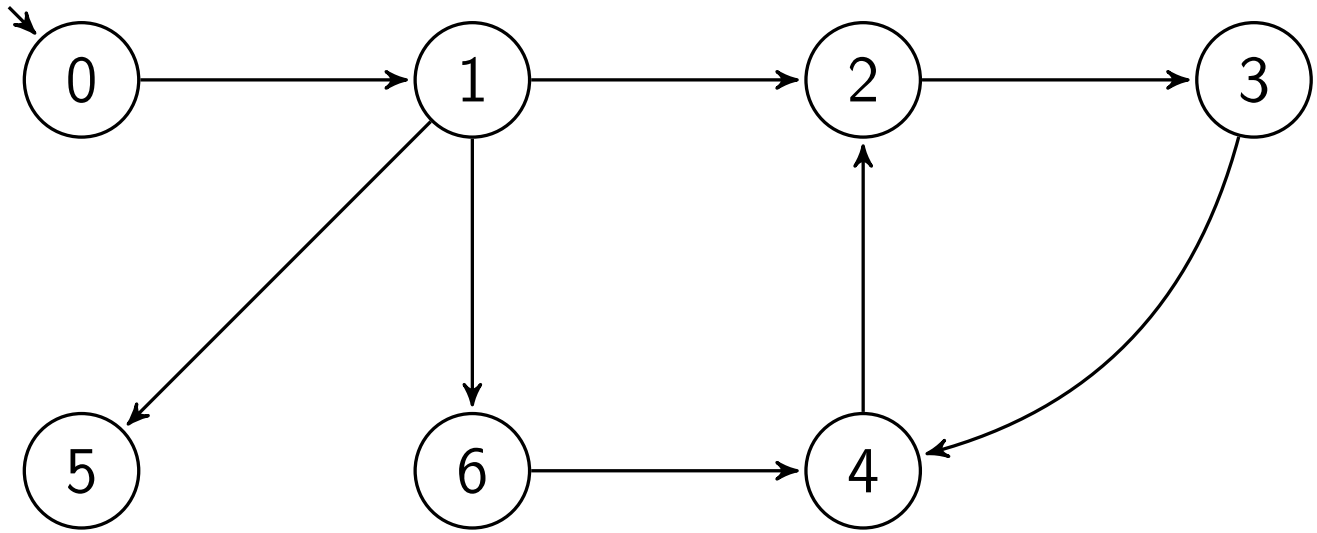
Exemple



On dépile jusqu'à l'état 1. Le cache contient donc les états 4, 3, et 2 (car ils ne sont plus sur le DFS).

Lorsque l'on dépile l'état 5, si le cache ne peut contenir que 3 états, on supprime l'état 4. Le cache contient donc les états 3, 2, 5.

Exemple



On dépile jusqu'à l'état 1. Le cache contient donc les états 4, 3, et 2 (car ils ne sont plus sur le DFS).

Lorsque l'on dépile l'état 5, si le cache ne peut contenir que 3 états, on supprime l'état 4. Le cache contient donc les états 3, 2, 5.

L'état 4 va être revisité !

Conclusion

- De nombreux emptiness checks existent
  - NDFS, SCC
  - Parallèles, Distribués
- Qui peuvent être combinés avec de nombreuses optimisations
- Toutes ces techniques sont complémentaires, il faut les combiner dans des approches porte-folio.