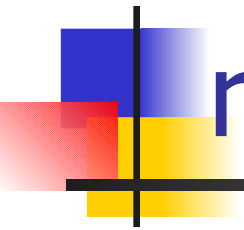
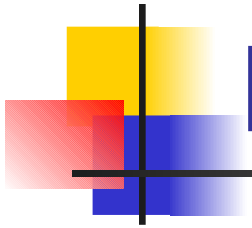


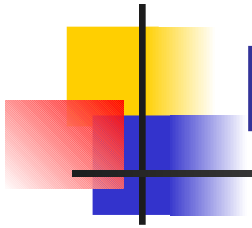
Méthodes informatiques de résolution du problème SAT

Anthony Monnet, doctorant
UQAM



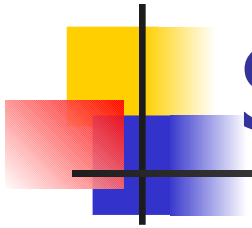
Plan

- Définition
- Encodage de problèmes
- Complexité théorique
- Algorithmes incomplets
- Algorithmes complets
  - DP
  - DPLL
    - Backjumping, apprentissage
  - Décompositions arborescentes



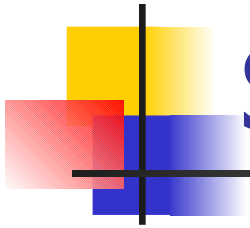
Formule propositionnelle

- Variables booléennes, connecteurs, pas de quantificateurs
- Définition récursive
  - Une variable booléenne x est une formule
  - Soient F et G deux formules
  - ¬F est une formule
  - F V G est une formule
  - F Λ G est une formule
  - F⟶G est une formule (équivalente à ¬F V G)
  - F↔G est une formule



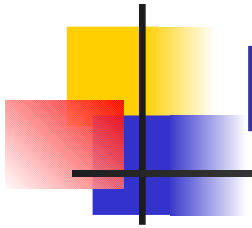
Satisfaisabilité

- Une instanciation $\theta=((x_1,b_1),(x_2,b_2),..)$ associe à des variables x_i des valeurs booléennes b_i.
- Appliquer θ à F consiste à remplacer toutes les variables x_i dans θ par les valeurs b_i correspondantes.
- Une instanciation complète θ satisfait la formule F ssi F(θ) ≡VRAI.
  - θ est un modèle de F.
- F est satisfaisable ssi il existe une instanciation complète qui la satisfait.



Satisfaisabilité

- $F=(a \rightarrow b) V \neg(a \vee b)$
- $\theta=((a,faux),(b,vrai))$
- θ est un modèle de F



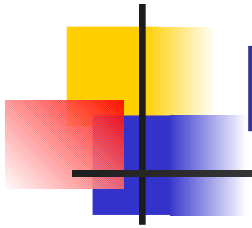
Forme normale conjonctive

- Un littéral: une variable ou une négation de variable
  - v, ¬x
- Une clause: une disjonction de littéraux
  - x V ¬z V y
- Formule en FNC: une conjonction de clauses
  - (x V ¬z V y) Λ (¬y V k) Λ (¬k V ¬x)
- Pour toute formule propositionnelle, il existe une formule en FNC logiquement équivalente

Le problème SAT

- Entrée: une formule propositionnelle F en forme normale conjonctive
- Sortie: F est-elle satisfaisable?
  - Si oui, fournir un modèle.
- Pourquoi en FNC?
  - Forme normale utilisée par les algorithmes
  - F est satisfaite ssi chaque clause a au moins un littéral vrai
  - Ensemble de clauses ≡ ensemble de contraintes à satisfaire

$(x \vee \neg z \vee y) \wedge (\neg y \vee k) \wedge (\neg k \vee \neg x)$

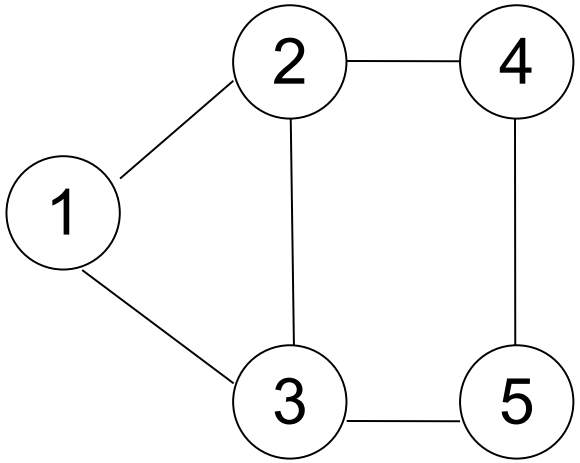


Encodage de problèmes

- Utilité pratique de SAT
- Encoder un problème en logique propositionnelle
- Utiliser un solveur SAT
- Retranscrire le modèle pour trouver une solution au problème

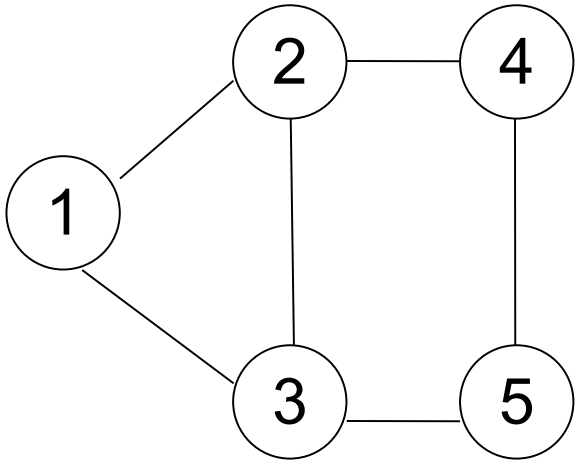
Encodage de problèmes

- Coloration de graphe
  - Définition de variables
    - j_i ≡ le noeud i est jaune
    - r_i ≡ le noeud i est rouge
    - b_i ≡ le noeud i est bleu



Encodage de problèmes

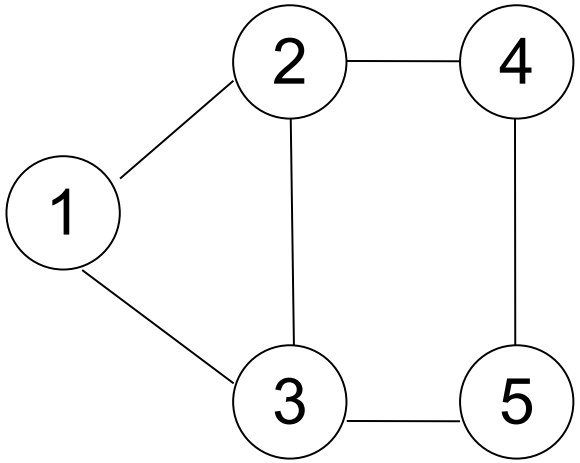
- Coloration de graphe



  - Définition de contraintes
    - Un noeud est d'une et une seule couleur
      - j_i V r_i V b_i
      - $\neg j_i$ V $\neg r_i$
      - $\neg j_i$ V $\neg b_i$
      - $\neg r_i$ V $\neg b_i$
    - Pour i=1..5

Encodage de problèmes

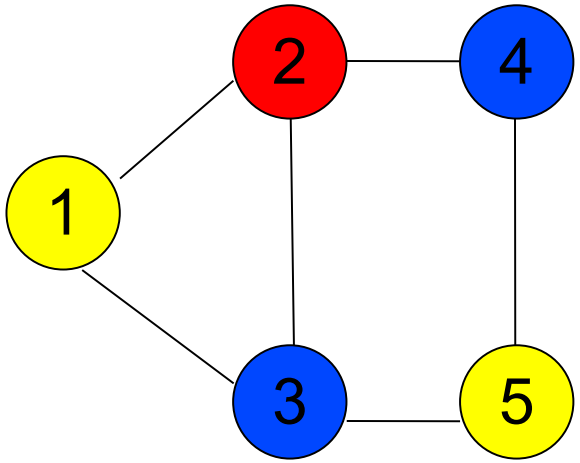
- Coloration de graphe



- Définition de contraintes
  - Deux noeuds adjacents ne peuvent pas être de la même couleur
    - $\neg j_i \vee \neg j_k$
    - $\neg r_i \vee \neg r_k$
    - $\neg b_i \vee \neg b_k$
  - Pour tous les couples de noeuds i et k adjacents

Encodage de problèmes

- Coloration de graphe



  - Le problème est satisfaisable
    - Une solution:

$j_1 \quad \neg r_1 \quad \neg b_1$

$\neg j_2 \quad r_2 \quad \neg b_2$

$\neg j_3 \quad \neg r_3 \quad b_3$

$\neg j_4 \quad \neg r_4 \quad b_4$

$j_5 \quad \neg r_5 \quad \neg b_5$

Encodage de problèmes

- Vérification de circuits électroniques

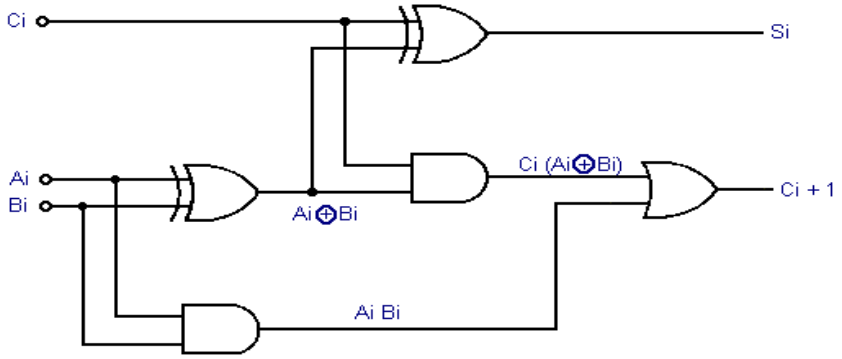


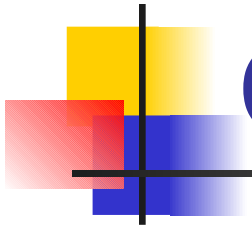
Fig. 7. - Exemple de schéma logique d'un addtionneur complet.

- Encoder logiquement le circuit, l'entrée et la sortie désirées. Si SAT, circuit correct
- Eviter coûts de construction d'un modèle réel

Encodage de problèmes

- Vérification de modèles (model checking)

- Modélisation d'un système (algorithme, protocole)
  - Etats + transitions possibles
- Exprimer une propriété à vérifier
- Vérification bornée (bounded model checking):
  - Encoder la modélisation + la négation de la propriété
  - SAT => il existe un contre-exemple



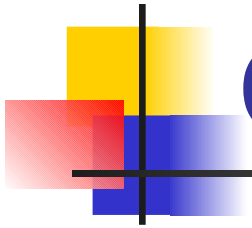
Complexité théorique

- Mesure de l'efficacité théorique d'un algorithme: complexité asymptotique
- Temps d'exécution borné par une fonction de la taille du problème
  - Logarithmique ou linéaire: très efficace
  - Polynomial: efficace
  - Exponentiel: inefficace

Complexité théorique

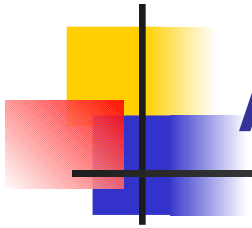
- Comparaison de comportements asymptotiques

Taille du problème	n	n^2	n^3	(2^n)/2
1	1ms	1ms	1ms	1ms
10	10ms	100ms	1s	512ms
20	20ms	400ms	8s	8min44s
30	30ms	900ms	27s	6j5h
100	100ms	10s	16min40s	20Md de Md d'années



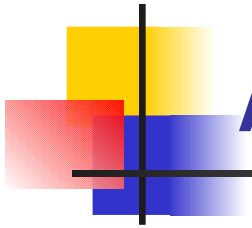
Complexité théorique

- SAT est un problème NP-complet (Cook, 1971)
- On ne connaît pas d'algorithme polynomial pour sa résolution, et on ne sait pas s'il peut en exister.
- On étudie expérimentalement l'efficacité des algorithmes sur des problèmes types.



Algorithmes incomplets

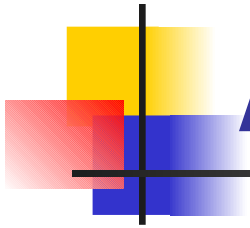
- Classification des algorithmes
  - Correct / incorrect
  - Complet / incomplet
- Un algorithme incomplet peut échouer à retourner une réponse
- Recherche locale: on ne cherche pas à tester exhaustivement toutes les instanciations possibles



Algorithmes incomplets

- **Hill Climbing**
- Départ: une instanciation aléatoire + un critère de qualité (nombre de clauses non satisfaites)
- A chaque étape
  - Choisir une variable
  - Inverser la valeur de la variable si cela améliore le critère.
  - Sinon, inverser avec une probabilité faible.

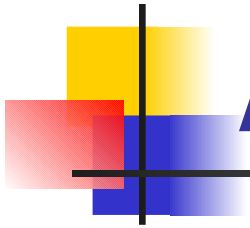
- Inversion possible sans amélioration: éviter les optimums locaux



Algorithmes incomplets

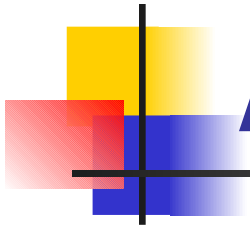
- **Algorithmes génétiques**
- Inspirés de la sélection naturelle
- Départ: une population d'instanciations aléatoire
- A chaque étape, brassage génétique
  - « croisement » des meilleures individus
  - « mutation » de certains individus
  - Élimination des solutions les plus faibles

- Autres variantes d'algorithmes incomplets: recuit simulé, colonies de fourmis…



Algorithmes incomplets

- Avantages:
  - Possible de trouver un modèle rapidement si le problème est satisfaisable
- Inconvénients:
  - Pas sûr de trouver un modèle, même si la formule est SAT
  - Si la formule est non-SAT, aucune réponse.
- A utiliser
  - Si on pense que la formule est SAT
  - Si on a besoin d'un modèle



Algorithmes complets

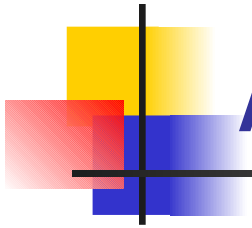
- Donnent toujours une réponse (si assez de temps et d'espace)

- Algorithme DP: résolution
- Algorithme DPLL: recherche
- Algorithmes de décomposition arborescente: recherche

Algorithme DP

- Davis-Putnam (1960)
- Résolution de deux clauses
  - $C_1 = x_1 \vee x_2 \vee .. \vee x_n \vee z$
  - $C_2 = y_1 \vee y_2 \vee .. \vee y_m \vee \neg z$
  - $C = x_1 \vee x_2 \vee .. \vee x_n \vee y_1 \vee y_2 \vee .. \vee y_m$ est la résolvante de C1 et C2

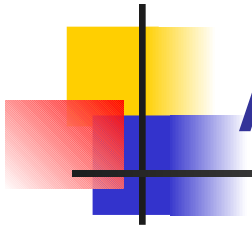
C est SAT <=> $C_1 \wedge C_2$ est SAT



Algorithme DP

- Pour toute variable v
  - Pour toute clause c1 contenant v
    - Pour toute clause c2 contenant ¬v
      - Ajouter la résolvante de c1 et c2
  - Retirer toutes les clauses contenant v ou ¬v

- La formule est satisfaisable ssi au cours des résolutions on n'obtient aucune clause vide.



Algorithme DP

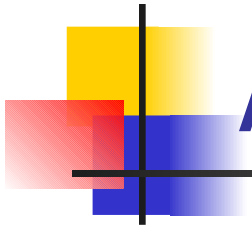
aV¬c

aV¬bVc

¬aVc

¬aVb

bVc



Algorithme DP

aV¬c

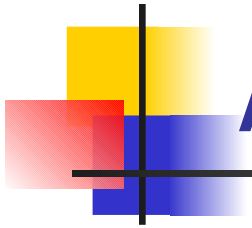
aV¬bVc

¬aVc

¬aVb

bVc

cV¬c



Algorithme DP

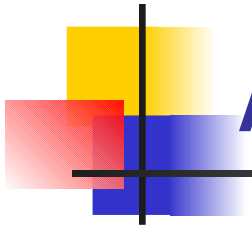
aV¬c

aV¬bVc bV¬c

¬aVc

¬aVb

bVc



Algorithme DP

aV¬c

aV¬bVc

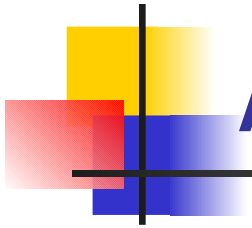
¬aVc

¬aVb

bVc

bV¬c

¬bVc



Algorithme DP

aV¬c

aV¬bVc

¬aVc

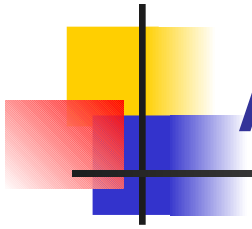
¬aVb

bVc

bV¬c

¬bVc

bV¬bVc



Algorithme DP

aV¬c

aV¬bVc bV¬c

¬aVc ¬bVc

¬aVb

bVc

Algorithme DP

bV¬c cV¬c

¬bVc

bVc

Algorithme DP

bV¬c c

¬bVc

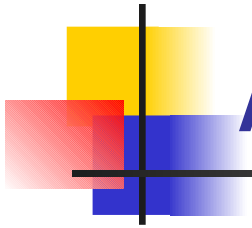
bVc

Algorithme DP

bV¬c c

¬bVc

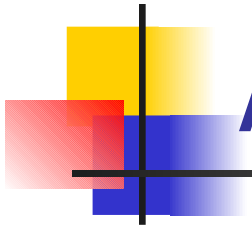
bVc



Algorithme DP

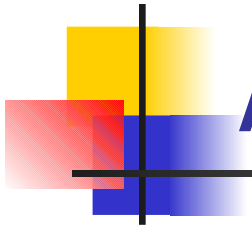
c

Formule satisfaisable



Algorithme DP

- Inconvénients:
  - Complexité temporelle et spatiale très mauvaise (explosion du nombre de clauses)
  - Il est prouvé que certains problèmes demandent un temps et un espace exponentiels selon le nombre de variables
  - Si la formule est SAT on peut retrouver un modèle, mais il faut avoir conservé toutes les clauses depuis le début!

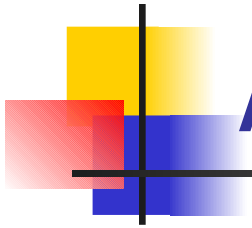


Algorithme DPLL

- Davis-Putnam-Logemann-Loveland (1962)
- Algorithme de recherche explicite et exhaustif dans l'espace des instanciations
- Recherche en profondeur:
  - Construire progressivement un modèle, variable par variable
  - Si une clause devient insatisfaisable, retour en arrière (backtrack)

Algorithme DPLL

- DPLL(F,θ)
  - Si F(θ) est vide, retourner VRAI
  - Si F(θ) contient une clause vide, retourner FAUX
  - Si F(θ) contient une clause unitaire p *(resp. ¬p)* , retourner DPLL(F,θ U (p,VRAI *(resp. FAUX)*))
  - Sinon,
    - Choisir une variable p et une valeur booléenne b
    - Si DPLL(F, θ U (p,b)) retourne VRAI alors retourner VRAI
    - Sinon retourner DPLL(F, θ U (p,¬b))



Algorithme DPLL

aVc

¬aV¬bVdVc

bVd

¬bV¬d

aV¬d

¬cV¬d

Algorithme DPLL

aVc

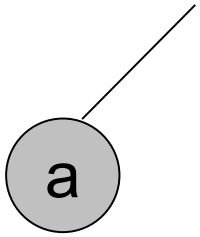
¬aV¬bVdVc

bVd

¬bV¬d

aV¬d

¬cV¬d



Algorithme DPLL

aVc

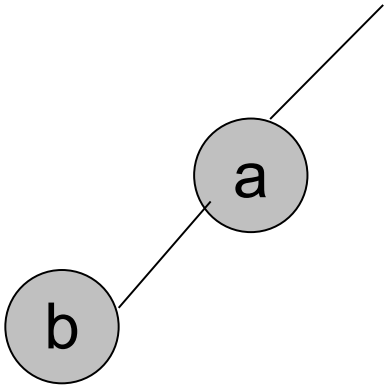
¬aV¬bVdVc

bVd

¬bV¬d

aV¬d

¬cV¬d



Algorithme DPLL

aVc

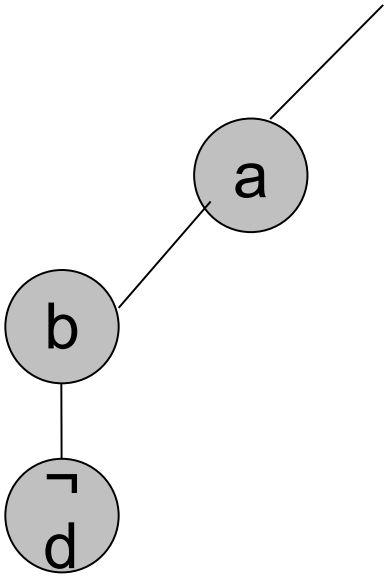
¬aV¬bVdVc

bVd

¬bV¬d

aV¬d

¬cV¬d



Algorithme DPLL

aVc

¬aV¬bVdVc

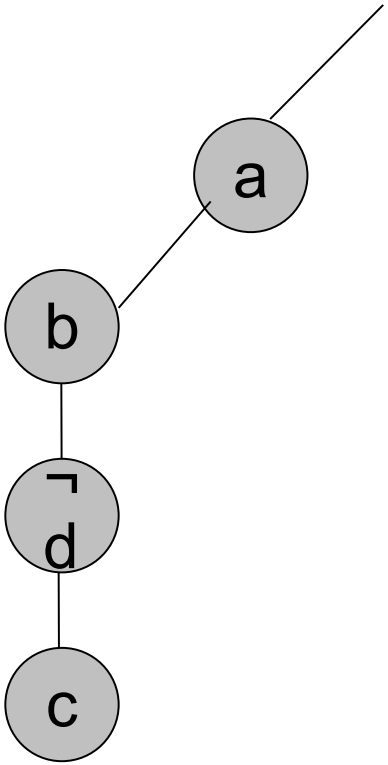
bVd

¬bV¬d

aV¬d

¬cV¬d

contradiction



Algorithme DPLL

aVc

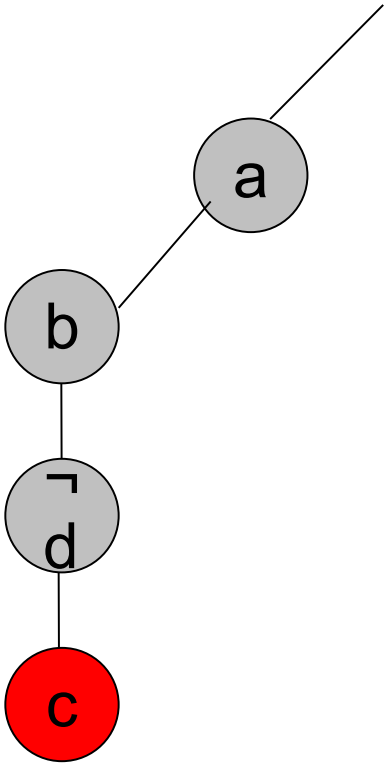
¬aV¬bVdVc

bVd

¬bV¬d

aV¬d

¬cV¬d



Algorithme DPLL

aVc

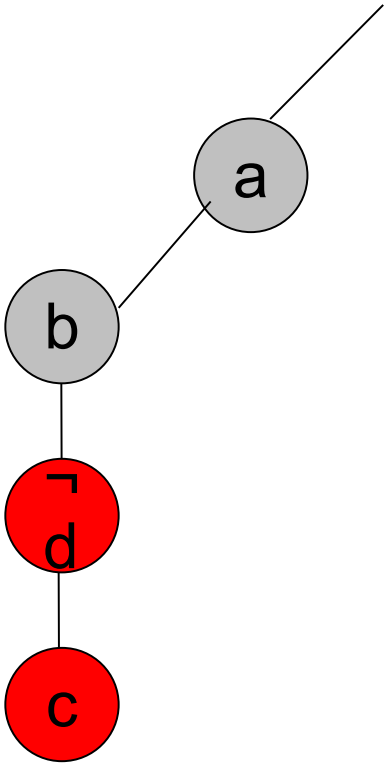
¬aV¬bVdVc

bVd

¬bV¬d

aV¬d

¬cV¬d



Algorithme DPLL

aVc

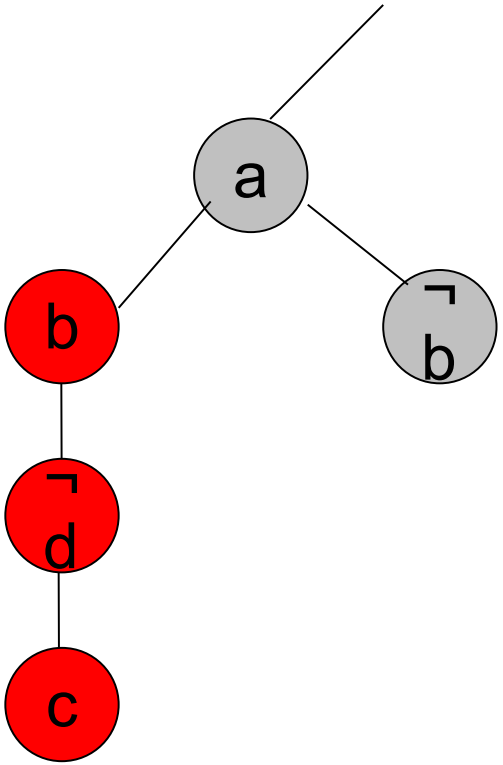
¬aV¬bVdVc

bVd

¬bV¬d

aV¬d

¬cV¬d



Algorithme DPLL

aVc

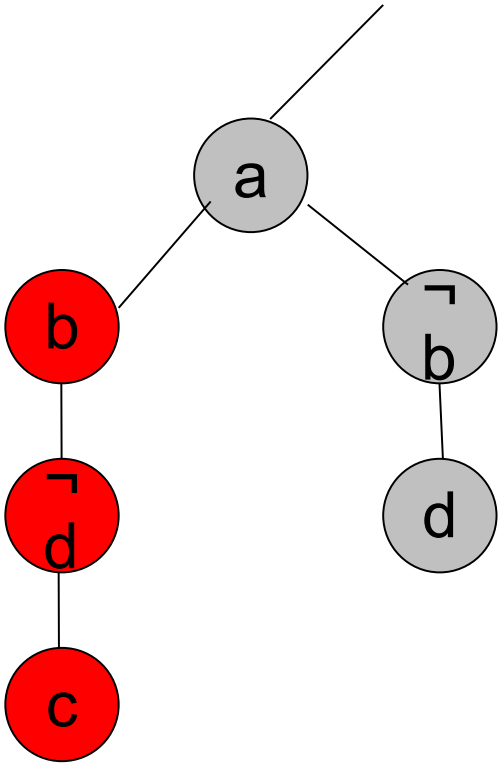
¬aV¬bVdVc

bVd

¬bV¬d

aV¬d

¬cV¬d



Algorithme DPLL

aVc

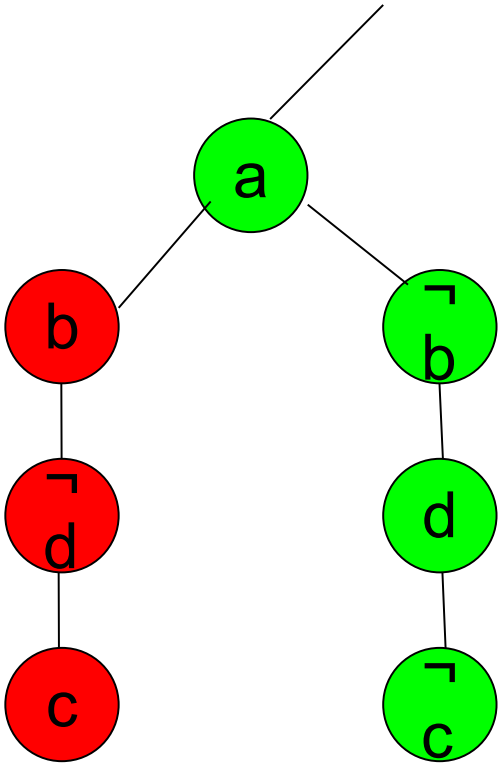
¬aV¬bVdVc

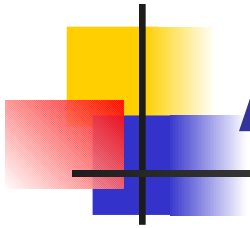
bVd

¬bV¬d

aV¬d

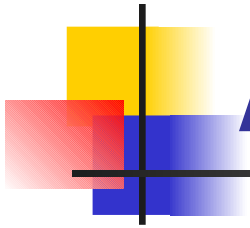
¬cV¬d





Algorithme DPLL

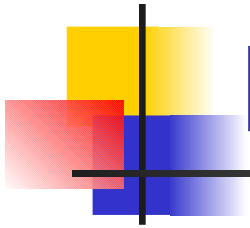
- Complexité temporelle exponentielle selon le nombre de variables
  - Au pire des cas, essayer les 2^n instanciations possibles
- Complexité spatiale linéaire
  - On stocke uniquement les clauses ainsi que l'instanciation courante



Algorithme DPLL

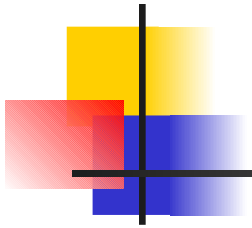
- Ordonnancement des variables
- Peut grandement affecter les performances de DPLL (selon les formules)

- Plusieurs heuristiques possibles
  - UP (unit propagation): choisir le littéral qui produit le plus de propagations unitaires
  - MOMS (maximal occurences in shortest clauses): le littéral qui apparaît le plus souvent, en favorisant les clauses les plus courtes
  - etc...



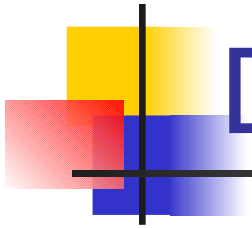
DPLL: backjumping

- Contradiction : provoquée par le dernier littéral de décision d, mais indépendante de c
- Eviter de retrouver plusieurs fois la même contradiction
- ¬d aurait dû être impliqué plus tôt dans la recherche
- Trouver à quel endroit grâce à résolution



DPLL: backjumping

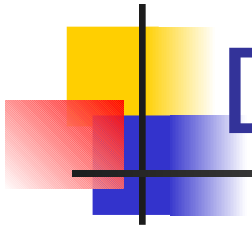
- Niveau de décision
  - Augmente lorsqu'une instanciation est décidée
  - Pas lorqu'elle est impliquée
- Raison d'une implication l: une clause devenue unitaire qui contient l
- Conflit: une clause unitaire qui contient ¬l et au moins un littéral du dernier niveau de décision
- => Résolution



DPLL: backjumping

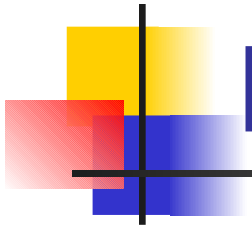
- Si la résolvante contient une variable impliquée de niveau supérieur, on la résout avec sa raison.
- Jusqu'à ce que la seule variable de niveau supérieur soit la variable de décision.

- Résultat:
  - Une clause conséquence de la formule d'origine
  - Elle justifie que la valeur de x aurait pû être impliquée par des assignations de niveaux inférieurs



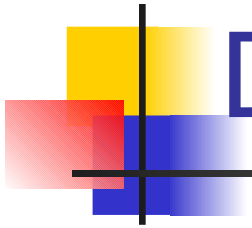
DPLL: backjumping

- Soit n le niveau de décision le plus élevé dans la clause mis à part celui de d
- On considère que ¬d est une implication de niveau n
- On ne défait pas les décisions de niveau inférieur à celui de d



DPLL: apprentissage

- Le backjumping détecte des implications dans la branche courante de la recherche
- Il est possible de retrouver le même cas dans une autre branche
- Solution: mémoriser la clause de conflit



DPLL: apprentissage

- Avantage: élaguer l'espace de recherche sans plus d'efforts par rapport au backjumping
- Inconvénient: espace mémoire!

- => Mémoriser seulement certaines clauses (n'affecte pas la correction de l'algorithme)
  - Pas plus grandes qu'une taille limite
  - Effacer celles qui n'ont pas servi depuis trop longtemps
- Paramétrable selon espace disponible

Décompositions arborescentes

- Outil d'algorithmique sur les graphes
- Représenter une FNC en graphe:
  - Une variable => un noeud
  - Une clause => une clique

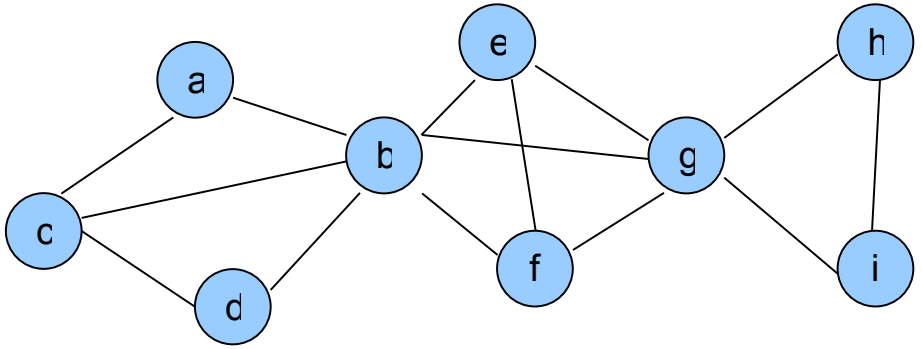
aV¬bVc

bvcV¬d

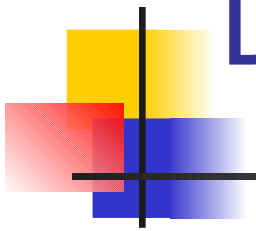
bveVfVg

¬eV¬f

¬gV¬hVi



Décompositions arborescentes



- Un arbre dans lequel
  - Chaque noeud contient un ensemble de variables
  - Chaque variable est dans au moins un noeud
  - Toutes les variables d'une clause sont dans un même noeud
  - Si deux noeuds contiennent une même variable, tous les autres noeuds sur leur chemin également (connectivité)
- Largeur d'aborescence: nombre max. de variables par noeud

Décompositions arborescentes

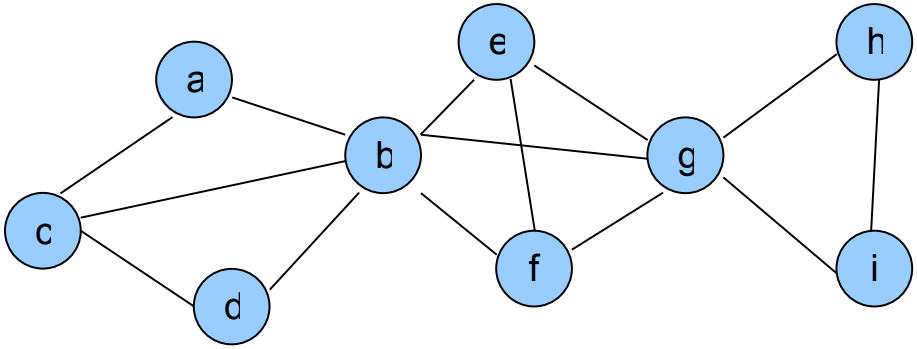
aV¬bVc

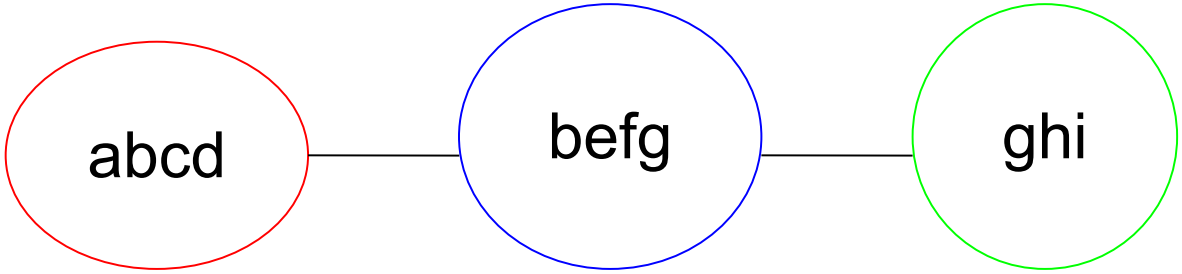
bvcV¬d

bveVfVg

¬eV¬f

¬gV¬hVi

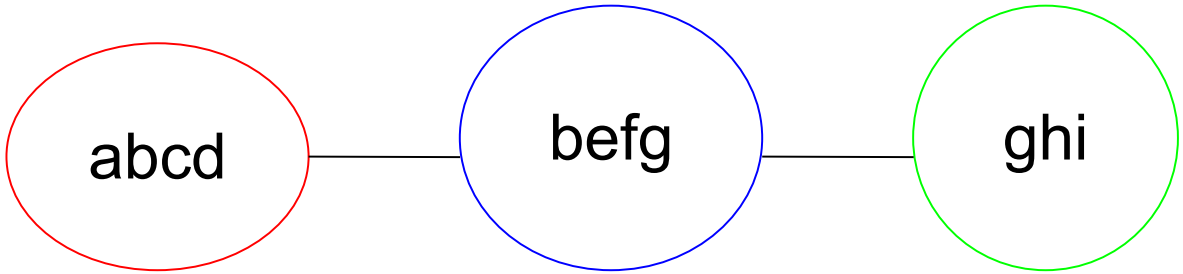




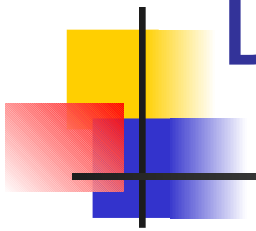
Décompositions arborescentes

- Utilisation directe (Dechter-Pearl, 88)
- Chaque noeud définit un sous-problème
- Trouver toutes les solutions de chaque sous-problème
- Essayer de reconstruire une solution globale à partir de sous-solutions compatibles

aV¬bVc
bvcV¬d
bveVfVg
¬eV¬f
¬gV¬hVi



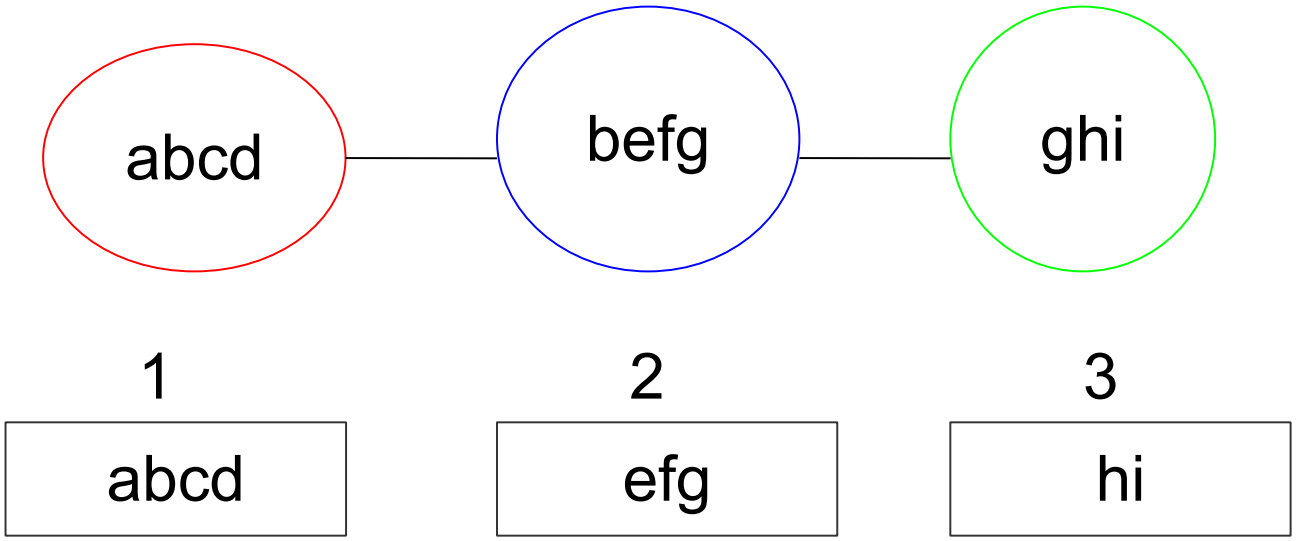
Décompositions arborescentes



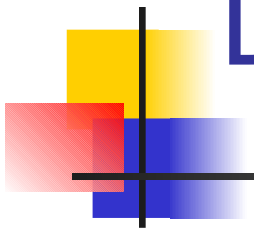
- Sous-problèmes: complexités temps et espace exponentielles selon largeur d'aborescence (stockage de toutes les solutions)
- Trouver une décomposition de plus petite largeur est NP-complet
  - Approximations
- Largeur minimale dépend de la formule

Décompositions arborescentes

- Utilisation indirecte:
  - Chercher une décomposition
  - Créer un ordonnancement partiel
  - L'utiliser dans une recherche DPLL



Décomposition



- Traiter consécutivement des variables « liées »
- Complexité temporelle: exponentiel selon largeur d'arborescence
- Complexité spatiale: toujours linéaire!
- Expérimentalement efficace sur des formules « structurées »
- Surcoût pour calculer la décomposition

Références

- DP: Davis, Martin; Putnam, Hillary (1960). "A Computing Procedure for Quantification Theory". Journal of the ACM 7 (3): 201–215.
- DPLL: Davis, Martin; Logemann, George, and Loveland, Donald (1962). "A Machine Program for Theorem Proving". Communications of the ACM 5 (7): 394–397.
- Décompositions arborescentes:
  - Dechter, Rina; Pearl, Judea (1989). Tree clustering for constraint networks. Artificial Intelligence, 38(3):353–366.
  - Huang, Jimbo; Darwiche, Adnan (2003). A structure-based variable ordering heuristic for SAT. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1167–1172.