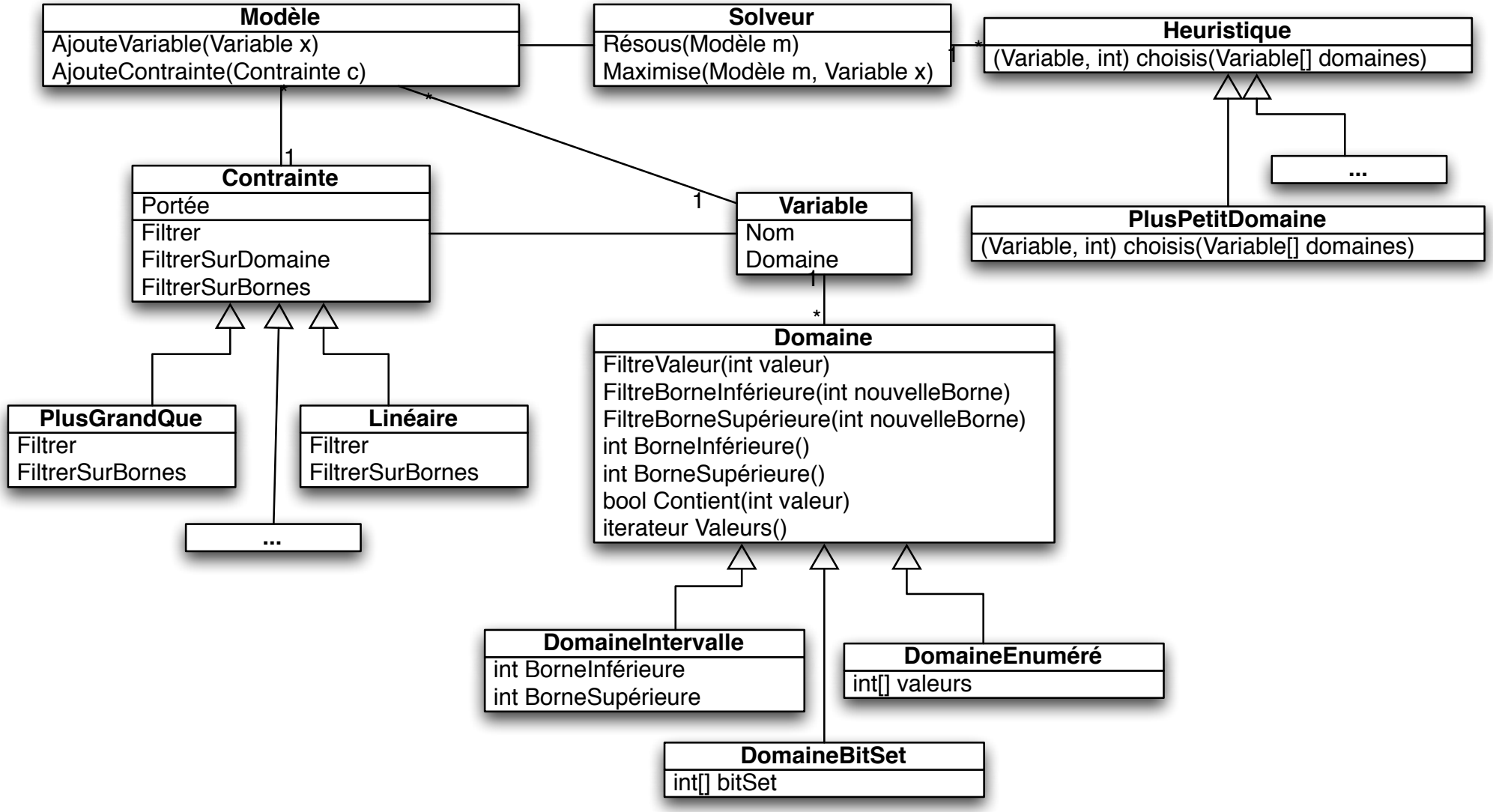
Chapitre 3
Autopsie d'un solveur

Claude-Guy Quimper

Introduction

- Il existe autant d'architectures de solveurs qu'il existe de solveurs.
- Cependant, certaines composantes sont communes à plusieurs solveurs.
- Nous présentons ces composantes et montrons comment elles interagissent entre elles.

Composantes d'un solveur de contraintes



Le modèle

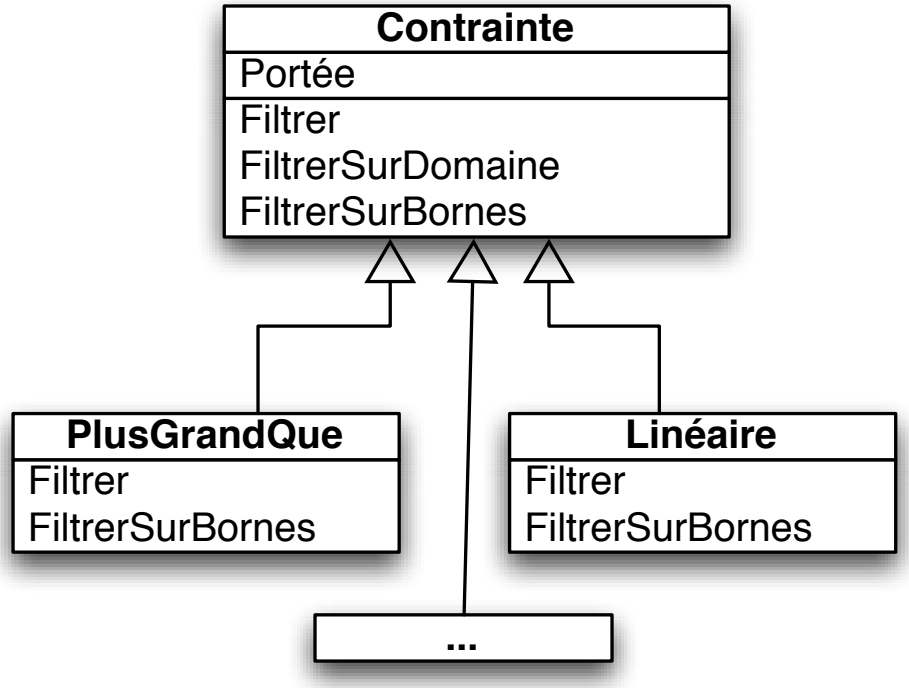
- Un modèle est l'ensemble des variables et des contraintes du problème. Cette composante n'est souvent qu'un conteneur.
- Toutes les variables et les contraintes doivent être ajoutées au modèle pour qu'elles soient considérées.
- Il peut y avoir un prétraitement sur le modèle afin de le rendre plus efficace. Par exemple, deux variables X et Y sujettes à la contrainte X = Y peuvent être remplacées par une seule et même variable.
- Le solveur procède généralement à une étape de filtrage initiale et remplace les variables n'ayant qu'une seule valeur dans leur domaine par une constante.

La variable

- La variable possède généralement un nom et toujours un domaine.
- Le nom sert à identifier la variable dans les interfaces graphiques et lors du débogage. Il n'est aucunement utilisé lors de la résolution du problème. Dans la plupart des systèmes, rien n'empêche deux variables d'avoir le même nom.
- Certains solveurs ne font pas la distinction entre la variable et le domaine.
- Certains solveurs traitent la contrainte d'égalité X = Y en faisant partager le même domaine aux variables X et Y.

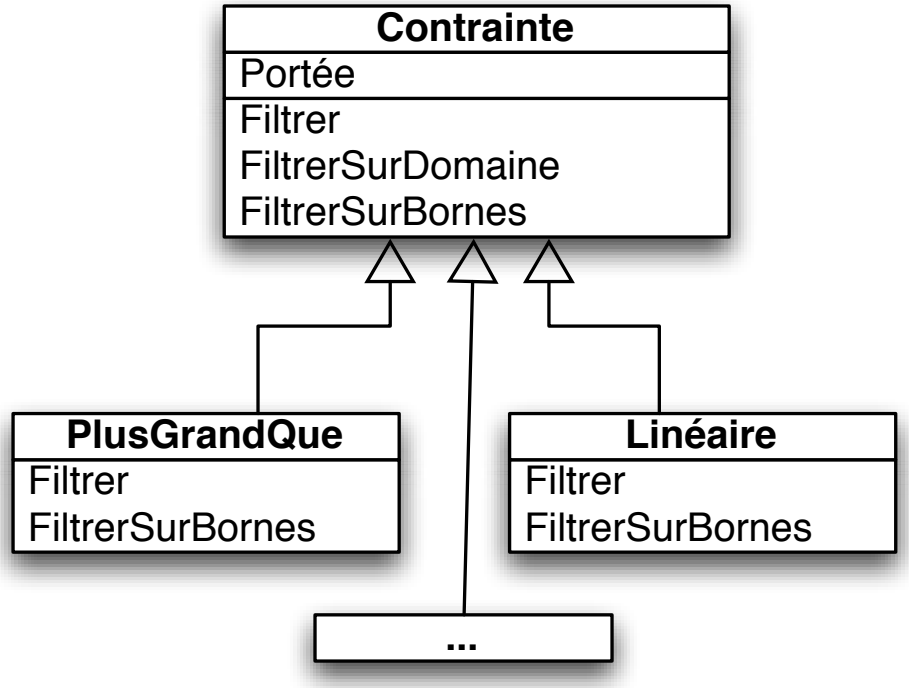
La contrainte

- Une contrainte a comme attribut la portée qui est un vecteur de variables.
- On a généralement trois méthodes de filtrage.



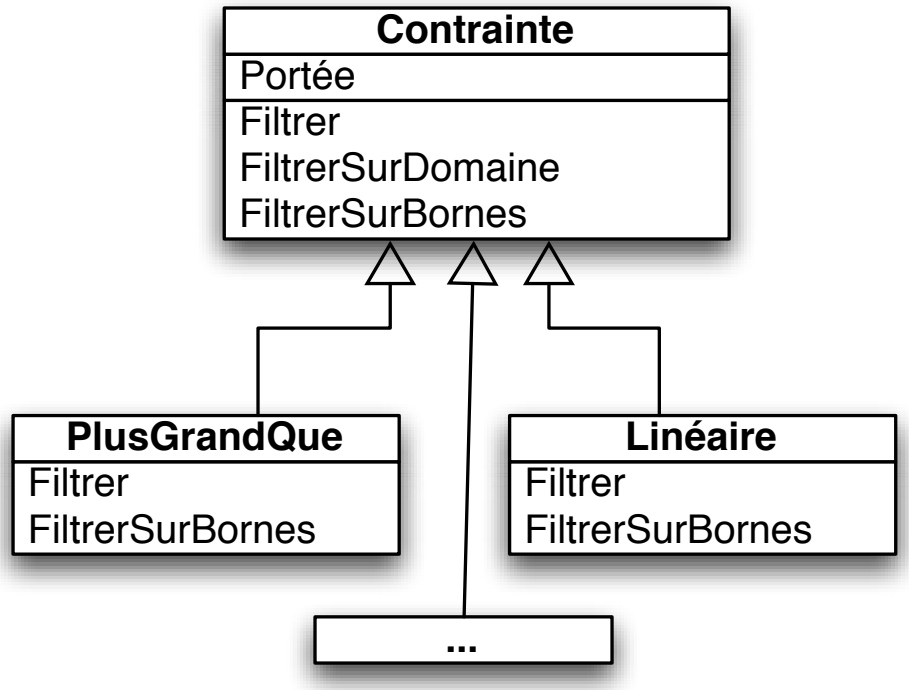
La contrainte

- **Filtrer**: Filtre la contrainte à partir de zéro.
- **FiltrerSurDomaine**: Filtre la contrainte lorsqu'une valeur a été retirée du domaine. La liste des domaines et des valeurs retirées est donnée en argument.
- **FiltrerSurBornes**: Filtre la contrainte lorsqu'une borne d'une variable de la portée a été modifiée. Les bornes et les variables modifiées sont données en argument.



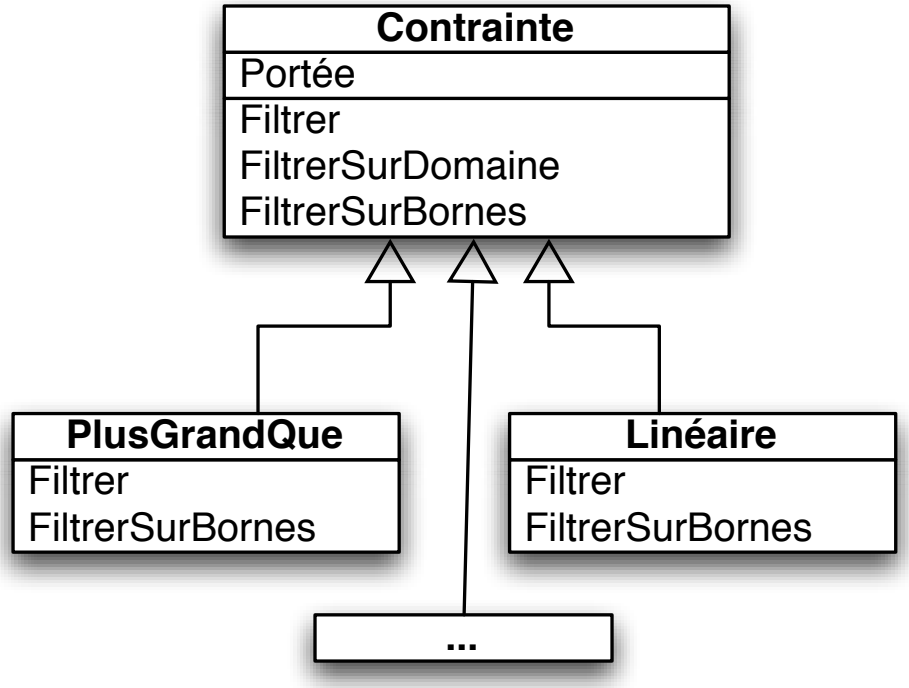
La contrainte

- La classe contrainte est une classe abstraite. C'est-à-dire que les fonctions Filtrer, FiltrerSurDomaine et FiltrersurBornes doivent être implantées par un descendant de la contrainte.
- Chaque contrainte va implanter ces trois fonctions.
- Par défaut, FiltrerSurDomaine et FiltrerSurBornes vont appeler Filtrer.



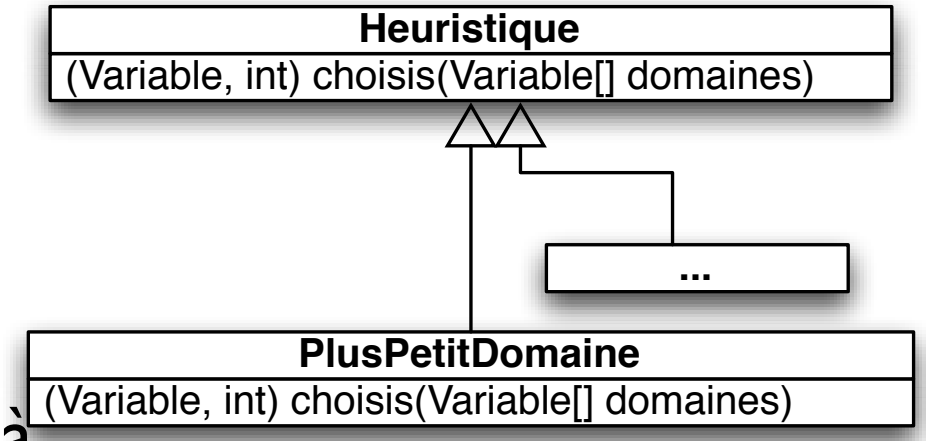
La contrainte

- Cette architecture permet à un utilisateur de créer ses propres contraintes.
- Il suffit de créer un descendant de la classe Contrainte et d'implanter les algorithmes de filtrage appropriés.



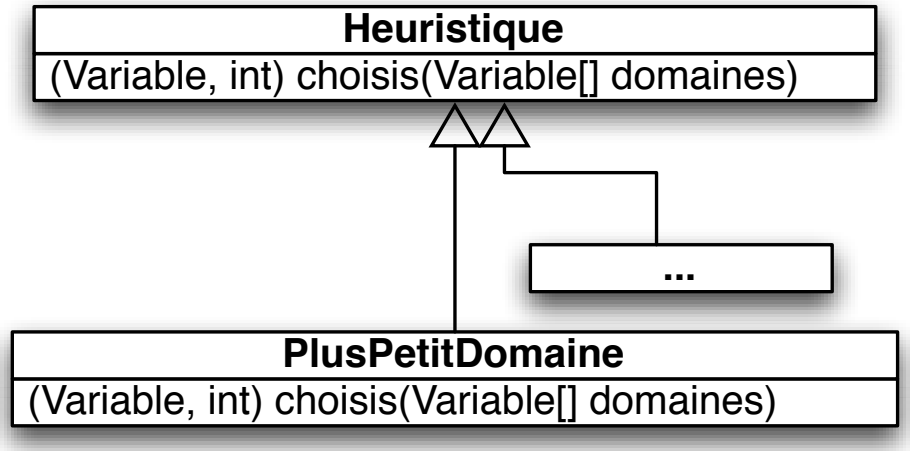
L'heuristique

- L'heuristique de recherche est simplement une fonction qui prend en paramètre l'ensemble des variables du problème et qui retourne un choix d'une variable et d'une valeur pour le branchement.
- Il existe plusieurs heuristiques déjà programmées dans le solveur et l'heuristique par défaut en est une qui fonctionne « bien » pour « la plupart » des problèmes.

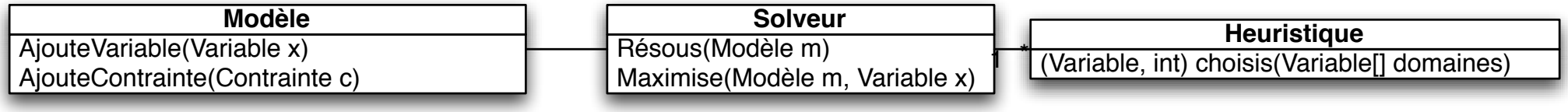


L'heuristique

- Pour des problèmes industriels, il est recommandé, voire nécessaire, de concevoir ses propres heuristiques de recherche.
- Il suffit alors de créer un descendant à la classe Heuristique et implanter sa méthode « choisis ».

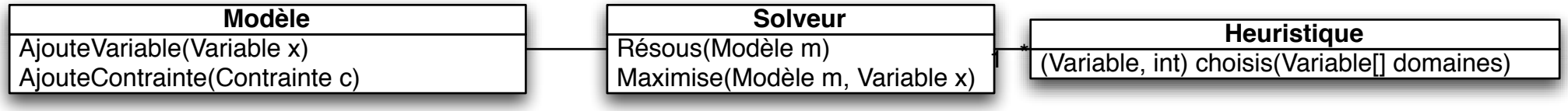


Le solveur



- Le solveur est le moteur du système. C'est lui qui crée l'arbre de recherche et l'explore.
- Les heuristiques définissent sa configuration.
- On lui passe en entrée un modèle et il retourne une solution.

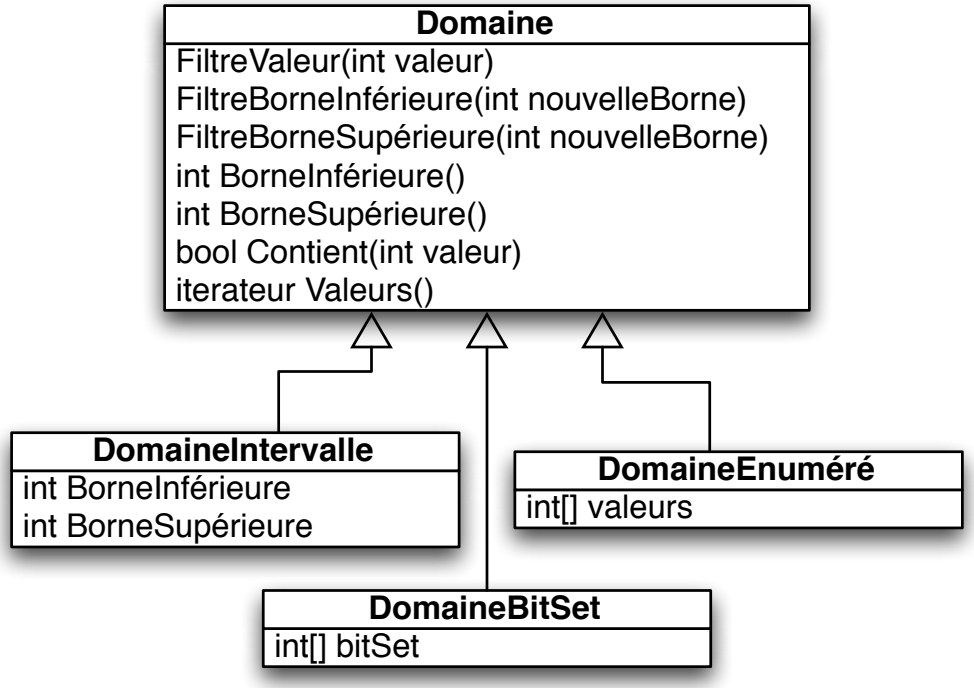
Le solveur



- Le solveur peut résoudre un problème de satisfaction par sa fonction « Résous ».
- Il peut résoudre un problème d'optimisation par sa fonction « Maximise ». Cette fonction retourne une solution où la valeur de la variable est maximale.
- **Question**: Que fait-on si on ne désire pas maximiser une variable, mais bien une somme de variables?
- **Réponse**: On déclare une nouvelle variable X. À l'aide d'une contrainte linéaire, on impose la variable X à être égale à la somme désirée. Puis on maximise X.

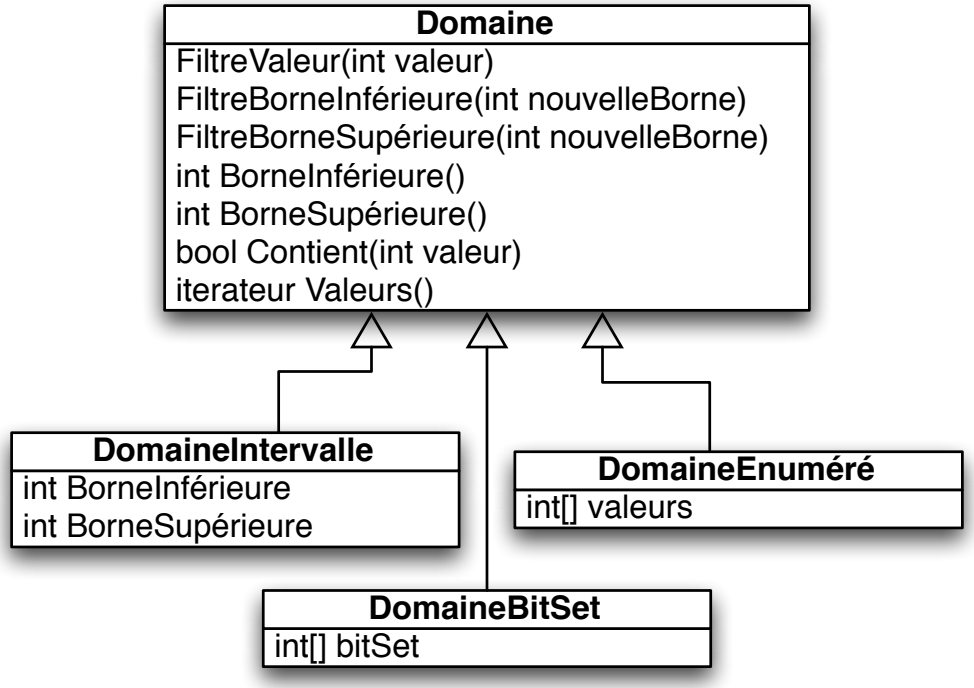
Le domaine

- Le domaine est un ensemble de valeurs.
- On doit donc être capable de tester si une valeur appartient au domaine.
- Il existe plusieurs fonctions répondant à ce besoin.



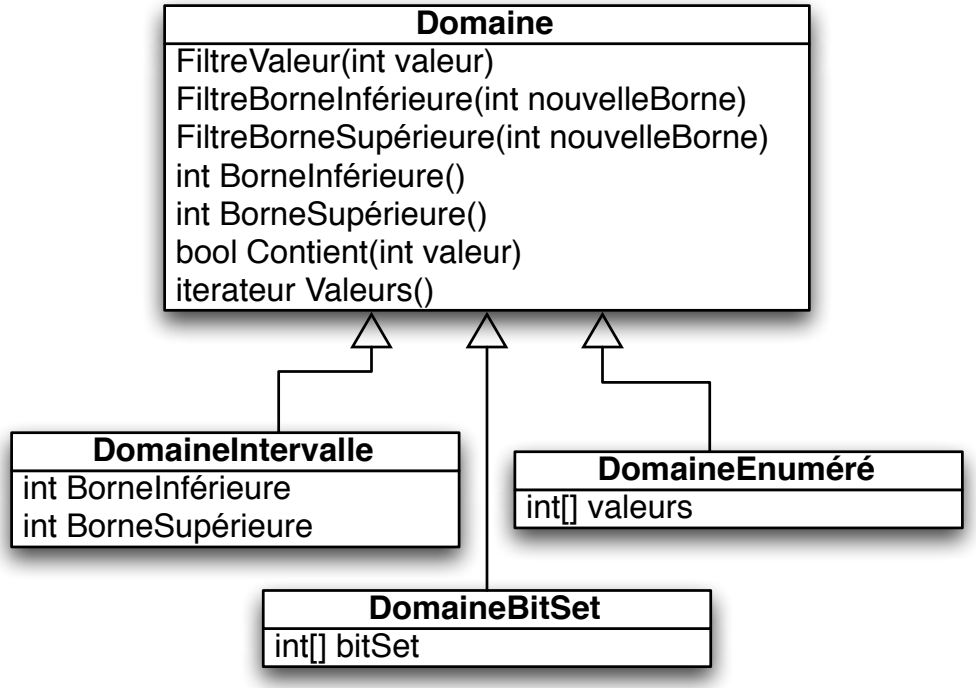
Le domaine

- **BorneInférieure**: Retourne la plus petite valeur du domaine.
- **BorneSupérieure**: Retourne la plus grande valeur du domaine.
- **Contient**: Retourne si oui ou non une valeur appartient au domaine.
- **Valeurs**: Retourne un itérateur permettant de lister les valeurs du domaine.



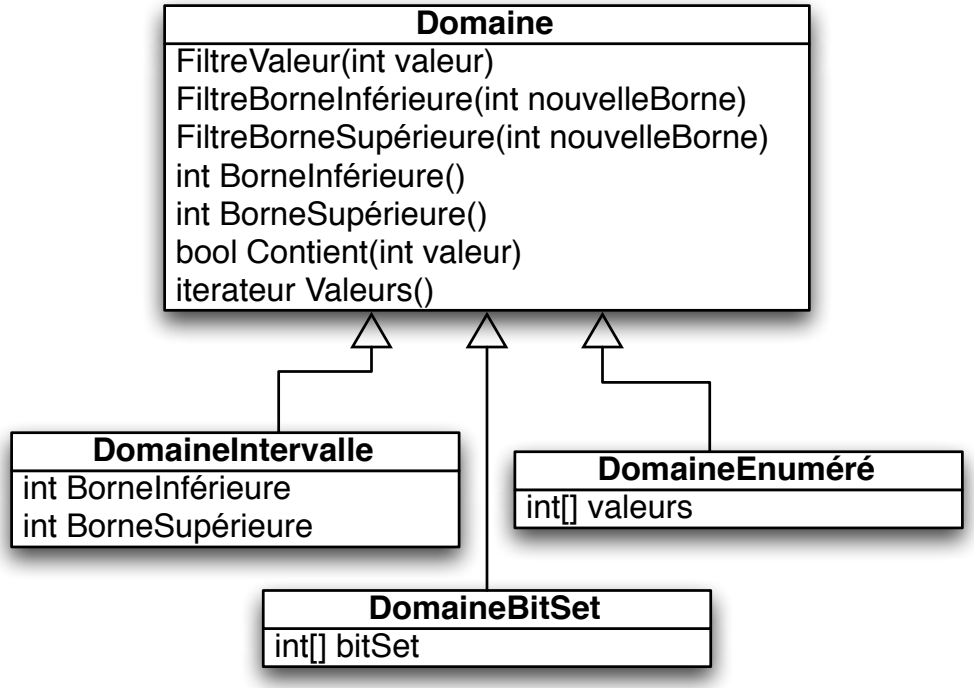
Le domaine

- Une contrainte doit être en mesure de retirer des valeurs des domaines lors du filtrage.
- **FiltreValeur**: retire une valeur spécifique du domaine.
- **FiltreBorneInférieure**: Augmente la borne inférieure d'un domaine.
- **FiltreBorneSupérieure**: Réduit la borne supérieure d'un domaine.
- Si un domaine devient vide, le solveur est notifié et un retour arrière est déclenché.



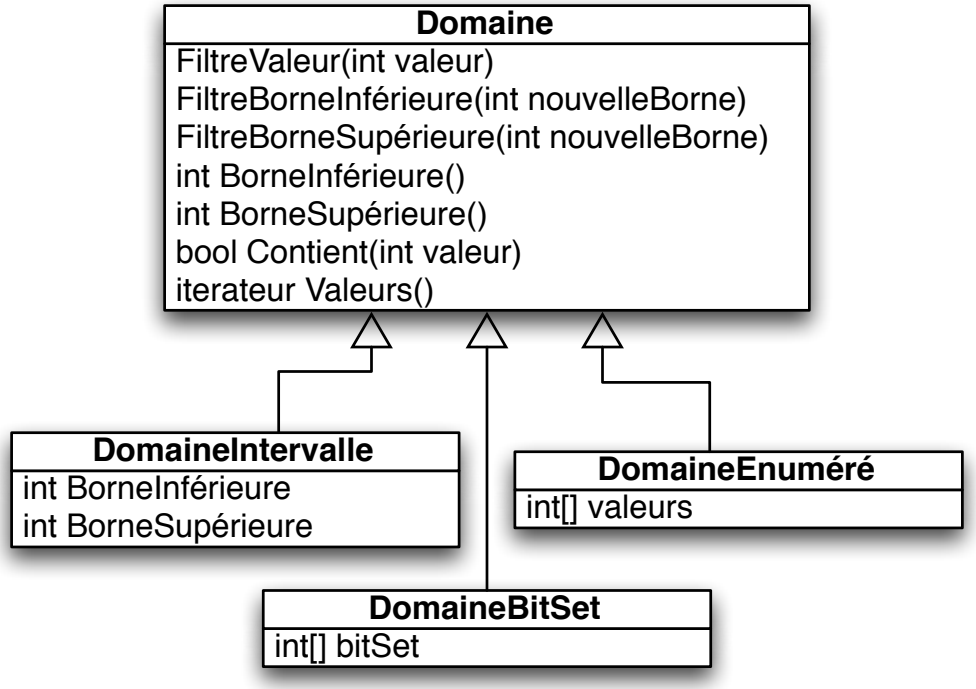
Le domaine

- Lors d'un retour arrière, le domaine doit être restauré à l'état où il était dans le noeud parent de l'arbre de recherche.
- La variable instanciée au niveau inférieur de l'arbre retrouve donc son domaine original au niveau supérieur.
- Une valeur qui a été filtrée lors d'une instanciation doit être réintroduite dans son domaine lors du retour arrière.



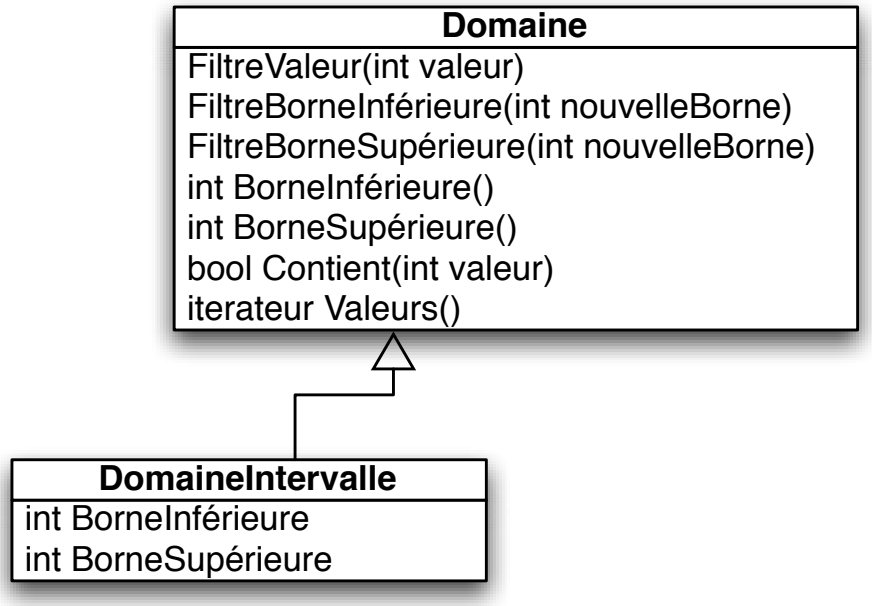
Le domaine

- Il existe plusieurs structures de données permettant d'encoder un ensemble de valeurs dont on peut retirer des valeurs et revenir à un état précédent.
- Chaque structure de donnée offre ses avantages et ses désavantages. Il faut donc les choisir judicieusement.
- Certains solveurs font le choix des structures de données automatiquement. La plupart laissent l'utilisateur faire ce choix.

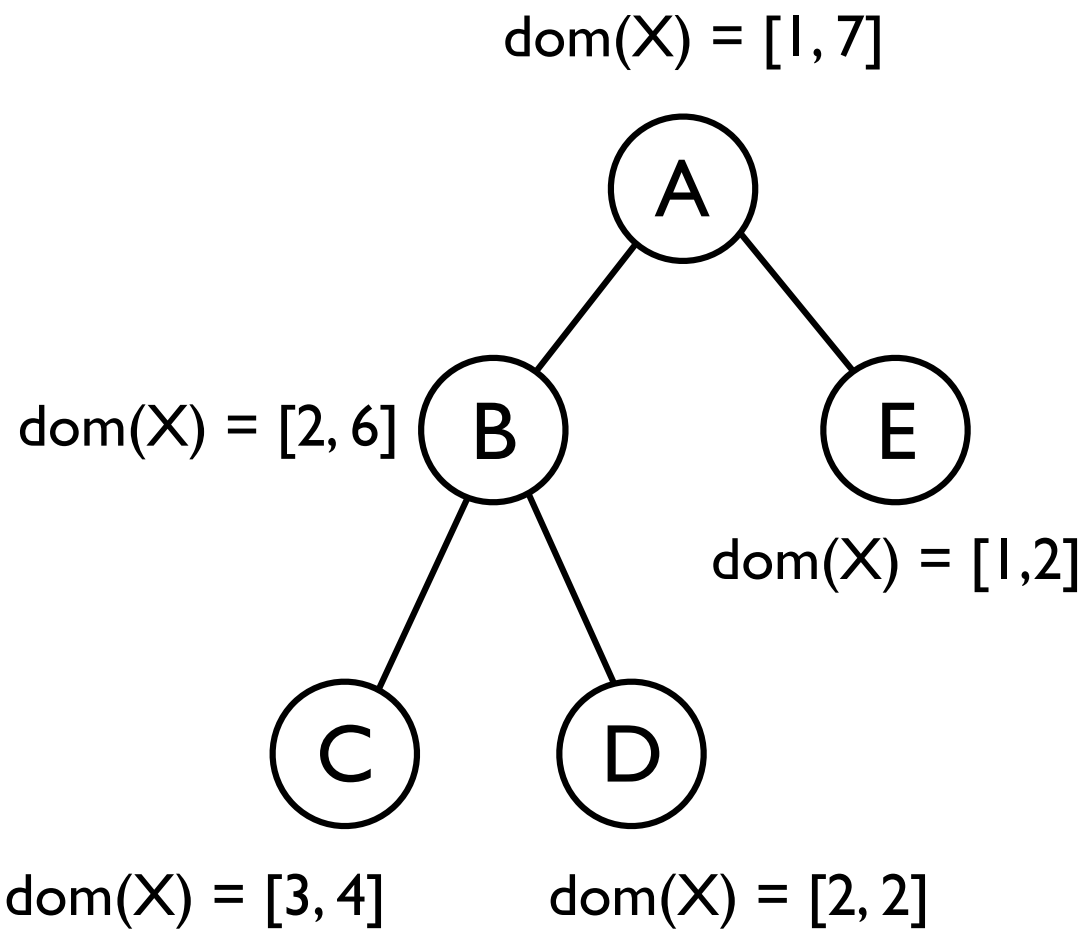


Le domaine intervalle

- Ce domaine ne garde que la borne inférieure et la borne supérieure du domaine.
- Le domaine {1, 3, 5} ne peut pas être représenté par cette structure.
- Seules les bornes peuvent être modifiées.
- Le domaine garde sur une pile toutes les valeurs de ses bornes pour chaque noeud de l'arbre de recherche entre la racine et le noeud courant.

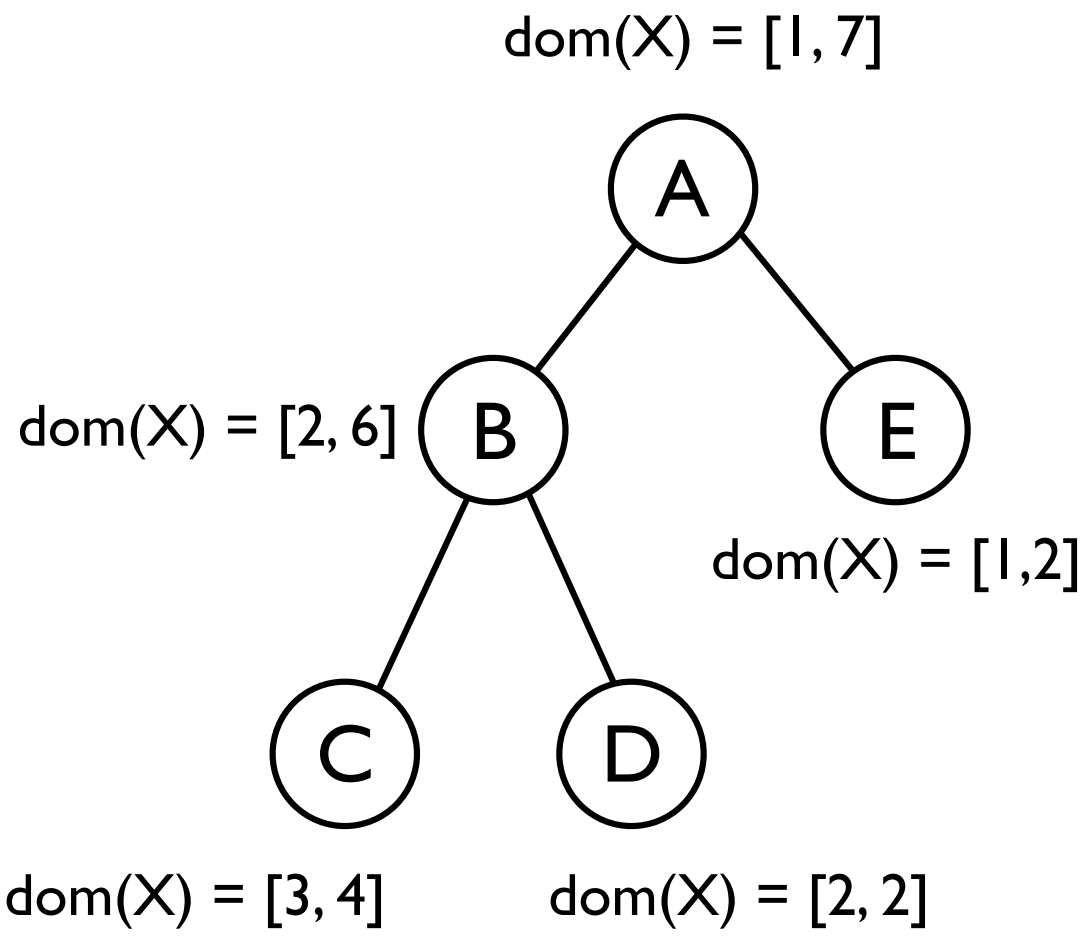


La pile d'un domaine



- Considérons une variable dont le domaine est filtré à chaque noeud de l'arbre de recherche.
- Chaque fois qu'on visite un enfant, le domaine ajoute sur sa pile le nouvel intervalle.
- Pour chaque retour arrière, on retrouve sur la pile le domaine du noeud parent.

La pile d'un domaine



Noeud	Pile
A	[1, 7]
B	[1, 7], [2, 6]
C	[1, 7], [2, 6], [3, 4]
B	[1, 7], [2, 6]
D	[1, 7], [2, 6],[2, 2]
B	[1, 7], [2, 6]
A	[1, 7]
E	[1, 7], [1, 2]

Le domaine intervalle

Opération	Algorithme	O
Tester l'appartenance d'une valeur	Tester les deux bornes	O(1)
Itérer sur une valeur	Trivial	O(1)
Retourner les bornes	Trivial	O(1)
Changer les bornes	Trivial	O(1)
Retirer une valeur	Non disponible	
Enregistrer les changements.	2 valeurs à sauvegarder	O(1)
Restaurer le domaine	2 valeurs à restaurer	O(1)

Quand utiliser le domaine intervalle?

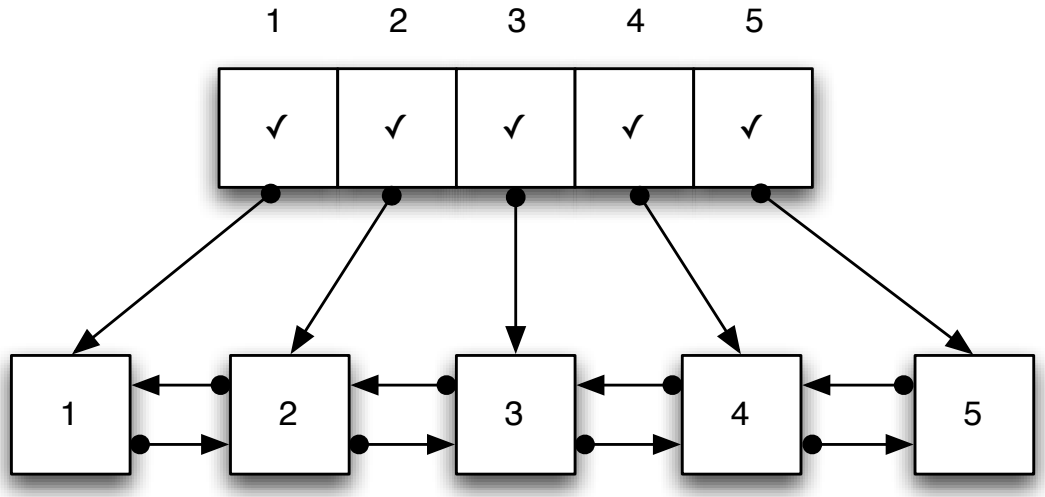
- On utilise le domaine intervalle lorsque les domaines sont très grands et qu'on ne peut pas se permettre d'énumérer toutes les valeurs.
- Certaines contraintes telles que les inégalités ($<$, $>$, $\leq$, $\geq$) peuvent seulement filtrer les bornes des domaines. Le domaine intervalle est donc une structure de données minimaliste et efficace pour encoder ces domaines.
- On les utilise aussi lorsque les algorithmes de filtrage font de la cohérence de bornes.

Les domaines énumérés

- La plupart du temps, les domaines doivent contenir des valeurs ne formant pas un intervalle.
- On utilise donc une structure de donnée qui permet d'énumérer les valeurs facilement et de tester efficacement si une valeur appartient au domaine.

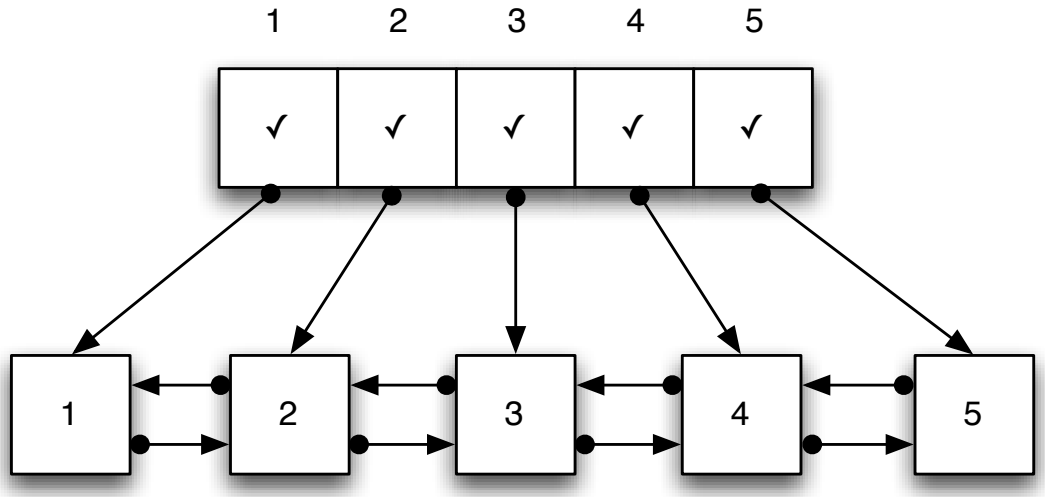
La structure de données

- La structure de données est une liste doublement chaînée des valeurs du domaine.
- Un tableau permet de rapidement trouver un noeud dans la liste chaînée.
- Une cellule du tableau contient un drapeau et un pointeur vers l'entrée dans la liste chaînée.



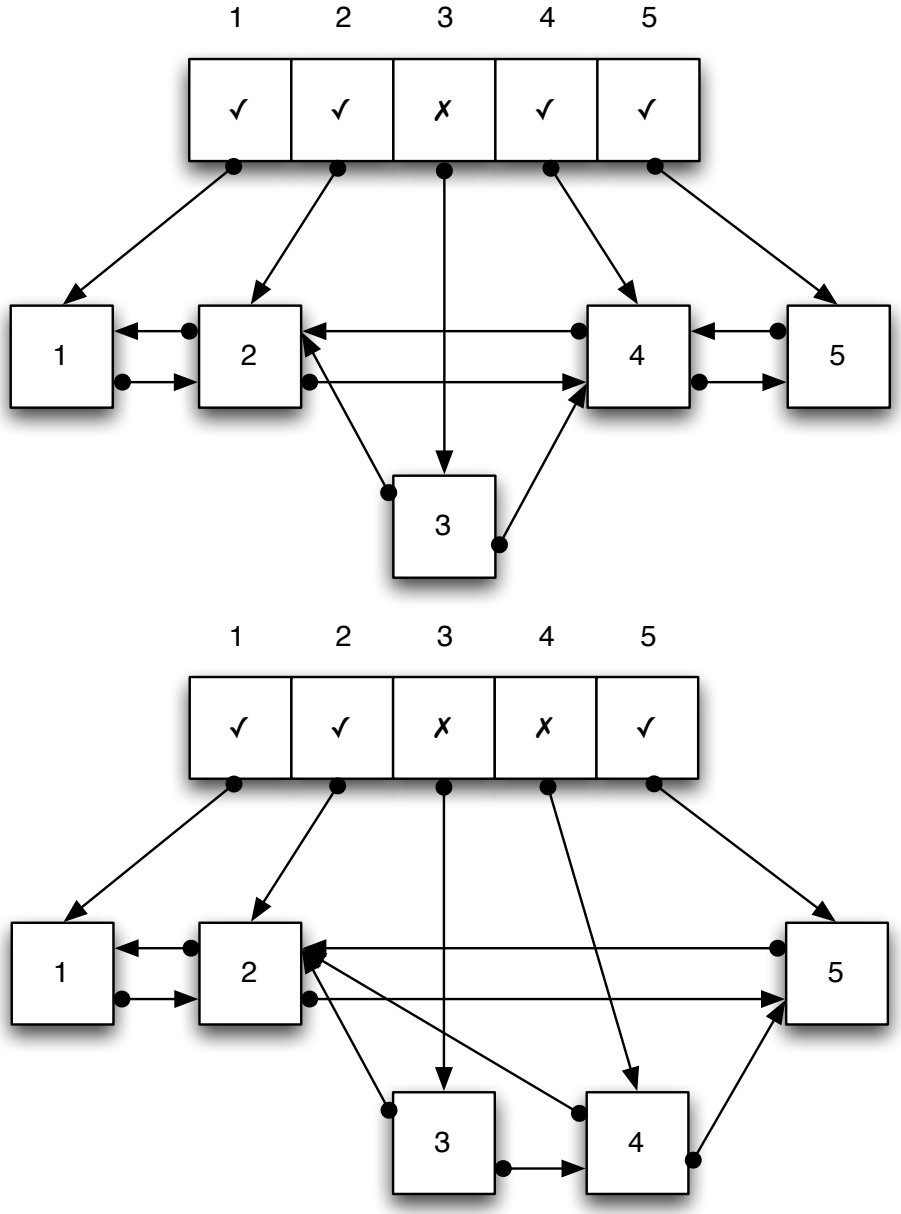
La structure de données

- On peut donc tester en temps constant l'appartenance d'une valeur au domaine en regardant le drapeau dans le tableau.
- On peut itérer les valeurs en parcourant la liste chaînée.



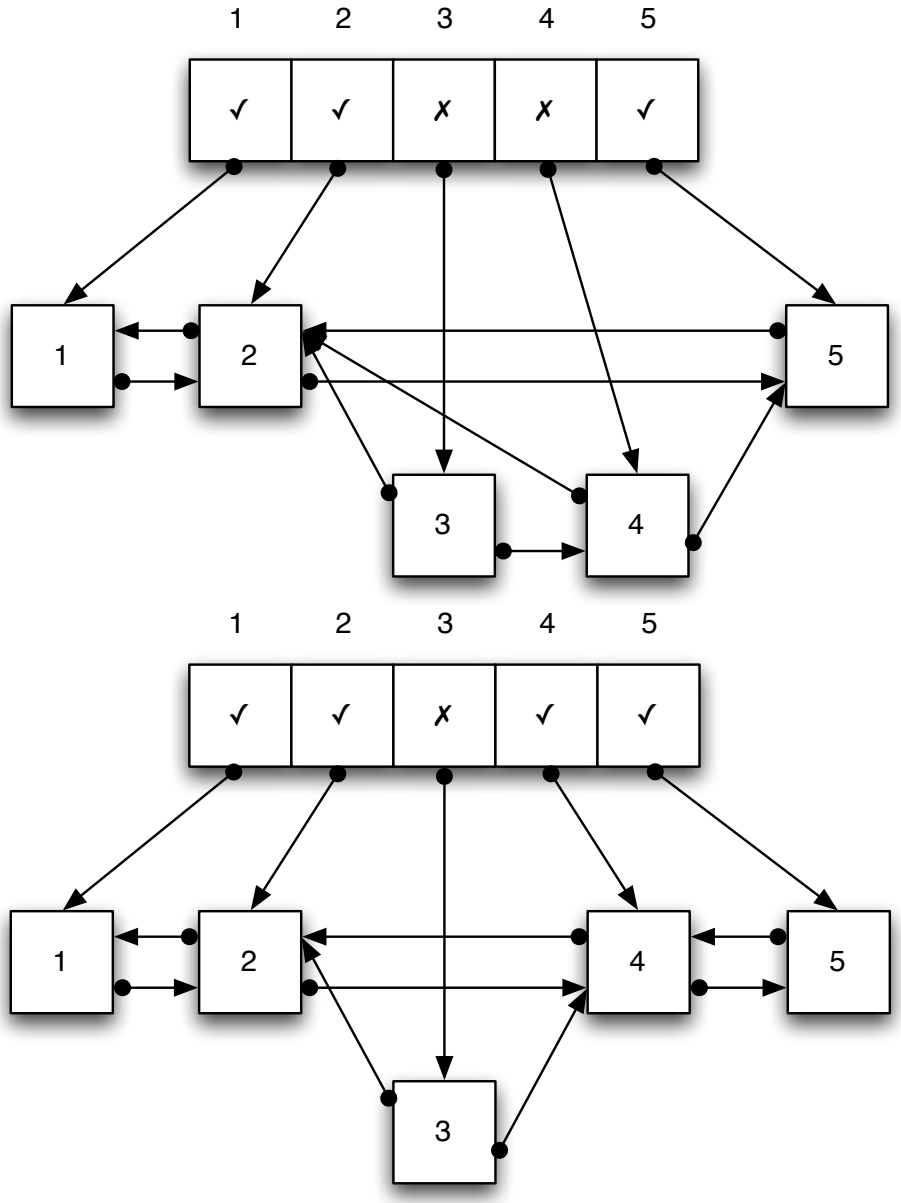
Le retrait d'une valeur

- Lors du retrait d'une valeur, on change le drapeau dans le tableau et on retire le noeud de la liste chaînée.
- Cependant, le noeud de la valeur retirée n'est pas supprimé et pointe toujours vers son successeur et son prédécesseur dans la liste chaînée.
- À droite, on présente la structure de donnée après les retraits consécutifs des valeurs 3 et 4.

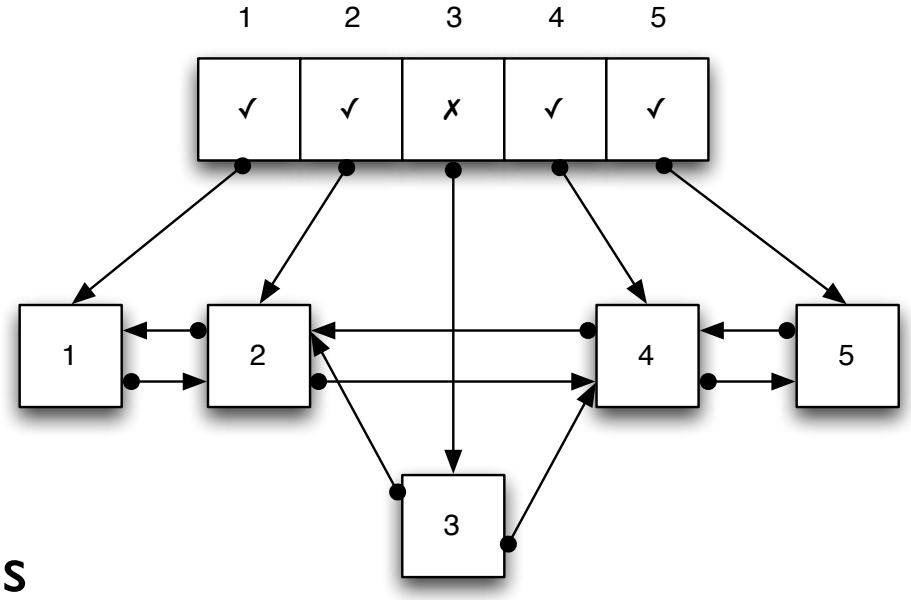


La réinsertion d'une valeur

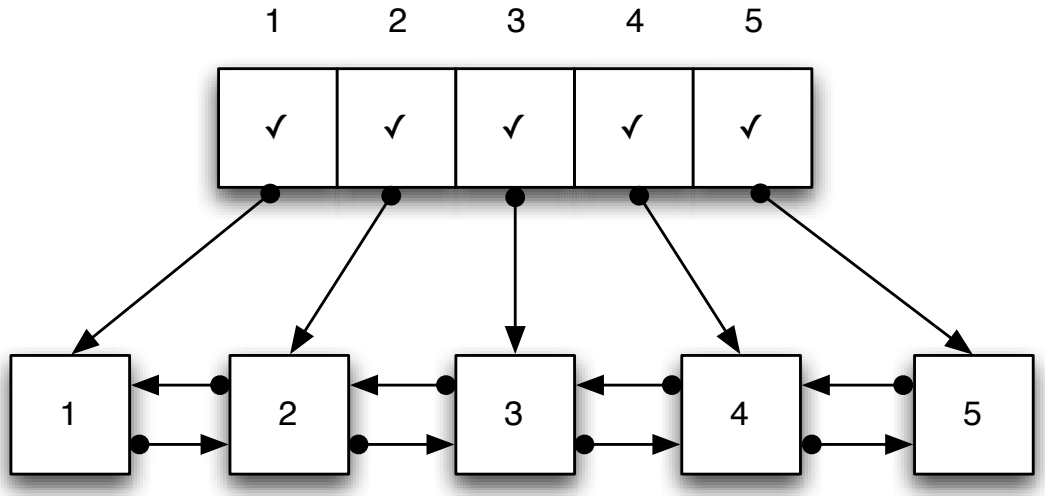
- On réinsère les valeurs dans l'ordre inverse qu'elles ont été retirées. C'est-à-dire que la première valeur à être réinsérée est la dernière à avoir été retirée.
- Un noeud est inséré entre son noeud suivant et son noeud précédent.
- Ici, la valeur 4 est réintroduite entre les valeurs 2 et 5.



La réinsertion d'une valeur



- La valeur 3 est réintroduite entre les valeurs 2 et 4.



Enregistrement des valeurs retirées

- Les valeurs retirées d'un domaine sont enregistrées dans chaque noeud de l'arbre de recherche. Ainsi, lors d'un retour arrière, il est possible de réinsérer les valeurs dans les domaines.

Le domaine énuméré

Opération	Algorithme	O
Tester une valeur	Vérifier le drapeau dans le tableau	O(1)
Itérer sur une valeur	Parcourir la liste chaînée	O(1)
Retourner les bornes	Retourner le premier et le dernier élément de la liste.	O(1)
Changer les bornes	Parcourir la liste chaînée et supprimer d valeurs.	O(d)
Retirer une valeur	Modifier 2 pointeurs	O(1)
Enregistrer k changements.	Écrire k valeurs dans le noeud de l'arbre de recherche.	O(k)
Restaurer k valeurs	Modifier 2k pointeurs	O(k)

Domaine énuméré avec retours arrière rapides

- Il existe une autre façon de représenter un domaine énuméré.
- Pour un domaine qui contient initialement les valeurs de 1 à d, on crée deux vecteurs A[1..d] et B[1..d] ainsi qu'une variable entière *cardinalité* initialisée à d qui représente la cardinalité du domaine.
- Initialement, nous avons A[i] = B[i] = i.

Structure de données

A	1	2	3	4	5	6
B	1	2	3	4	5	6

cardinalité = 6

- Les *cardinalité* premières entrées du vecteur B sont les valeurs du domaine.
- La valeur A[i] indique la position de la valeur i dans B.
- Nous avons donc comme invariant: B[A[i]] = i.

Test d'appartenance

A	1	2	3	4	5	6
B	1	2	3	4	5	6

cardinalité = 6

- Pour vérifier si une valeur *v* appartient au domaine, on teste si A[v] ⩽ *cardinalité*.

Retrait d'une valeur

- Pour retirer la valeur v, on interchange la valeur v dans B avec la valeur B[cardinalité].
- On met à jour le vecteur A afin de maintenir l'invariant.
- On décrémente la variable *cardinalité* de un.

Exemple

A	1	2	3	4	5	6
B	1	2	3	4	5	6

Avant le retrait de 3

cardinalité = 6

A	1	2	6	4	5	3
B	1	2	6	4	5	3

Après le retrait de 3

cardinalité = 5

Exemple

A	1	2	6	4	5	3
B	1	2	6	4	5	3

Avant le retrait de 4

cardinalité = 5

A	1	2	6	5	4	3
B	1	2	6	5	4	3

Après le retrait de 4

cardinalité = 4

Exemple

A	1	2	6	5	4	3
B	1	2	6	5	4	3

Avant le retrait de 2

cardinalité = 4

A	1	4	6	5	2	3
B	1	5	6	2	4	3

Après le retrait de 2

cardinalité = 3

Réinsertion d'une valeur

- Pour réinsérer les *k* dernières valeurs à avoir été retirées, on ajoute *k* à la variable *cardinalité*.
- Cette opération se fait en temps constant.
- Il n'y a aucun besoin de savoir quelles valeurs ont été retirées. Il suffit de savoir leur nombre.
- On note, dans chaque noeud de l'arbre de recherche, le nombre de valeurs retirées pour chaque domaine.
- C'est dans cette opération que se trouve l'efficacité de cette structure de données.

Désavantages

- Cette structure de données comporte quelques désavantages.
- Il est inefficace de déterminer le plus petit élément et le plus grand élément du domaine. Il faut parcourir le vecteur A et tester l'appartenance de chaque élément.
- On peut itérer sur les éléments du domaine en parcourant le vecteur B, mais les valeurs ne sont pas en ordre croissant. Certains algorithmes de filtrage perdent de leur efficacité si les valeurs ne sont pas ordonnées.

Le domaine énuméré avec retours arrière rapides

Opération	Algorithme	O
Tester une valeur	Vérifier si A[v] ⩽ cardinalité	O(1)
Itérer sur une valeur	Parcourir le vecteur B	O(1)
Retourner les bornes	Parcourir le vecteur A	O(d)
Changer les bornes	Parcourir le vecteur A et retirer chaque valeur appartenant au domaine.	O(*d*)
Retirer une valeur	Modifier 4 entrées et décrémenter la variable *cardinalité*.	O(1)
Enregistrer k changements.	On enregistre le nombre de valeurs filtrées dans chaque noeud de l'arbre de recherche.	O(1)
Restaurer k valeurs	Ajouter k à la variable *cardinalité*.	O(1)

Comparaison

- Il est beaucoup plus lent de changer les bornes du domaine avec un domaine énuméré qu'avec un domaine intervalle.
- Les solveurs améliorent ces structures de donnée à leur façon pour les rendre plus efficaces.
- Lors de la création du modèle et de ses variables, il faut se poser la question si on désire une mise à jour rapide des bornes quitte à ne pas permettre le retrait de valeur à l'intérieur d'un intervalle.

Comparaison

- Dans le cas où nous désirons avoir des domaines énumérés, il faut savoir si les algorithmes de filtrage ont besoin de parcourir les valeurs du domaine en ordre croissant.
- Si ce n'est pas une exigence, alors on peut utiliser une structure de données avec retours arrière rapides.
- Généralement, les solveurs laissent le choix à l'utilisateur entre des domaines intervalles et des domaines énumérées mais n'implante qu'un seul type de domaine énuméré.

Conclusion

- Nous avons vu les différentes composantes d'un solveur de contraintes.
- Nous avons vu trois types de domaines: les domaines intervalle, les domaines énumérés et les domaines énumérés avec retours arrière rapides.