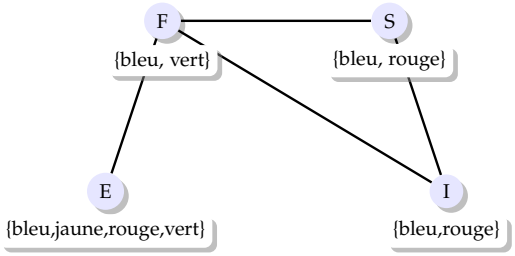
Raisonnement par Contraintes

On considère le problème de coloration de graphe suivant :



On va résoudre "à la main" ce problème avec trois algorithmes. Le premier est l'algorithme de base. Les deux autres font un travail de propagation des contraintes plus important. Le but est de comparer ces trois algorithmes.

- Le premier `Bactrack` effectue une recherche en profondeur d'abord en cherchant à affecter à chaque noeud de l'arbre une couleur. A chaque fois qu'on essaie une couleur, on va vérifier si elle satisfait bien les contraintes.
- Le second `Forwardchecking` marche de manière similaire sauf qu'une fois qu'on a affecté une variable avec une couleur, on va mettre à jour les contraintes des variables voisines. Ce petit travail devrait nous permettre de detecter plus tôt qu'on ne peut pas affecter une variable ou bien qu'on ne pourra pas utiliser une couleur.
- Le troisième algorithme, `BacktrackAC3` va plus loin que `Backtrack` : au lieu de propager les contraintes aux voisins seulement, on va maintenant rendre le CSP arc-cohérent. On va donc, après chaque affectation, rendre le CSP arc-cohérent.

1. Utiliser l'algorithme `Backtrack` ci-dessous pour résoudre le problème. On examinera les variables dans l'ordre F, E, S, I ; et on examinera les variables dans l'ordre lexicographique.

```
1  BackTrack (CSP net, Affectation a)
2    si l'affectation a est complète alors retourne a
3    var ← variable suivante non affectée
4    Pour toutes valeurs var ∈ D(var)
5      si {var=val} ne viole aucune contrainte
6          a = a ∪ {var=val}
7          result ← Backtrack(net, a)
8          si result ≠ échec
9              retourne result
10         sinon retourne échec
```

2. Une fois qu'on a mis à jour l'affectation, on va maintenant utiliser la propagation de contraintes pour limiter le nombre de valeurs possibles.

```
1  ForwardChecking (CSP net, Affectation a)
2    si l'affectation a est complète alors retourne a
3    var ← variable suivante non affectée
4    Pour toutes valeurs var ∈ D(var)
5      si {var=val} ne viole aucune contrainte
6          a = a ∪ {var=val}
7          pour toutes variables v connectée à var
8              verifier arc-cohérence de var et v
9          result ← Backtrack(net, a)
10         si result ≠ échec
11             retourne result
12         sinon retourne échec
```

Utiliser `ForwardChecking` pour trouver une solution en conservant les mêmes ordres que dans la question 1 pour examiner les variables et les valeurs.

3. utiliser des heuristiques pour le choix des valeurs et des variables
   — choisir la variable qui a le moins de valeurs disponibles
   — choisir la variable qui a le plus de contraintes avec des variables non affectées.
   — choisir la valeur qui contraint le moins les voisins

4. `ForwardCheckAC3`

```
Returns false if an inconsistency is found, and true otherwise
1  AC3(CSP csp)
2      while queue is not empty do
3          (Xi,Xj) → queue.pop();
2          if Revise(csp, Xi, Xj)
2              if (Di.size() == 0)
2                  return false;
2              for each Xk in Xi.neighbors \{Xj}
2                  queue.add((Xk,Xi));
2    return true;
```

```
                    returns true iff we revise the domain of Xi
1  Revis(CSP csp, Variable Xi, Variable Xj)
2    boolean revised leftarrow false
2     for each x in Di do
3         if (no value y in Dj allows (x,y) to satisfy the constraints between Xi and Xj )
2             delete x from Di
2             revised ← true
2    return revised
```

Exercice Implementation

Le but de cet exercice est d'écrire un solveur pour résoudre des problèmes de programmation par contraintes. On va traiter des problèmes avec un seul type de contraintes binaires : deux variables ne peuvent pas avoir la même valeur. On va donc pouvoir résoudre des problèmes tels que le sudoku ou les problèmes de coloration.
Pour résoudre un CSP, on doit combiner deux idées : utiliser une méthode de recherche et raisonner sur les contraintes. L'algorithme `ForwardChecking` effectue une sorte de recherche en

profondeur d'abord – à chaque étape, on tente d'affecter une variable – et on raisonne sur les contraintes impliquant seulement la variable que l'on vient d'affecter.
Avec `ForwardChecking`, on vérifie donc l'arc cohérence [1] mais on limite la vérification à une petite partie des variables. On peut aussi vérifier l'arc cohérence pour toutes les variables : cela va coûter plus cher, mais peut restreindre plus vite le domaine des valeurs possibles pour les autres variables. On a présenté un algorithme simple appelé `ac3` pour rendre un CSP arc cohérent (ou découvrir que le problème n'a pas de solutions).

Le premier but du projet est d'implémenter `ForwardChecking` avec l'utilisation des trois heuristiques vues en cours pour choisir la variable suivante et l'ordre des valeurs. On cherchera à estimer le gain de ces heuristiques sur un jeu de données : ici, on vous fournit 50 jeux de sudoku.
Le second but est d'implémenter une variante de `ForwardChecking` qui, au lieu de raisonner simplement sur les contraintes impliquant la variable qui vient d'être affectée utilise `ac3` pour rendre le CSP arc-cohérent. On utilisera le même jeu de données pour savoir si cela a un impact sur les performances.
On fournit un code java pour modéliser une variable (classe `Variable<T>`, une contrainte binaire (classe `BinaryConstraint<T>`) et un problème de satisfaction par contraintes (CSP, classe `BinaryCSP<T>`). Le paramètre `T` représente le type des variables. On fournit aussi des classes pour les applications sur le sudoku (on aura besoin des classes `Sudoku` et `Digit`) et pour les problèmes de coloration (classe `GraphColouring`).

Travail à effectuer

Complétez la classe `BinaryCSP<T>` avec les méthodes `forwardCheck()` et `forwardCheckAC3()`.
A priori, vous n'avez pas besoin de modifier les autres classes (à part les méthodes `main`).
Le fichier `sudokus.txt` contient 50 jeux, la classe `Sudoku` contient des méthodes pour lire ce fichier, créer une instance, et lancer l'exécution des méthodes `ForwardChecking` et `ForwardCheckingAC3`.
On fournit aussi deux exemples de problème de coloration : `australia.txt` correspond à l'exemple vu en cours, et `gc.txt` contient un problème beaucoup plus difficile (instance connue sous le nom de `myciel7)`. La classe `GraphColouring` contient des méthodes pour lire ces fichiers et appeler le solveur.

Evaluez le changement de performance dû à l'utilisation des heuristiques pour choisir les variables et les valeurs.
Evaluez les différences de performance entre le `ForwardChecking` classique et sa variante qui utilise `ac3`. Pouvait-on s'attendre à ces résultats ?

1. une variable est arc coherente si toutes les valeurs associees a la variable satisfont toutes les contraintes binaires ; le CSP est arc coherent si toutes les variables sont arc-cohérentes.