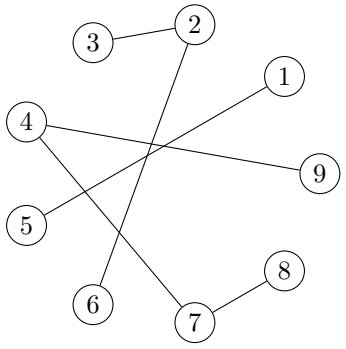
Exercice 1

1. – réflexive, avec $k = 0$, $u = w_0 = w_1 = v$;
   – symétrique, en inversant le $(k+2)$-uplet ;
   – transitive en concaténant les uplets.
2. On s'aide de la question 3 : on obtient le graphe :



   On obtient donc les classes d'équivalence : $\{\{1,5\},\{2,3,6\},\{4,7,8,9\}\}$.
3. Les classes d'équivalence correspondent assez naturellement aux composantes connexes de ce graphe.
4. On écrit d'abord une fonction récursive de parcours en profondeur.

```
Fonction DFS(s, X, comp)
    Si s ∉ X Alors
        X ← X ∪ {s}
        comp ← comp ∪ {s}
        Pour tous les sommets t voisins de s Faire
            DFS(t, X, comp)
```

   Dès lors, on obtient la fonction de calcul de composantes connexes :

```
Fonction Composantes(G)
    X ← ∅
    CC ← ∅
    Pour chaque sommet s Faire
        Si s ∉ X Alors
            comp ← ∅
            DFS(s, X, comp)
            CC ← CC ∪ {comp}
    Renvoyer CC
```

5. Le graphe G_f correspond à un ensemble de composantes connexes de tailles 2.
6. Les classes d'équivalences sont soit de taille 2, soit des cycles de taille paire. En effet, dans G, les chemins dans une composante de taille > 2 seraient formés d'une alternances d'arêtes de G_φ et de G_ψ, et n'auraient que des sommets de degré 2 (d'où le cycle).

Exercice 2

1. On a plus ou moins la réponse dans le code compagnon : F_1 et F_3 sont des formules de Horn, mais F_2 n'en n'est pas une (il y a une clause qui contient x_1 et x_0 qui sont deux littéraux positifs).
2. On commence par une fonction qui teste si une clause est vide :

```
let clause_vide (x, y) = (x = None) && (y = [])
```

et on l'utilise sur chaque clause de la formule :

```
let avoir_clause_vide f = List.exists clause_vide f
```

3. – Pour F_1 : x_2 est une clause unitaire. En propageant x_2, on obtient :

$$F_1' = (\neg x_0 \vee \neg x_1 \vee \neg x_3) \wedge (x_0 \vee \neg x_1) \wedge (x_0 \vee \neg x_3) \wedge \neg x_3$$

Il n'y a plus de clause unitaire ni de clause vide, donc la formule est satisfiable.

– Pour F_3 : x_1 est une clause unitaire. En propageant x_1, on obtient :

$$F_3' = \neg x_4 \wedge (\neg x_0 \vee \neg x_3 \vee \neg x_4) \wedge x_0 \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_4 \vee \neg x_0)$$

x_0 devient une clause unitaire. En propageant, on obtient :

$$F_3'' = \neg x_4 \wedge (\neg x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \wedge x_4$$

x_4 devient une clause unitaire. En propageant, on obtient :

$$F_3''' = () \wedge \neg x_3 \wedge (x_2 \vee \neg x_3)$$

Cette formule contient une clause vide, donc elle n'est pas satisfiable.

4. On parcourt la formule jusqu'à trouver une clause unitaire.

```
let rec trouver_clause_unitaire = function
    | [] -> None
    | (Some i, []) :: _ -> Some i
    | _ :: f -> trouver_clause_unitaire f
```

5. On remarque que la propagation ne peut pas faire apparaître de littéral positif dans une clause, et que la formule reste en FNC. C'est donc toujours une formule de Horn.

Pour propager, on supprime les clauses contenant x_i, et on supprime i des listes des littéraux négatifs sinon.

```
let rec propager f i = match f with
    | [] -> []
    | (Some j, _) :: g when i = j -> propager g i
    | (pos, neg) :: g -> (pos, List.filter ((<>) i) neg) :: propager g i
```

6. On utilise toutes les questions précédentes :

```
let rec etre_satisfiable f = match trouver_clause_unitaire f with
    | None -> not (avoir_clause_vide f)
    | Some i -> etre_satisfiable (propager f i)
```

On pense à tester sur F_1 et F_3.

7. On détaille la complexité des fonctions, pour une formule à m clauses et n variables :
   - `avoir_clause_vide` est en $\mathcal{O}(m)$;
   - `trouver_clause_unitaire` également ;
   - `propager` est en $\mathcal{O}(m \times n)$ (il faut parcourir les littéraux négatifs de chaque clause) ;
   - `etre_satisfiable` est en $\mathcal{O}(m \times n^2)$ (dans le pire cas, on propage n fois).

   La complexité est polynomiale, ce qui n'est pas le cas pour les algorithmes connus qui résolvent SAT, qui est NP-complet.

8. (a) une valuation qui attribue 0 à chaque variable convient ;

(b) comme il y a équisatisfiabilité entre une formule et celle obtenue après propagation, il suffit juste de vérifier que les cas d'arrêt sont corrects. S'il y a une clause vide, elle ne peut pas être satisfaite, donc la formule n'est pas satisfiable, et sinon, la question précédente permet de répondre.

(c) Pour chaque clause unitaire x_i rencontrée au cours de l'algorithme, on pose $\mu(x_i) = 1$. Lorsqu'il n'y a plus de clause unitaire, on pose $\mu(x_j) = 0$ pour toutes les variables x_j restantes.