Outils pour le Calcul des Prédicats

De – backer Cuvelier Sébastien
Llobregat Nicolas

Validation de syllogismes complexes à l'aide des diagrammes de Venn.

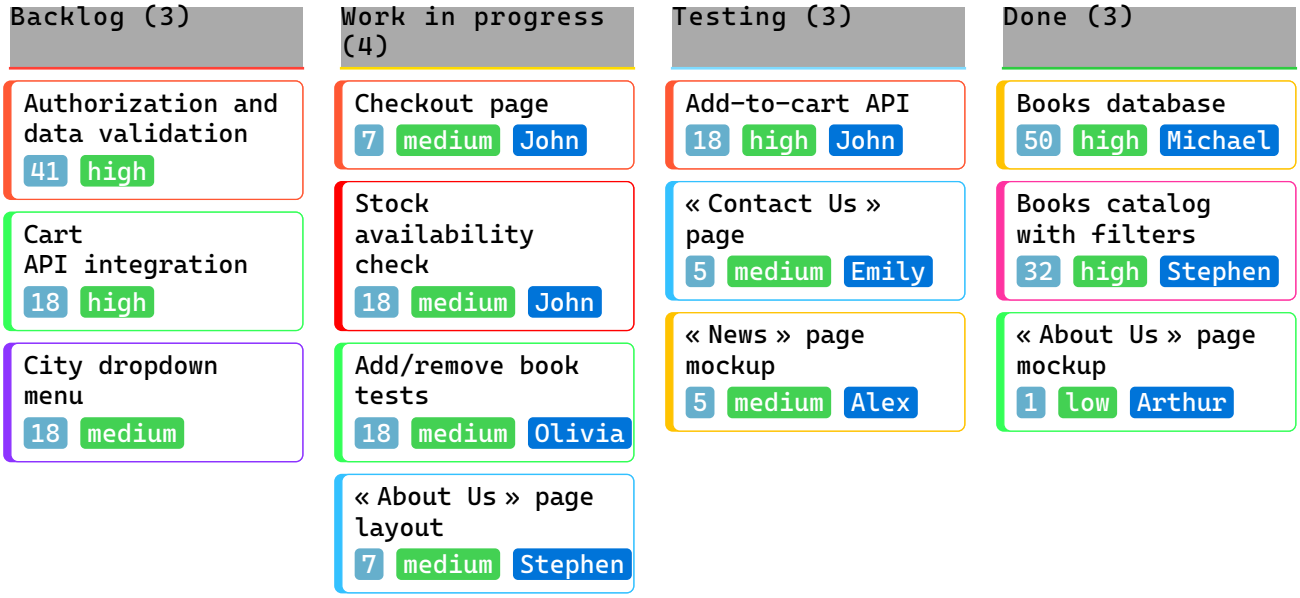
December 21, 2025

Table des matières

1. Gestion de projet

1.1. Organisation du travail

Afin d'organiser notre travail nous avons décider de suivre l'avancée du projet et des futures tâches à réaliser grâce à un tableau dit « kanban » sur Notion, dont voici un exemple:



Ainsi nous avons pu avoir une vue d'ensemble sur les tâches à réaliser, ainsi que le travail déjà effectué. De plus nous avons utiliser un logiciel de versionning afin de pouvoir collaborer de manières efficace sur les mêmes codes sources. Il s'agit de GIT, nous avions donc un repository Github sur lequel nous effectuons une pull request à chaque feature implémentée. Une fois approuvé par l'autre nous pouvions merge la pull request sur la branche de travail principale.

2. Réponses aux questions

Expliquons pour chacune des affirmations qui vont suivre pourquoi elles sont valides.

2.1. Question 1.1 ——————————————————————————

Si des diagrammes D et D' sont contradictoires, alors D n'est pas compatible avec D'.

Revenons sur la notion de contradiction de deux diagrammes. On dit que deux diagrammes D et D' sont contradictoires s'il existe une zone qui est non-vide dans D et vide dans D', ou inversement. Par exemple ci-dessous sont contradictoires car la zone A est vide dans le Diagramme D et non-vide dans le diagramme D'.

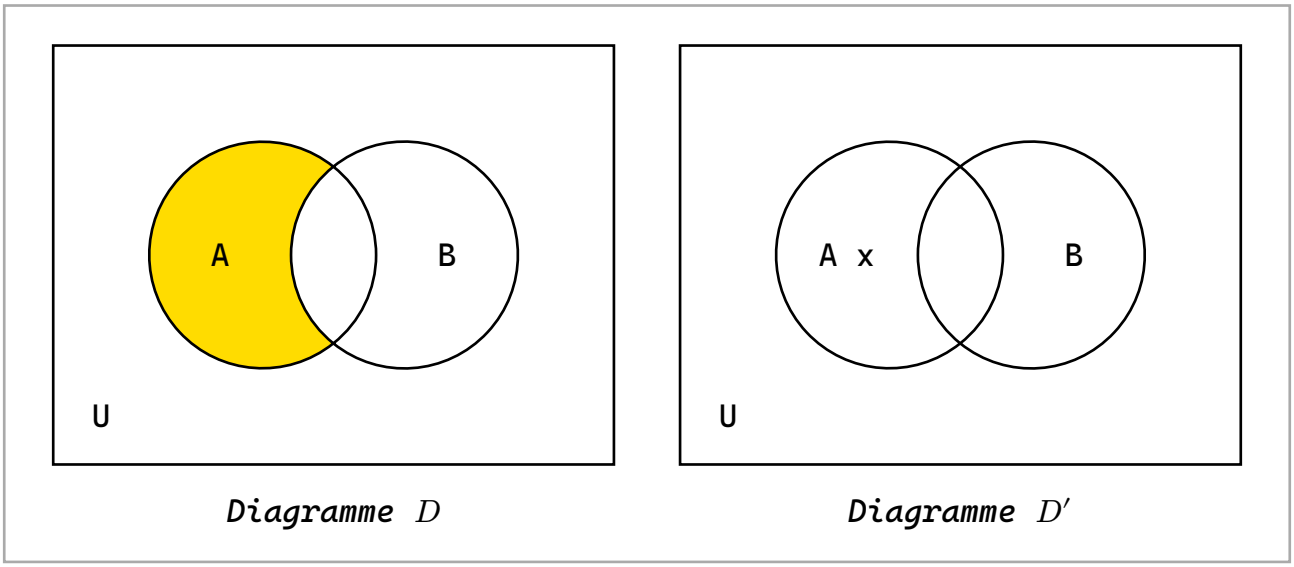


Figure 1.1 : Exemple de diagrammes contradictoires

Ils ne sont donc pas compatibles, car il n'est en aucun cas possible que $D \wedge D' = \text{Vrai}$ car $D \wedge D' \equiv \text{Faux}$. Ce qui termine l'explication.

2.2. Question 1.2 ——————————————————————————

Si des diagrammes D et D' sont contradictoires alors toute extension de D est en contradiction avec D'.

Commençons par définir ce qu'est une extension. Une extension dans le contexte d'un syllogisme fait référence à l'idée d'agrandir ou d'étendre un diagramme représentant une prémisse ou une conclusion afin de couvrir d'autres les possibilités entre les différentes zones de l'univers considéré. Plus spécifiquement, il s'agit de compléter ou ajouter des informations dans le diagramme. Cela semble en effet trivial, reprenons notre exemple de la question 1.1 (Figure 1.1). Ajoutons cependant une information supplémentaire à D, par exemple $\exists x, a(x) \wedge b(x)$.

Il est en effet évident que le diagramme obtenu après conjonctions soit toujours en contradiction avec D puisque de toutes manières la zone menant à la contradiction ne change pas.

2.3. Question 1.3 ——————————————————————————

Un diagramme D est compatible avec un diagramme D' si et seulement si D est une extension de D'.

D'après les questions précédentes, nous savons déjà que si D et D' sont contradictoires, alors toute extension de D' sera également contradictoire, et donc incompatible avec D. De plus, étant donné que le nombre d'extensions possibles est fini, cela signifie que D et D' seront incompatibles dans ce cas.

Maintenant, supposons que D et D' ne sont pas contradictoires. Cela signifie qu'il existe au moins une extension de D' qui ne contredit pas D. Posons donc D comme étant le diagramme possédant le plus de

contraintes parmi toutes les extensions possibles de D'. Si D possède le plus de contraintes, cela signifie que toute extension de D' qui satisfait les contraintes de D doit nécessairement être compatible avec D. En d'autres termes, D est une extension de D', et donc les deux diagrammes sont compatibles.

Ainsi, nous avons montré que si D est une extension de D', alors D et D' sont compatibles. Réciproquement, si D et D' sont compatibles, alors D doit être une extension de D'. Par conséquent, un diagramme D est compatible avec un diagramme D' si et seulement si D est une extension de D'.

2.4. Question 1.4 ——————————————————————————

Le nombre d'extensions (respectivement extensions complètes) d'un diagramme de Venn est 3^k (respectivement 2^k), où k est le nombre de zones non définies dans le diagramme.

Dans un diagramme possédant k zones chaque zone peut-être dans dans trois états distincts:

- **Existence** (symbolisée par une croix)
- **Exclusion** (symbolisée par une zone pleine)
- **Neutre** (symbolisée par une zone vide)

Soit pour chaque zone trois choix possible, donc 3^k. Dans le cas d'extensions complètes nous n'avons pas de zones indéfinies donc plus que deux choix possibles pour chaque zones donc 2^k possibilités.

2.5. Question 1.5 ——————————————————————————

Si D est un diagramme complet, alors deux diagrammes D et D' ne sont pas contradictoires si et seulement si D est compatible avec D'.

Si D et D' sont compatibles, cela signifie qu'il existe au moins une extension de D' qui ne contredit pas D.

Si les deux diagrammes sont complets, il n'y a aucune zone indéfinie. Par conséquent, l'extension de D' doit exactement correspondre aux zones de D, sans générer de contradiction. Cela prouve que les deux diagrammes ne sont pas contradictoires.

Si D et D' ne sont pas contradictoires, cela signifie qu'il existe une situation où les zones correspondantes des deux diagrammes peuvent coexister sans contradiction. Si les deux diagrammes sont complets, cela signifie que, pour chaque zone, leur valeur est fixée de manière compatible, ce qui montre que D et D' sont compatibles (au moins une extension de D' existe qui correspond à D).

2.6. Question 2.1 ——————————————————————————

La conjonction d'une liste de diagrammes l avec la négation d'un diagramme D est vide si et seulement si tout diagramme de l est compatible avec D.

Soit l, une liste de diagrammes, et D, un diagramme quelconque. On suppose d'abord que la conjonction de cette liste avec la négation de D est vide. Cela signifie qu'il n'existe aucun diagramme de l qui possède une zone contrainte en commun avec la négation de D. Donc tous les diagrammes de l ont leurs zones contraintes compatibles avec celles de D, ce qui veut dire que tous les diagrammes de l sont compatibles avec D.

Réciproquement, si tous les diagrammes d'une liste l sont compatibles avec D, alors toutes les zones contraintes de tous les diagrammes de l sont compatibles avec celles de D. Donc lorsque que l'ont calcule la négation de D, on remarque qu'aucun des diagrammes de l n'est compatible avec la négation de D. Ce qui signifie que lorsque l'on fait la conjonction de l et D on obtient une liste vide.

2.7. Question 2.2 ——————————————————————————

La conjonction d'une liste de diagrammes l avec la négation d'une liste de diagrammes l' ne contient que des extensions de diagrammes de l en contradiction avec les diagrammes de l'.

Soit l et l' deux liste de diagrammes, on peut calculer la négation de l' et en faire la conjonction avec l qu'on note C. On peut remarquer que tous les diagrammes de C sont compatibles à la fois avec les diagrammes de l et les diagrammes de la négation de l', car en faisant la conjonction des deux listes, les contraintes des diagrammes de C satisfont les contraintes de chaque diagramme de l et/ou de la négation de l'. Et

comme les diagramme de C sont compatibles avec les diagrammes de la négation de l', nous pouvons en conclure qu'ils sont donc incompatibles avec les diagrammes de l'.

3. Implémentation

Nous allons commenter les fonctions demandées explicitement dans le sujet afin d'avoir une idée précise de ce quelles réalisent.

3.1. DiagVenn.ml

3.1.1. negate_diag

La fonction negate_diag calcule la négation logique d'un diagramme de Venn. Elle prend un diagramme en entrée et renvoie une liste de diagrammes qui représentent la négation. La négation d'un diagramme avec plusieurs régions donne plusieurs possibilités alternatives (disjonction) donc nous devons bien renvoyer une liste de diagrammes.

Nous allons donc parcourir tout le diagramme d, et regarder chaque valeur, si nous avons une région Vide, dans ce cas nous allons ajouter à la liste un diagramme où cette région ne l'est pas respectivement pour une zone NonVide nous allons ajouter un diagramme ou la zone sera vide.

```OCaml
(** neg_reg reg : renvoie le complementaire d'une zone **)
let neg_reg (reg : fill) : fill =
  match reg with NonVide -> Vide | Vide -> NonVide

(** negate_diag d renvoie la négation du diagramme d*)
let negate_diag (d : diagramme) : diagramme list =
  if Diag.is_empty d then []
  else Diag.fold (fun k v acc -> Diag.singleton k (neg_reg v) :: acc) d []
```

3.1.2. negate_diag_list

Une liste de diagrammes représente une disjonction: au moins un des diagrammes doit être satisfait. Si nous avons donc une liste de

diagrammes l = [d1, d2, d3, ..., dn] cela représente en réalité: d1 ∨ d2 ∨ d3 ∨ ... ∨ dn. La négation devient alors la conjonction des négations de chaque di. Soit ¬d1 ∧ ¬d2 ∧ ¬d3 ∧ ... ∧ ¬dn. Il nous faut donc fusionner dans une seule liste l'ensemble des négations.

```OCaml
(** negate_diag_list ds renvoie la négation de la liste de diagrammes ds *)
let negate_diag_list (ds : diagramme list) : diagramme list =
  match ds with
  | [] -> [ Diag.empty ]
  | ds -> (
      let neg = List.map negate_diag ds in
      match neg with
      | [] -> []
      | hd :: tl -> List.fold_left conj_diag_list hd tl)
```

3.1.3. conj_diag

```OCaml
let conj_diag (d1 : diagramme) (d2 : diagramme) : diagramme option =
  let result =
    Diag.merge
      (fun _ v1 v2 ->
        match (v1, v2) with
        | Some Vide, Some Vide -> Some (Some Vide)
        | Some NonVide, Some NonVide -> Some (Some NonVide)
        | Some Vide, Some NonVide -> Some None
        | Some NonVide, Some Vide -> Some None
        | Some v, None -> Some (Some v)
        | None, Some v -> Some (Some v)
        | None, None -> None)
      d1 d2
  in
  if Diag.exists (fun _ v -> v = None) result then None
  else
    Some (Diag.filter_map (fun _ v -> v) result)
```

3.1.4. conj_diag_list

La conjonction de deux listes de diagrammes est la combinaison cartésienne des conjonctions des diagrammes pour lesquelles la conjonction est définie (en faisant attention au cas ou la fonction conj_diag nous renvoie None), conj_diag d1 d2 calcule la conjonction de deux diagrammes, c'est-à-dire un diagramme contenant les contraintes définies par les deux diagrammes. Si la même zone est vide dans l'un et

non vide dans l'autre, alors cette conjonction est impossible et la fonction renvoie None (fonction partielle).

```OCaml
let conj_diag_list (ds1 : diagramme list) (ds2 : diagramme list) :
    diagramme list =
  let cartesian_product =
    List.concat (List.map (fun a -> List.map (fun b -> (a, b)) ds2) ds1)
  in
  List.filter_map
    (fun (d1, d2) ->
      match conj_diag d1 d2 with
      | None -> None
      | Some combined_diag -> Some combined_diag)
    cartesian_product
```

3.1.5. disj_of_diag_list

La disjonction l et l' est la concaténation des deux listes de diagrammes :

```OCaml
let disj_of_diag_list (ds1 : diagramme list) (ds2 : diagramme list) :
    diagramme list =
  ds1 @ ds2
```

3.1.6. diags_of_bool_comb

La fonction traite chaque cas du type boolCombSyllogismes définit comme suit dans le fichier Formule_Syllogisme.ml:

```OCaml
type boolCombSyllogismes =
  | Vrai
  | Faux
  | Base of formule_syllogisme
  | Et of boolCombSyllogismes * boolCombSyllogismes
  | Ou of boolCombSyllogismes * boolCombSyllogismes
  | Non of boolCombSyllogismes
```

Dans la fonction diags_of_bool_comb nous devons donc traiter chacun de ces cas, sachant que:

Vrai: Représente une tautologie, donc aucune contrainte ce qui correspond au diagramme vide

Faux: Représente une contradiction, donc aucun diagramme valide ce qui correspond a la liste vide

Base f: Pour une formule de syllogisme de base, on utilise la fonction existante diag_from_formule (c'est ici qu'on introduit alpha).

Et (b1, b2): La conjonction logique correspond à combiner les diagrammes des deux sous-formules avec conj_diag_list

Ou (b1, b2): La disjonction logique correspond à prendre l'union des diagrammes avec disj_of_diag_list

Non b': La négation inverse tous les diagrammes avec negate_diag_list

La fonction auxilliaire permettant de recupérer les atomes de b, on la fusionnera avec la liste alpha le cas échéant dans le fonction principale.

```OCaml
let rec atomes_of_bool_comb (b : boolCombSyllogismes) : string list =
  match b with
  | Vrai -> []
  | Faux -> []
  | Base f -> ( match f with PourTout x -> atomes x | IlExiste x -> atomes x)
  | Et (a, b) -> atomes_of_bool_comb a @ atomes_of_bool_comb b
  | Ou (a, b) -> atomes_of_bool_comb a @ atomes_of_bool_comb b
  | Non a -> atomes_of_bool_comb a
```

```OCaml
let diags_of_bool_comb (alpha : string list) (b : boolCombSyllogismes) :
    diagramme list =
  let all_atomes = List.sort_uniq compare (alpha @ atomes_of_bool_comb b) in
  let rec aux b =
    match b with
    | Vrai -> [ Diag.empty ]
    | Faux -> []
    | Base f -> diag_from_formule all_atomes f
    | Et (b1, b2) -> conj_diag_list (aux b1) (aux b2)
    | Ou (b1, b2) -> disj_of_diag_list (aux b1) (aux b2)
    | Non b' -> negate_diag_list (aux b')
  in
  aux b
```

3.2. ValiditeNaive.ml

Pour valider un syllogisme complexe nous avons besoin de savoir si les diagrammes des prémisses ne sont pas en contradiction avec ceux de la conclusion.

3.2.1. complete_diags

Complete_diags permet de créer la liste de toutes les extensions complètes du diagramme d en considérant la listes d'atomes ats pour considérer les zones à compléter.

- **Exemple**:
  - Si un diagramme n'est composé que des zones {∅, A}; et que l'on souhaite l'étendre avec les atomes {A, B, C}. On aura dans le diagramme les zones {∅, A, B, C, AB, AC, BC, ABC}

```OCaml
let complete_diags (d : diagramme) (ats : string list) : diagramme list =
let ud =
    List.filter (fun reg -> not (Diag.mem reg d)) (all_prem_sublist ats)
  in
  let d_list = List.map (fun p -> diag_from_undef d p) ud in
  match d_list with
  | [] -> [ d ]
  | _ ->
      List.fold_left
        (fun acc dl -> conj_diag_list acc dl)
        (List.hd d_list) (List.tl d_list)
```

3.2.1.1. Fonctions auxiliaires

```OCaml
let diag_from_undef (d : diagramme) (p : Predicate_set.t) : diagramme list =
  [ Diag.add p NonVide d; Diag.add p Vide d ]
```

diag_from_undef, permet de créer toutes les extensions d'un diagramme donné à partir d'une zones non définie. Pour l'exemple ci-dessous, les extensions de D sur la zones A ∧ B sont les diagramme E et E'

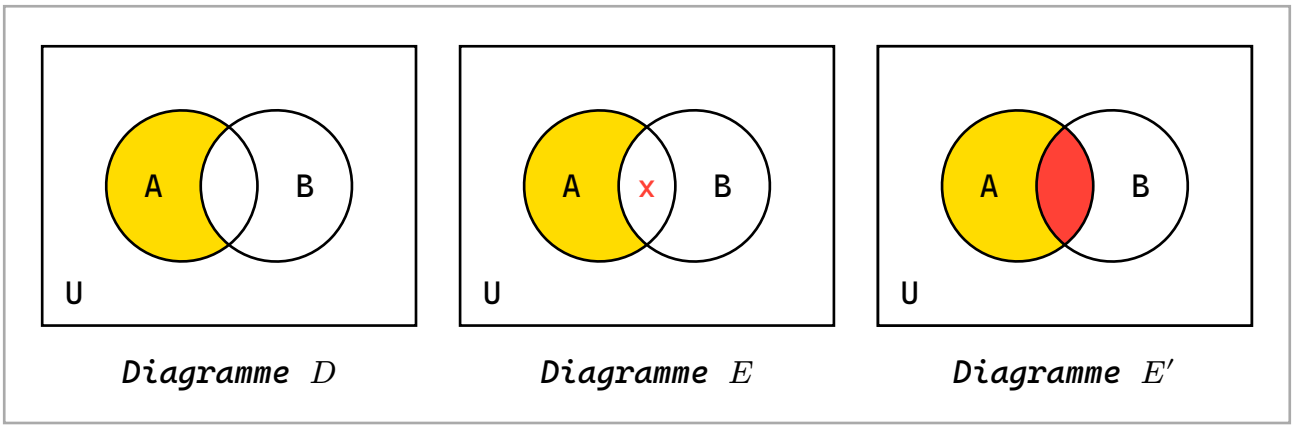


Figure 1.2 : Exemple d'exécution de diag_from_undef

```OCaml
let all_prem_sublist (ats : string list) : Predicate_set.t list =
  List.map
    (fun at ->
      List.fold_left
        (fun acc a -> Predicate_set.add a acc)
        Predicate_set.empty at)
    (all_sublists ats)
```

Cette fonction permet simplement de construire la liste de toutes les sous listes d'ats mais sous forme de liste de prédicats.

3.2.2. is_contradiction

Teste si les diagrammes d1 et d2 sont contradictoires, c'est-à-dire qu'il existe au moins une zone commune au deux diagrammes qui ont des contraintes incompatibles.

```OCaml
let is_contradiction (d1 : diagramme) (d2 : diagramme) : bool =
  let check_zone pre v1 =
    match Diag.find_opt pre d2 with
    | None -> false
    | Some v2 -> v1 <> v2
  in
  let contradiction_d1_to_d2 = Diag.exists check_zone d1 in
  let check_zone_inverse pre v2 =
    match Diag.find_opt pre d1 with
    | None -> false
```

```
    | Some v1 -> v2 <> v1
  in
  let contradiction_d2_to_d1 = Diag.exists check_zone_inverse d2 in
  contradiction_d1_to_d2 || contradiction_d2_to_d1
```

3.2.3. est_valid_premiss_conc

Teste naïvement la validité d'un syllogisme complexe, défini par deux valeurs de types **boolCombSyllogismes** (une pour les prémisses et une pour la conclusion). Cette fonction fait appel à **temoins_invalidite_premisses_conc**. Si le résultat de cet appel est une liste vide alors aucun diagramme contradictoire n'a été trouvé et donc b_1 valide b_2.

```OCaml
let est_valid_premiss_conc (b1 : boolCombSyllogismes)
    (b2 : boolCombSyllogismes) : bool =
(temoins_invalidite_premisses_conc b1 b2) = []
```

3.2.4. temoins_invalidite_premisses_conc

Cette fonction construit la liste des diagrammes étendues de b_1 contredisant au moins un diagramme de b_2

```OCaml
let temoins_invalidite_premisses_conc (b1 : boolCombSyllogismes)
    (b2 : boolCombSyllogismes) : diagramme list =
  let atomes_b1 = atomes_of_bool_comb b1
  and atomes_b2 = atomes_of_bool_comb b2 in
  let atomes = List.sort_uniq String.compare (atomes_b1 @ atomes_b2) in
  let db1 =
    List.concat_map
      (fun d -> complete_diags d atomes)
      (diags_of_bool_comb atomes b1)
  and db2 =
    List.concat_map
      (fun d -> complete_diags d atomes)
      (diags_of_bool_comb atomes b2)
  in
  List.filter
    (fun d1 -> List.for_all (fun d2 -> is_contradiction d1 d2) db2)
    db1
```

3.3. ValiditeViaNegation.ml

3.3.1. temoins_invalidite_premisses_conc'

Comme dit dans le sujet nous pouvons utiliser la méthode de la conjonction avec les diagrammes inverses de la conclusion

$$[d_1, ..., d_n] \overset{?}{\rightarrow} [d'_1, ..., d'_m] \Leftrightarrow [d_1, ..., d_n] \wedge \neg[d'_1, ..., d'_m] \overset{?}{\rightarrow} []$$

Il nous faut donc créer la liste des diagrammes associés à la combinaison booléenne b1 de formules pour les syllogismes, sur les prédicats issus de b1.

Pour ce faire nous avons déjà implémenter la fonction: diags_of_bool_comb que nous pouvons appelé avec pour argument alpha la liste vide.

Il nous faut également construire la liste des diagrammes associés à b2 puis la nier avec negate_diag_list. Enfin nous faisons la conjonctions de ces deux dernières listes de diagrammes avec la fonction que nous avons réaliser auparavant: conj_diag_list. Si le résultat est la liste vide alors il n'existe pas de diagramme de la conjonction des diagrammes de b1 avec la négation de b2, qui contredisent un diagramme de b2 sinon on a nos témoins.

```OCaml
let temoins_invalidite_premisses_conc' (b1 : boolCombSyllogismes)
    (b2 : boolCombSyllogismes) : diagramme list =
  let diags_b1 = diags_of_bool_comb [] b1 in
  let diags_b2 = diags_of_bool_comb [] b2 in
  let diags_not_b2 = negate_diag_list diags_b2 in
  conj_diag_list diags_b1 diags_not_b2
```

3.3.2. est_valid_premiss_conc'

En réalité la fonction est identique à temoins_invalidite_premisses_conc' mais nous comparons le résultat à la liste vide, si elle l'est nous n'avons pas de témoins donc le syllogisme est valide (respectivement invalide si non vide).

```OCaml
let est_valid_premiss_conc' (b1 : boolCombSyllogismes)
    (b2 : boolCombSyllogismes) : bool =
  temoins_invalidite_premisses_conc' b1 b2 = []
```

4. Exemple complet

Dans la suite nous allons étudier ces deux formules:

```OCaml
(* b1 : pour tout x, a(x) ou b(x)*)
let b1 = Base (PourTout (Ou (Atome "a", Atome "b")))
(* b2 : (pour tout x, a(x)) ou (pour tout x, b(x)) *)
let b2 = Ou (Base (PourTout (Atome "a")), Base (PourTout (Atome "b")))
```

En passant par les tables de vérités:

a	b	a ∨ b
0	0	0
0	1	1
1	0	1
1	1	1

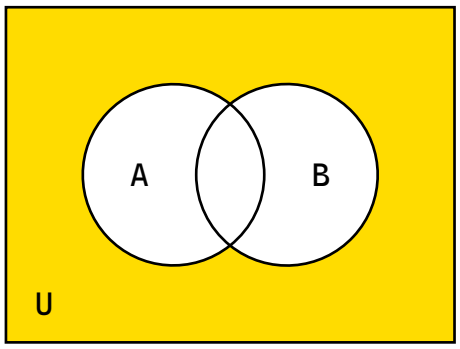


Diagramme p_1

a	b	a
0	0	0
0	1	0
1	0	1
1	1	1

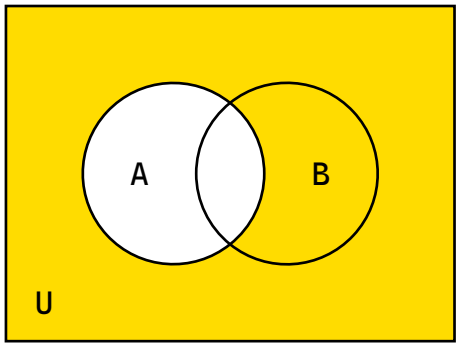


Diagramme p_2

a	b	b
0	0	0
0	1	1
1	0	0
1	1	1

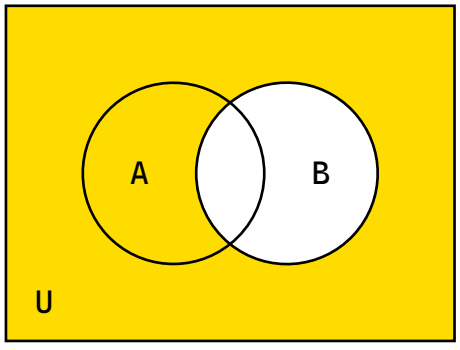


Diagramme p_3

4.1. ValiditeViaNegation

On doit calculer la négation des diagrammes de la conclusion

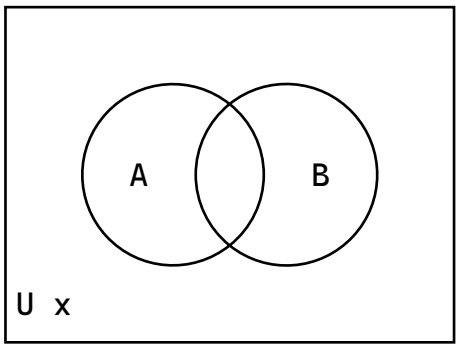


Diagramme p_4

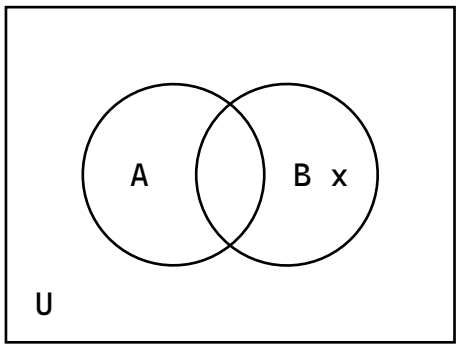


Diagramme p_5

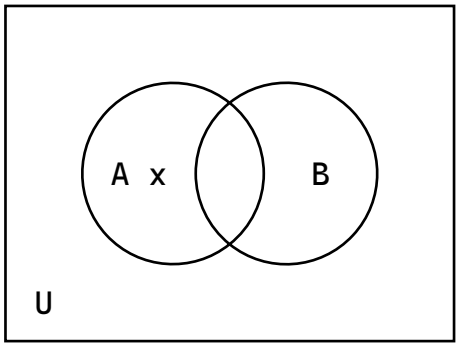
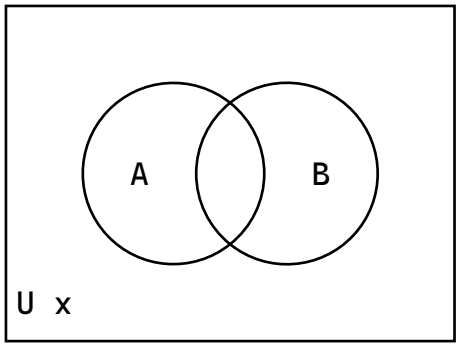


Diagramme p_6



On peut remarquer par ailleurs que le premier et le dernier diagramme sont identiques on en considère donc qu'un seul.

On doit maintenant faire la conjonction entre la liste des diagrammes des prémisses et la négation de la liste des diagrammes de la conclusion. Le cas échéant si la liste résultante est vide le syllogisme est valide, dans le cas contraire nous avons une liste de témoins d'invalidité.

Ici on vois que les diagrammes p_1 et p_4 ne sont pas compatibles:

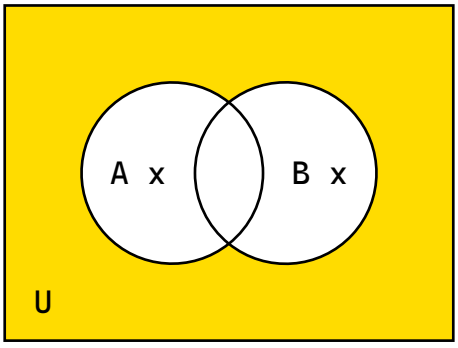
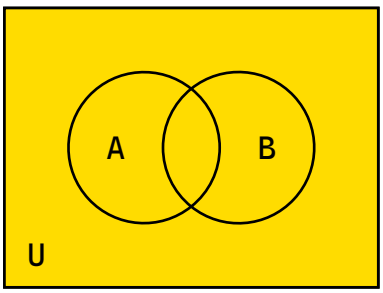


Diagramme p_7 *témoin d'incompatibilité*

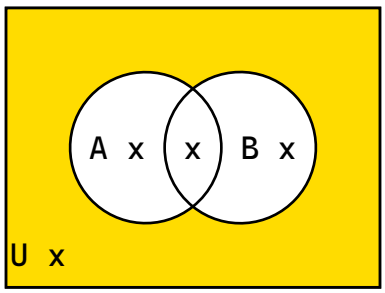
Ce diagramme est une extension du diagramme p_1 en contradiction avec ceux de la conclusion étendue $p_2 \vee p_3$. Le syllogisme est donc invalide.

4.2. ValiditeNaive

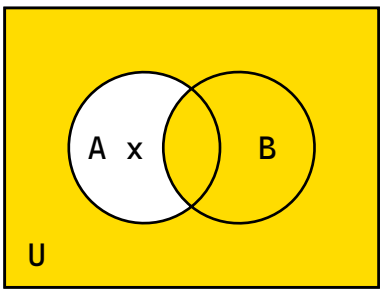
On doit d'abord calculer toutes les extensions complètes possibles du diagramme b_1, c'est-à-dire construire des diagrammes contenant toutes les contraintes initialement dans d et toutes les possibilités de contraintes pour toutes les zones non définies. Dans notre exemple, nous avons 3 zones non définies dans le diagramme b_1 et donc le nombre de diagrammes à construire est 2^3 = 8:



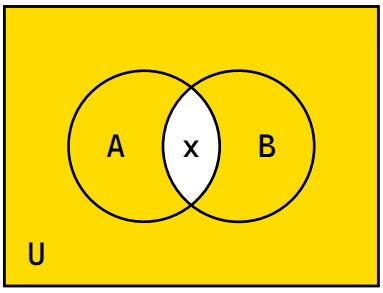
E_1



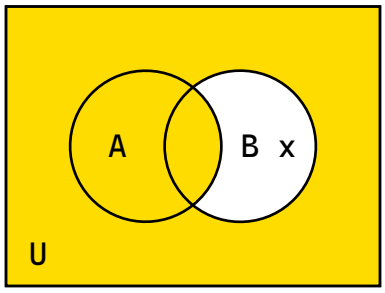
E_2



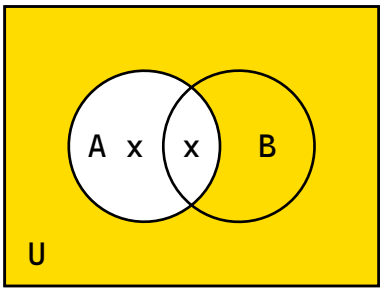
E_3



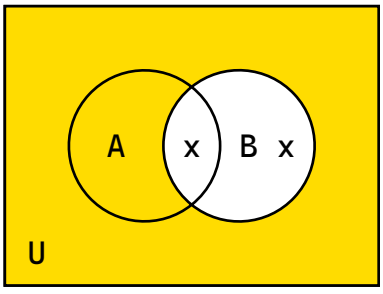
E_4



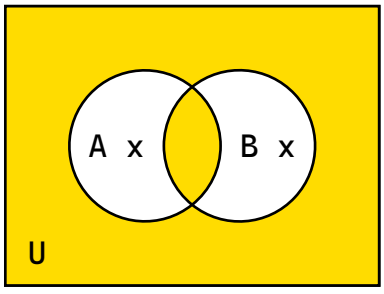
E_5



E_6



E_7



E_8

Maintenant que nous avons tous les diagrammes étendus de b_1, nous devons vérifier que tous ces diagrammes valident au moins 1 diagramme de b_2. Il faut donc que toutes les contraintes du diagrammes de b_2 soient incluent dans le diagramme de b_1:

- E_1, E_4, E_5 et E_7 incluent les contraintes de p_3 donc ces diagrammes valident b_2
- E_3, E_1, E_4, E_6 incluent les contraintes de p_2 donc ces diagrammes valident b_2
- E_2, E_8 n'incluent aucun des diagrammes de b_2. Donc E_2 et E_8 sont les contres exemples du syllogismes.

Ici la liste des diagrammes étendus de b_1 en contradiction avec b_2 n'est pas vide ce qui implique que le syllogisme est invalide.