Corrigé du sujet 0 CCINP

I Partie I : logique.

1. On peut utiliser B pour "marquer un but", C pour "être content", F pour "faire la fête" et G pour "l'équipe gagne".
2. (i) $F1 = B \Rightarrow C \wedge F$
   (ii) $F2 = G \Rightarrow C \vee F$
   (iii) $F3 = \neg G \Rightarrow \neg C \vee B$
   (iv) $F4 = \neg B \wedge F \Rightarrow C$
   (v) $F5 = \neg C$
3. (i) $\neg B \vee (C \wedge F)$ qui est équivalente à $(\neg B \vee C) \wedge (\neg B \vee F)$
   (ii) $\neg G \vee C \vee F$
   (iii) $G \vee \neg C \vee B$
   (iv) $B \vee \neg F \vee C$
   (v) $\neg C$
4. En trouvant une valeur de vérité permettant de satisfaire la formule qui correspond à la conjonction des quatres affirmations et donc des quatres formules ci-dessus, on disposera de faits plausibles (en cohérence avec l'énoncé). L'énoncé laisse sous-entendre qu'il n'y aura qu'une seule valuation permettant de satisfaire cette formule et ainsi on saura si leu joueur a marqué etc...
   Ainsi on souhaite trouver une valeur de satisfiabilité de la formule F définie par :

   $$F = (\neg B \vee C) \wedge (\neg B \vee F) \wedge (\neg G \vee C \vee F) \wedge (G \vee \neg C \vee B) \wedge (B \vee \neg F \vee C) \wedge \neg C$$

   On s'intéresses donc ici à un problème de satisfiabilité d'une formule qui est sous forme normale conjonctive.
5. $$\mathcal{C} = \{(\neg B \vee C), (\neg B \vee F), (\neg G \vee C \vee F), (G \vee \neg C \vee B), (B \vee \neg F \vee C), \neg C\}$$

   On évalue C à faux et on obtient :

   $$\mathcal{C}_1 = \{\neg B, (\neg B \vee F), (\neg G \vee F), (B \vee \neg F)\}$$

   On évalue G à faux et on obtient :

   $$\mathcal{C}_2 = \{\neg B, (\neg B \vee F), (B \vee \neg F)\}$$

   On évalue B à faux et on obtient :

   $$\mathcal{C}_3 = \{\neg F\}$$

   On évalue B à faux et on obtient :

   $$\mathcal{C}_4 = \varnothing$$

   On renvoie vrai.

II Partie II : Bases de données

6.

7. III Partie III

III.1 Structures de données

8. On peut utiliser le type structuré suivant :
   `typedef struct element{Domino d; struct element* suivant;} element;`
9. Une chaîne est alors l'adresse du premier élément de la liste et on peut définir le type suivant : `typedef element* chaine;`
10. On peut ajouter l'élément récursivement dès lors que la liste n'est pas vide ou bien parcourir itérativement la liste jusqu'à tomber sur le dernier élément (celui dont le champ suivant est `NULL`).

```
element* ajoutElement(element* l,Domino d){
   //si la liste est vide on renvoie une nouvelle liste.
   if(l==NULL){
      element* res = malloc(sizeof(element));
      res->d=d;
      res->suivant=NULL;
      return res;
   }
    //sinon on accroche récursivement
    else{
              l->suivant = ajoutElement(l->suivant, d);
              return l;
     }
}
```

On peut aussi proposer une version itérative :

```
element* ajoutElement(element* l, Domino d){

   if (l==NULL){
      element* new = malloc(sizeof(element));
      new->d=d;
      new->suivant=NULL;
      return new;
}

   element* courant = l; //courant n'est pas NULL
   while (courant->suivant != NULL){
      courant=courant->suivant; //courant reste non NULL
   }
   //on s'arrete sur le dernier élément de la liste
   //qui est à l'adresse courant
   element* new = malloc(sizeof(element));
```

```
   new->d=d;
   new->suivant=NULL;
   courant->suivant=new;
   return l;
}
```

11. On supposera que l'élément est effectivement dans la chaine (qui est donc non nulle). On propose une fois encore une version récursive puis une version itérative. Ici on doit garder le maillon `précédent` en mémoire dans la version itérative afin de pouvoir supprimer le maillon `courant`.
    On a aussi besoin d'une fonction qui permet de savoir si deux dominos sont ou non égaux.

```
bool egale_dom(Domino d1, Domino d2){
    return(d1.x==d2.x && d1.y==d2.y);
}

element* retireElement(element* l,Domino d){
   assert(l != NULL);
   if (egale_dom(l->d,d)){
      return l->suivant;
   }
   else{
      l->suivant = retireElEment(l->suivant,d);
      return l;
   }
}
```

    Et en itératif :

```
element* retireElement(element* l,Domino d){
   assert(l != NULL);
   element* precedent = NULL;
   element* courant = l;
   while(courant != NULL && !(egale_dom(courant->d , d))){
      precedent=courant;
      courant=courant->suivant;
   }
     //normalement on s'arrête en ayant
     //trouvé d dans le maillon courant
   assert(courant != NULL);

   //on raccroche ce qui précède le maillon
   //courant et ce qui lui succède
   precedent->suivant = courant->suivant;
   return l;
}
```

12. Cette fonction ressemble à la précédente sauf qu'on peut ne pas trouver l'élément et donc aboutir sur une liste courante `NULL`. On remarquera la simplicité de la version récursive.

```
bool rechercheElement(element *l, Domino d){
   if (l==NULL)
      return false;
   else{
      return (egale_dom(l->d,d) || rechercheElement(l->suivant,d));
   }
}

bool rechercheElement(element *l, Domino d){
      element* courant = l;
      while(l != NULL && !(egale_dom(l->d, d))){
         courant=courant->suivant;
      }
      //on a trouvé d ssi courant n'a pas atteint
      // la fin de la liste en devenant NULL
      return(courant !=NULL);
}
```

III.2 Existence d'une chaine de taille n.

13. Il me semble qu'il y a ici une erreur dans l'énoncé, devoir parcourir le sac pour déterminer qui sont les dominos numéros i et j me semble inutilement coûteux et la question suivante laisser suggérer qu'on s'intéresse plutôt à la situation où on prend les dominos en entrée. De plus, pour être cohérente avec la suite, ma fonction teste si D_j peut être placé à droite de D_i.

```
bool possible(Domino Di,Domino Dj){
   return(Di.y==Dj.x);
}
```

14. Ce pointeur permet de modifier la valeur de `Dj` (passage par adresse) et ainsi de faire effectivement la rotation sur `Dj` et donc de modifier le contenu de cette variable.
15. Je choisis de renvoyer `true` uniquement dans les cas où on fera effectivement une rotation.

```
bool possibleAvecRotation(Domino Di,Domino *Dj){
   if(Di.y==Dj->y){
      Dj->y=Dj->x;
      Dj->x=Di.y;
      return true;
   }
   return false;
}
```

16. Pour passer d'une $k$chaine à une $k+1$ chaine, il suffit d'ajouter un domino dont l'une des deux valeurs est égale à la dernière valeur du dernier domino de la chaine (avec ou sans rotation, on pourra utiliser les deux fonctions `possible` pour décider cela).
17. On peut considérer la stratégie suivante :
    — On part de la chaine vide.

— A chaque étape, on détermine l'ensemble des dominos possibles pour aggrandir la chaine courante à l'aide des fonctions `possible` et `possibleAvecRotation` et on essaie de compléter avec chacun d'entre eux avant de lancer un appel récursif.

— Si à une étape, aucun domino restant n'est possible, alors on s'arrête et on fait un retour sur trace.

18. Cela donne le code suivant (cette question est beaucoup trop difficile posée telle quelle dans le sujet).
La fonction suivante prend en entrée un sac et une solution partielle ainsi que le dernier domino et renvoie, si elle existe, une solution complétée avec les éléments du sac.

```
element* cherche_sol(element* sac, element* res, Domino dernier){
    if (sac==NULL){
        return res;
    }
    else{
    //on commence par construire la liste des dominos
    //que l'on peut utiliser pour la prochaine place
    element* parcours_sac = sac;
    element* atester=NULL;
    while(parcours_sac != NULL){
        if (possible (dernier, parcours_sac->d) ||
        possibleAvecRotation(dernier, &(parcours_sac->d))){
            atester=ajoutElement(atester, parcours_sac->d);
        }
        parcours_sac=parcours_sac->suivant;
    }
        // on les teste un part un
        while(atester !=NULL){
            element* res_poss=
               cherche_sol(retireElement(sac,atester->d),
               ajoutElement(res,atester->d), atester->d);
            if (res_poss != NULL){
                return res_poss;
            }
            else{//on n'oublie pas de retablir la situation
                sac = ajoutElement(sac,atester->d);
                res = retireElement(res, atester->d);

            }
        atester=atester->suivant;

    }
    //si aucun ne convient, on renvoie NULL
    return NULL;
    }
}

//on écrit la fonction finale
element* cherche_solution(element* sac){
    if (sac==NULL){
```

```
        return NULL;
    }
    //on essaie avec chaque élément comme début
    else{element* sol_deb=sac;
        while(sol_deb !=NULL){
            Domino dernier = sol_deb->d;
            sac = retireElement(sac,dernier);
            element* res = ajoutElement(NULL,dernier);
            element* sol_possible = (cherche_sol(sac,res,dernier));
            if (sol_possible != NULL) return sol_possible;
            sac = ajoutElement(sac,dernier);
            sol_deb=sol_deb->suivant;
        }
        return NULL;
    }
}
```

19. Dans le pire cas, à chaque étape, on teste tous les dominos restants comme successeurs ce qui donne une complexité en $O(n!)$ où n est le nombre de dominos. ??? atteint ???

III.3 Nombre de chaines de taille n

20. On compte ici le nombre de couples non ordonnés de deux entiers de $[|0, N|]$ avec répétition éventuelle donc $S_N = N + 1 + \binom{N+1}{2}$.
21. graphe complet à trois sommets.
22. Chaque sommet est relié aux N autres sommets et donc le degré d'adjacence de chaque sommet vaut N.
23. La somme des degrés est un nombre pair car vaut 2 fois le nombre d'arêtes : quand on somme tous les degrés, on compte chaque arrête exactement une fois. Si on réduit cette somme modulo 2, on obtient donc que le nombre de sommets de degrés impairs est congru à 0 modulo 2 et donc est pair (quand on réduit modulo deux la somme des degrés, chaque degré pair vaut alors 0 et chaque degré impair vaut 1 modulo 2 ce qui garantit que la somme des degrés des sommets est congrue au nombre de sommets de degrés impairs modulo 2).
24. Si G comporte un chemin eulérien alors tout sommet qui n'est pas une extrémité du chemin doit être de degré pair (car le chemin doit emprunter les arêtes deux par deux à chaque fois que l'on traverse ce sommet puisqu'il ne peut pas emprunter deux fois une même arête). Ainsi un chemin eulérien possède au plus deux sommets de degré impair.
25. Soit G un graphe connexe, on définit G' dans lequel on ajoute une arête entre $s1$ et $s2$ (cette arrête pouvant déjà exister on se retrouve ici avec un multigraphe et non pas un graphe, ceci constitue un défaut de l'énoncé mais les résultats restent corrects), ainsi tous les sommets de G' sont de degrés pairs et alors G' admet un chemin eulérien que l'on va noter C. C passe par toutes les arêtes donc contient $(s1, s2)$ et on va noter $C = (u_1, \ldots, s_1, s_2, \ldots, u_n)$. Puisuqe toutes les arêtes sont utilisées dans C et que tous les degrés sont pairs alors nécessairement $u_1 = u_n$ car sinon ces deux sommets seraient de degrés impairs pour les mêmes raisons que précédemment. Ainsi, C est un cycle eulérien et en retirant l'arête a du cycle on obtient un chemin eulérien de G.
26. Si C n'est pas une boucle alors il a forcément son premier sommet pour lequel seul un nombre impair d'arêtes adjacentes n'ont été empruntées : une arête est utilisée pour quitter

le premier sommet puis on utilise un nombre pair d'arêtes à chaque passage (je suppose ici que les chemins considérés ne réutilisent pas deux fois une même arête ce qui me semble implicite mais non précisé dans l'énoncé). Ainsi, puisque tous les degrés sont pairs, il reste une arête adjacente au premier sommet ce qui permettrait de prolonger C et qui contredirait le fait qu'ils soit de longueur maximale.

27. S'il existe au moins une arête qui n'est pas dans C alors puisque le graphe G est connexe, il y a un sommet de C qui est adjacent à une arête qui n'est pas dans C. Considérons $s \in C$ un tel sommet et $(s,t) \notin C$ une arête qui n'est pas dans C. Dans ce cas, en considérant le cycle C de s à s que l'on prolonge par l'arête (s,t) on aurait un chemin plus long que C ce qui est exclu.
28. Dans K_7, tous les sommets sont de degré 6 donc le graphe admet une chemin et même un cycle eulérien.
29. Un cycle eulérien de longueur k donne lieu à k chaines (une pour chaque point de départ) ne contenant que des dominos non doubles. On va commencer par insérer les dominos doubles au sein des cycles obtenus : pour chaque cycle, on dispose de $N+1$ dominos doubles et chacun pourra s'intercaler entre deux des N dominos contenant sa valeur. Il y a donc $N/2$ emplacements possibles. Ainsi, pour chacune des $N+1$ valeurs, on doit choisir un des $N/2$ emplacements ce qui donne $\frac{N}{2}^{N+1}$. On obtient ainsi $\frac{N}{2}^{N+1} E_{N+1}$ cycles comprenant tous les dominos où k est la longueur du chemin eulérien. Il reste à voir combien de chaines de dominos différentes on obtient avec un cycle de dominos qui en définitive n'est riend d'autre qu'une chaine de dominos fermée. Il y a en a le nombre total de dominos puisqu'il suffit de choisir le domino où on va "ouvrir" le cycle pour en faire une chaine linéaire. On obtient donc bien en définitive : $S_n \frac{N}{2}^{N+1} E_{N+1}$ avec $S_N = N + 1 + \binom{N+1}{2}$ comme vu à la question 20. Tout à coup, on parle de circuit???

III.4 Recherche de la plus longue sous-séquence.

Ici sous-séquence semble qualifier des dominos consécutifs dans le sac initial.??

30.
31.
32. $l_{1,0} = 1 = l_{1,1}$.
33. Il suffit de rajouter un champ booléen au type indiquant si le domino est ou non retourné :

```
typedef struct {int x; int y; bool ret;} Domino;
```

34. III.5 Recherche du nombre de sous-séquences

35. On fixe i, j, k et l dans $[|1, n|]$ et on calcule $\bar{M}_{ij}.\bar{M}_{kl} = \bar{M}_{ij}\bar{M}_{kl} + \bar{M}_{kl}\bar{M}_{ij}$. On distingue plusieurs cas :
    — Si $i \neq j$ et $k \neq l$ alors on obtient : $(M_{ij} + M_{ji})(M_{kl} + M_{lk}) + (M_{kl} + M_{lk})(M_{ij} + M_{ji})) = M_{ij}M_{kl} + M_{ji}M_{kl} + M_{ji}M_{lk} + M_{ij}M_{lk} + M_{kl}M_{ij} + M_{kl}M_{ji} + M_{lk}M_{ij} + M_{lk}M_{ji}$.
        (a) Si $j \neq k$ et $i \neq k$ et $j \neq l$ et $i \neq l$ alors tous les termes sont nuls car on sait que le produit de deux matrices élémentaires vaut : $M_{ij}M_{kl} = \delta_{j=k}M_{il}$ et la somme est alors nulle.
        (b) Si $j = k$ alors $j \neq l$ et $i \neq k$ donc on obtient $M_{il} + M_{ji}M_{lk} + M_{kl}M_{ij} + M_{li} = \bar{M}_{il} + M_{ji}M_{lk} + M_{kl}M_{ij} = \bar{M}_{il} + \delta_{i=l}\bar{M}_{kj}$.

(c) Si $i = l$ alors $i \neq k$ et $j \neq l$ donc on obtient de même, $\bar{M}_{jk} + \delta_{j=k}\bar{M}_{il}$.

— Si $i = j$ et $k = l$ alors on obtient $M_{ii}M_{kk} + M_{kk}M_{ii} = 2\delta_{i=k}M_{ii}$.

— Si $i = j$ et $k \neq l$ alors on obtient $M_{ii}M_{kl} + M_{ii}M_{lk} + M_{kl}M_{ii} + M_{lk}M_{ii}$.

— Si $i \neq j$ et $k = l$ alors on obtient $M_{ij}M_{kk} + M_{ji}M_{kk} + M_{kk}M_{ij} + M_{kk}M_{ji}$.

On peut alors reprendre la distinction de cas faite dans l'énoncé :

— Si $i = k$ et $j \neq l$ alors on obtient dans le premier cas ($i \neq j$ et $k \neq l$) $M_{jl} + M_{lj} = \bar{M}_{jl}$, le second cas ($i = j$ et $k = l$) est impossible, dans le troisième cas ($i = j$ et $k \neq l$), on obtient $\bar{M}_{il} = \bar{M}_{jl}$ et dans le quatrième cas ($i \neq j$ et $k = l$), on obtient $\bar{M}_{jk} = \bar{M}_{jl}$.

— Si $i = k$, $j = l$ et $i \neq j$ alors on est forcément dans le premier cas et on obtient $M_{jl} + M_{ik} + M_{ki} + M_{lj} = 2M_{jj} + 2M_{ii} = 2\bar{M}_{jj} + 2\bar{M}_{ii}$.

— Si $i = j = k = l$ alors on a déjà vu qu'on obtient $2\bar{M}_i i$.

— Enfin, si $\{i, j\} \cap \{k, l\} = \varnothing$ alors dans chacun des quatre cas tous les termes sont nuls et la somme est bien nulle.

36. On peut faire une preuve par récurrence sur k avec une initialisation avec $k = 1$ qui est valide car $\bar{M}_{ij} \neq 0$ et qu'un seul domino forme toujours une chaine.

37.

38. Soit k fixé. On peut décomposer les manières de procéder comme l'union disjointe des manières de construire la chaine en positionnant en premier le $i$eme domino pour i allant de 1 à k. Ce nombre (à i fixé) vaut $\binom{k-1}{i-1}$ car correspond à choisir une liste de longueur $k-1$ comprenant deux valeurs : "avant" et "arrière" permettant de déterminer dans quel ordre on positionne les différents voisins. On obtient alors : $N(k) = \sum_{i=1}^{k}\binom{k-1}{i-1} = \sum_{i=0}^{k-1}\binom{k-1}{i} = 2^{k-1}$ d'après la formule du binome de Newton.

39.

40.

41.

42.

43.