french

hyperrefOption 'colorlinks' set 'true' hyperrefHyper figures OFFhyperrefLink nesting OFFhyperrefHyper index ONhyperrefPlain pages OFFhyperrefBackreferencing OFF hyperrefImplicit mode ON; LaTeX internals redefined hyperrefBookmarks ON

hyperrefHyper figures OFFhyperrefLink nesting OFFhyperrefHyper index ONhyperrefbackreferencing OFFhyperrefLink coloring ONhyperrefLink coloring with OCG OFFhyperrefPDF/A mode OFF

hyperrefDriver (autodetected): hluatex

rerunfilecheckFeature "pdfmdfivesum is not available(e.g. pdfTeX or LuaTeX with package 'pdftexcmds').Therefore file contents cannot be checked efficientlyand the loading of the package is aborted

automata, positioning, arrows, shapes.geometric matrix

[mic]cstyle=friendly, breaklines=true, autogobble [mio]ocamlstyle=friendly, breaklines=true, autogobble [miq]sqlstyle=friendly, breaklines=true, autogobble [mib]bashstyle=friendly, breaklines=true, autogobble

InEntrée OutSortie Returnrenvoyer IfElifElsesialorssinon sisinon SwitchCaseDefaultdistinguer seloncas oùautres cas Forpour Whiletant que

- 2mm stylelegende

normaltext

Sélection de sujets posés lors de la session 2024

Exercices de type A

Exercice 1 Une relation d'équivalence (type A)
On fixe $n \in \mathbb{N}^\star$. Soient $\varphi : 1, n \to 1, n$ et $\psi : 1, n \to 1, n$. Soit u et v deux éléments de $1, n$. On dit que u et v sont (φ, ψ)-équivalents s'il existe $k \in \mathbb{N}$, un tuple $(w_0, w_1, \cdots, w_{k+1}) \in 1, n^{k+2}$ avec $w_0 = u, w_{k+1} = v$ et vérifiant :

$$\forall i \in 0, k, \varphi(w_i) = \varphi(w_{i+1}) \text{ ou } \psi(w_i) = \psi(w_{i+1}).$$

L'objectif est d'écrire un algorithme en pseudo-code permettant de calculer les différentes classes d'équivalence engendrées par cette relation.

1. Justifier rapidement que "être (φ, ψ)-équivalent" est une relation d'équivalence sur l'ensemble $1, n$.

2. Pour cette question, on considère les applications φ et ψ définies par :

$i =$	1	2	3	4	5	6	7	8	9
$\varphi(i) =$	3	2	2	9	6	4	9	5	7
$\psi(i) =$	5	1	3	4	5	1	7	7	4

   Calculer les différentes classes d'équivalence.

3. On revient au cas au général. On définit le graphe $G = (S, A)$ par :

$$S = 1, n, A = \{(x, y) \in S^2 | x \neq y \text{ et } (\varphi(x) = \varphi(y) \text{ ou } \psi(x) = \psi(y))\}.$$

   On fixe x et y deux sommets différents de S. Traduire sur le graphe G le fait que les sommets x et y sont (φ, ψ)-équivalents et en déduire que le calcul des classes d'équivalence de G se traduit en un problème classique sur les graphes que l'on précisera.

4. Donner en pseudo-code un algorithme permettant de résoudre le problème correspondant sur les graphes.

On fixe n un nombre pair. On considère deux applications φ et ψ de $1, n$ où tout élément de l'image de φ admet exactement deux antécédents par φ et où tout élément de l'image de ψ admet exactement deux antécédents par ψ.
Pour $f \in \{\varphi, \psi\}$, on note G_f le graphe $(S, \{(x, y) \in S^2 | x \neq y \text{ et } f(x) = f(y)\})$.

[resume]Préciser la forme du graphe G_f pour $f \in \{\varphi, \psi\}$. Expliciter la forme des différentes classes d'équivalence dans le graphe G correspondant.

Proposition de corrigé

1. La réflexivité correspond au cas $k = 0$, la symétrie consiste à réindexer à l'envers, la transitivité consiste à mettre bout à bout les deux séquences.

2. On trouve comme classes : $C_1 = \{1, 5\}, C_2 = \{2, 3, 6\}, C_3 = \{4, 7, 8, 9\}$.

3. Il existe alors un chemin commençant par x et terminant par y. Sachant que le graphe est en réalité symétrique, calculer les classes d'équivalence revient alors à calculer les composantes connexes du graphe correspondant (on peut aussi calculer les composantes fortement connexes si on n'a pas remarqué la symétrie).

4. Dans le cas où on remarque la symétrie : un parcours du graphe suffit (n'importe lequel).

   0.6cm [H] compComposantes_connexes parcoursparcours_largeur graphe G tableau numComp avec numComp[i] égal au numéro d'une composante connexe numComp = [0, 0, ..., 0] (indexation de 1 à n inclus)
   i,num file = [i] file non vide a = defiler(file) numComp[a] vaut 0 numComp[a] := num
   x voisin de a enfiler(x)

   num = 0 i = 1 à n numComp[i] = 0 num := num + 1
   (i,num)

   numComp

   Dans le cas où la symétrie n'est pas remarquée, on peut utiliser un algorithme de calcul des composantes fortement connexes, comme l'algorithme de Kosaraju.

5. On reconnaît un graphe biparti où chaque sommet est de degré 1 (c'est le graphe induit par un couplage parfait).

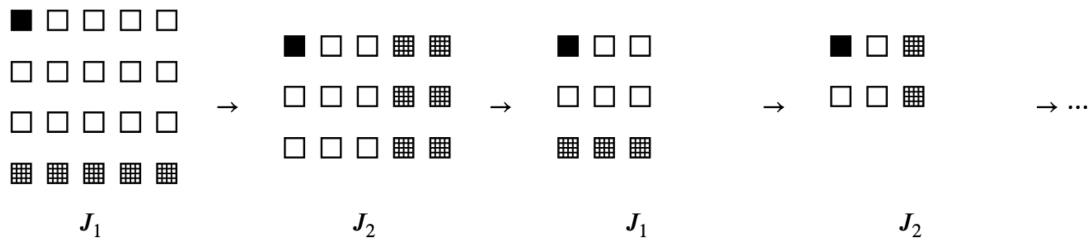
6. On obtient deux types de composantes connexes :

   - composantes à deux sommets (même image par φ et ψ)
   - des cycles ayant un nombre pair de sommets (on peut remarquer qu'il y a alternance entre arêtes créees par φ et créees par ψ).

   On pourra remarquer qu'un sommet est soit d'arité 1, soit d'arité 2. Lorsqu'il est d'arité 1, cela signifie que son voisin et lui ont même image par φ et ψ. Donc ces deux sommets constituent une composante connexe.

   Dans le cas où un sommet est d'arité 2, par la contraposition de la remarque précédente, tous les sommets dans sa composante connexe sont d'arité 2. Ainsi, on a uniquement des cycles ayant un nombre pair de sommets.

Exercice 2 Jeu de Chomp (type A)
On considère une variante du jeu de Chomp : deux joueurs J_1 et J_2 s'affrontent autour d'une tablette de chocolat de taille $l \times c$, dont le carré en haut à gauche est empoisonné. Les joueurs choisissent chacun leur tour une ou plusieurs lignes (ou une ou plusieurs colonnes) partant du bas (respectivement de la droite) et mangent les carrés correspondants. Il est interdit de manger le carré empoisonné et le perdant est le joueur qui ne peut plus jouer.
Dans la figure ci-dessous, matérialisant un début de partie sur une tablette de taille 4×5, le carré noir est le carré empoisonné, le choix du joueur J_i est d'abord matérialisé par des carrés hachurés, qui sont ensuite supprimés.



On associe à ce jeu un graphe orienté $G = (S, A)$. Les sommets S sont les états possibles de la tablette de chocolat, définis par un couple $s = (m, n), m \in [\![1, l]\!],\ n \in [\![1, c]\!]$. De plus, $(s_i, s_j) \in A$ si un des joueurs peut, par son choix de jeu, faire passer la tablette de l'état s_i à l'état s_j. On dit que s_j est un successeur de s_i et que s_i est un prédécesseur de s_j.

1. Dessiner le graphe G pour $l = 2$ et $c = 3$. Les états de G pourront être représentés par des dessins de tablettes plutôt que par des couples (m, n).

On va chercher à obtenir une stratégie gagnante pour le joueur J_1 par deux manières.

Utilisation des noyaux de graphe
Soit $G = (S, A)$ un graphe orienté. On dit que $N \subset S$ est un noyau de G si :

- pour tout sommet $s \in N$, les successeurs de s ne sont pas dans N,
- tout sommet $s \in S \setminus N$ possède au moins un successeur dans N

[resume]Donner tous les noyaux possibles pour les graphes suivants :

1.

[scale=0.7, every node/.style=transform shape] [shape=circle,draw=black] (A) at (1,2) s_1; [shape=circle,d

Dans la suite, on ne considère que des graphes acycliques.

2. Montrer que tout graphe acyclique admet un puits, c'est-à-dire un sommet sans successeur.

   Dans le cas général, le noyau dun graphe $G = (S, A)$ est souvent difficile à calculer. Si la dimension du jeu nest pas trop importante, on peut toutefois le faire en utilisant l'algorithme suivant :

   0.6cm [H] $N = \emptyset$ il reste des sommets à traiter Chercher un sommet $s \in S$ sans successeur $N = N \bigcup \{s\}$ Supprimer s de G ainsi que ses prédécesseurs

3. Justifier que cet algorithme termine et renvoie un noyau.
4. Démontrer que ce noyau est unique. Conclure que le graphe du jeu de Chomp possède un unique noyau N.
5. Appliquer cet algorithme pour calculer le noyau du jeu de Chomp à 2 lignes et 3 colonnes. Que peut-on dire du sommet (1,1) pour le joueur qui doit jouer ? En déduire à quoi correspondent les éléments du noyau.

6. Montrer que, dans le cas d'un graphe acyclique, tout joueur dont la position initiale nest pas dans le noyau a une stratégie gagnante. Le joueur J_1 a-t-il une stratégie gagnante pour ce jeu dans le cas $l = 2$ et $c = 3$?

Utilisation des attracteurs
On modélise le jeu par un graphe biparti : pour ce faire, on dédouble les sommets du graphe précédent : un sommet s_i génère donc deux sommets s_i^1 et s_i^2, s_i^j étant le sommet i contrôlé par le joueur J_j. On forme alors deux ensembles de sommets $S_1 = \{s_i^1\}_i$ et $S_2 = \{s_i^2\}_i$, et on construit le graphe de jeu orienté $G = (S, A)$ avec $S = S_1 \bigcup S_2$ et $S_1 \bigcap S_2 = \emptyset$. De plus, $(s_i^1, s_j^2) \in A$ si le joueur 1 peut, par son choix de jeu, faire passer la tablette de l'état s_i^1 à l'état s_j^2. On raisonne de même pour $(s_i^2, s_j^1) \in A$.
On rappelle la définition d'un attracteur : soit F_1 lensemble des positions finales gagnantes pour J_1. On définit alors la suite d'ensembles $(\mathcal{A}_i)_{i\in\mathbb{N}}$ par récurrence : $\mathcal{A}_0 = F_1$ et

$$(\forall i \in \mathbb{N})\ \mathcal{A}_{i+1} = \mathcal{A}_i \cup \{s \in S_1 / \exists t \in \mathcal{A}_i, (s,t) \in A\} \cup \{s \in S_2 \text{ non terminal}, \forall t \in S, (s,t) \in A \Rightarrow t \in \mathcal{A}_i\}$$

et $\mathcal{A} = \bigcup_{i=0}^{\infty} \mathcal{A}_i$ est l'attracteur pour J_1.

[resume]Que représente l'ensemble $\mathcal{A}_i$? Dans le cas du jeu de Chomp à deux lignes et trois colonnes (question 1), calculer les ensembles $\mathcal{A}_i$. Le joueur J_1 a-t-il une stratégie gagnante ? Comment le savoir à partir de $\mathcal{A}$?

Proposition de corrigé

1. On obtient le graphe suivant (plus lisible avec des tablettes) :

[¿=latex, node distance = 1cm] [] (0) (2, 3); [] (1) [right = of 0] (2, 1); [] (2) [above = of 1] (2, 2); [] (3) [below = of 1] (1, 3); [] (4) [right = of 1] (1, 2); [] (5) [below = of 4] (1, 1);

```
-latex](0)
edge(1)(0)edge(2)(0)edge(3)(1)edge(5)(2)edge(1)(2)edge(4)(3)edge(4)(3)
edge(5)(4)edge(5);\end{tikzpicture}\end{center}\itemPourlepremiergraphe,
lenoyauest$(s_2,s_4,s_5)$.Pourlesecondilyadeuxnoyaux$(s_1,s_3)$et$(s_2,s_4)
$.\itemSoit$G$ungrapheacycliquesanspuits.Soit$(s_0\cdotss_k)$unchemindans$G$.
Comme$s_k$n'estpasunpuits,ilexisteunarc$(x_k,y)$quiprolongelecheminprécédent.
Ilyadoncunchemindelongueurarbitrairedans$G$.Parleprincipedestiroirs,
toutcheminassezlongcontientnécessairementdeuxsommetsquicoïncident,
onadoncuncycle,encontradictionavec$G$acyclique.
Doncilexisteaumoinsunpuitsdans$G$.\itemSupprimerdessommetsd'
ungrapheacycliquelelaisseacycliquedonclaquestion3assurequ'
ontrouveratoujoursunsommet$s$convenableenligne3.Parconséquent,
lenombredesommetsdans$G$décroîtstrictementàchaqueitérationcequiassurelaterminaisondel'
algorithme.Parailleurs,
unsommetsanssuccesseurdoitêtredansunnoyauàcausedeladeuxièmepropriétéetdonctoussesprédécess
Cesdeuxcontraintesnécessairessontgarantiesparl'algorithme.
Ellessontsuffisantesdanslecadred'
ungrapheacycliquecartoutsommetquidevraitnécessairementêtredansunnoyauetnécessairementnepas
uncycle.\itemTravaillonsparrécurrencefortesurlenombre$n$desommetsde$G$.Si$n=
1$ceseulsommet(puits)constituelenoyau.Sinon,soit$s$unpuitsde$G$(quiexisted'
aprèslaquestion3).Cesommetfaitnécessairementpartiedetouslesnoyauxde$G$.
Onnote$P(s)$l'ensembledesprédécesseursde$s$.
Legraphe$G$privéde$s$etdessommetsde$P(s)$resteacycliqueetdonc,
parhypothèsederécurrence,aunnoyauunique$N$.L'ensemble$N\bigcup\{s\
}$estunnoyaude$G$etparunicitéde$N$etnécessitédufaitque$s$soitdanstoutnoyau,c'
estleseul.\itemLessommets(2,2)et(1,1)constituentlenoyau.Laposition(1,1)
estperdantepourlejoueurquidoitjouer(ilmangelechocolatempoisonné)
,lesélémentsdunoyausontlessommetsperdantsetlesautressommetslespositionsgagnantes.\itemOnm
```

1$danslenoyaunepeutpasperdre.Si$s_1$napasdesuccesseur,
ladversairenepeutplusjouer,ilaperdu.Sinon,ladversairevachoisir$s_
2$danslessuccesseursde$s_1$.Onadonc$s_2\inS\setminusN$donc$s_
2$admetaumoinsunsuccesseurdans$N$.\item$\mathcalA_{i}$estl'
ensembledessommetsde$S$àpartirdesquelsJ_1peutforcerlapartieàarriveren$F_
1$enmoinsde$i$coups.\item$\mathcalA_0=\{S_6^2\},\mathcalA_1=\{S_6^2,S_2^1,S_
4^1\}=\mathcalA$.L'
attracteurcontientexactementtouteslespositionsgagnantespour$J_
1$.\end{enumerate}\newpage\section*{ExercicesdetypeB}\
exocommand{\texttt{HORNSAT}(typeB)}\preambuleocaml\
textit{Onattendunstyledeprogrammationfonctionnel.L'
utilisationdesfonctionsdumodule\mio{List}estautorisée;
celledesfonctionsdumodule\mio{Option}estinterdite.
}Uneformuleducalculpropositionnelestune\textit{formuledeHorn}s'ils'agitd'
uneformulesousformenormaleconjonctive(FNC)
danslaquellechaqueclause(éventuellementvide,
auquelcaslaclauseenquestionestladisjonctiond'
unensemblevidedelittérauxetestdoncsémantiquementéquivalenteà$\bot$)
contientauplusunlittéralpositif.Danslasuite,onconsidèrequ'uneclaused'
unetelleformulecontientauplusuneoccurrencedechaquevariable(enparticulier,
lesclausessontsansdoublons).\begin{enumerate}\
itemLesformulessuivantessont-ellesdesformulesdeHorn?\begin{enumerate}a)]

$$F_1 = (\neg x_0 \vee \neg x_1 \vee \neg x_3) \wedge (x_0 \vee \neg x_1) \wedge (\neg x_2 \vee x_0 \vee \neg x_3) \wedge (\neg x_0 \vee \neg x_3 \vee x_2) \wedge x_2 \wedge (\neg x_3 \vee \neg x_2).$$

$$F_2 = (x_0 \vee \neg x_1) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_0) \wedge \neg x_1 \wedge (x_1 \vee \neg x_1 \vee x_0) \wedge (\neg x_0 \vee x_2).$$

$$F_3 = (\neg x_1 \vee \neg x_4) \wedge x_1 \wedge (\neg x_0 \vee \neg x_3 \vee \neg x_4) \wedge (x_0 \vee \neg x_1) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_4 \vee \neg x_0 \vee \neg x_1).$$

On utilise le type suivant pour manipuler les formules de Horn : une formule de Horn est une liste de clauses de Horn ; une clause étant la donnée d'un int option valant None si la clause ne contient pas de littéral positif et Some i si x_i en est l'unique littéral positif et d'une liste d'entiers correspondants aux numéros des variables intervenant dans les littéraux négatifs.

roundcorner=2pt, topline=true, leftline=true, bottomline=true, rightline=true, innertopmargin=4pt, innerbottommargin=4pt, innerrightmargin=4pt, innerlinewidth=1pt, backgroundcolor=gray!10, linecolor=gray!40, linewidth=0.1pt, [] [style=friendly, escapeinside= , mathescape=true, numbersep=0pt, autogobble, breaklines=true, xleftmargin=-2mm, framesep=1.5mm, linenos=false]ocaml
type clause$_horn = int option * int list type formule_horn = clause_horn list$

[resume]Écrire une fonction avoir$_clause_vide : formule_horn -> bool qui renvoie true si et seulement si la formul$

On appelle *clause unitaire* une clause réduite à un littéral positif. Par ailleurs, *propager* une variable x_i dans une formule F sous FNC consiste à modifier F comme suit :

(a) • Toute clause de F qui ne fait pas intervenir la variable x_i est conservée telle quelle.

- Toute clause de F qui fait intervenir le littéral x_i est supprimée entièrement.
- On supprime le littéral $\neg x_i$ de toutes les clauses de F qui font intervenir ce littéral.

On souligne que supprimer $\neg x$ d'une clause C qui ne fait intervenir que ce littéral ne revient pas à supprimer la clause C. On s'intéresse à l'algorithme $\mathcal{A}$ suivant dont on admet (pour le moment) qu'il permet de déterminer si une formule de Horn F est satisfiable :

0.2cm [H] il y a une clause unitaire x_i dans F $F \leftarrow$ propager x_i
dans F F contient une clause vide faux vrai

[resume]À l'aide de cet algorithme déterminer si les formules de Horn de la question 1 sont satisfiables. On utilisera ces formules pour tester les fonctions implémentées aux questions suivantes. Écrire une fonction trouver$_{c}lause_{u}nitaire : formule_{h}orn- > intoptionrenvoyantNonesila$
est l'une des clauses unitaires sinon. Justifier que propager une variable dans une formule de Horn donne une formule de Horn. Écrire une fonction propager : formule$_{h}orn- > int- >$
$formule_{h}ornquiprendenentreuneformuledeHorn$F$etunentierietcalculelaformulersultatdelapropag$
dans F. Déduire des questions précédentes une fonction etre$_{s}atisfiable : formule_{h}orn- >$
$boolrenvoyanttruesietseulementsilaformuledeHornenentreestsatisfiable.Quelleestlacomplexitdevc$
$ondiredesproblmesdedcisionSATet$HORN-SAT$(dontladfinitionestlammequecelledeSAT$ceci$prsquelesfor$

(d) (e)

(f) Si F est une clause de Horn sans clause unitaire ni clause vide, donner une valuation simple qui satisfait F.

(g) On admet que si F est une formule de Horn faisant intervenir une clause unitaire x_i et F' est le résultat de la propagation de x_i dans F, alors que F est satisfiable si et seulement si F' est satisfiable. En déduire la correction de l'algorithme $\mathcal{A}$.

2. Expliquer comment on pourrait modifier les fonctions précédentes afin de déterminer une valuation satisfaisant une formule de Horn dans le cas où elle existe plutôt que de juste dire si elle est satisfiable ou non. On ne demande pas d'implémentation.

Proposition de corrigé

1. F_1 et F_3 sont des formules de Horn mais pas F_2, ce qui est implicitement suggéré par le code.

2. Avec notre modélisation, la clause vide est (None, []) d'où : roundcorner=2pt, topline=true, leftline=true, bottomline=true, rightline=true, innertopmargin=4pt, innerbottommargin=4pt, innerrightmargin=4pt, innerlinewidth=1pt, backgroundcolor=gray!10, linecolor=gray!40, linewidth=0.1pt, [] [style=friendly, escapeinside= , mathescape=true, numbersep=0pt, autogobble, breaklines=true, xleftmargin=-2mm, framesep=1.5mm, linenos=false]ocaml let avoir$_{c}lause_{v}ide(f : formule_{h}orn) : bool = List.mem (None, [])f$

3. La formule F_1 est satisfiable d'après cet algorithme mais pas F_3.

   Techniquement on peut arrêter les propagations dès qu'on produit une clause vide.

4. On propose : roundcorner=2pt, topline=true, leftline=true, bottomline=true, rightline=true, innertopmargin=4pt, innerbottommargin=4pt, innerrightmargin=4pt, innerlinewidth=1pt, backgroundcolor=gray!10, linecolor=gray!40, linewidth=0.1pt, [] [style=friendly, escapeinside= , mathescape=true, numbersep=0pt, autogobble, breaklines=true, xleftmargin=-2mm, framesep=1.5mm, linenos=false]ocaml let rec trouver$_{c}lause_{u}nitaire(f : formule_{h}orn) : intoption = matchfwith|[]- > None|(Somev, l) :: q- > ifl = []thenSomevelsetrouver_{c}lause_{u}nitaireq|_{:} : q- > trouver_{c}lause_{u}nitaireq$

5. La fonction auxiliaire enlever$_{n}egsupprimesielleexistelaseuleoccurrenced'unentierdansunelisted'entiers(on$

roundcorner=2pt, topline=true, leftline=true, bottomline=true, rightline=true, innertopmargin=4pt, innerbottommargin=4pt, innerrightmargin=4pt, innerlinewidth=1pt, backgroundcolor=gray!10, linecolor=gray!40, linewidth=0.1pt, [] [style=friendly, escapeinside= , mathescape=true, numbersep=0pt, autogobble, breaklines=true, xleftmargin=-2mm, framesep=1.5mm, linenos=false]ocaml let rec propager (f:formule$_h orn)(v : int) : formule_h orn = letrecenlever_n eg(l : intlist) : intlist = matchlwith|[]- > []|t :: qwhent = v- > q|t :: q- > t :: (enlever_n egq)inmatchfwith|[]- > []|(i,l) :: q- > matchiwith|Sometwhent = v- > propagerqv|_- > (i, enlever_n egl) :: (propagerqv)$

La formule résultant d'une propagation dans une formule de Horn reste une formule de Horn : c'est toujours une FNC et on ne fait que supprimer des clauses / littéraux (donc s'il y avait au plus un littéral positif par clause, en supprimant des choses cette propriété est conservée).

6. On applique l'algorithme donné par l'énoncé en imbriquant les fonctions précédentes : round-

   corner=2pt, topline=true, leftline=true, bottomline=true, rightline=true, innertopmargin=4pt, innerbottommargin=4pt, innerrightmargin=4pt, innerlinewidth=1pt, backgroundcolor=gray!10, linecolor=gray!40, linewidth=0.1pt, [] [style=friendly, escapeinside= , mathescape=true, numbersep=0pt, autogobble, breaklines=true, xleftmargin=-2mm, framesep=1.5mm, linenos=false]ocaml let rec etre$_s atisfiable(f : formule_h orn) : bool = match(trouver_c lause_u nitairef)with|None- > not(avoir_c lause_v idef)|Somev- > etre_s atisfiable(propagerfv)$

7. La fonction trouver$_c lause_u nitaireestlinaireennlatailledelaformuleenentre.Unepropagationsefaitaussiline$ $propagation" omestlenombredeclauses.$

   De plus, la recherche d'une clause vide dans une formule se fait linéairement en le nombre de clauses. On aboutit donc grossièrement (c'est-à-dire sans compter qu'au fil des propagations la taille de la formule réduit - de toutes façons ça ne changerait probablement pas grand chose) à une complexité pour etre$_s atisfiableen$O(mn + m) = O(n^2).

   On en déduit que `HORN-SAT` $\in$ P alors qu'on sait que `SAT` est NP-complet. Sous réserve que P $\neq$ NP, étudier la satifiabilité de formules de Horn est donc un problème bien plus facile que lorsqu'il n'y a pas d'hypothèse sur les formules.

   8. [a)]

      La valuation qui assigne faux à toutes les variables convient. En effet, s'il n'y a ni clause unitaire ni clause vide, toutes les clauses contiennent au moins une variable niée.

   (b) Si F est une formule de Horn, notons $\Pi(F)$ la formule de Horn obtenue après toutes les propragations successives de la boucle tant que dans l'algorithme $\mathcal{A}$. La propriété de l'énoncé permet de montrer par récurrence que F et $\Pi(F)$ sont équisatisfiables.

      Or $\Pi(F)$ est satisfiable si et seulement si cette formule de Horn ne contient pas la clause vide : le sens direct est immédiat puisque une FNC contenant une clause vide n'est jamais satisfiable (cette clause ne l'étant pas) ; le sens réciproque est fourni par la question 7a.

9. Le principe est de garder trace des variables que l'on propage : on les assigne toutes à vrai. Toutes les variables non propagées sont quant à elles assignées à faux.

Exercice 3 Mots de Dyck (type B)

Cet énoncé est accompagné dun ou plusieurs codes compagnons en C fournissant certaines des fonctions mentionnées dans lénoncé : il sont à compléter en y implémentant les fonctions demandées.
*La ligne de compilation gcc -o main.exe -Wall *.c -lm vous permet de créer un exécutable main.exe à partir du ou des fichiers C fournis. Vous pouvez également utiliser lutilitaire make. En ligne de commande, il suffit d'écrire make. Dans les deux cas, si la compilation réussit, le programme peut être exécuté avec la commande ./main.exe.*
*Il est possible d'activer davantage d'avertissements et un outil d'analyse de la gestion de la mémoire avec la ligne de compilation gcc -o main.exe -g -Wall -Wextra -fsanitize=address *.c -lm ou en écrivant make safe. Lexaminateur pourra vous demander de compiler avec ces options.*
Si vous désirez forcer la compilation de tous les fichiers, vous pouvez au préalable nettoyer le répertoire en faisant make clean et relancer une compilation.
La compilation du code compagnon initial avec make safe provoque des warnings attendus qui seront résolus lors de l'implémentation des fonctions demandées par le sujet.
On s'intéresse dans cet exercice aux mots de Dyck, c'est-à-dire aux mots bien parenthésés. Dans ce type de mots, toute parenthèse ouverte "(" est fermée ")" et une parenthèse ne peut être fermée si elle ne correspond pas à une parenthèse préalablement ouverte.
Par exemple pour deux couples de parenthèses, "(())" et "()()" sont des chaînes de parenthèses bien formées. "())(" et ")()(" ne le sont pas.
On admet que le nombre de mots bien parenthésés à n couples de parenthèses est donné par les nombres de Catalan définis par la formule suivante :

$$C_n = \frac{(2n)!}{(n+1)!n!} \text{ pour } n \geq 0$$

On rappelle que le type `uint64_t` est un type entier non signé codé sur 64 bits.

1. Complétez dans le code compagnon la fonction dont le prototype est `uint64_t catalan(int n)`. Vous pouvez utiliser une fonction auxiliaire si cela vous semble pertinent.

2. Que va-t-il se passer si on tente dafficher `catalan(n)` pour `n` un peu grand ? Le constatez-vous ici ?

On cherche maintenant à afficher le nombre de mots (chaînes) bien parenthésés avec n fixé couples de parenthèses, ainsi que les mots eux-mêmes.
Un algorithme de force brute pour déterminer toutes les chaînes à n couples de parenthèses bien formées consiste à générer toutes les possibilités puis à ne garder que les chaînes bien formées.

3. Complétez dans le code compagnon la fonction dont le prototype est `bool verification(char * mot)`. Cette fonction renvoie `true` si le mot fourni en paramètre `mot` est bien parenthésé, `false` sinon.

4. Quelle est la complexité de cette vérification ?

5. Quelle est la complexité finale de l'algorithme de force brute ?

On appelle `n` le nombre de couples de parenthèses voulu. Dans le fichier compagnon fourni, le nombre de couples a été limité à 18.
On vous propose de coder l'énumération des chaînes de parenthèses bien formées en appliquant l'algorithme de backtracking suivant, dont on admet qu'il est correct : on compte le nombre de parenthèses ouvertes `o` et le nombre de parenthèses fermées `f` dans une chaîne de caractères courante (vide au départ).

- Si `o = f = n`, on a trouvé une chaîne bien formée.
- Si `o < n`, on ajoute une parenthèse ouvrante et on relance.

- Si `f < o`, on ajoute une parenthèse fermante et on relance.

Cet algorithme est à implémenter dans la fonction dont le prototype est `void dyck(char s[N], int o, int f, int n)` qui affiche sur la sortie standard les chaînes de parenthèses bien formées avec `n` couples de parenthèses lorsque `s` est la chaîne de caractère courante, `o` est son nombre de parenthèses ouvrantes et `f` est son nombre de parenthèses fermantes.

6. Compléter la fonction `dyck` pour afficher les chaînes bien parenthésées avec 5 couples de parenthèses.

7. Adapter la fonction `dyck` pour calculer le nombre de mots obtenus. Combien de mots trouvez-vous pour 16 couples de parenthèses ?

8. Adapter la fonction `dyck` pour stocker les mots bien parenthésés dans une liste chaînée et les afficher après l'appel à la fonction. Vous trouverez dans le code compagnon une structure qui peut vous aider.

Proposition de corrigé

1. Une solution et d'implémenter une factorielle. On peut aussi simplifier la formule donnée (ce qui permet de repousser un peu plus loin le dépassement). roundcorner=2pt, topline=true, leftline=true, bottomline=true, rightline=true, innertopmargin=4pt, innerbottommargin=4pt, innerrightmargin=4pt, innerlinewidth=1pt, backgroundcolor=gray!10, linecolor=gray!40, linewidth=0.1pt, [] [style=friendly, escapeinside= , mathescape=true, numbersep=0pt, autogobble, breaklines=true, xleftmargin=-2mm, framesep=1.5mm, linenos=false]c uint64$_t fact(intn) uint64_t r = 1; for(inti =$
   uint64$_t catalan1(intn) return fact(2*n)/fact(n+1)/fact(n);$

2. On obtient un dépassement d'entier. Expérimentalement, si on utilise la fonction factorielle, le résultat devient faux pour $n = 11$. En simplifiant la formule en $2n \times ... \times (n + 2)/n!$, c'est pour $n = 16$ que ça coince. Dans tous les cas, le dépassement finira par arriver provoquant un comportement indéfini.

3. On utilise un compteur qui indique le nombre de parenthèses ouvertes : on l'incrémente lorsqu'on rencontre une parenthèse ouvrante et on le décrémente lorsqu'on rencontre une parenthèse fermante. Si ce compteur devient négatif, le mot n'est pas bien parenthésé puisqu'il y a trop de parenthèses fermantes et on peut donc interrompre l'exécution (même si on peut se passer de cette subtilité). En fin de décompte, le compteur devrait être nul si le mot est bien parenthésé. roundcorner=2pt, topline=true, leftline=true, bottomline=true, rightline=true, innertopmargin=4pt, innerbottommargin=4pt, innerrightmargin=4pt, innerlinewidth=1pt, backgroundcolor=gray!10, linecolor=gray!40, linewidth=0.1pt, [] [style=friendly, escapeinside= , mathescape=true, numbersep=0pt, autogobble, breaklines=true, xleftmargin=-2mm, framesep=1.5mm, linenos=false]c bool verification(char * mot) int i = 0; bool fin = false; int compteur = 0; bool resultat = false; while (!fin (mot[i] != '')) if (mot[i]=='(') compteur++; else if (mot[i] == ')') compteur = compteur - 1; if (compteur¡0) fin = true; i = i + 1; if (compteur == 0) resultat = true; return resultat;

4. On obtient un algorithme linéaire en la taille du mot en entrée.

5. Il y a 4^n mots à générer (puisqu'un mot à n couples de parenthèses possède $2n$ lettres). Pour chacun, la vérification de s'il est bien formé se fait en $O(2n) = O(n)$. À supposer que la construction des mots ne soit pas coûteuse, on obtient une complexité pour l'algorithme naïf en $O(4^n \times n)$.

6. Voici une proposition qui répond aux questions 6, 7 et 8.

roundcorner=2pt, topline=true, leftline=true, bottomline=true, rightline=true, innertopmargin=4pt, innerbottommargin=4pt, innerrightmargin=4pt, innerlinewidth=1pt, backgroundcolor=gray!10, linecolor=gray!40, linewidth=0.1pt, [] [style=friendly, escapeinside= , mathescape=true, numbersep=0pt, autogobble, breaklines=true, xleftmargin=-2mm, framesep=1.5mm, linenos=false]c
void dyck(char s[N], int o, int f, int n, uint64$_t{*}r, struct liste_s{**}liste) if (f == n) if (liste != NULL) struct liste$

7. Une façon de faire est d'ajouter un paramètre passé par pointeur pour compter ce nombre. On pourrait aussi utiliser une variable globale. On trouve 35357670 mots corrects pour $n = 16$.

8. Voir la proposition en question 6.