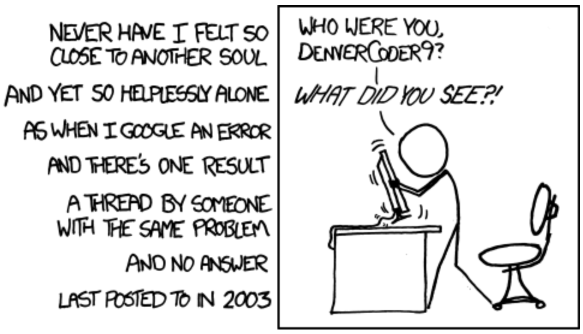
MPI* Info - TP3

Exercices CCINP et Automates en C-Grep



O. Caffier

1 Exercices CCINP

Exercice 1

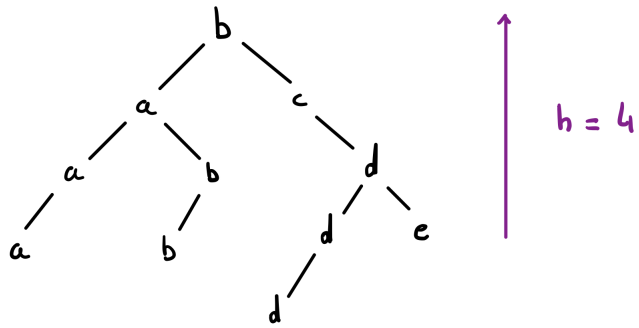
Dans cet exercice, on autorise les doublons dans un arbre binaire de recherche (ABR) et pour le cas d'égalité, on choisira le sous-arbre gauche. On ne cherche pos à équilibrer les arbres.

1. Rappeler la définition d'un ABR
   Corrigé :

   Définition - ABR

   Soient E un ensemble d'étiquettes ordonné par une relation $\leq_E$.
   Alors :
   - L'arbre vide est un ABR : $\perp$ est un ABR
   - Une feuille est un ABR : $\forall x \in E, \mathcal{N}(\perp, x, \perp)$ est un ABR
   - Soient $\mathcal{A}_1, \mathcal{A}_2$ deux ABR et $x \in E$, alors $\mathcal{N}(\mathcal{A}_1, x, \mathcal{A}_2)$ est un ABR ssi :
     - ➢ Pour toute étiquette $y \in \mathcal{A}_1$, on a $y \leq_E x$
     - ➢ Pour toute étiquette $z \in \mathcal{A}_2$, on a $x \leq_E z$

2. Insérer successivement et une à une dans un arbre binaire de recherche initialement vide toutes les lettres du mot *bacddabdbae*, en utilisant l'ordre alphabétique sur les lettres. Quelle est la hauteur de l'arbre ainsi obtenu ?
   Corrigé :



3. Montrer que le parcours en profondeur infixe d'un arbre binaire de recherche de lettres est un mot dont les lettres sont rangées dans l'ordre croissant. On pourra procéder par induction structurelle.
   Corrigé :
   - **Assertions** : arbre vide et feuilles → immédiat
   - **Règle d'inférence** : Soient $\mathcal{A}_1, \mathcal{A}_2$ deux ABR et $x \in E$ tels que $\mathcal{N}(\mathcal{A}_1, x, \mathcal{A}_2)$ est un ABR.
     Notons $p_1 = x_1, \cdots, x_n$ les étiquettes rencontrées lors du parcours infixe de $\mathcal{A}_1$.
     De même, notons $p_2 = x_{n+1}, \cdots, x_p$ celles rencontrées lors du parcours infixe de $\mathcal{A}_2$.
     On a bien, par hypothèse, les relations suivantes :

     $$x_1 \leq \cdots \leq x_n$$
     $$x_{n+1} \leq \cdots \leq x_p$$

     Enfin, comme $\mathcal{N}(\mathcal{A}_1, x, \mathcal{A}_2)$ est un ABR, on a $x_n \leq x$ et $x \leq x_{n+1}$.
     Un parcours infixe visitant d'abord les éléments du s.a.g, puis le noeud, puis les éléments du s.a.d, on a bien le résultat voulu.

4. Proposer un algorithme qui permet de compter le nombre d'occurences d'un élément x dans un ABR. Quelle est sa complexité ?
   Corrigé :

Algorithme

— si $\mathcal{A} = \mathcal{N}(\perp, y, \perp)$ est une feuille : renvoyer $\delta_{[y=x]}$

— Si $\mathcal{A} = \mathcal{N}(\mathcal{A}_{\}}, y, \perp)$:

  — si $y < x$: renvoyer 0

  — si $y \geq x$: renvoyer $\delta_{[y=x]}$ + critère du s.a.g (cf. en dessous)

— Si $\mathcal{A} = \mathcal{N}(\perp, y, \mathcal{A}_d)$:

  — si $y \leq x$: renvoyer $\delta_{[y=x]}$ + critère du s.a.d (cf. en dessous)

  — si $y > x$: renvoyer 0

— Si $\mathcal{A} = \mathcal{N}(\mathcal{A}_{\}}, y, \mathcal{A}_d)$:

  — $y_g \leftarrow$ plus grand élément de $\mathcal{A}_g$

  — $y_d \leftarrow$ plus petit élément de $\mathcal{A}_d$

  — si $y_g < x$:

    — si $y_d > x$: renvoyer $\delta_{[y=x]}$

    — sinon : renvoyer $\delta_{[y=x]}$ + `occurences`$(x, \mathcal{A}_d)$

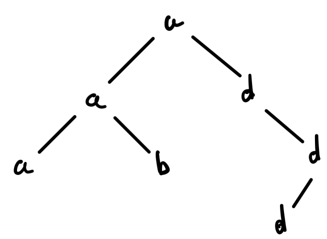
  — sinon si $y_d > x$: renvoyer $\delta_{[y=x]}$ + `occurences`$(x, \mathcal{A}_g)$

  — sinon : renvoyer $\delta_{[y=x]}$ + `occurences`$(x, \mathcal{A}_g)$ + `occurences`$(x, \mathcal{A}_d)$

On a, au maximum, $C(n) = 2C(\lfloor \frac{n}{2} \rfloor)$. D'où $C(2^n) = 2C(2^{n-1}) = 2^n C(1) = 2^n$.

D'où la complexité en $\mathcal{O}(n)$ avec n le nombre d'étiquettes dans l'ABR.

5. On souhaite supprimer une occurrence d'une lettre donnée d'un arbre binaire de recherche de lettres. Expliquer le principe d'un algorithme permettant de résoudre ce problème et le mettre en œuvre sur l'arbre obtenu à la question 2. en supprimant successivement une occurrence des lettres *e, b, b, c* et *d*. Quelle en est la complexité ?
Corrigé : Pour supprimer un élément, on effectue une recherche dans l'ABR en $\mathcal{O}(h)$, puis en supprimant l'étiquette concernée on prend celle du minimum du s.a.d (ou le maximum du s.a.g) qu'on supprime également de son emplacement pour pouvoir la mettre là où était placé le sommet à supprimer. Ces deux opérations sont en $\mathcal{O}(h)$, d'où la complexité recherchée.
Finalement, on obtient :



Exercice 2

Définition - Jeu

Un *jeu* est un triplet (S, A, s_0) où S est un ensemble fini d'états, $A \subset S^2$ est un ensemble de transitions et $s_0 \in S$ est un état initial tels que le graphe (S, A) est acyclique.

Définition - Stratégie

Soit (S, A, s_0) un jeu. Une *stratégie* est une fonction partielle $\varphi : S \to S$ telle que pour tout état s où elle est définie, $(s, \varphi(s)) \in A$. On peut faire jouer deux stratégies φ_0 et φ_1 l'une contre l'autre en les faisant jouer alternativement. On définit ainsi la séquence d'états $s_1 = \varphi_0(s_0), s_2 = \varphi_1(s_1), \cdots, s_k = \varphi_{(k-1) \bmod 2}(s_{k-1})$. On admet que la séquence (s_k) est finie.
Pour deux stratégies φ_0 et φ_1 jouées par les joueurs 0 et 1, le joueur perdant est le premier joueur pour lequel la stratégie n'est plus définie.

Définition - Stratégie gagnante

Une *stratégie gagnante* pour le joueur $i \in \{0, 1\}$ est une stratégie qui garantit que le joueur i gagne quand il joue suivant cette stratégie, quelle que soit la stratégie jouée par l'autre joueur.

1. Parmis les jeux suivants (où l'état initial est le sommet 0), quels sont ceux qui ont une stratégie gagnante pour le joueur 0, pour le joueur 1 ?
   Corrigé :
   - Pour le graphe 1 :
     - Joueur 0 : $\varphi_0(0) = 2$
     - Joueur 1 : même en jouant $\varphi_1(1) = 2$, le joueur 1 n'est pas garanti de gagner car le joueur dispose d'une stratégie gagnante.
   - Pour le graphe 2 :
     - Joueur 0 : même en suivant une stratégie quelle qu'elle soit, elle fera face à celle du joueur 1
     - Joueur 1 : $\varphi_1(1) = 2$
   - Pour le graphe 3 :
     - Joueur 0 : $\varphi_0(0) = 1, \varphi_0(2) = \varphi_0(3) = 4$
     - Joueur 1 : impossible

   En somme :

	Joueur 0	Joueur 1
Graphe 1	Oui	
Graphe 2		Oui
Graphe 3	Oui	

2. Faut-il libérer la mémoire allouée pour créer les jeux exemples, une fois l'exécution terminée ? Justifier.
   Corrigé : Non il ne faut pas libérer la mémoire allouée car il s'agit de variables globales, c'est donc de l'allocation statique.
3. Écrire une fonction `int nombre_coups_possibles(jeu J, int s)` qui prend en argument un jeu `J` et un état `s` et renvoie le nombre de coups possibles depuis l'état `s`.
   Corrigé :

```c
int nombre_coups_possibles(jeu J, int s){
    int ind = 0;
    while ((ind < J.taille) && (J.graphe[s][ind] != 100)){
        ind ++;
    }
    return ind;
}
```

4. Écrire une fonction `bool est_voisin(jeu J, int s, int t` telle que `est_voisin(J,s,t)` renvoie `true` si et seulement si `t` est un état accessible depuis `s` dans le jeu `J`.
Corrigé :

```
bool est_voisin(jeu J, int s, int t){
    int ind = 0;
    while ((ind < J.taille) && (J.graphe[s][ind] != 100)){
        if (J.graphe[s][ind] == t){
            return true;
        }
        else{
            ind++;
        }
    }
    return false;
}
```

Définition - Valeur de Grundy

Pour un jeu (S, A, s_0), on définit pour $s \in S$ sa *valeur de Grundy*, notée $G(s)$, par :
$$G(s) = \min\left(\mathbb{N} \setminus \{G(t) \mid (s,t) \in A\}\right)$$

5. Écrire une fonction `int plus_petit_absent(int* tab, int k)` qui prend en argument un pointeur vers un tableau d'entiers positifs ou nuls et un entier `k` et renvoie le plus petit entier naturel qui n'apparaît pas parmi les `k` premiers éléments du tableau.
Corrigé :

```
    int plus_petit_absent(int* tab, int k){

    bool* tmp_tab = malloc(k *sizeof(bool));
    for (int i = 0; i < k ; i++){
        tmp_tab[i]= true;
    }

    for (int id_tab = 0 ; id_tab < k ; id_tab++){
        if ((tab[id_tab] < k) && tab[id_tab] >= 0){
            int val = tab[id_tab];
            tmp_tab[val] = false;
        }
    }

    for (int entier = 0; entier < k ; entier++){
        if (tmp_tab[entier]){
            return entier;
        }
    }

    free(tmp_tab);
    return k;
}
```

6. Compléter la fonction `int nb_grundy(jeu J, int G[100], int s)` du fichier `grundy.c` en conséquence. Quelle est la complexité de la fonction `grundy` ?
Corrigé :

```
int nb_grundy(jeu J, int G[100], int s){
    /* Renvoie le nombre de Grundy du sommet s et modifie le tableau
     G en consequence pour que G[s] contienne la valeur renvoyee. */
    if (G[s] == -1){
        int k = nombre_coups_possibles(J,s);
        int ind_tmp =0;
        int* nb_grundy_utiles = malloc(k*sizeof(int));
        for (int t =0; t< J.taille; t++){
           if (est_voisin(J,s,t)){
                nb_grundy_utiles[ind_tmp] = nb_grundy(J,G,t);
                ind_tmp++;
           }
        }
        G[s] = plus_petit_absent(nb_grundy_utiles,k);
    }
    return G[s];
}
```

7. Montrer que le joueur 0 possède une stratégie gagnante dans le jeu $J = (S, A, s_0)$ si et seulement si $G(s_0) \neq 0$. **Corrigé :**
Soit $J = (S, A, s_0)$ un jeu. Supposons premièrement que le joueur 0 y possède une stratégie gagnante. Nous noterons φ_0 une telle stratégie.

Remarquons premièrement que pour toute "feuille" f (nœud sans descendant), alors $G(f) = 0$ par définition. Ainsi, tout père p d'une feuille est tel que $G(p) \geq 1$.

Il semble se profiler un lemme intermédiaire :

Lemme

Tout nœud n avec $G(n) = 0$ est perdant pour le joueur 1, et réciproquement.

(i.e J_1 perd si et seulement s'il se trouve sur un nœud avec $G(n) = 0$, ceci est également valable pour J_0).

En effet, par induction sur la profondeur du nœud (défini comme étant sa distance MAX à l'ensemble des feuilles) :
Ceci est évident pour une profondeur $k = 0$.

Si tout nœud n de profondeur $\leq k$ tel que $G(n) = 0$ est perdant pour 1, et réciproquement (H.R), alors soit n de profondeur $k+1$.

— Supposons que $G(n) = 0$, montrons que ce nœud est perdant pour J_1 :

  En particulier, chacun des fils de n doit avoir un nombre de Grundy différent de 0 par définition. Ainsi, pour toute stratégie, J_1 amène J_0 sur un nœud de profondeur $\leq k$, avec un nombre de Grundy plus grand strictement que 0 (par définition du nombre de Grundy). Par hypothèse de récurrence, J_1 perd car J_0 est capable d'envoyer J_1 sur un fils dont le nombre de Grundy est 0 (toujours par définition du nombre de Grundy), et cette position est par (H.R) perdante pour J_1. D'où ce sens.

— Réciproquement, si ce nœud est perdant pour J_1 : Alors peu importe le choix émis par J_1, J_0 sera capable d'envoyer J_1 sur une position perdante (J_0 possède en somme une stratégie gagnante sur ce nœud). Or, ceci aura pour effet de diminuer la profondeur du nœud considéré strictement, donc nous pouvons appliquer (H.R). Ainsi, pour tout fils de n, il existe un petit-fils p tel que J_1 perde sur p, donc tel que $G(p) = 0$ par H.R, ce qui implique que chaque fils de n possède un nombre de Grundy strictement plus grand que 0, donc que $G(n) = 0$.

  D'où l'équivalence souhaitée.

Ainsi, si J_0 possède une stratégie gagnante, alors $G(s_0) \neq 0$, car sinon tout fils de s_0 possède $G(p) \geq 1$, donc aucune de ces positions n'est perdante pour J_1, ceci est absurde par hypothèse.

Réciproquement, si $G(s_0) \neq 0$, alors il existe un fils p de s_0 avec $G(p) = 0$, ce qui est une position perdante pour J_1. Donc J_0 possède une stratégie gagnante qui consiste à envoyer J_1 sur un tel fils.

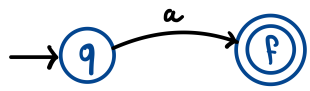
2 Automates en C-Grep

2.1 Construction de l'automate de Thompson

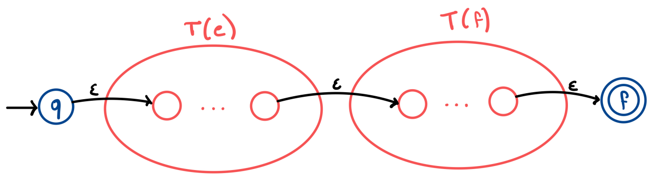
Question 1.

1. Rappeler, sous forme de schémas, les constructions de Thomson pour des expressions de la forme :

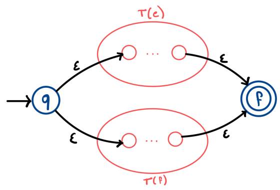
   (a) a où $a \in \Sigma$
   Corrigé :



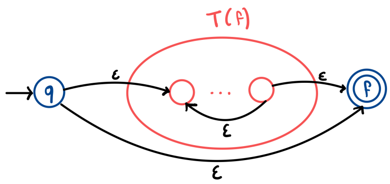
   (b) ef où e et f sont des expressions régulières
   Corrigé :



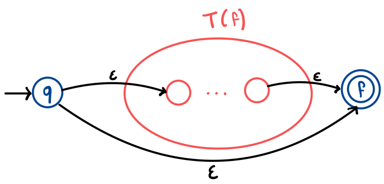
   (c) $e|f$ où e et f sont des expressions régulières
   Corrigé :



   (d) $e*$ où e est une expression régulière
   Corrigé :



2. Proposer une construction pour e? où e est une expression régulière et le ? signifie "zéro ou une fois".
   Corrigé :



3. Combien de transitions sortantes étiquetées par une lettre un état de l'automate de Thompson peut-il posséder ? Et combien de transitions sortantes étiquetées par ε ? Peut-il posséder à la fois les deux types de transition sortante ?

4. Justifier que si la représentation postfixe de e est de longueur n (en tant que chaîne de caractères), alors l'automate de Thompson associé possède au plus $2n$ états.
   Corrigé : On remarque que chaque opération effectuée par l'algorithme de construction ne crée que au plus 2 états par rapport aux automates précédemment créés. On en conclut que l'automate associé au langage entré en paramètre comprendra au maximum $2n$ états.

```
truct state {
int c;
struct state *out1;
struct state *out2;
int last_set;
};

typedef struct state state_t;

struct nfa {
state_t *start;
state_t *final;
int n;
};

typedef struct nfa nfa_t;
```

Question 2.

1. Écrire la fonction `new_state` renvoyant un pointeur vers un nouvel état (alloué sur le tas). Comme dit plus haut, on initialisera `last_set` à `-1`.

   Corrigé :

```
state_t *new_state(int c, state_t *out1, state_t *out2){
state_t* res = malloc(sizeof(state_t));
res->c=c;
res->out1=out1;
res->out2=out2;
res->last_set=-1;
return res;
}
```

2. Écrire la fonction `character` qui renvoie un `nfa_t` reconnaissant le caractère donné. Attention, on renvoie bien un `nfa_t`, par valeur, et pas un `nfa_t*`.
   Corrigé :

```
nfa_t character(int c){
state_t* qf = new_state(MATCH,NULL,NULL);
state_t* qi = new_state(c,qf,NULL);

nfa_t res;
res.start = qi;
res.final = qf;
res.n = 2;

return res;
}
```

3. Écrire une fonction `all` qui renvoie un `nfa_t` reconnaissant n'importe quel mot de longueur 1. On utilisera le même automate que pour `character`, sauf que le champ `c` de l'état initial sera mis à la valeur `ALL`.
   Corrigé :

```
nfa_t all(void){
return character(ALL);
}
```

4. Écrire les fonctions `concat`, `alternative`, `star` et `maybe` qui correspondent aux différentes constructions du dernier exercice. Autrement dit, dans l'appel `concat(a, b)`, on suppose que `a` et `b` sont deux automates de Thompson, d'ensembles d'états disjoints, reconnaissant deux expressions régulières e et f, et l'on demande de renvoyer l'automate de Thompson pour ef.
   Corrigé :

```c
nfa_t maybe(nfa_t a){
    state_t* qi = new_state(EPS,a.start,a.final);

    nfa_t res;
    res.start=qi;
    res.final=a.final;
    res.n = a.n +1;

    return res;
}

nfa_t build(char *regex){
    int len = strlen(regex);
    int i = 0;
    stack_tt* p = stack_new(2*len);
    while(regex[i] != '\0'){
        if (regex[i]=='@'){
            // Concatenation
            nfa_t a = pop(p);
            nfa_t b = pop(p);
            nfa_t tmp = concat(b,a);
            push(p,tmp);
            i+=1;
        }
        else if (regex[i]=='*'){
            // Etoile de Kleene
            nfa_t a = pop(p);
            nfa_t tmp = star(a);
            push(p,tmp);
            i+=1;
        }
        else if (regex[i]=='|'){
            nfa_t a = pop(p);
            nfa_t b = pop(p);
            nfa_t tmp = alternative(a,b);
            push(p,tmp);
            i+=1;
        }
        else if (regex[i]=='.'){
            nfa_t tmp = all();
            push(p,tmp);
            i+=1;
        }
        else if (regex[i]=='?'){
            nfa_t a = pop(p);
            nfa_t tmp = maybe(a);
            push(p,tmp);
            i+=1;
        }
        else{
            nfa_t tmp = character(regex[i]);
            push(p,tmp);
            i+=1;
        }
    }
    nfa_t res = pop(p);
    stack_free(p);
    return res;
}
```

5. Déterminer la complexité temporelle de `build` en fonction de la longueur m de la chaîne donnant l'écriture postfixe de

l'expression régulière.

2.2 Exécution de l'automate

Question 4.

1. Proposer une fonction `bool backtrack(state_t *state, char *s)` qui renvoie `true` ssi la lecture du mot `s` depuis l'état pointé par `state` nous amène dans un état final. On considérera que le mot s'arrête au premier caractère nul ou `'\n'` rencontré, exclu.
   Corrigé :

```c
bool backtrack(state_t *state, char *s){
    if (state == NULL){
        return false;
    }
    else if (s[0]=='\0' || s[0]=='\n'){
        if (state->c==EPS){
            return backtrack(state->out1,s) || backtrack(state->out1,s);
        }
        else{
            return state->c == MATCH;
        }
    }
    else{
        if (state->c == s[0] || state->c == ALL){
             return backtrack(state->out1,&s[1]);
        }
        else if (state->c == EPS){
            return backtrack(state->out1,s) || backtrack(state->out2,s);
        }
        else{
            return false;
        }
    }
}

bool accept_backtrack(nfa_t a, FILE* in){
    return backtrack(a.start,s);
}
```

2. **Corrigé :**

```c
int main(int argc, char* argv[]){
    if (argc==2 || argc==3){
        FILE* f;
        char* regex = argv[1];
        nfa_t automate = build(regex);
        if (argc==2){
            f = stdin;
        }
        else{
            char* fichier = argv[2];
            f = fopen(fichier,"r");
        }

        match_stream_backtrack(automate,f);
        return 0;
    }
    else{
        return 1;
    }
}
```

Question 5.

1. Écrire une fonction `void add_state(set_t *set, state_t *s)`.

```c
void add_state(set_t *set, state_t *s){
    if (s!=NULL&& s->last_set!=set->id){
        set->states[set->length]=s;
        set->length++;
        s->last_set=set->id;
        if (s->c==EPS){
            add_state(set,s->out1);
            add_state(set,s->out2);
        }
    }
}
```

2. Écrire une fonction `void step(set_t *old_set, char c, set_t *new_set)`.

```c
void step(set_t *old_set, char c, set_t *new_set){
    new_set->length=0;
    new_set->id=old_set->id + 1;
    for (int i=0;i<old_set->length;i++){
        if(old_set->states[i]->c==c||old_set->states[i]->c==ALL){
            add_state(new_set,old_set->states[i]->out1);
        }

    }

}
```

3. Déterminer la complexité en temps en fonction de la fonction `step`.
4. Écrire une fonction `bool accept(nfa_t a, char *s, set_t *s1, set_t *s2)` qui prend en entrée un automate et une chaîne et renvoie `true` ou `false` suivant que le mot (la chaîne) est reconnu ou non. À nouveau, on considérera que le mot se termne juste avant le premier caractère `'\n'` ou `'\0'` rencontré.

```c
bool accept_rec(nfa_t a, char* s, set_t* s1, set_t* s2){
    if (s[0]=='\n' || s[0]=='\0'){
        return a.final->last_set == s1->id;


    }
    else{
        step(s1,s[0],s2);
        accept_rec(a,&s[1],s2,s1);
    }
}

bool accept(nfa_t a, char* s, set_t *s1, set_t *s2){
    s1->length =0;
    add_state(s1,a.start);
    return accept_rec(a,s,s1,s2);
}
```