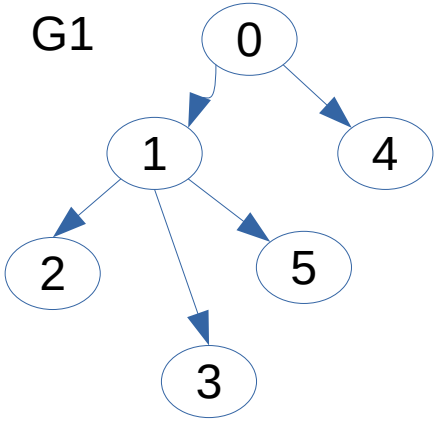
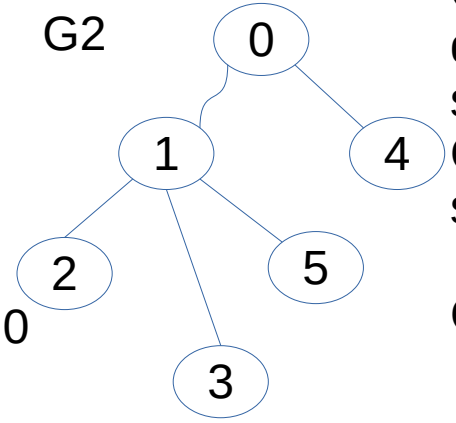
G1



G1 c’est une arborescence

G1 c’est l’orientation de l’arbre G2 (avec 0 comme racine)

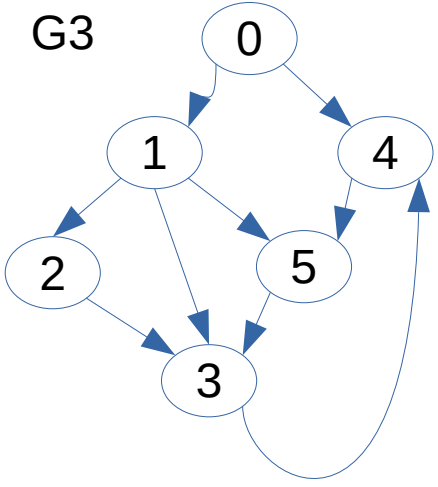
G2



G2 c’est une arbre qui a pour racine le sommet 0
G2 connexe (car un seul tenant)

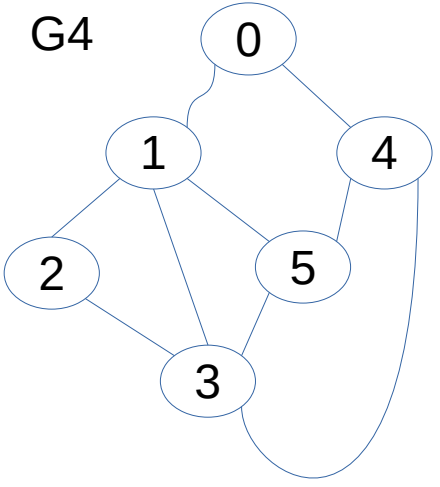
G2 est sans cycle

G3



G3 graphe orienté avec un unique circuit

G4



G4 est non orienté, connexe avec plusieurs cycles :

- (3,4,5)
- (1,3,5)
- (0,1,5,4)

Parcours de graphes

- 1.3 :
  Parcours1 : 0 4 1 5 3 2
  Parcours1 avec G2 donne une boucle infinie (la fonction boucle sur "0 4")
- 1.4 :
  On ajoute en paramètre (int u) de la fonction le sommet courant et on s'empêche de revisiter ce sommet :
  if(v!=u) aff_parcours2(G,v,r) ;
  Parcours2 : 0 4 1 5 3 2 → affiche un arbre

Parcours de graphes

- 1.5 :
  Parcours2 : 0 4 5 3 2 1 5 3 2 1.... (G4)
  Parcours2 : 0 4 5 3 4 5 3 .... (G3)
  On peut donc aussi avoir des problèmes de boucles infinies avec des graphes orientés

Parcours de graphes

- 1.6 : On utilise un tableau de marquage

```
void aff_parcours3(GrapheSimple* G, int r, int* visit) {
    visit[r]=1 ;
    printf("%d ", r);
    Cellule* cour = G→tabS[r];
    while( cour != NULL ) {
            int v = cour→v;
                    if(visit[ ]==0) aff_parcours3(G,v,visit);
            cour = cour→suiv;
    }
}
```

Parcours3: 0 4 5 3 1 2

Ce code permet de trouver un sous-parcours dans un graphe orienté quelconque

Parcours de graphes

- 1.7 : On utilise un tableau de marquage

```
int detecte_circuit_desc_r(GrapheSimple* G, int r, int* visit) {
    int detect=0;
    visit[r]=1;
    Cellule cour= G->tabS[r];
    while( cour!=NULL && !detect) {
        int v = cour->v;
        if(visit[v] == 0) detect = detecte_circuit_desc_r(G,v,visit);
        else if(visit[v] == 1) detect = 1;

        cour = cour->suiv;
    }
    visit[r] = 2;
    return detect;
}
```

→ Un arc (k,l) correspondant à la détection d'un circuit est détecté si on trouve un arc (r,v) avec v de statut toujours 1

Parcours de graphes

- 1.8 : On utilise un tableau de marquage

```
int detecte_circuit(GrapheSimple* G) {
    int* visit = (int*)malloc(G->nbsom * sizeof(int));
    int r=0;
    int detect=0;
    for(int i=0;i<G->nbsom;i++) {
        visit[i]=0;
    } // boucle for qui posait problème en TD !
    while( r<G->nbsom && !detect) {
        detect = detecte_circuit_desc_r(G, r, visit);
        while( r<G->nbsom && visit[r]!=0 ) r++;
    }
    return detect;
}
```

→ Pour détecter un circuit il faut relancer le parcours pour tout sommet non visité : en effet, un graphe orienté peut ne pas avoir de racine ou on ne connaît pas forcément cette racine (idem pour graphes non orientés non connexes)

Parcours de graphes

- 1.9 :

```
int trouve_circuit_desc_rec(GrapheSimple* G, int r, int* visit, int* pred, int *k, int* l)
{
    int detect=0;
    visit[r]=1;
    Cellule* cour = G->tabS[r];
    while(cour!=NULL && !detect){
        int v=cour->v;
        if(visit[v]==0){
            pred[v]=r;
            detect=trouve_circuit_desc_rec(G,v,visit,pred,k,l);
        } else if(visit[v]==1){
            detect=1;
            *k=r;
            *l=v;
        }
        cour=cour->suiv;
    }
    visit[r]=2;
    return detect;
}
```

```
int trouve_circuit(GrapheSimple* G, int* pred, int* k, int* l) {
    int* visit= malloc(...);
    int r=0;
    int detect=0;
    for(int i=0;i<G->nbsom;i++) visit[i]=0;
    while(r<G->nbsom && !detect) {
      detect=trouve_circuit_desc_rec(G,r,visit,pred,k,l);
      while(r<G->nbsom && visit[r]!=0) r++;
    }
    return detect;
}
```

Parcours de graphes

- 1.10 :

```
void aff_circuit(int* pred, int k, int l) {
   printf("(%d,%d) ",k,l);
   int i=k;
   while(pred[i]!=1){
       printf("(%d,%d) ",pred[i],i);
       i = pred[i];
   }
   printf("(%d,%d)\n",l,i);
}
```

- Pour l'avoir à l'endroit : liste chaînée avec insertion en tête

Parcours de graphes

- 1.11 :

  cycle pour G2: (3,4) (5,3) (4,5)
  cycle pour G4: (2,3) (3,2)

  Pour G4, il détecte les arêtes comme étant des circuits de taille 2. Il faut conserver le sommet d'où l'on vient par paramètre et ajouter un test (comme précédemment)

CFC

- 2.1:

```
void creeGraphe(Graphe* G, int n) {
    G->nbsom=n;
    G->t_som= malloc(...);
    for(int i=0;i < g->nbsom;i++) {
        G->t_som[i].u=i;
        G->t_som[i].L-succ=NULL;
        G->t_som[i].L_prec=NULL;
    }
}
void ajoutArc(Graphe* G, int i, int j) {
    Arc *a = malloc(...);
    a->v=j;
    a->suiv=G->t_som[i].L_succ;
    G->t_som[i].L_succ=a;

    *a = malloc(...);
    a->v=i;
    a->suiv=G->t_som[j].L_prec;
    G->t_som[j].L_prec=a;
}
```

CFC

- 2.2 : En utilisant un algorithme de (sous)parcours enraciné en k, par exemple un parcours en largeur. Sur la figure, un parcours en largeur peut donner alors (1,0,3,4,2,7,5,6): on peut donc dire que tous les sommets du graphe sont au bout d'un chemin débutant en 1

CFC

- 2.3 :

```
void liste_descendants(Graphe* G, int r, int* marquage) {
    Arc* cour;
    int u,v;
    File F;
    initFile(&F);
    enfile(&F,r);
    while(!estFileVide(F)) {
        u=defile(&F);
        cour=G->t_som[u].L_succ;
        while(cour!=NULL){
            v=cour->v;
            if(marquage[v]==0){
                marquage[v]=1;
                enfile(&F,v);
            }
            cour=cour->suiv;
        }
    }
}
```

CFC

- 2.3 :

```
void CFC(Graphe* G, int* CFC) {
    int* descendants = malloc();
    int* ascendants = malloc();
    int r=0;
    while(r < G->nbsom) {
        for(int j=0;G->nbsom;j++){
            descendants[j]=0;
            ascendants[j]=0;
        }
        liste_ascendants(G,r,ascendants);
        liste_descendant(G,r,descendants);
        for(int j=0;G->nbsom;j++){
            if(descendants[j] && ascendants[j]) CFC[j]=r;
        }
        while( r < G->nbsom && CFC[r]!=-1) r++;
    }
}
```