– $f-(m+1)+1 = f - \left\lfloor \frac{d+f}{2} \right\rfloor \leqslant f - \frac{d+f-1}{2} = \frac{f-d+1}{2}$;

– $(m-1)-d+1 = \left\lfloor \frac{d+f}{2} \right\rfloor - d \leqslant \frac{d+f}{2} - d = \frac{f-d}{2} \leqslant \frac{f-d+1}{2}$.

Comme le corps de la boucle s'effectue en temps constant, on en déduit la complexité attendue.

Exercice 5

1. Il suffit de parcourir tous les sommets du graphe, vérifier s'ils sont dans la partie, et conserver le degré maximal.

```
int degre_max(graph* g, bool* partie) {
    int dmax = -1;
    for (int i=0; i<g->n; i++){
        if (partie[i]){
            if (g->degre[i] > dmax) { dmax = g->degre[i]; }
        }
    }
    return dmax;
}
```

2. On écrit une fonction auxiliaire qui fait un parcours en profondeur depuis un sommet s, en ayant accès au tableau des sommets déjà visités.

```
void dfs(graph* g, bool* vu, int s){
    if (!vu[s]){
        vu[s] = true;
        for (int i=0; i<g->degre[s]; i++){
            dfs(g, vu, g->voisins[s][i]);
        }
    }
}
```

Dès lors, on peut écrire la fonction demandée en créant un tableau de faux et en lançant un appel à la fonction précédente :

```
bool* accessibles(graph* g, int s) {
    bool* vu = malloc(g->n * sizeof(bool));
    for (int i=0; i<g->n; i++){
        vu[i] = false;
    }
    dfs(g, vu, s);
    return vu;
}
```

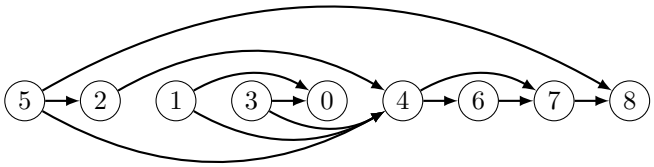
3. Il suffit de faire appel aux deux fonctions précédentes. On pense à libérer le tableau avant de renvoyer la valeur :

```
int degre_etoile(graph* g, int s) {
    bool* vu = accessibles(g, s);
    int d_etoile = degre_max(g, vu);
    free(vu);
    return d_etoile;
}
```

La fonction `degre_max` a une complexité linéaire en $|S|$ (simple boucle avec des opérations en

temps constant). La fonction `accessibles` a une complexité en $\mathcal{O}(|S|+|A|)$ (création du tableau + parcours du graphe). C'est également la complexité de la fonction `degre_etoile` qui enchaîne les deux fonctions.

4. On obtient par exemple le graphe :



5. On suppose un tableau DET de taille $n = |S|$ créé, contenant des -1. On commence par écrire une fonction récursive qui renvoie $d^*(s)$ pour un sommet s donné, et modifie $DET[s]$ pour qu'il contienne $d^*(s)$:
   - si $DET[s] = -1$:
     * si s est un puits (de degré sortant nul), poser $DET[s] = 0$;
     * sinon, lancer un appel récursif sur tous les voisins de s, puis poser $DET[s] = \max(d^+(s), \max\{d^+(t) \mid t \text{ voisin de } s\}$.
   - renvoyer $DET[s]$.

   Dès lors, on peut lancer un appel à cette fonction récursive depuis chaque sommet du graphe. Grâce à la mémoïsation des résultats, on s'assure pour chaque sommet de ne faire qu'un calcul supplémentaire linéaire en son degré. Cela garantit la complexité $\mathcal{O}(|S| + |A|)$.
6. On commence par remarquer que d^* est constant dans chaque composante fortement connexe du graphe (car les mêmes sommets sont accessibles). On peut alors calculer le graphe des composantes fortement connexes (qui est acyclique), et procéder de manière similaire à l'algorithme précédent, en posant, pour chaque composante fortement connexe, un d^* qui vaut le max entre le plus grand degré sortant maximal de la composante, et les d^* des composantes voisines.

Exercice 6

1. On peut utiliser des tableaux d'entiers pour les poids et les valeurs, et un tableau de booléens pour les indicateurs.
2. On crée un tableau de booléens. On utilise la variable `pmax` pour garder en mémoire le poids restant. On parcourt les objets, et si un objet rentre, on modifie son indicateur en conséquence et on modifie le poids restant.

```
bool* glouton(int* poids, int n, int pmax){
    bool* indic = malloc(n * sizeof(bool));
    for (int i=0; i<n; i++){
        if (poids[i] <= pmax){
            indic[i] = true;
            pmax -= poids[i];
        } else {
            indic[i] = false;
        }
    }
    return indic;
}
```

3. On écrit plusieurs fonctions intermédiaires pour simplifier le code :
   - une fonction qui calcule un total (de poids ou valeurs) selon un tableau d'indicateurs :