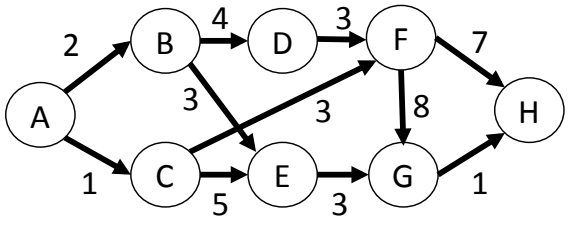
IFT 615 : Devoir 1
Travail individuel

Remise : 16 février 2012, 16h20 (**au plus tard**)

1. **[2 points]** Pour les 2 jeux suivants, déterminer les 6 caractéristiques de leur environnement :
   (a) Pong (`http://fr.wikipedia.org/wiki/Pong`),
   (b) Monopoly (`http://fr.wikipedia.org/wiki/Monopoly`).
   Justifiez brièvement votre choix pour chaque caractéristique.

2. **[2 points]** Soit le graphe suivant :



n	*h(n)*
A	7
B	5
C	4
D	7
E	4
F	2
G	0
H	0

   où le noeud de départ est A, le noeud de destination (but) est H, le nombre près de chaque arrête est le coût de la traverser et le tableau à gauche décrit l'heuristique à utiliser.
   (a) Simulez à la main l'exécution de A*, en donnant la valeur de *open* au début de chaque itération, ainsi que la solution retournée (voir acétate 21 du cours sur la recherche heuristique).
   (b) Est-ce que l'heuristique est admissible ? Pourquoi ?
   (c) Est-ce que l'heuristique est cohérente (*consistent*) ? Pourquoi ?

3. **[2 points]** Prouvez que, lors de la sortie d'un noeud n de *open* durant l'exécution de A*, la valeur de $f(n)$ est plus petite ou égale à la vraie valeur $f^*(n)$ du chemin optimal passant par n. Pour ce faire, faites une preuve séparée pour les deux cas possibles, soient :
   (a) n est déjà attaché à son chemin optimal (cas trivial),
   (b) n n'est pas encore attaché à son chemin optimal (conseil : faites une preuve par contradiction).

4. **[2 points]** Dans le contexte du problème de coloriage de la carte de l'Australie vu en cours, montrez comment l'algorithme AC-3 peut détecter que l'assignation partielle $WA{=}G$, $V{=}R$ n'est pas compatible avec les contraintes du problème.

5. **[2 points]** Programmez un joueur optimal de *tic-tac-toe* en implémentant l'algorithme de recherche alpha-beta. La recherche alpha-beta devra donc se faire jusqu'à la profondeur maximale.

   Le programme doit être dans le langage Python. Plus spécifiquement, vous devez remettre **un seul fichier nommé** `solution.py` contenant une fonction nommée `joueur_alpha_beta_tic_tac_toe` prenant 4 arguments (dans l'ordre) :

   – `etat` : objet représentant l'état actuel du jeu,
   – `transitions` : une fonction acceptant un état comme argument et donnant en sortie un dictionnaire qui associe chaque action possible (clé du dictionnaire) à l'état qui en résulte (valeur du dictionnaire),
   – `but` : une fonction d'utilité prenant un état en argument et qui retourne l'utilité associée à l'état (grand nombre positif si le joueur `'X'` gagne, grand nombre négatif s'il perd, 0 si c'est une partie nulle). Si l'état ne correspond pas à une partie terminée, `but` retourne `None`,
   – `joueur` : le rôle joué par ce joueur, c'est-à-dire `'X'` ou `'O'` (sous forme d'une chaîne de caractères).

   La fonction `joueur_alpha_beta_tic_tac_toe` doit retourner l'action prise par votre joueur. Cette action doit donc être une clé valide du dictionnaire retourné par `transitions`.

   Le script Python `devoir_1.py` importera la fonction `joueur_alpha_beta_tic_tac_toe` contenue dans `solution.py` (qui doit être dans le même répertoire) et l'utilisera pour simuler le joueur `'X'`. Pour tester votre code, vous pouvez jouer vous-même le joueur `'O'`, invoquer un joueur aléatoire ou utiliser la même fonction `joueur_alpha_beta_tic_tac_toe`. Voici comment utiliser `devoir_1.py` :

   ```
   Usage: python devoir_1.py jeu adversaire

   où "jeu" est "tic-tac-toe" ou "connect4"
   et "adversaire" est "humain", "soi-meme" ou "aleatoire".
   ```

   La correction de votre code se fera de façon automatique, en vérifiant automatiquement l'ensemble des états visités par votre recherche alpha-beta. Puisque l'ensemble des états visités dépend de l'ordre dans lequel les états successeurs sont générés, on vous demande de toujours utiliser l'ordre suggéré par Python en itérant sur les items du dictionnaire :

   ```
   for action,nouvel_etat in transitions(etat).items():
       # Votre code
       ...
   ```

   Votre fichier `solution.py` doit être remis via l'outil **turnin** avant la date limite de remise.

6. **[BONUS]** Implémenter également un joueur pour le jeu Connect 4 (Puissance 4 : `http://fr.wikipedia.org/wiki/Puissance_4`). Normalement, votre code pour le tic-tac-toe devrait également fonctionner pour Connect 4. Par contre, une recherche jusqu'à la profondeur maximale est trop lente dans le cas de Connect 4. Ainsi, vous devrez l'ajuster afin de limiter la profondeur de la recherche et développer une heuristique pour compenser. Plus votre heuristique sera bonne, plus les chances de gagner de votre joueur seront bonnes.

   Pour ce faire, ajoutez une fonction `joueur_alpha_beta_connect4` à votre fichier `solution.py`. Cette fonction doit prendre les mêmes arguments que `joueur_alpha_beta_tic_tac_toe`, en plus d'un cinquième argument :

   – `temps_maximal` : le temps maximal, en secondes, accordé au joueur pour retourner une action.

   Si le joueur prend plus de 50% du temps maximal accordé pour choisir une action, il sera éliminé et perdra par défaut. Le temps excédant pris par le joueur est également retranché au temps accordé à son tour suivant.

Pour tenir compte du temps écoulé, vous pouvez utiliser le module standard `time` et sa fonction `time()` qui retourne le temps (en secondes) écoulé depuis le 1er janvier 1970 (pour les systèmes Unix). Plus de détails sur le module `time` sont disponibles ici : `http://docs.python.org/library/time.html`.

Pour développer votre heuristique, vous pourrez évaluer la qualité d'un état à partir du champ `etat.tableau` de celui-ci. Ce champs est un tableau Numpy de caractères. Il contient l'information sur l'occupation de chaque case du jeu, à toutes les positions possibles : chaque entrée vaut donc `'X'`, `'O'` ou `' '` (pour une case non-occupée). Par exemple, l'état du jeu

```
| | | | | | | |
| | | | | | | |
| |X| | | | | |
| |X| |O| | | |
| |X| |O| | | |
| |X| |O| | |O|X|
-----------------
 0 1 2 3 4 5 6 7
```

correspond à la valeur de `etat.tableau` suivante :

```
[[' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' 'X' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' 'X' ' ' ' 'O' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' 'X' ' ' ' 'O' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' 'X' ' ' ' 'O' ' ' ' ' ' 'O' 'X']]
```

L'évaluation de votre solution se fera sous la forme d'un tournoi, où tous les joueurs soumis auront à joueur un contre l'autre. **Deux points boni** seront accordés aux 5 meilleurs joueurs du tournoi. **Un point boni** ira aux 5 suivants. Par contre, pour obtenir ces points, les joueurs devront également avoir fait mieux qu'un joueur (gardé secret) utilisant une heuristique très simple.