# Algorithmes de recherches
# TD à rendre

TD Basé sur les supports de Zdravko Markov (http://www.usna.edu/Users/cs/delooze/teaching/AI/Documents/day4.html).

## Consignes générales

Vous allez rendre ce travail à la date indiquée sur le serveur. Vous rendrez donc vos fichiers .pl commentés et un fichierr texte qui répondra aux questions qui ne demandent pas d'implémentation.

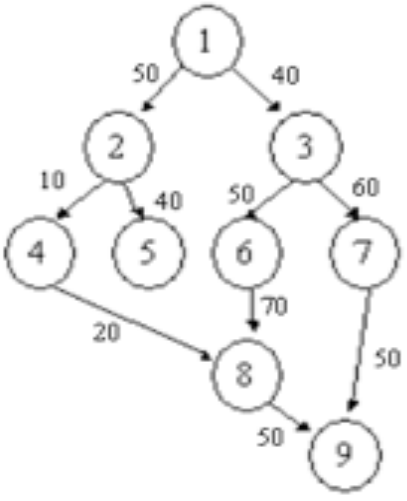
## 1. River Test (suite et fin)

1. Ajouter de nouveau prédicats basés sur les prédicats solution/3, resourdre/2 et resultat/2 pour gérer un nouveau paramètre : **la profondeur limite**.
2. A partir de quelle longueur a-t-on des solutions ? Le temps d'exécution apparent vous semble-t-il cohérent pour les valeurs suivantes de Lim : 5, 10, 12,17 ?
3. Ajouter ensuite le prédicat suivant pour mesurer ce temps de façon précise :

```
1: count(N,Lim) :-
2:  time(( findall( _,
3:         resoudre_avec_profondeur_limite( S, N, Lim ),
4:         L ),
5:  length( L, T ), write_ln(T) )).
```

4. Expliquez ligne par ligne ce que fait le prédicat ci-dessus.
5. Utilisez-le pour les valeurs suivantes de Lim : 5, 10, 12, 13. Donnez les résultats.
6. Que pensez-vous qu'il va se passer pour 17 ? Justifiez sans exécuter !

## 2. Algorithmes non informés




1. Créer un fichiers arcs.pl décrivant le graphe ci-contre (10 arêtes), par ex. : arc(1,2,50).
2. Récupérez et chargez dans prolog le fichier **search1.pl** sur celene.insa-cvl.fr. Analysez le code de la recherche en profondeur : `depth_first`. Comprendre notamment la gestion de pile pour la `Queue`.
3. Effectuez ensuite une recherche en profondeur du noeud 1 au 9 pour trouver un chemin :
   `?- depth_first([[1]],9,P,N).`
4. En se rappelant que la recherche en largeur utilise une file

au lieu d'une pile, compléter l'écriture de `breadth_first(+[[Start]], +Goal,-Path,-NbNodes)`.

5. Effectuez ensuite enfin une recherche de chemin en largeur du noeud 1 au 9 :
   `?- breadth_first([[1]],9,P,N).`
6. Ajoutez un arc entre le noeud 4 et le noeud 1 (cycle). Est-ce que cela change quelque chose pour la recherche en profondeur ? Et en largeur ? Pourquoi ?
7. Ces deux algorithmes trouvent-ils le plus court chemin ?
8. Afficher à l'écran (avec **writeln**) le contenu de **Path, Queue, NewPaths** et **NewQueue** pour évaluer l'encombrement mémoire de chaque algorithme (refaire alors les questions 3 et 4). De ce point de vue, pour ce graphe particulier, lequel des 2 est le plus efficace ?
9. Ecrire une famille de prédicats de démarrage **solve_x/5** qui renvoie le coût réel de chaque chemin solution. S'inspirer du prédicat **solve** vu en cours. Le **x** reflétera les prédicat de recherche appelé (sa première lettre).
   Par exemple, écrivez les prédicats :
   - **solve_d(+Start, +Goal, -SolPath, -ExploredNodes, -Cost)** pour la recherche en profondeur et le prédicat
   - **solve_b(+Start, +Goal, -SolPath, -ExploredNodes, -Cost)** pour la recherche en largeur.

   Utilisez-les et rediscutez éventuellement votre réponse de la question 7.
   Si besoin, débuggez pour voir encore plus précisément ce qui se passe en tapant la command «**trace.**» Pour stopper les traces, tapez «**notrace.**» puis « **nodebug** ».

## 3. Algorithmes informés

10. Récupérez et chargez dans prolog le fichier **map.pl** sur celene.insa-cvl.fr qui contient la carte de la Roumanie vue en cours. Comprendre le code. Expliquez la différence entre le cout apparaissant sur les arcs et celui dans straight_line_distance.
11. Cherchez le meilleur chemin de Arad à Bucharest, en ***profondeur*** d'abord puis en ***largeur***. Notez le coût réel, taille des chemins et nombre de noeuds explorés pour chaque parcours.
12. Récupérez et chargez dans prolog le fichier **search2.pl** sur celene.insa-cvl.fr. Regardez succinctement le code de :
    - *recherche meilleur d'abord* : `best_first`
    - *recherche A** : `a_star`
13. Modifiez `solve_b` et `solve_d` pour qu'ils fonctionnent à présent avec la carte de la Roumanie et écrivez les prédicats `solve_m` et `solve_a` pour lancer respectivement `a_star` et `best_first` sur le même modèle.
14. Modifier le code des fonctions heuristique de **search2.pl** pour pouvoir l'appliquer à la base de **map.pl**.
15. Cherchez le meilleur chemin de Arad à Bucharest, avec meilleur d'abord puis avec A*. Notez le coût réel, taille des chemins et nombre de noeuds explorés pour chaque parcours. Comparez avec les algorithmes non informés de la question 13.

## 4. Cout uniforme et profondeur itérative

16. Récupérez le fichier **blind-search2.pl.** Regardez et comprenez ce que fait le code.
17. Appliquer la recherche en coût uniforme au voyage en Roumanie. Comparez avec les algorithmes précédents.
18. Appliquer la recherche en profondeur itérative au voyage en Roumanie. Utilisez-le avec différentes de limite et comparez* avec les algorithmes précédents.

## 5. Beam-search ('recherche faiseau')

19. Récupérez et chargez dans prolog le fichier **beam-search.pl**. Regardez et comprenez ce que fait le code.
20. Appliquer beam-search au voyage en Roumanie avec les valeur de *BeamSize* suivantes : 100, 10, 1. Pourquoi ne peut-il trouver le meilleur chemin avec la valeur 1 ?
21. Expliquez son comportement. Comparez* avec les précédents.

## 6. Taquin (8-puzzle)

22. Récupérez et chargez dans prolog le fichier **8-puzzle.pl** sur celene.insa-cvl.fr. Regardez succinctement le code.
23. Faîtes en sorte de pouvoir appliquer les 4 algorithmes précédents au problème du 8-puzzle pour résoudre l'instance suivante du problème :
    *état initial* : **board(2,3,5,0,1,4,6,7,8)**
    =>
    *état final* : **board(0,1,2,3,4,5,6,7,8)**
24. Expliquez ce que calcule le prédicat distance actuel.
25. Ajoutez une nouvelle fonction heuristique dans **8-puzzle.pl** : la somme des distances de Manhattan (voir cours si besoin).

***NB : lorsque ce n'est pas explicité, une comparaison d'un algorithme avec les précédents nécessite de faire une évaluation mémoire et temporelle de l'exécution sur des choses comparables.**