Rapport TP - Algorithme A* - Application au Taquin

Balayssac Antoine, Rassat Théo

Semestre 2 - Année 2018/2019

Introduction :

Dans ce rapport, nous répondrons aux questions spécifiées dans le sujet, et donnerons aussi des observations et analyses qui nous ont semblé pertinentes.

I. Familiarisation avec le problème du Taquin 3x3

1.2.a

La clause finale qui permettrait de représenter la situation finale du Taquin 4x4 est :

```
final_state4x4([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12],
            [13, 14, 15, vide]]).
```

En affichant cet état, on obtient bien la grille suivante (en remplaçant les nombres par leur lettre correspondante pour garder l'affichage intact) :

```
 ---------------
| a | b | c | d |
 ---------------
| e | f | g | h |
 ---------------
| i | j | k | l |
 ---------------
| m | n | o |   |
 ---------------
```

A la fin de notre travail, pour éprouver la généricité de notre code, nous avons testé notre algorithme sur une grille 4x4.

1.2.b

```
initial_state(Ini), nth1(L,Ini,Ligne), nth1(C,Ligne, d).
```

-> Permet de savoir quels sont les numéros de ligne (L) et de colonne (C) du symbole d sur la grille de la situation Ini.

```
final_state(Fin), nth1(3,Fin,Ligne), nth1(2,Ligne,P)
```
-> Permet de savoir quel est le symbole à la troisième ligne et à la deuxième colonne de la grille de la situation Ini.

1.2.c

Cette requête permet de savoir si la pièce a est bien placée dans l'état initial Ini par rapport à F.

```
initial_state(Ini), nth1(L,Ini,LigneI), nth1(C,LigneI, a),
final_state(Fin), nth1(L,Fin,LigneF), nth1(C,LigneF, a).
```

Dans la requête suivante, la variable P peut prendre les valeurs de toutes les pièces bien placées dans l'état initial Ini par rapport à F :

```
initial_state(Ini), nth1(L,Ini,LigneI), nth1(C,LigneI, P),
final_state(Fin), nth1(L,Fin,LigneF), nth1(C,LigneF, P).
```

1.2.d
Pour trouver une situation finale de l'état initial Ini :

```
initial_state(Ini), rule(Action, 1, Ini, Successeur).
```

On constate bien trois états successeurs possibles :

```
-------------  -------------  -------------
| b | h | c |  | b | h | c |  | b | h | c |
-------------  -------------  -------------
| a |   | d |  | a | f | d |  | a | f | d |
-------------  -------------  -------------
| g | f | e |  |   | g | e |  | g | e |   |
-------------  -------------  -------------
```

1.2.e
Pour avoir les 3 solutions regroupées dans une liste :

```
initial_state(Ini),
findall(Successeur, rule(_, 1, Ini, Successeur), Successeurs).
```

Cette action permet de compiler les différentes valeurs prises par la variable Successeur lors de la résolution de rule dans une même liste Successeurs.

1.2.f
On peut améliorer la requête précédente en rangeant dans la liste SuccEtAct les actions ayant permis d'atteindre les états successeurs de l'état initial :

```
initial_state(Ini),
findall([Action, Successeur], rule(Action, 1, Ini, Successeur),
SuccEtAct).
```

II. Développement des 2 heuristiques :

2.1 Heuristique du nombre de pièces mal placées :

Pour le développement de l'heuristique 1, nous choisissons l'approche a, l'évaluation récursive du nombre des différences entre les matrices représentant les grilles.
En résumé, la clause d'évaluation de la différence entre matrices appelle celle pour les lignes, qui appelle celle pour les éléments.

```
%les deux matrices sont vides (cas trivial)
diff_matrice([], [], 0).
diff_matrice([L1|M1], [L2|M2], X):-
    diff_ligne(L1, L2, X1),
    diff_matrice(M1, M2, X2),
    X is X2+X1.
```

```
%les deux lignes sont vides (cas trivial)
diff_ligne([], [], 0).
diff_ligne([E1|L1], [E2|L2], X) :-
    diff_element(E1, E2, X1),
    diff_ligne(L1, L2, X2),
    X is X2+X1.
```

```
/*fonction pour évaluer l'égalité
    entre deux éléments :
    > si E1\=E2 et E1\=vide => R=1
    > sinon R=0
*/
diff_element(E1,E2,R) :-
    ( E1\=E2 ->
        (E1\=vide ->
            R=1
        ;
            R=0)
    ;
        R=0
    ).
```

Nous calculons à la main les valeurs de l'heuristique 1 sur les différentes situations initiales fournies, et éprouvons notre implémentation de heuristique1 par des test unitaires qui se révèlent probants (cf test_heuristique dans testUni.pl). De plus, la coïncidence est également vérifiée pour notre état final.

```
?- test_heuristique1.
*****TEST UNITAIRE DE L'HEURISTIQUE 1*****
TEST No1 : SUCCES ; Calculé = Attendu ; 4
TEST No2 : SUCCES ; Calculé = Attendu ; 2
TEST No3 : SUCCES ; Calculé = Attendu ; 7
TEST No4 : SUCCES ; Calculé = Attendu ; 6
TEST No5 : SUCCES ; Calculé = Attendu ; 8
TEST No6 : SUCCES ; Calculé = Attendu ; 2
TEST Coïncidence : SUCCES ; l'heuristique
est coïncidente avec comme état final :
   ------------
  | a | b | c |
   ------------
  | h |   | d |
   ------------
  | g | f | e |
   ------------
****************************************
```

2.2 Heuristique basée sur la distance de Manhattan :

Pour le calcul de cette heuristique, nous avons fabriqué une liste à partir de la matrice représentant la situation (prédicat construitListe), pour pouvoir la parcourir plus facilement. Nous calculons ensuite la distance de Manhattan pour chaque pièce de la liste, puis faisons la somme de ces distances pour trouver H.

```
heuristique2(U, H) :-
    % on transforme la matrice U en liste pour la parcourir
    construitliste(U, Liste),
    final_state(F),
    % on construit la liste des distances de Manhattan
    findall(DManh, (member(Piece, Liste), d_manh(U, F, Piece, DManh)), ListeDM),
    % on somme les éléments de la liste
    sumlist(ListeDM, H).
```

Nous calculons la distance de Manhattan d'une pièce avec ses coordonnées dans la matrice de la situation concernée et dans la matrice de la situation finale (avec le prédicat coordonnees).

```
coordonnees([L,C], Mat, Elt) :-
    nth1(L, Mat, Row),
    nth1(C, Row, Elt) .
```

```
%la distance de Manhattan de vide est nulle peu importe les matrices
d_manh(_, _, vide, 0).
d_manh(M1,M2,Value,DManh) :-
    Value\=vide,
    % on récupère les coordonnées dans M1
    coordonnees([X1, Y1], M1, Value),
    % on récupère les coordonnées dans M2
    coordonnees([X2, Y2], M2, Value),
    % on calcule la distance de Manhattan
    DManh is (abs(X1-X2)+abs(Y1-Y2)).
```

Pour le cas trivial, si l'élément est vide on renvoie une distance nulle.

Des test unitaires ont été faits, comparant les valeurs calculées sur les exemples de situation initiale à celles attendues, spécifiées dans le fichier. La coïncidence est également vérifiée pour notre état final.

```
?- test_heuristique2.
*****TEST UNITAIRE DE L'HEURISTIQUE 2*****
TEST No1 : SUCCES ; Calculé = Attendu ; 5
TEST No2 : SUCCES ; Calculé = Attendu ; 2
TEST No3 : SUCCES ; Calculé = Attendu ; 10
TEST No4 : SUCCES ; Calculé = Attendu ; 16
TEST No5 : SUCCES ; Calculé = Attendu ; 24
TEST No6 : SUCCES ; Calculé = Attendu ; 2
TEST Coïncidence : SUCCES ; l'heuristique
est coïncidente avec comme état final :
  ------------
 | a | b | c |
  ------------
 | h |   | d |
  ------------
 | g | f | e |
  ------------
*****************************************
```

(Les tests des heuristiques pour les deux situations extrêmes U0 et F sont compris dans les tests unitaires.)

III. Implémentation de A* :

3.1 Implémentation de P et Q par des arbres AVL :

La structure d'AVL fournie permet d'ordonner nos situations, afin de déterminer facilement celles qui valent le coup d'être développées. Dans notre algorithme A*, nous utiliserons deux AVL pour représenter l'ensemble P des états non encore développés : Pu (ordonné lexicographiquement) et Pf (ordonné selon f et h, utile pour déterminer l'état de f minimum).

Après les tests grâce aux arbres disponibles en fin de fichier, nous comprenons que :
empty/1 permet de créer un AVL nul (nil).
Il y a également les prédicats insert/3 et suppress/3, qui nous seront utiles pour mettre à jour l'arbre.
Le prédicat suppress_min/3 sera utile pour trouver dans Pf l'état qui vaut le coup d'être exploré.
D'autres prédicats comme put_flat/1 et put_90/1 nous ont été utiles pour le débuggage.

3.2 Algorithme A* adapté aux structures AVL choisies :

Dans ce qui suit, certaines explications se trouvent dans les commentaires présents sur les captures. Sous celles-ci, nous expliquerons brièvement les clauses dont nous n'avons pas jugé nécessaire de mettre le code.

main/0 :

La fonction main initialise les AVL et lance A*

```
main :-
    initial_state(S0),
    % on initialise les trois arbres à nil
    empty(Pu0),
    empty(Pf0),
    empty(Q),
    % on calcule l'heuristique de l'état initial
    heuristique(S0, H0),
    %on insère [F0=H0+G0, H0, G0=0] dans Pu & Pf
    insert([[H0, H0, 0], S0], Pf0, Pf),
    insert([S0, [H0, H0, 0], nil, nil], Pu0, Pu),
    % lancement de A*
    aetoile(Pf,Pu,Q).
```

aetoile/3 :

> cas trivial de l'état final inatteignable : Pf et Pu sont vides.

```
% cas trivial, Pf et Pu sont vides donc pas de solution
aetoile(nil, nil, _) :-
    writeln("PAS de SOLUTION: L'ETAT FINAL N'EST PAS ATTEIGNABLE!").
```

> cas trivial : la matrice du noeud extrait correspond à la situation finale : on peut afficher la solution.

```
%cas trivial, on a trouvé la solution
aetoile(Pf, Pu, Q):-
    % l'état du noeud minimum extrait correspond à la situation finale
    final_state(Sf),
    suppress_min([_,UFMin], Pf, _),
    UFMin==Sf,

    % on supprime le noeud frère dans Pu est récupère au passage son père et l'action y menant
    suppress([UFMin, [FUFMin,HUFMin,GUFMin], PereUFMin, ActionUFMin], Pu, _),

    % on ajoute l'état final dans Q
    insert([UFMin, [FUFMin,HUFMin,GUFMin], PereUFMin, ActionUFMin],Q,Q1),

    affiche_solution(Q1, Sf, _),!.
```

> cas général :

```
aetoile(Pf, Pu, Q) :-
    % on cherche et supprime l'état UFMin de F minimum dans Pf
    suppress_min([_,UFMin] , Pf, Pf2),

    % on supprime le noeud frère dans Pu
    suppress([UFMin, [FUFMin,HUFMin,GUFMin], PereUFMin, ActionUFMin], Pu, Pu2),

    % on cherche les successeurs
    successeursEtActions(UFMin, SetA),
    splitSetA(SetA,Successeurs,Actions),

    % on calcule les valeurs [F, H, G] pour chaque successeur
    expand(Successeurs, [Fs,Hs,Gs] , GUFMin),

    % on parcourt chaque successeur pour les traiter, et on récupère les nouveaux AVL Pu3 et Pf3
    loop_successors(Successeurs, [Fs, Hs, Gs], Pu2, Pf2, Pu3, Pf3, Q, UFMin, Actions),

    % U ayant été développé et supprimé de P, on insère son noeud dans
    insert([UFMin,[FUFMin,HUFMin,GUFMin], PereUFMin, ActionUFMin], Q, Q2),

    % on rappelle aetoile avec les nouveaux AVL
    aetoile(Pf3,Pu3,Q2).
```

Dans la clause successeurEtActions(Etat, SetA) , SetA est une liste contenant des éléments de la forme [Successeur, Action]. splitSetA/3 permet d'obtenir une liste Successeurs et une liste Actions à partir de SetA, ce qui est plus facile à traiter.

expand/3 :

```
expand(Successeurs, [Fs, Hs, Gs], Gu):-
    %calcul de H pour chaque successeur
    findall(H, (member(S, Successeurs), heuristique(S, H)), Hs),
    %calcul de G pour chaque successeur
    findall(G, (member(S, Successeurs), G is Gu+1), Gs),
    %calcul de F en faisant G+H pour chaque successeur
    sommeListe(Gs, Hs, Fs).
```

Notre prédicat expand calcule le trio de valeur [F, H, G] associé à chaque successeur.
Pour tester expand, nous lançons des requêtes avec les successeurs de l'état initial :

```
initial_state(Ini),
findall(Successeur, rule(_, 1, Ini, Successeur), Successeurs),
expand(Successeurs, [Fs, Hs, Gs], 1).
```

traiter_successeur/9 (loop_successors) :

Basiquement, notre clause loop_successors ne sert qu'à parcourir les successeurs pour appliquer à chacun traiter_successeur, il nous semble donc plus pertinent de détailler cette dernière.

```
traiter_successeur(Succ, [FNew, HNew, GNew], Pu0, Pf0, Pu1, Pf1, Q, Pere, Action) :-
    %si le successeur est dans Q, on ignore, les AVL restent inchangés
    (belongs([Succ, _, _, _], Q) -> Pu1 = Pu0 ,
                                    Pf1 = Pf0
        ;
        %si est S connu dans Pu on compare les deux noeuds et on garde la meilleure évaluation
        (belongs([Succ, [FOld,HOld,_], _, _],Pu0) ->
                                compareEtats(Succ, FOld, HOld, FNew, HNew, GNew, Pu0, Pu1, Pf0, Pf1, Pere, Action)
            ;
            % le successeur n'est connu ni dans Q, ni dans Pu, on l'insère donc dans Pu et Pf
            insert([Succ,[FNew, HNew, GNew],Pere, Action],Pu0,Pu1),
            insert([[FNew, HNew, GNew],Succ],Pf0,Pf1)
            )
        ).
```

On distingue donc trois cas :

1. on ignore le successeur s'il a déjà été développé (i.e. s'il est dans Q).
2. sinon, on compare son évaluation à celle déjà enregistrée dans P, et on la remplace si elle vaut plus le coup.
3. on l'ajoute dans Pu et Pf s'il n'y est pas présent.

Le cas 2 fait appel à la clause compareEtats, consultable dans le code : elle met à jour Pu et Pf si FNew < FOld ou si FNew=FOld et HNew < FOld, et les laisse inchangés sinon.

Nous avons réalisé de solides test unitaires pour traiter_successeur, qui couvre tous les cas possibles (successeur connu dans Q, successeur inconnu dans P et Q, successeur connu dans P avec meilleurr évaluation (à F égaux ou non), successeur connu dans P avec moins bonne évaluation (à F égaux ou non).

```
?- test_traiter_successeur.
*****TEST UNITAIRE DE TRAITER SUCCESSEUR*****
>Traitement successeur quand Q vide, Pu vide et Pf vide
Test No1 : SUCCES ; Calculé = Attendu ; avl(nil,[x,[3,2,1],p,up],nil,0)
Test No2 : SUCCES ; Calculé = Attendu ; avl(nil,[[3,2,1],x],nil,0)
>Traitement noeud successeur déjà connu mais moins intéressant
  avec FNew > FOld
Test No3 : SUCCES ; Calculé = Attendu ; avl(nil,[x,[3,2,1],p,up],nil,0)
Test No4 : SUCCES ; Calculé = Attendu ; avl(nil,[[3,2,1],x],nil,0)
  avec FNew==FOld et HNew > HOld
Test No5 : SUCCES ; Calculé = Attendu ; avl(nil,[x,[3,2,1],p,up],nil,0)
Test No6 : SUCCES ; Calculé = Attendu ; avl(nil,[[3,2,1],x],nil,0)
>Traitement noeud successeur déjà connu et plus intéressant
  avec FNew < FOld
Test No7 : SUCCES ; Calculé = Attendu ; avl(nil,[x,[2,0,2],p,up],nil,0)
Test No8 : SUCCES ; Calculé = Attendu ; avl(nil,[[2,0,2],x],nil,0)
  avec FNew==FOld et HNew < HOld
Test No9 : SUCCES ; Calculé = Attendu ; avl(nil,[x,[3,0,3],p,up],nil,0)
Test No10 : SUCCES ; Calculé = Attendu ; avl(nil,[[3,0,3],x],nil,0)
>Traitement noeud successeur déjà connu par Q
Test No11 : SUCCES ; Calculé = Attendu ; nil
Test No12 : SUCCES ; Calculé = Attendu ; nil
*********************************************
```

affiche_solution/3 :

Une fois que l'état final est atteint, nous appliquons notre clause affiche_solution. L'idée est de remonter de fils en père jusqu'à l'état initial. Nous avons développé une clause affiche_etat/1 pour afficher_solution, qui permet de voir très distinctement la grille, et de deviner plus facilement les actions effectuées pour passer de l'une à l'autre.

```
% cas trivial dans lequel l'état initial est trouvé
affiche_solution(Q, U, 0):-
    belongs([U, [F, H, G], nil, nil], Q),
    initial_state(U),
    writeln("Etat initial : "),
    affiche_etat(U),
    write("F = "),
    write(F),
    write(" ; H = "),
    write(H),
    write(" ; G = "),
    writeln(G),
    writeln("***************************").
```

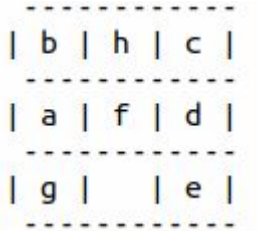
```
% cas général
affiche_solution(Q, U, NumeroEtape) :-
    belongs([U, [F, H, G], Pere, Action], Q),
    affiche_solution(Q, Pere, NE),
    NumeroEtape is NE+1,
    write("Etape "),
    write(NumeroEtape),
    writeln(" : "),
    write("Action : "),
    writeln(Action),
    affiche_etat(U),
    write("F = "),
    write(F),
    write(" ; H = "),
    write(H),
    write(" ; G = "),
    writeln(G),
    writeln("***************************").
```

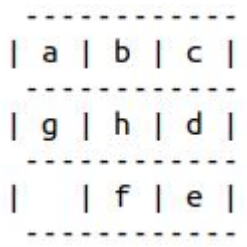
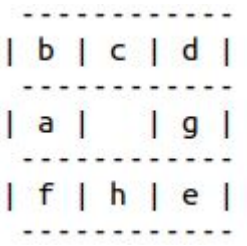
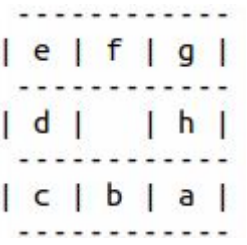
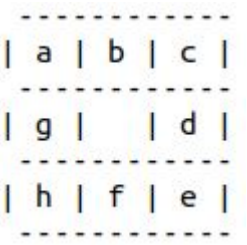
Pour un problème simple, affiche_solution donne ce résultat :

```
Etat initial :
  -------------
 | a | b | c |
  -------------
 | g | h | d |
  -------------
 |   | f | e |
  -------------
F = 2 ; H = 2 ; G = 0
**********************
Etape 1 :
Action : up
  -------------
 | a | b | c |
  -------------
 |   | h | d |
  -------------
 | g | f | e |
  -------------
F = 2 ; H = 1 ; G = 1
**********************
Etape 2 :
Action : right
  -------------
 | a | b | c |
  -------------
 | h |   | d |
  -------------
 | g | f | e |
  -------------
F = 2 ; H = 0 ; G = 2
**********************
```

3.3 Analyse expérimentale :

> Noter le temps de calcul de A* et l'influence du choix de l'heuristique : quelle taille de séquences optimales (entre 2 et 30 actions) peut-on générer avec chaque heuristique (H1, H2) ? Présenter les résultats sous forme de tableau.

Etat initial	Heuristique 1	Heuristique 2
	H(I1) = 4 0.001 seconds 4245 inferences 5 coups	H(I1) = 5 0.009 seconds 27 718 inferences 5 coups

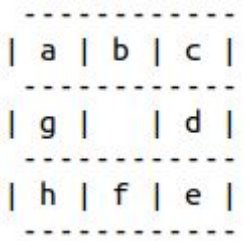
	H(I2) = 2 0.001 seconds 1182 inferences 2 coups	H(I2) = 2 0.003 seconds 3805 inferences 2 coups
	H(I3) = 7 0.007 seconds 18 657 inferences 10 coups	H(I2) = 10 0.009 seconds 20 312 inferences 10 coups
	H(I4) = 6 0.932 seconds 2 771 963 inferences 20 coups	H(I4) = 16 0.032 seconds 153 983 inferences 20 coups
	H(I5) = 8 ne trouve pas de solution	H(I4) = 24 0.987 seconds 5 082 392 inferences 30 coups
	H(I6) = 2 ne trouve pas de solution	H(I6) = 2 ne trouve pas de solution

La taille de séquence optimale maximal que nous avons pu générer avec H1 est de 20 (avec I4), tandis qu'elle est de 30 avec H2 (avec I5).
Par ailleurs, on peut remarquer que H1 est beaucoup plus efficace que H2 pour des problèmes peu compliqués. En observant le fonctionnement des heuristiques, on peut voir que H1 a tendance à être plus optimiste, et à ne pas donner de poids aux difficultés : deux pièces mal placées rajouteront chacune la même valeur 1 à l'évaluation de l'heuristique, même si la place finale de l'une est à côté, et celle de l'autre à l'autre bout du plateau. H2 fait cette distinction avec la distance de Manhattan : ce n'est pas binaire, une pièce peut être plus ou moins bien placée en fonction de son éloignement par rapport à sa place dans la situation finale.

> Quelle longueur de séquence peut-on envisager de résoudre pour le Taquin 4x4 ?

Ce qui empêche de générer une séquence de longueur infinie est la mémoire finie (out of local stack). Dans cet algorithme, la mémoire est remplie par les AVL, remplis par les successeurs générés.
Pour le Taquin 3x3, une situation a en moyenne 2.7 successeurs, contre 3 pour le Taquin 4x4. Si l'on se base que sur ce facteur, la taille de la séquence optimale maximale pour un Taquin 4x4 devrait avoisiner les 27, si celle du Taquin 3x3 est 30. (2.7/3 * 30).
Cependant, d'autres facteurs doivent entrer en compte : notamment qu'une place mal placée doive faire en moyenne plus de déplacements pour gagner sa place.

> A* trouve-t-il la solution pour la situation initiale suivante ?

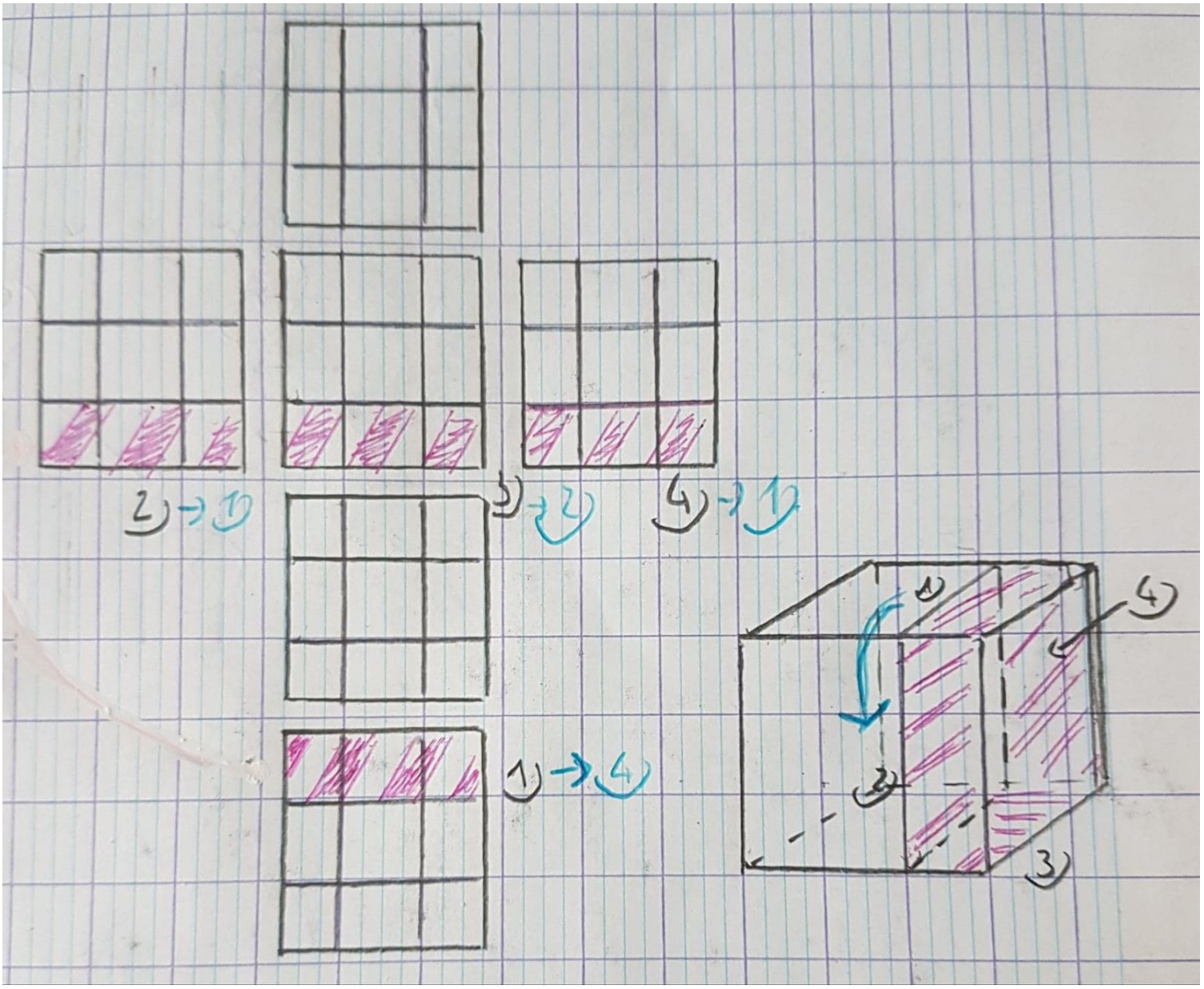


Notre A* ne trouve pas de solution pour cette situation initiale, mais celle ci est non connexe avec l'état final donc il n'existe pas de séquence l'y menant.

> Quelle représentation de l'état du Rubik's Cube et quel type d'action proposeriez-vous si vous vouliez appliquer A*?

Dans un Rubik's cube de taille N, chaque situation a N*6 successeurs (-1 en enlevant sa situation père).

Pour représenter l'état du Rubik's Cube, nous proposons six matrices, chacune représentant une face. Ces six matrices doivent être corrélées entre elles, pour que l'algorithme puisse voir quels sont les alignements qui font le tour du cube. (cf schéma ci dessous, qui montre un alignement en rose sur les six matrices et la représentation 3D). Avec cette représentation, l'algorithme pourra établir les successeurs de chaque situation en lui appliquant chaque coup. Sur notre schéma, en bleu, on peut voir que pour le mouvement opéré sur le cube en représentation 3D à droite, une rotation est modélisée : l'alignement 1 devient 4, le 2 devient 1 etc.



Comme heuristique, on peut imaginer de calculer pour chaque face le nombre de couleurs différentes présentes dessus, et faire la somme pour tout le cube.
Ou bien, pour rebondir sur la comparaison entre H1 et H2 du taquin et se rapprocher d'une heuristique comme H2, qui semble plus appropriée pour un problème de cette complexité, on pourrait évaluer l'éparpillement pour chaque couleur. Par exemple, si un carré vert est seul sur une face mais qu'il peut en rejoindre 4 autres en un coup, la valeur de l'heuristique pourra en être atténuée.

Conclusion :

Dans ce projet, nous avons éprouvé le fonctionnement de A* un sur exemple simple : le jeu du taquin. Il était très intéressant de voir qu'une heuristique peut être très efficace pour résoudre des problèmes simples, et inefficace quand ils se compliquent (et vice versa pour d'autres). Ainsi, même pour le problème du taquin, le choix entre deux heuristiques semble avoir une grande importance : même si les deux peuvent résoudre le problème, il faut bien le définir pour voir laquelle est la plus adaptée.

L’ouverture du Rubik’s Cube est intéressante, car elle permet de se rendre compte qu’il n’est pas toujours facile de définir une heuristique en abordant un nouveau problème, bien que celle-ci soit déterminante de l’efficacité de la résolution.