

INSTITUT DE LA FRANCOPHONIE POUR L'INFORMATIQUE



INTELLIGENCE ARTIFICIELLE

TRAVAIL PRATIQUE 1

Encadrement : TRAN Duc Khanh

Etudiant : LE Viet Man

Promotion 15, IFI

Hanoï, Janvier 2010

1. INTRODUCTION

Le problème est d'implémenter les algorithmes de recherche pour résoudre le Puzzle 8 et le Puzzle 15. En particulier :

- Implémenter trois algorithmes : la recherche en largeur d'abord, la recherche en profondeur d'abord et la recherche A* pour le Puzzle 8.
- Implémenter deux algorithmes : IDA* et SMA* pour le Puzzle 15.

De plus, dans mon TP, j'ai encore programmé l'algorithme en profondeur itérative pour le Puzzle 8 et l'algorithme A* pour le Puzzle 15 pour comparer l'efficacité entre ces algorithmes.

Pour programmer les algorithmes ci-dessus, j'ai utilisé le langage C++ et la librairie Qt de Nokia (<http://qt.nokia.com/>). J'ai testé mes implémentations sur les systèmes d'exploitation Windows 7 et Mac OS X.

2. IMPLEMENTATION

2.1. Implémentation générale

2.1.1. Organisation des données :

Pour stocker des états dans le processus de calcul, j'ai construit une classe **State** avec des propriétés suivantes :

| Propriété | Définition |
|--------------------------|---|
| int State[9]; | un array «int» avec la taille 9 ou 16 qui correspondent au Puzzle 8 ou au Puzzle 15. Ce array a pour but de stocker des valeurs de la matrice Puzzle. La case vide est représenté par la valeur 0. Alors, au lieu de stocker en matrice, j'ai utilisé un array. |
| State *parent; | un pointeur qui pointe l'état de père. |
| QList<State *> children; | une liste des états successives. |

La classe **State** possède les méthodes suivantes :

| Méthodes | Définition |
|---|--|
| State(State *parent, int stateNew[9]); | c'est un constructeur |
| void addChildren(State *stateChildren); | une méthode a pour but d'ajouter un nouveau enfant |
| State *getParent(); | recevoir l'état de père |
| int at(int index); | recevoir la valeur à la case «index» |
| QString toString(); | recevoir la chaîne des valeurs de l'état |
| bool equal(State *stateOther); | vérifier l'égalité entre l'état avec l'état «stateOther» |
| QList<State *> successeur(State *parent); | recevoir des états successives |

Ci-dessous est la classe State pour le Puzzle 8 :

```

class State
{
public:
    State(State *parent, int stateNew[9]);
    void addChildren(State *stateChildren);
    State *getParent();
    int at(int index);
    QString toString();
    bool equal(State *stateOther);
    QList<State *> successeur(State *parent);

private:
    void copy(int state[9]);
    void exchangeValue(int posi, int posj);

    int state[9];
    State *parent;
    QList<State *> children;
};

```

2.1.2. Interface

Puisque j'ai seulement consacré à l'algorithme, les interfaces sont simples. L'utilisateur entrera des données à le textbox **Input**.

La manière d'entrer des puzzles :

soit le puzzle suivant :

1 2 3

4 5 6

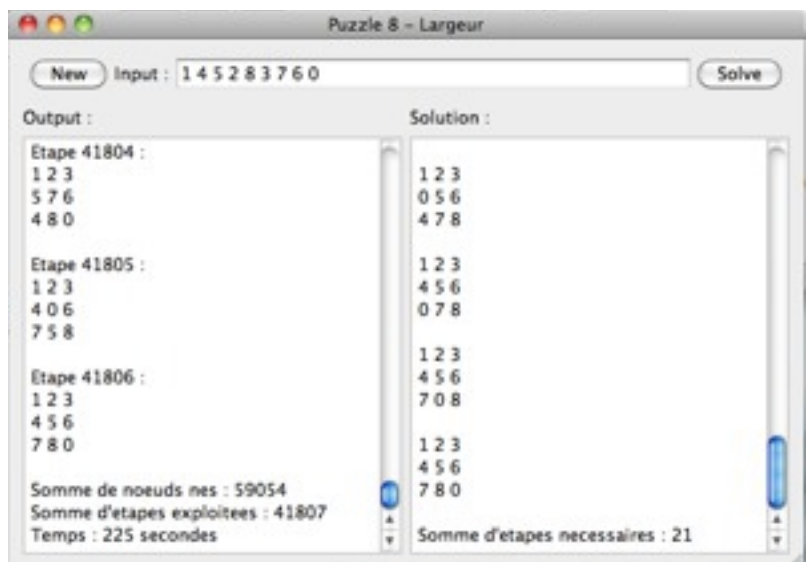
7 8 0

entrer comme :

1 2 3 4 5 6 7 8 0

Note : pas entrer des espaces superflus.

Après avoir touché le bouton **Solve**, le logiciel exécutera et le résultat affichera dans deux textbox, **Output** et **Solution** où Output présentera tous les états traversés et Solution présentera tous les états nécessaires pour résoudre le problème. Si l'algorithme ne peut pas le résoudre, la phrase «Ce probleme ne peut pas resoudre !» affichera.



2.1.3. Organisation des résultats

Les logiciels affichent principalement des résultats dans deux textbox Output et Solution. Mais j'ai enregistré les résultats dans les fichiers en organisant un dossier **Resultat**. Dans ce dossier Resultat, j'ai mis aussi des puzzles pour tester.

2.2. Recherche en largeur d'abord - Puzzle 8

Recherche en largeur d'abord est une des algorithmes de recherche simples. Elle est aussi facile à implémenter. J'ai utilisé une queue **open** pour stocker des états successives non plus traversés et une liste **close** pour stocker des états traversés.

Mon algorithme est pareil l'algorithme origine, mais je l'ai amélioré. Au lieu d'ajouter tous les états successives à la queue **open**, je vérifie tout d'abord qu'il existe ces états dans la queue **open** et la liste **close**. Si non, je l'ajoute à **open**. Alors, l'algorithme esquivra de reexploiter les états traversés.

L'algorithme est implémenté en détail comme ci-dessous :

```
open.enqueue(initialState);

while (!open.isEmpty())
{
    State *temp = open.dequeue();
    close.append(temp);

    if (temp->equal(finalState))
    {
        return true;
    }
    else
    {
        QList<State *> succ = temp->successeur(temp);
        for (int i = 0; i < succ.length(); i++)
        {
            bool check = false;
            for (int j = 0; j < open.length(); j++)
                if (succ.at(i)->equal(open.at(j)))
                {
                    check = true;
                    break;
                }
        }
    }
}
```

```

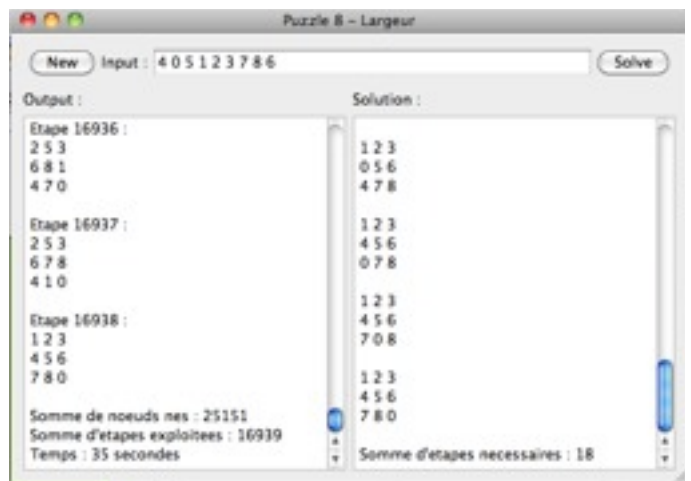
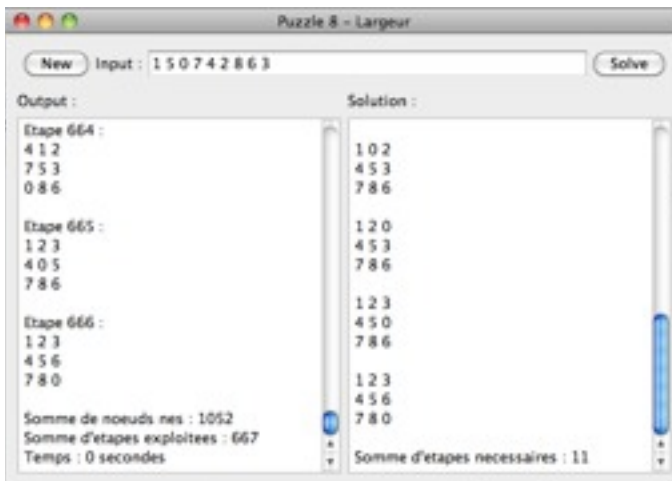
    if (!check)
        for (int j = 0; j < close.length(); j++)
            if (succ.at(i)->equal(close.at(j)))
            {
                check = true;
                break;
            }
    if (!check)
    {
        temp->addChildren(succ.at(i));
        open.enqueue(succ.at(i));
    }
}
}

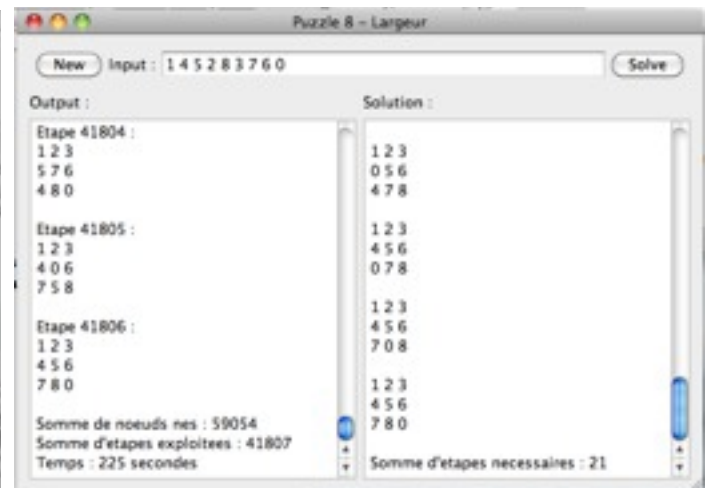
return false;

```

Evaluation

Quelques testes et les résultats :





Remarque : la recherche en largeur d'abord est seulement efficace pour les problèmes dont le nombre des étapes traversées sont bases. Avec des problèmes dont le nombre des étapes traversées sont plus hautes, l'espace de mémoire et le temps de calcul augmenteront en multiplicateur.

2.3. Recherche en profondeur d'abord - Puzzle 8

Au contraire de la recherche en largeur d'abord, j'ai utilisé un stack open pour stocker des états successives non plus traversés. Comme mon algorithme en largeur d'abord, j'ai aussi vérifié l'existence des états successives dans le stack **open** et la liste **close**.

L'algorithme est implémenté en détail comme ci-dessous :

```
open.push(initialState);

while (!open.isEmpty())
{
    State *temp = open.pop();
    close.append(temp);

    if (temp->equal(finalState))
    {
        return true;
    }
    else
    {
        QList<State *> succ = temp->successeur(temp);
        for (int i = 0; i < succ.length(); i++)
        {
            bool check = false;
            for (int j = 0; j < open.toList().length(); j++)
                if (succ.at(i)->equal(open.at(j)))
```

```

        {
            check = true;
            break;
        }
    if (!check)
        for (int j = 0; j < close.length(); j++)
            if (succ.at(i)->equal(close.at(j)))
                {
                    check = true;
                    break;
                }
    if (!check)
    {
        temp->addChildren(succ.at(i));
        open.push(succ.at(i));
    }
}
}

return false;

```

Evaluation

| Output : | Solution : |
|--|---|
| Etape 232 : 1 2 3 4 0 6 7 5 8 | 1 2 3 4 6 0 7 5 8 |
| Etape 233 : 1 2 3 4 0 6 4 5 6 7 0 8 | 1 2 3 4 0 6 7 5 8 |
| Etape 234 : 1 2 3 4 5 6 7 8 0 | 1 2 3 4 5 6 7 0 8 |
| Somme de noeuds nes : 428 Somme d'etapes exploitees : 235 Temps : 0 secondes | 1 2 3 4 5 6 7 8 0 Somme d'etapes necessaires : 214 |

| Output : | Solution : |
|---|--|
| Etape 1173 : 1 2 3 4 0 6 7 5 8 | 1 2 3 4 6 0 7 5 8 |
| Etape 1174 : 1 2 3 4 0 6 4 5 6 7 0 8 | 1 2 3 4 0 6 7 5 8 |
| Etape 1175 : 1 2 3 4 5 6 7 8 0 | 1 2 3 4 5 6 7 0 8 |
| Somme de noeuds nes : 2127 Somme d'etapes exploitees : 1176 Temps : 48 secondes | 1 2 3 4 5 6 7 8 0 Somme d'etapes necessaires : 1086 |

Quelques testes et les résultats :

La tableau suivante compare les résultats de deux algorithmes :

| Problème | Largeur d'abord | Profondeur d'abord |
|-------------------|--|--|
| 1 0 2 4 5 3 7 8 6 | Somme de noeuds nes : 26 Somme d'etapes exploitees : 13 Temps : 0 secondes Somme d'etapes necessaires : 4 | Somme de noeuds nes : 428 Somme d'etapes exploitees : 235 Temps : 0 secondes Somme d'etapes necessaires : 214 |
| 4 1 2 0 5 3 7 8 6 | Somme de noeuds nes : 64 Somme d'etapes exploitees : 35 Temps : 0 secondes Somme d'etapes necessaires : 6 | Somme de noeuds nes : 2127 Somme d'etapes exploitees : 1176 Temps : 46 secondes Somme d'etapes necessaires : 1086 |
| 1 5 0 7 4 2 8 6 3 | 0 secondes | trop long |
| 4 0 5 1 2 3 7 8 6 | 35 secondes | trop long |
| 1 4 5 2 8 3 7 0 6 | 133 secondes | trop long |
| 1 4 5 2 8 3 7 6 0 | 225 secondes | trop long |

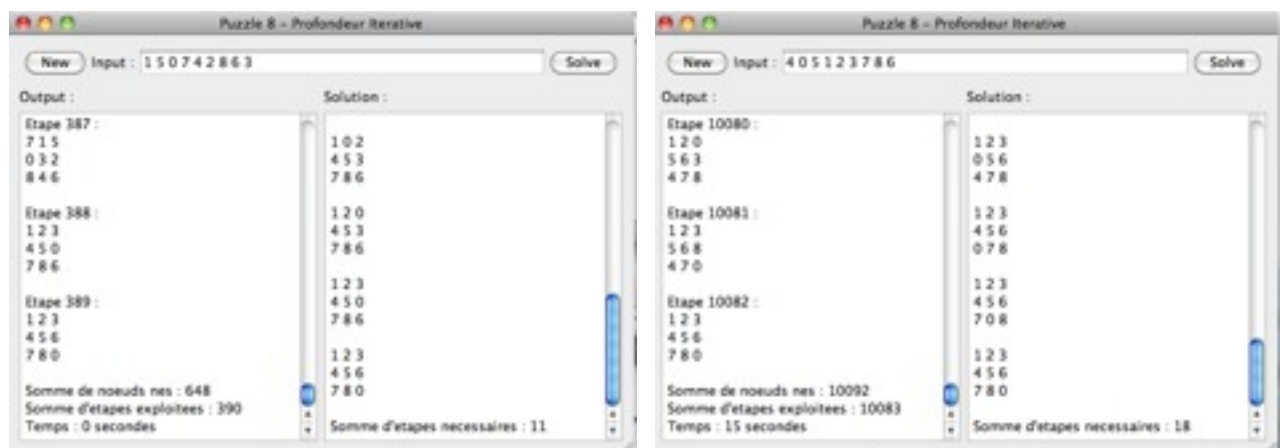
Remarque : L'algorithme en profondeur d'abord n'est pas efficace pour la plupart des problèmes.

2.4. Recherche en profondeur itérative - Puzzle 8

A partir de la raison ci-dessus, j'ai implémenté l'algorithme de la recherche en profondeur itérative. Dans mon algorithme, j'ai amélioré pour ne pas retraverser tous les états traversés dans l'itération précédente. J'ai utilisé deux stack **open** et **open1** pour stocker tour à tour des états successives non plus traversés dans chaque itération.

Evaluation

Quelques testes et les résultats :



| Problème | Largeur d'abord | Profondeur itérative |
|-------------------|---|--|
| 1 0 2 4 5 3 7 8 6 | Somme de noeuds nes : 26 Somme d'etapes exploitees : 13 Temps : 0 secondes Somme d'etapes necessaires : 4 | Somme de noeuds nes : 17 Somme d'etapes exploitees : 9 Temps : 0 secondes Somme d'etapes necessaires : 4 |
| 4 1 2 0 5 3 7 8 6 | Somme de noeuds nes : 64 Somme d'etapes exploitees : 35 Temps : 0 secondes Somme d'etapes necessaires : 6 | Somme de noeuds nes : 61 Somme d'etapes exploitees : 34 Temps : 0 secondes Somme d'etapes necessaires : 6 |
| 1 5 0 7 4 2 8 6 3 | Somme de noeuds nes : 1052 Somme d'etapes exploitees : 667 Temps : 0 secondes Somme d'etapes necessaires : 11 | Somme de noeuds nes : 648 Somme d'etapes exploitees : 390 Temps : 0 secondes Somme d'etapes necessaires : 11 |
| 4 0 5 1 2 3 7 8 6 | Somme de noeuds nes : 25151 Somme d'etapes exploitees : 16939 Temps : 41 secondes Somme d'etapes necessaires : 18 | Somme de noeuds nes : 19971 Somme d'etapes exploitees : 12650 Temps : 22 secondes Somme d'etapes necessaires : 18 |
| 1 4 5 2 8 3 7 0 6 | Somme de noeuds nes : 47589 Somme d'etapes exploitees : 33035 Temps : 169 secondes Somme d'etapes necessaires : 20 | Somme de noeuds nes : 29037 Somme d'etapes exploitees : 17419 Temps : 51 secondes Somme d'etapes necessaires : 20 |



Se baser sur les résultats, je trouve que l'espace des états et le temps de calcul sont diminués de façon appréciable dans l'algorithme en profondeur itérative.

2.5. L'algorithme A* - Puzzle 8

Pour implémenter l'algorithme A*, j'ai ajouté une fonction heuristique **calculateH()** à la classe **State** pour calculer la valeur h en utilisant la distance « Manhattan ». Dans mon algorithme A*, j'ai utilisé une liste **open** pour stocker tour à tour des états successives non plus traversés au lieu d'une queue. Donc, j'ai implémenté une fonction **choisirState** pour recevoir le meilleur état avant la valeur **f** plus bas.

Evaluation

Quelques testes et les résultats :

| | |
|--|--|
|  <p>Puzzle 8 - A*</p> <p>New Input : 1 4 5 2 0 6 7 3 8 Solve</p> <p>Output : Etape 997 : 1 2 3 4 0 5 7 8 6</p> <p>Etape 998 : 1 2 3 1 2 3 4 0 5 7 8 6</p> <p>Etape 999 : 1 2 3 4 5 6 7 8 0</p> <p>Somme de noeuds nes : 1610 Somme d'etapes exploitees : 1000 Temps : 0 secondes</p> <p>Solution : 1 0 3 4 2 5 7 8 6</p> <p>1 2 3 4 5 0 7 8 6</p> <p>1 2 3 4 5 0 7 8 6</p> <p>1 2 3 4 5 6 7 8 0</p> <p>Somme d'etapes necessaires : 21</p> |  <p>Puzzle 8 - A*</p> <p>New Input : 6 3 0 5 7 1 8 4 2 Solve</p> <p>Output : Etape 9962 : 1 2 3 1 2 3 4 0 6 7 5 8</p> <p>Etape 9963 : 1 2 3 1 2 3 4 0 6 7 5 8</p> <p>Etape 9964 : 1 2 3 1 2 3 4 5 6 7 0 8</p> <p>Etape 9965 : 1 2 3 4 5 6 7 8 0</p> <p>Somme de noeuds nes : 15395 Somme d'etapes exploitees : 9965 Temps : 14 secondes</p> <p>Solution : 1 2 3 4 6 0 7 5 8</p> <p>1 2 3 4 5 6 7 5 8</p> <p>1 2 3 4 5 6 7 0 8</p> <p>1 2 3 4 5 6 7 8 0</p> <p>Somme d'etapes necessaires : 29</p> |
|--|--|

| Problème | Largeur d'abord | A* |
|-------------------|---|---|
| 1 0 2 4 5 3 7 8 6 | Somme de noeuds nes : 26 Somme d'etapes exploitees : 13 Temps : 0 secondes Somme d'etapes necessaires : 4 | Somme de noeuds nes : 7 Somme d'etapes exploitees : 4 Temps : 0 secondes Somme d'etapes necessaires : 4 |
| 4 1 2 0 5 3 7 8 6 | Somme de noeuds nes : 64 Somme d'etapes exploitees : 35 Temps : 0 secondes Somme d'etapes necessaires : 6 | Somme de noeuds nes : 10 Somme d'etapes exploitees : 6 Temps : 0 secondes Somme d'etapes necessaires : 6 |
| 1 5 0 7 4 2 8 6 3 | Somme de noeuds nes : 1052 Somme d'etapes exploitees : 667 Temps : 0 secondes Somme d'etapes necessaires : 11 | Somme de noeuds nes : 1022 Somme d'etapes exploitees : 638 Temps : 0 secondes Somme d'etapes necessaires : 18 |
| 4 0 5 1 2 3 7 8 6 | Somme de noeuds nes : 25151 Somme d'etapes exploitees : 16939 Temps : 41 secondes Somme d'etapes necessaires : 18 | Somme de noeuds nes : 1022 Somme d'etapes exploitees : 638 Temps : 0 secondes Somme d'etapes necessaires : 18 |
| 1 4 5 2 8 3 7 0 6 | Somme de noeuds nes : 47589 Somme d'etapes exploitees : 33035 Temps : 169 secondes Somme d'etapes necessaires : 20 | Somme de noeuds nes : 1199 Somme d'etapes exploitees : 740 Temps : 0 secondes Somme d'etapes necessaires : 20 |
| 1 4 5 2 0 6 7 3 8 | trop long | Somme de noeuds nes : 1610 Somme d'etapes exploitees : 1000 Temps : 0 secondes Somme d'etapes necessaires : 21 |
| 6 3 0 5 7 1 8 4 2 | trop long | Somme de noeuds nes : 15395 Somme d'etapes exploitees : 9965 Temps : 14 secondes Somme d'etapes necessaires : 29 |

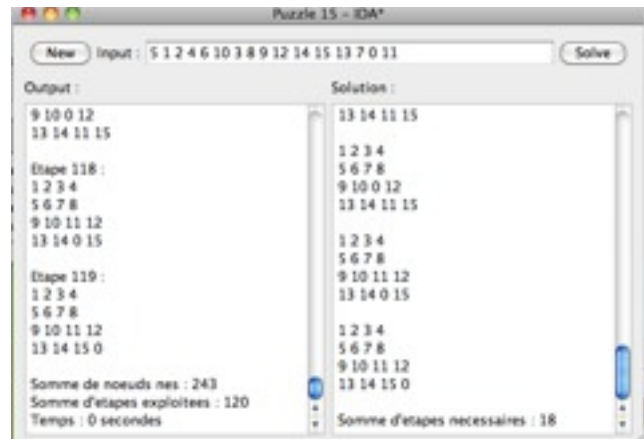
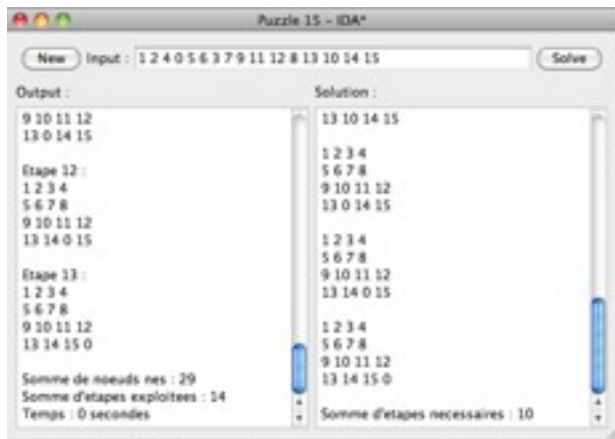
2.6. L'algorithme IDA* - Puzzle 15

L'algorithme IDA* est une extension de l'algorithme de la recherche en profondeur itérative qui utilise la limite de **f** au lieu de la limite de profondeur. Comme mon algorithme de la recherche en profondeur itérative que j'ai présenté dans 2.4, j'ai implémenté deux stack **open** et **open1** pour mon algorithme IDA*.

Evaluation

| Problème | A* | IDA* |
|--|---|---|
| 1 2 4 0 5 6 3 7 9 11 12 8 13 10 14 15 | Somme de noeuds nes : 23 Somme d'etapes exploitees : 11 Temps : 0 secondes Somme d'etapes necessaires : 10 | Somme de noeuds nes : 29 Somme d'etapes exploitees : 14 Temps : 0 secondes Somme d'etapes necessaires : 10 |

| Problème | A* | IDA* |
|--|---|---|
| 5 1 2 4 6 10 3 8 9 12 14 15 13 7 0 11 | Somme de noeuds nes : 232 Somme d'etapes exploitees : 115 Temps : 0 secondes Somme d'etapes necessaires : 18 | Somme de noeuds nes : 243 Somme d'etapes exploitees : 120 Temps : 0 secondes Somme d'etapes necessaires : 18 |



2.7. L'algorithme SMA* - Puzzle 15

L'idée de l'algorithme SMA* est la limitation de la taille de mémoire de la queue **open**. Un état successif est seulement ajouté à la queue **open** quand la queue open n'est pas pleine. Si elle est pleine, on doit supprimer un état qui a la valeur f plus haut.

Dans mon algorithme, j'ai amélioré cet algorithme avec deux idées suivantes :

- Tout d'abord, j'ajoute seulement le meilleur état successif à la queue **open**, c'est-à-dire ne pas ajouter tous les états successifs.

```

if (!succ.isEmpty())
{
    // lay ra thang tot nhat
    int posBest = this->choisirStateMinF(succ);
    State *best = succ.at(posBest);
    temp->addChildren(best);
    open.append(best);
    succ.removeAt(posBest);

    // lay ra thang tot nhi
    if (succ.isEmpty()) // khi khong con thang nao
    {
        open.removeAt(pos);
    }
    else // khi con nhieu thang trong succ

```

```

        {
            posBest = this->choisirStateMinF(succ);
            temp->setF(succ.at(posBest)->getF());
        }
    }
    else // succ la rong
    {
        open.removeAt(pos);
    }
}

```

- Ensuite, j'ai limité la taille de la queue open en 100 records. Si la queue est vide, je supprime l'état qui a la valeur f plus haut.

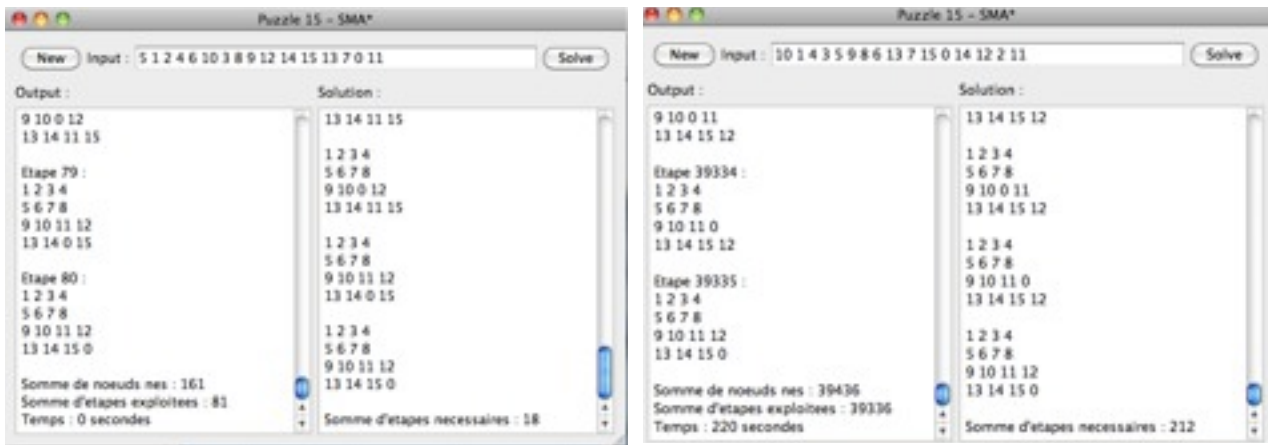
```

while (open.length() > 100)
{
    int posMaxF = this->choisirStateMaxF(open);
    if (this->constraint(close, open.at(posMaxF)))
    {
        close.append(open.at(posMaxF));
    }
    open.removeAt(posMaxF);
}

```

Evaluation

| Problème | A* | SMA* |
|--|---|--|
| 1 2 4 0 5 6 3 7 9 11 12 8 13 10 14 15 | Somme de noeuds nes : 23 Somme d'etapes exploitees : 11 Temps : 0 secondes Somme d'etapes necessaires : 10 | Somme de noeuds nes : 29 Somme d'etapes exploitees : 14 Temps : 0 secondes Somme d'etapes necessaires : 10 |
| 5 1 2 4 6 10 3 8 9 12 14 15 13 7 0 11 | Somme de noeuds nes : 232 Somme d'etapes exploitees : 115 Temps : 0 secondes Somme d'etapes necessaires : 18 | Somme de noeuds nes : 161 Somme d'etapes exploitees : 81 Temps : 0 secondes Somme d'etapes necessaires : 18 |
| 10 1 4 3 5 9 8 6 13 7 15 0 14 12 2 11 | trop long | Somme de noeuds nes : 39436 Somme d'etapes exploitees : 39336 Temps : 220 secondes Somme d'etapes necessaires : 212 |



3. CONCLUSION

J'ai implémenté et évalué les algorithmes de la recherche. Je trouve que l'algorithme SMA* est la meilleure. Dans mon TP, j'ai amélioré les algorithmes pour diminuer l'exigence de la mémoire et pour résoudre rapidement le problème. Mais l'utilisation la liste **close** pour stocker des états traversés est très coûteuse. Je peut utiliser une fonction « hacher » pour coder les états. Alors, il coûte peu de mémoire.

4. REFERENCE

Russell, Stuart et Peter Norvig. 1995. Artificial Intelligence - A Modern Approach. Prentice Hall, Englewood Cliffs, New Jersey 07632, p. 946.