Search methods for general permutation problems (Bachelor's thesis)

Arthur Toenz

July 31, 2012

Contents

1 Introduction **1**

2 Preliminaries **3**
2.1 Permutation Problems 3
2.2 Modelling 4
2.3 Permutation Representation 6
2.3.1 The Cauchy's two-line notation 6
2.3.2 The one-line notation 7
2.3.3 The cycle notation 7

3 The Algorithm & Strategy selection **9**
3.1 The idea within the algorithm 9
3.2 The sift-algorithm 9
3.3 Strategy selection 11
3.4 Example 11

4 Experiments **15**
4.1 The Models 15
4.1.1 The Pancake Problem 15
4.1.2 The TopSpin Puzzle 15
4.2 Results 16
4.3 Discussion 17

5 Conclusion **18**

Abstract

A permutation problem considers the task where an initial order of objects (ie, an initial mapping of objects to locations) must be reordered into a given goal order by using *permutation operators*. Permutation operators are 1:1 mappings of the objects from their locations to (possibly other) locations. An example for permutation problems are the well-known *Rubik's Cube* and *TopSpin Puzzle*. Permutation problems have been a research area for a while, and several methods for solving such problems have been proposed in the last two centuries. Most of these methods focused on finding *optimal* solutions, causing an exponential runtime in the worst case.

In this work, we consider an algorithm for solving permutation problems that has been originally proposed by M. Furst, J. Hopcroft & E. Luks in 1980 ([1]). This algorithm has been introduced on a theoretical level within a proof for "Testing Membership and Determining the Order of a Group", but has not been implemented and evaluated on practical problems so far. In contrast to the other abovementioned solving algorithms, it only finds suboptimal solutions, but is guaranteed to run in polynomial time. The basic idea is to iteratively reach subgoals, and then to let them fix when we go further to reach the next goals. We have implemented this algorithm and evaluated it on different models, as the Pancake Problem and the TopSpin Puzzle .

Chapter 1

Introduction

In combinatoric mathematics, a permutation is the action of rearranging objects from an order into another. This means that all present objects of the considered world are still present after the permutation, as it is just a reordering. And thus, a permutation on objects in a world is a 1 to 1 transformation of the world in itself. Permutation problems have first appeared in games and logic puzzles, quickly followed by the Artificial Intelligence research domain. Permutations games are single-player games. One of the first permutation games/problems was the *Taquin*, also named the Fifteen Puzzle (about 1870 [4]). It is a sliding-puzzle composed of a 4×4 box containing 15 numbered squares tiles, arranged in a random order. The goal is, using sliding moves (thanks to the empty space left by the 15 tiles in the 16-places square), to place the tiles in increasing order, from left to right and downwards. There are some variations of this game, with a larger or fewer number of tiles.
In permutation problems, we are looking for a strategy, consisting of a sequence of moves with the help of which we can reach the goal configuration.
It is absolutely possible, depending on the problem, that the possible moves do not allow to reach a given goal, and thus this goal may be unreachable. But in this paper, we will only consider problems with a reachable goal, this means solvable problems.

There are several possible approaches for solving permutation problems. The existent permutation problem solvers always focused on finding optimal strategies, which are strategies with a shortest move sequence. But the problem of this approach is that it needs exponential runtime. The nearest-to optimum way to solve them seems to be the use of heuristic search algorithms like A* ([2]) or IDA* ([3]) together with relevant heuristics [**?**]. A* algorithm is an extension of Edgar Dijkstra's 1959 algorithm, and achieves better performance by using heuristics combined with a best-first search.

But the algorithm we are interested in here belongs to the class of the suboptimal algorithms, but has the interesting property to run in polynomial runtime. It is based on the assumption that the length of the move-sequence to the goal is less important than the speed with which we can find it.

The contribution of this thesis is the investigation of a suboptimal but polynomial algo-

rithm for solving general permutation problems. This algorithm has been proposed by Sirs Furst, Hopcroft & Luks in 1980 ([1]) in the context of a proof on group appartenance, but has not been implemented and evaluated so far. The idea of the algorithm is to place iteratively each object on its goal place, letting every previously successfully placed objects fix. It uses the fact that permutations are cyclic. We have implemented this algorithm and evaluated it on some benchmark problems: the pancake sorting problem and the TopSpin puzzle. The results show that the strategies selected by the algorithm are non-optimal, but not very far from the optimal ones. The results show also that the algorithm is quick, which correlates with our expectations.
The remainder of the thesis is organized as follows. In chapter 2, we provide the preliminaries for this work. In chapter 3, we explain the algorithm and the strategy selection, and we go through an example. In chapter 4 we detail the experiments and the results. In chapter 5, we sum up the discussion and propose further working directions and possibilities.

Chapter 2

Preliminaries

2.1 Permutation Problems

Formally, what is a Permutation Problem ?
A permutation problem is a problem in which we want to transform a world from a world state (*initial state*) into another one (*goal state*) using *generators* (the elementar permutations, the *possible moves*).

Definition (Permutation Problem)
A permutation problem is a tuple (O, W_0, W_f, G), where:

- O is a set of n objects to rearrange
- W_0 is the initial state of the world W (W is a vector consisting of n places, labeled W[0] to W[n])
- W_f is the to goal state of the world to reach
- G is the set of possible actions, the generators

It is important to know that $generator(state) = state$
We are looking for a strategy S, consisting of a sequence (i.e. a composition) of generators with the help of which we can reach the goal configuration. Mathematically, this is equivalent to: $S = \prod^k G_i$ (composition of generators) and $S(W_0) = W_f$ (the strategy transforms the initial world state in the goal world state). This means S = $G_i(G_j(G_k(...(W_0)...))$. There can be more than one possible strategy.

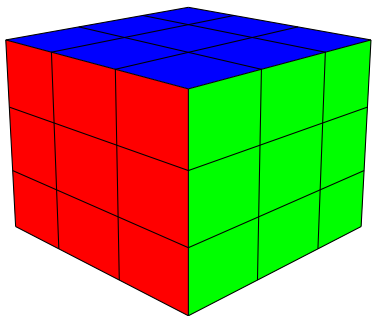
The optimal strategy $S_{optimal}$ is the strategy with the shortest move sequence. It would be in our notation $S_{optimal} = \prod^k G_i$ with k minimal (i.e. with the minimal sequence of moves) and $S(W_0) = W_f$.

We will see in the next sections how we model all these components.

2.2 Modelling

A world W is represented as a vector (each place assimilated to a label). This will allow us to use special notations for permutations, which will dramatically simplify the task. The object set O is represented as different objects named O_1 to O_n. Each one is placed in a unique location of W.

Let's see how we are going to model a world into a vector:
Example:
Let's represent the 3×3 Rubik's Cube world in a model. We choosed the Rubik's Cube to show how we are going to model any problem, because it is a worldwide-known problem, and its 3-dimensionality will show best the method to represent any problem world as a 1-dimensional vector.
Here is the famous Rubik's Cube (invented in 1974 by the hungarian Ernõ Rubik):



The 3×3 Rubik's Cube is composed of $3 \cdot (3^2) - 1 = 26$ (there is no central cube) cubes, which all seem to be able to move on each face of the Cube and seem not to be bound to the Cube without even falling apart. In fact, there is an axis system in the center of it, patented by Rubik, which allows the small cubes to rotate around the Cube. There are in total $6 * 9 = 54$ colored faces on the cube (the inner faces can never be seen). The goal of this puzzle is to reconstitute the Rubik's Cube as above, with each 3×3 face of the same color. But how could a 3-dimensional problem world be represented in a (1-dimensional) Vector ? For this purpose, we need to label each small face. Let's describe for instance the front face as follow:

1	2	3
4	5	6
7	8	9

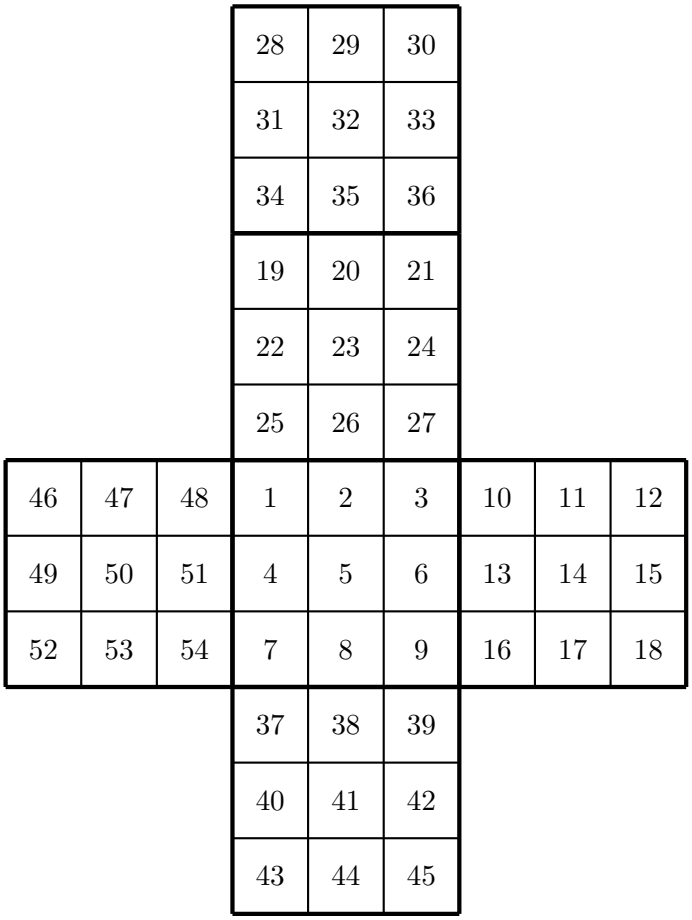
Then, the right-face will be:

10	11	12
13	14	15
16	17	18

And so on, until the down-face. We will then have labelled 54 places.

With that, the 3×3-Rubik's Cube World is represented.
Plus, as it is now represented as an array, we will be able to represent every state of W, W_i, as a vector of length 54. The first place of the array will represent the first place of the start-cube, and will contain the number of the current subface on it, and likewise for the second, third places, and so on.
For example: Let's consider an already solved Cube as start Cube. The 1-9 face (front face) will for instance be all red, the 10-18 face (right face) all green, the 28-36 face (back face) all white and the 46-54 face (left face) all black. If we rotate the first row of the front face left, we will get three green squares and six red ones on the front face. And as the labels correspond to the colors, the new World-representation-vector after this first move, W_1, will be as follows:
⟨10,11,12,4,5,6,7,8,9,36,35,34,13,14,15,16,17,18,25,22,19,26,23,20,27,24,21,28,29,30,
31,32,33,48,47,46,37,38,39,40,41,42,43,44,45,1,2,3,49,50,51,52,53,54⟩
The 3x3 Rubik's Cube World is represented, but not yet ready to be used.
It lacks on **generators** (this will come in next section).

Here the labelled 3x3 Rubik's Cube in 2D-representation:



2.3 Permutation Representation

There exist several different notations for representing a permutation. In the following, we will introduce three of them:

→ The Cauchy's two-line notation

→ The one-line notation

→ The cycle notation

2.3.1 The Cauchy's two-line notation

Let's take an example to explain it:

$$p = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix}$$

The first line of the two-line notation from Cauchy represent the *locations*. In the second line we can find each location's image according to the permutation.
For instance, if we want to apply p on a set $s = \{3\ 4\ 2\ 1\}$, we will have:

$p(3) = 2$, $p(4) = 3$, $p(2) = 4$, $p(1) = 1$
which means
"object 3 (in location 1) goes to the location 2",
"object 4 (in location 2) goes to the location 3",
"object 2 (in location 3) goes to the location 4" and
"object 1 (in location 4) goes to the location 1"

$p(s) = p(\langle 3, 4, 2, 1\rangle) = \langle 1, 3, 4, 2\rangle$

2.3.2 The one-line notation

The one-line is just the second line of the Cauchy two-line notation. It is true that the first line of the two-line notation is not really important, if we admit that it is always 1..n, the numbering of the locations. The one-line notation corresponding to the example in the two-line notation would be:
(2 3 4 1)

2.3.3 The cycle notation

This other notation uses the cyclic nature of permutations and expresses it as a product of cycles. Those cycles correspond to the "orbits" of the permutation. To begin, here is an example that illustrates the cyclic nature of permutations:
We will use here the one-line notation. We consider a world w_o and a permutation p
$w_0 = \langle 1, 2, 3, 4\rangle$
$p = (3\ 4\ 1\ 2)$
$p(w_0) = w_1 = \langle 3, 4, 1, 2\rangle$
$p(w_1) = p(p(w_o)) = \langle 1, 2, 3, 4\rangle \rightarrow p(p(w_0)) = w_0$
After a finite number of moves (here, 2), we are back in the same position. Then, the permutation is cyclic. The same will work with any permutation, but it might take longer than two moves.

The cycle notation:
The example above will be noted as follows in cycle notation: $p = (1\ 3)(2\ 4)$
Each "cycle", we will call it **subpermutation**, is represented in parenthesis, and is a permutation, namely the one mapping the values in itself to the following values in itself; and mapping the other (non-mentionned in the subpermutation) objects into themselves. For the first subpermutation of p, p_1, we have $p_1 = (1\ 3)$. This means the object on the place 1 will go to place 3, and as it is a cycle, the object on place 3 will go to place 1 (the next value for the last value is the first one).
On another example, $p = (1\ 2\ 3)$, object on place 1 will go to place 2, object on place 2 to place 3 and object on place 3 to place 1.

Properties:

i) the cycles are all **disjunct** (since they are part of a unique permutation)

ii) $(a\ b\ c)$ is fully equivalent to $(b\ c\ a)$ or even $(c\ a\ b)$ (only order is important in the cycle)

iii) $(a\ b)(c\ d)$ is fully equivalent to $(c\ d)(a\ b)$ (resulting directly from property i), as cycles are disjunct)

iv) the identity permutation, which maps all locations to themselves, can simply be noted () or id

v) a subpermutation will be at least from length 2, otherwise it is just an identity permutation

vi) $((a\ b\ c)(d\ e))^{-1} = (c\ b\ a)(e\ d)$ (just invert the order to get the inverse of the permutation). This means $(a\ b)^{-1} = (b\ a) = (a\ b)$, following property ii).

Concatenation/Sequencing:

During our future calculations, we will need to concatenate permutations. To concatenate two permutations, an easy method is to take a standard-world, to apply both permutations and to abstract the concatenation from the result. We will write it then from right to left (order is here important). This means $p_2.p_1(w_0)$ will be $w_1 = p_1(w_0)$ followed by $w_2 = p_2(w_1)$.
With $w_0 = \{a\ b\ c\}$, $p_1 = (1\ 3)$ and $p_2 = (2\ 3\ 1)$:
$\rightarrow w_2 = p_2.p_1(w_0) =?$
$p_1(w_0) = \{c\ b\ a\} = w_1$
$p_2(w_1) = \{a\ c\ b\} = w_2$
Then, we compare w_2 with w_0, and it appears that $p_2.p_1 = (2\ 3)$, which is much more compact than $(2\ 3\ 1).(1\ 3)$, and respects the cycle notation.

Inverse of a concatenation:
The inverse $(p_2.p_1)^{-1}$ of a concatenation of permutations is simply the concatenation of the inverses of the permutations, in reversed order:

Proof. $(p_2.p_1)^{-1} =?$
We know that $p.p^{-1} = id$ for any permutation p, then we have:
$p_?.p_2.p_1 = id$
$p_1^{-1}.p_2^{-1}.p_2.p_1 = p_1^{-1}.(p_2^{-1}.p_2).p_1 = p_1^{-1}.id.p_1 = p_1^{-1}.p_1 = id$ □

What are the advantages of using the cycle notation ?

- We just need to note the moving objects, while Cauchy's notation must always be complete.
- The property vi) is very useful to avoid complicated calculations for the inverses.
- The concatenation method makes thing much more clear (although it would be possible too with the Cauchy notation)

Chapter 3

The Algorithm & Strategy selection

3.1 The idea within the algorithm

Because of the cyclic nature of permutations (see 2.3.3), it might be comfortable to accept to introduce cycles in the strategy sequence, if it makes it possible to avoid extra-calculations. This means, cycles will have no impact on the accuracy of the strategy.
(Consequently, a way to improve the output-strategy could be to transform the strategy after using the suboptimal algorithm, by detecting cycles in it).
The basic idea of the sift algorithm is to place iteratively each object on its goal location, letting every successfully placed objects (from the former steps) fix. The problem size decreases then virtually in each step.

We have choosen the cycle notation for permutations (2.3.3), as it is very handy and has several useful properties (2.3.3).

3.2 The sift-algorithm

The sift algorithm was proposed by Sirs Furst, Hopcroft and Luks in 1980, within a mathematical proof on group appartenance ([1]). Its purpose is to produce a permutation-matrice that will allow us to reach any goal state ou of this start position. This means: a problem $\rightarrow$ a permutation-matrix, for whatever we want as a goal state. We will name the permutation-matrice **M**, the Matrice Expand of the generator set G.
Each cell $c_{i,j}$ of M, in row i and column j, will represent a particular action:
Letting all objects successfully placed in rows 1 to i-1 fix, map the object on location j on location i. (this means that we want to be able to map an object to the place i, using a permutation of this row, letting all previous successfully placed objects fix on their places)
As a result, we will have an upper-triangular matrice, with *id*'s on the diagonal (mapping j to i where i==j is nothing more than to do nothing):

$$M = \begin{pmatrix} id & * & ... & & \\ 0 & id & * & ... & \\ 0 & 0 & id & * & ... \\ 0 & ... & ... & ... & ... \\ 0 & ... & ... & 0 & id \end{pmatrix}$$

It is an upper triangular matrice, because after each step of the strategy selection (i.e. after each choice of permutation, iteratively on the rows), an object more stays fix (because it is now in its goal location). This means we have reduced the problem size for the next subgoal and no more object needs to be placed on the location were we fixed the object.

In order to fill this upper-triangular matrice, we need the *sift-procedure*:

For a given permutation x, we will go iteratively over the rows. In each row, we will check whether there exists a permutation y that does the same action than the permutation to add or not. If it is the case, we have to go further in the iteration, and check with next row, but this time not for just x, but $y^{-1}.x$, in order to conserve the work accomplished in the last row (and then as it is iterative, in every other preceding rows). If not, then it means that we can insert it in the row, at the column j where we checked if object at location j could already be placed at location i (the row index), with any permutation of the row.

Here in pseudocode:
sift(x):
i $\leftarrow 0$
while ($i \neq n-1$ **and**
there is a y in row i such that
y and x map $i+1$ to the same letter)
do
$i \leftarrow i+1$
$x \leftarrow y^{-1}x$
if x is not a member of row i
then insert x in row i

We have now the sift-algorithm, it is time to populate the matrice. This can be done within two steps:

1. Sift all generators.
2. Close the table, by running through all pairs (x,y) from the table, and sift their product.

M. Furst, J. Hopcroft and E. Luks have showed in their work that the entire algorithm's running time is $O(n^6)$. It is because the coset representatives in M is at most n^2, and therefore the number of calls to sift is at most $(n^2)(n^2)=n^4$. And, as the sift of a permutation is maximal $O(n^2)$ itself, the entire algorithm will be $O(n^4.n^2)=O(n^6)$ in the worst

case, a polynomial in n ([1]).

Now we have a matrice which contains actions which allow to reach any goal state from the start position. It is time to extract a strategy out of it.

3.3 Strategy selection

Now that we have a Matrice **M** which allows us to reach every possible goal state, we are ready to go. We know W_f, the goal state to reach. The selection of the Strategy **S** will be straight forward, and as follows:

iteratively on i:
For each line i, select the permutation that will bring the object that should be at $W_f[i]$ from its current location to the location i.
Per definition, it will let all the objects from $W_f[0]$ to $W_f[i-1]$ fix (thanks to the "$y^{-1}x$" hat-trick in the sift-procedure).

3.4 Example

Here we will go through an example for the pancake problem with 5 pancakes 4.1.1. It will illustrate and clarify everything we previously explained.
Pancakes will be represented by p_i, with p_1 the biggest pancake and p_5 the smallest pancake. The goal state will be a pyramid of pancakes, with the biggest pancake at the bottom and the smallest at the top. This means $W_f = \begin{bmatrix} p1 & p2 & p3 & p4 & p5 \end{bmatrix}$.
Let's take a random start state $W_0 = \begin{bmatrix} p5 & p3 & p1 & p2 & p4 \end{bmatrix}$.
The generators will be as follows:

- $g_1 = (4\ 5)$ (flip the two top-pancakes)
- $g_2 = (3\ 5)$ (flip three pancakes)
- $g_3 = (2\ 5)(3\ 4)$ (flip four pancakes)
- $g_4 = (1\ 5)(2\ 4)$ (flip all five pancakes)

We then begin with M:

$$M = \begin{pmatrix} id & 0 & 0 & 0 & 0 \\ 0 & id & 0 & 0 & 0 \\ 0 & 0 & id & 0 & 0 \\ 0 & 0 & 0 & id & 0 \\ 0 & 0 & 0 & 0 & id \end{pmatrix}$$

The populating method says:

1. Sift all generators.
2. Close the table, by running through all pairs (x,y) from the table, and sift their product.

Let's sift all generators to begin.
sift(g_1):
$i = 0$
$i \neq 4$ AND there is a y in row i such that y and g_1 map the same letter to 1 (g_1 maps 1 to 1, and id does the same) $\surd$
$i = 1$
$x = y^{-1}.x = id^{-1}.g_1 = g_1$
$i \neq 4$ AND there is a y in row i such that y and g_1 map the same letter to 2 (g_1 maps 2 to 2, and id does the same) $\surd$
$i = 2$
$x = y^{-1}.x = id^{-1}.g_1 = g_1$
$i \neq 4$ AND there is a y in row i such that y and g_1 map the same letter to 3 (g_1 maps 3 to 3, and id does the same) $\surd$
$i = 3$
$x = y^{-1}.x = id^{-1}.g_1 = g_1$
$i \neq 4$ AND there is a y in row i such that y and g_1 map the same letter to 4 (g_1 maps 5 to 4, and there is no other permutation that does the same) $\times$
g_1 is not a member of row 3 $\surd$
then insert g_1 in row 2, column (mapping from 4)-1.

$$M = \begin{pmatrix} id & 0 & 0 & 0 & 0 \\ 0 & id & 0 & 0 & 0 \\ 0 & 0 & id & 0 & 0 \\ 0 & 0 & 0 & id & g_1 \\ 0 & 0 & 0 & 0 & id \end{pmatrix}$$

sift(g_2):
$i = 0$
$i \neq 4$ AND there is a y in row i such that y and g_2 map the same letter to 1 (g_2 maps 1 to 1, and id does the same) $\surd$
$i = 1$
$x = y^{-1}.x = id^{-1}.g_2 = g_2$
$i \neq 4$ AND there is a y in row i such that y and g_2 map the same letter to 2 (g_2 maps 2 to 2, and id does the same) $\surd$
$i = 2$
$x = y^{-1}.x = id^{-1}.g_2 = g_2$
$i \neq 4$ AND there is a y in row i such that y and g_2 map the same letter to 3 (g_2 maps 5 to 3, and there is no other permutation that does the same) $\times$
g_2 is not a member of row 2 $\surd$
then insert g_2 in row 2, column (mapping from 3)-1.

$$M = \begin{pmatrix} id & 0 & 0 & 0 & 0 \\ 0 & id & 0 & 0 & 0 \\ 0 & 0 & id & 0 & g_2 \\ 0 & 0 & 0 & id & g_1 \\ 0 & 0 & 0 & 0 & id \end{pmatrix}$$

sift(g_3):
$i = 0$
$i \neq 4$ AND there is a y in row i such that y and g_3 map the same letter to 1 (g_3 maps 1 to 1, and id does the same) $\surd$
$i = 1$
$x = y^{-1}.x = id^{-1}.g_3 = g_3$
$i \neq 4$ AND there is a y in row i such that y and g_3 map the same letter to 2 (g_3 maps 5 to 2, and there is no other permutation that does the same) $\times$
g_3 is not a member of row 1 $\surd$
then insert g_3 in row 1, column (mapping from 2)-1.

$$M = \begin{pmatrix} id & 0 & 0 & 0 & 0 \\ 0 & id & 0 & 0 & g_3 \\ 0 & 0 & id & 0 & g_2 \\ 0 & 0 & 0 & id & g_1 \\ 0 & 0 & 0 & 0 & id \end{pmatrix}$$

sift(g_4):
$i = 0$
$i \neq 4$ AND there is a y in row i such that y and g_4 map the same letter to 1 (g_4 maps 5 to 1, and there is no other permutation that does the same) $\times$
g_4 is not a member of row 0 $\surd$
then insert g_4 in row 0, column (mapping from 1)-1.

$$M = \begin{pmatrix} id & 0 & 0 & 0 & g_4 \\ 0 & id & 0 & 0 & g_3 \\ 0 & 0 & id & 0 & g_2 \\ 0 & 0 & 0 & id & g_1 \\ 0 & 0 & 0 & 0 & id \end{pmatrix}$$

All generators sifted. The exact same method will be used to close the table by sifting all possible concatenation of the entries of the matrice (which are for now only generators).
sift(g_4g_1):
$g_4g_1 = (4\ 1\ 5\ 2)$ (thanks to the concatenation method). $i = 0$
$i \neq 4$ AND there is a y in row i such that y and g_4 map the same letter to 1 (g_4 maps 4 to 1, and there is no other permutation that does the same) $\times$
g_4 is not a member of row 0 $\surd$
then insert g_4 in row 0, column (mapping from 1)-1.

$$M = \begin{pmatrix} id & 0 & 0 & g_4g_1 & g_4 \\ 0 & id & 0 & 0 & g_3 \\ 0 & 0 & id & 0 & g_2 \\ 0 & 0 & 0 & id & g_1 \\ 0 & 0 & 0 & 0 & id \end{pmatrix}$$

It would have been very long and uninteresting to do the whole table closure here. So, here is how it looks like at the end:

$$M = \begin{pmatrix} id & g_4g_3 & g_4g_2 & g_4g_1 & g_4 \\ 0 & id & g_3g_2 & g_3g_1 & g_3 \\ 0 & 0 & id & g_2g_1 & g_2 \\ 0 & 0 & 0 & id & g_1 \\ 0 & 0 & 0 & 0 & id \end{pmatrix}$$

Now we are able to extract the strategy. In row 0, we will choose g_4g_2, because it maps the object on location 3 (p_1) to the location 1.
$g_4g_2 = (1\ 5\ 3)(2\ 4)$ (thanks to the concatenation method)
$W_1 = g_4g_2(W_0) = g_4g_2(\begin{bmatrix} p5 & p3 & p1 & p2 & p4 \end{bmatrix}) = \begin{bmatrix} p1 & p2 & p4 & p3 & p5 \end{bmatrix}$ p_1 is now placed. It will stay there.

In row 1, we will choose id, because $p2$ is already well placed.

In row 2, we will choose g_2g_1, because it maps the object on location 4 (p_3) to the location 3.
$g_2g_1 = (3\ 4\ 5)$ (thanks to the concatenation method)
$W_2 = g_2g_1(W_1) = g_2g_1(\begin{bmatrix} p1 & p2 & p4 & p3 & p5 \end{bmatrix}) = \begin{bmatrix} p1 & p2 & p3 & p5 & p4 \end{bmatrix}$ p_3 is now placed. It will stay there.

In row 3, we will choose g_1, because it maps the object on location 5 (p_4) to the location 4.
$W_3 = g_1(W_2) = g_1(\begin{bmatrix} p1 & p2 & p3 & p5 & p4 \end{bmatrix}) = \begin{bmatrix} p1 & p2 & p3 & p4 & p5 \end{bmatrix}$ every object is now placed.

In row 4, we choose id, because there is no other choice.

The strategy is then: S = $g_1.g_2.g_1.g_4.g_2$
The found strategy has length 5.
We compared with the Pancake Solver from Gabriele Roeger ([**?**]), which found the optimal solution with length 4.
It would have been $S_{optimal} = g_4.g_1.g_2.g_3$
This shows the non-optimality of the solver (we will come back to this point later).

Chapter 4

Experiments

4.1 The Models

There were lots of models with which we could test our implementation. We choosed the Pancake Problem and the TopSpin Puzzle, with different problem sizes, because they are common benchmark problems, and thus it would be easy to compare the results with another existing solver.

4.1.1 The Pancake Problem

The Pancake Problem is a sorting problem. The start world is a stack of n pancakes, all from different sizes. The only possible actions are to flip the k top pancakes with a spatula. The goal is, starting from a random position, to make a pyramid of pancakes, ie to order them in increasing order.

We decided to model it as follows:
Let n be the number of pancakes. The pancakes will be represented by p_i with p_1 the biggest pancake and p_n the smallest pancake. The goal state to reach will be
$W_f = \begin{bmatrix}1 & 2 & 3 & ... & n\end{bmatrix}$
For this purpose, we will use the generators. These are $(n-1\ n)$ (flip the two pancakes on the top of the pile), $(n-2\ n)$ (flip the three pancakes on the top of the pile), $(n-3\ n)(n-2\ n-1)$ (flip the four pancakes on the top of the pile), and so on, ...

4.1.2 The TopSpin Puzzle

Original TopSpin Puzzle:
The TopSpin Puzzle was invented by Ferdinand Lammertink, and patented on 3 Oct 1989. "It consists of 20 numbered round pieces in one long looped track. You can slide all the pieces of the loop along. There is also a turntable in the loop which can rotate any four adjacent pieces so that they will be in reverse order. This in effect swaps two adjacent pieces and the two pieces on either side of them. The aim is of course to place the pieces in numerical order. In this game, as there are 20 numbered pieces, there are then 20! possible positions." (source: http://www.jaapsch.net/puzzles/topspin.htm)

We will focus on smaller TopSpin Puzzles ($n = 3..7$), with different sized turntables ($size = 2..4$).
Note that there exist some unsolvable TopSpin Puzzles ([**?**]).

To model the TopSpin Puzzle with n pieces and a k-sized turntable, we will do as follows: The pieces will be labeled o_i with i from 1 to n. The world's start configuration could be for instance $\begin{bmatrix} o_3 & o_2 & o_1 & o_4 & o_6 & o_5 \end{bmatrix}$ for $n = 6$. The generators would be:
- Turn all the pieces clockwise around the ovale : (1 2 3 ... n)
- Rotate the turntable clockwise : ($n-k$ $n-k+1$ ... n)
- Rotate the turntable counter-clockwise : (n $n-1$ ... $n-k$)

4.2 Results

Calculations were made on the base of 100 random start positions. We calculated the average time needed to find a solution and the average solution length. Every calculation was made 10 times and we took the average values of them.

I) Pancake Problem:

n	av. time (ms)	av. solution length (moves)
3	0.90	1.42
4	2.3	2.6
5	6.44	4.16
6	15.81	5.6
7	36.52	8.79
8	81.51	10.75
9	167.61	14.96
10	311.2	12.3

II) TopSpin Puzzle:

n	turn table size	av. time (ms)	av. solution length (moves)
4	2	3.16	7.83
5	2	8.02	30.21
5	3	8.33	44.81
7	2	43.61	116.86
7	3	64.81	348.23

Note: by the TopSpin Puzzle the variance in the average solution lengthes was big (sometimes +/- 5 moves). By this problem there is an implementation problem: the strategy would correctly be selected, but the program is not able to convert the strategy in a succession of generators (it contains inverses of permutations). The strategies lengthes are therefore not representative.

In order to be able to compare the results, and specifically the length of found solutions, with the Pancake Solver from M. Helmert and G. Roeger we have tested specific permutation problems on both solvers. The goal was to estimate the "distance" from our solver to an optimal solver.
$n = 3$: Start world state: (o_3 o_1 o_2). Our solution: g2.g1 (length 2). Optimal solution: g2.g1 (length2)
$n = 4$: Start world state: (o_4 o_1 o_3 o_2). Our solution: g1.g2.g1.g3.g2 (length 5). Optimal solution: g3.g2.g1 (length 3)
$n = 5$: Start world state: (o_5 o_3 o_1 o_2 o_4). Our solution: g1.g2.g1.g4.g2 (length 5). Optimal solution: g4.g1.g2.g3 (length 4)
$n = 6$: Start world state: (o_6 o_3 o_5 o_2 o_4 o_1). Our solution: g2.g1.g3.g2.g4.g3.g5 (length 7). Optimal solution has length 6.
Naturally, it could be of total fortuity, but we can see here that our solver finds quite accurate solutions.

4.3 Discussion

As the results are averages from averages (except for the comparison with the pancake solver from M. Helmert and G. Roeger), they are quite reliable. Variances were always very small (it means that the algorithm is stable and the method sure), except for the average solution lengthes for the TopSpin Puzzle. This shows that the problem is not very stable, and that variations in the start position can lead to complications in the solving.

For the Pancake sorting problem, the theory has shown that the maximal number of flips required to sort any stack of n pancakes must lie between $\frac{15}{14}n$ and $\frac{18}{11}n$ (source: $http://en.wikipedia.org/wiki/Pancake\_sorting$). As we can see here, the algorithm lies between this range.

Chapter 5

Conclusion

In this thesis, we have investigated a suboptimal algorithm for solving general permutation problems. The results show that the strategies selected by the algorithm are non-optimal, but not very far from the optimal ones. The results show also that the algorithm is quick, which correlates with our expectations.

The big length of solutions for the TopSpin Puzzle is due to the fact that lots of cycles are done.
Consequently, a way to **improve** the output-strategy could be to **transform the strategy after using the suboptimal algorithm, by detecting and deleting cycles in it**.

Everything has been correctly implemented and the algorithm delivers correct solutions for solvable problems. The only problem lies in the transformation of the strategies in sequences of generators. Despite this fact, we reached the goals we had been fixing to ourselves : the algorithm is quick but not optimal (speed: see the results; non-optimality: see the example).

Future work: As above explained, a post-algorithm cycle-detection unit could be a big improvement for the program, especially for the TopSpin Puzzle, in which there were lots of cycles.

Bibliography

[1] Merrick Furst, John Hopcroft, and Eugene Luks. *Polynomial-Time Algorithms for Permutation Groups*. Department of Computer Science (Cornell University), Ithaca, New York 14853, 1980.

[2] P. E. Hart, N. J. Nilsson, and B. Raphael. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE, 1968.

[3] Richard Korf. *Depth-first Iterative-Deepening: An Optimal Admissible Tree Search*. Artificial Intelligence 27: 97109, 1985.

[4] Jerry Slocum and Dic Sonneveld. *The 15 Puzzle book*. Slocum Puzzle Foundation, 2006.

Other sources (we got a problem with bibTex on the last minute):
M. Helmert and G. Roeger, "Relative-Order Abstractions for the Pancake Problem", 2010
S. Kaufmann, "A Mathematical Analysis Of The Generalized Ovale Track Puzzle", 2011