TopSpin Puzzle Solution

Well, as we now know, this was not an easy thing to do. Let us start with a straightforward, simple solution that can solve a six-piece problem (such as turning \([4,8,3,1,5,2,6,7]\) into \([1,2,3,4,5,6,7,8]\)) easily but would need lots of time and space for the original problem.

The \(h'\) function simply computes how many pieces are not followed by their correct right neighbor (i.e., piece \((n + 1)\) for a given piece \(n\)). This test wraps around the list, i.e., it tests whether the first element is compatible with the last one in this regard. If all elements have their correct right neighbor, then we know that the pieces are already ordered correctly and just need to be shifted into the correct position. So if we are already in the goal state, then \(h'(p) = 0\), otherwise \(h'(p) = 1\). If there are incorrect neighbors, then we need to note that one 180° flip (or reversal) changes the right neighbors of five pieces (the four pieces being flipped and the one immediately to their left). So, as many as five mismatching neighbors could be fixed within a single move. In order to obtain an optimistic estimator, we thus need to divide the number of mismatches by five and add it to \(h'(p)\). This is not a very sophisticated function. In order to devise a better one, we would have to do a more thorough mathematical analysis of this problem. Anyway, here it is (see also file “TopSpin8.hs” on the course homepage:

```haskell
h'Spin :: [Int] -> [Int] -> Double
h'Spin state goal
  | state == goal = 0
  | otherwise = 1 + 0.2*mismatches (last state) state
  where mismatches _ [] = 0
        mismatches x (y:ys)
          | y == (mod x $ length state) + 1 = mismatches y ys
          | otherwise = 1.0 + mismatches y ys

The genStatesSpin function assumes that the “flipping range” consists of the first four elements in the list. So it first generates the flipping move and then appends all possible shifting moves:

```haskell
genStatesSpin :: [Int] -> [[Int]]
  where shiftStates [x] _ = []
          shiftStates (x:xs) ys = (xs ++ ys ++ [x]):shiftStates xs (ys ++ [x])
```

Well, before we can compute a solution, we must adapt our aStar3 and aStarID functions to floating-point values of \(h'\), and for better efficiency we also want to hard code the solution to be \([1, 2, 3, \ldots, n]\):

```haskell
aStar3 :: [Int] -> ([Int]->Float) -> ([Int]->[[Int]]) -> Int -> [[Int]]
aStar3 start h' genStates maxDepth =
  expand [(h' start, [start]), (h' (spin start) + 1, [spin start, start])]
  where expand [] = []
        expand ((score, path):nodes)
          | fromIntegral (length path) == score + 1 = reverse path
          | length path > maxDepth = expand nodes
          | otherwise = expand $ sortBy (compare `on` fst) (nodes ++ newNodes)
  where newNodes = [(fromIntegral (length path) + h' state, state:path)
                      | state <- genStates $ head path, state `notElem` path]

aStarID start h' genStates maxDepthID = deepen 0
  where deepen depth
    | depth > maxDepthID = []
    | aStarResult /= [] = aStarResult
    | otherwise = deepen (depth + 1)
  where aStarResult = aStar3 start h' genStates depth
With these functions, we can at least solve some eight-piece problems such as this one:

\[
\text{solutionSpin} = \text{aStarID} \ [4, 8, 3, 1, 5, 2, 6, 7] \ h'\text{Spin} \ \text{genStatesSpin} \ 7
\]

This is the solution we get:

\[
[[4, 8, 3, 1, 5, 2, 6, 7], [1, 3, 8, 4, 5, 2, 6, 7], [1, 7, 6, 2, 3, 8, 4, 5], [8, 3, 2, 6, 4, 5, 1, 7], [2, 3, 8, 7, 6, 4, 5, 1],
[6, 7, 8, 3, 4, 5, 1, 2], [1, 5, 4, 3, 2, 6, 7, 8], [2, 3, 4, 5, 6, 7, 8, 1]]
\]

In order to solve the original eight-piece problem, we need to further optimize our program (unless we want to optimize the \( h' \) function instead). The main step is to realize that we need to perform an alternating sequence of shift and spin moves; it does not lead to an optimal solution if we perform multiple shifts in a row or multiple spins in a row. So we can define one move to be a shift followed by a spin.

We may want to be able to start the solution with a spin, and therefore we start the search with two open nodes, one being the start state and the other one the state that we reach by spinning the start state. Another problem is that after executing these double moves, we may only be one shift away from the goal. To resolve this issue, we just define all those states that are only one shift away from the goal as goal states, i.e., their \( h' \) = 0. Once we get there, it is trivial to make the final shift to solve the puzzle. As a final optimization, we no longer specify the goal state but assume that it is always \([1, 2, 3, \ldots, n]\). Here is the complete code that solves our eight-piece problem:

```haskell
import Data.List
import Data.Function

main = putStrLn $ show solutionSpin

solutionSpin = aStarID [4,8,3,1,5,2,6,7] h'Spin genStatesSpin 7

aStar3 :: [Int] -> ([Int]->Float) -> ([Int]->[[Int]]) -> Int -> [[Int]]
aStar3 start h' genStates maxDepth =
  expand [(h' start, [start]), (h' (spin start) + 1, [spin start, start])]
  where expand [] = []
    expand ((score, path):nodes)
      | fromIntegral (length path) == score + 1 = reverse path
      | length path > maxDepth = expand nodes
      | otherwise = expand $ sortBy (compare `on` fst) (nodes ++ newNodes)
      where newNodes = [(fromIntegral (length path) + h' state, state:path) |
        state <- genStates $ head path, state `notElem` path]

aStarID start h' genStates maxDepthID = deepen 0
  where deepen depth
    | depth > maxDepthID = []
    | aStarResult /= [] = aStarResult
    | otherwise = deepen (depth + 1)
    where aStarResult = aStar3 start h' genStates depth

h'Spin :: [Int] -> Float
h'Spin state = 0.2*mismatches (last state) state
  where mismatches _ [] = 0
    mismatches x (y:ys)
      | y == (mod x $ length state) + 1 = mismatches y ys
      | otherwise = 1.0 + mismatches y ys
```
genStatesSpin :: [Int] -> [[Int]]
genStatesSpin xs = map spin $ shiftStates xs []
    where shiftStates [x] _ = []
          shiftStates (x:xs) ys = (xs ++ ys ++ [x]):shiftStates xs (ys ++ [x])

spin :: [Int] -> [Int]

After starting this using GHC (you must then import Data.Function), taking a coffee break, and returning to our computer, we get the following result:

[[4,8,3,1,5,2,6,7],[1,3,8,4,5,2,6,7],[1,7,6,2,3,8,4,5],[8,3,2,6,4,5,1,7],[2,3,8,7,6,4,5,1],[6,7,8,3,4,5,1,2],[1,5,4,3,2,6,7,8],[2,3,4,5,6,7,8,1]]

By the way, you can speed up execution by making a multithreaded executable using GHC. When you compile, use the -threaded switch, and when you run the executable from the command line, use the arguments +RTS -N. This way – if possible - the execution of the program will be distributed across all cores of your system. The solution can also be found in “TopSpin8.hs” on the course homepage.