Question 1: Living in Another World

Imagine a world that is much more exciting than the one with the three toy blocks. This world consists of four beer bottles A, B, C, and D (Amstel, Becks, Corona, and Duff). They can be arranged in any order from left to right, except that bottle A can never be further to the right than bottle D. For example, ABCD, CBAD, and CADB are possible states of our world, whereas DCBA, CDAB, or BCDA can never occur. The world can be manipulated by the schema swap(x, y), which swaps the bottles in positions x and y. For example, swap(1, 2) turns state BCAD into CBAD. However, swap(1, 2), swap(2, 3), and swap(2, 4) are the only three available operators.

a) Draw the state-space graph of this world. You do not need to draw any bottles; just use four-letter sequences to describe states.

b) Assume that your world is in the state ADBC, but you would like it to be in state CBAD. Use best-first search to find a solution that requires a minimum number of operations. To do this, first define your estimation function \( f'(n) = g'(n) + h'(n) \). Then write down the resulting search tree, indicate the order in which nodes were created, and for each node \( n \) give the value of \( f'(n) \).
Simplest solution:

\[ f'(n) = \text{number of operations already performed} + \frac{\text{number of bottles in incorrect position}}{2} \]

Without the division by two it would not be an optimistic evaluator.

The number to the left of the letters indicates the order of creation, the number in brackets to the right is the value of \( f'(n) \) written as \( g'(n) + h'(n) \):

1: ADBC (0 + 2)  
   \[ \text{swap}(2, 3) \]
   2: ABDC (1 + 1.5)  
   \[ \text{swap}(1, 2) \]
   3: ACBD (1 + 1.5)  
   \[ \text{swap}(2, 4) \]
   4: BADC (2 + 2)  
   5: ACDB (2 + 2)  
   6: CABD (2 + 1)  
   7: ABCD (2 + 1)  
   8: CBAD (3 + 0)  
   \[ \text{done!} \]

The solution is:

\[ \text{swap}(2, 4) \]
\[ \text{swap}(1, 2) \]
\[ \text{swap}(2, 3) \]
**Question 2: Be a Game-Playing Computer**

It is your turn to do some of the alpha-beta pruning. The tree below indicates the complete Minimax tree for a particular problem (first move by MIN, then MAX, and then MIN again – notice that is different from our previous examples, where MAX started). The number at each leaf \( p \) indicates the value of the static evaluation function \( e(p) \) if it were computed at that leaf.

a) Now your job is to check the boxes under those leaves that do **not** need to be created and evaluated thanks to the alpha-beta pruning.

b) Which move (the left or right one) should MIN make, and why, i.e., what exactly is the advantage of making this move over making the other one?

MIN should make the left move, because it guarantees MIN a score of -2 or less, whereas the right move would lead to a score of 1 if MAX plays optimally.
Question 3: Making the Eight-Puzzle less Puzzling with A*

(a) The imperfection of the code is so severe that even the compiler or interpreter nudges you and gives you a friendly warning that the pattern matching for the expand function within aStar is incomplete. Clearly, it is not defined for an empty list, which it would encounter if there were no more nodes to expand, i.e., no solution was found for the given problem. So please write a new function aStar2 that fixes this problem and outputs an empty list if no solution was found.

```haskell
aStar2 :: Eq a => a -> a -> (a->a->Int) -> (a->[a]) -> [a]
  where expand [] = []
    expand ((score, path):nodes)
        | head path == goal = reverse path
        | head path > maxDepth = expand $ sortBy (compare `on` fst) (nodes ++ newNodes)
        | otherwise = expand $ sortBy (compare `on` fst) newNodes
    where newNodes = [(length path + h' state goal, state:path) |
                     state <- genStates $ head path, state `notElem` path]
```

(b) Unfortunately, you will notice that the compiler is happy now but that for an unsolvable problem (for example, try switching the first two numbers, 2 and 8, in the start state) you still do not receive an empty list. Depending on your compiler/interpreter and its parameters you will get a timeout, a stack overflow, or a desperate attempt to solve the problem that could take an eternity. It would in fact take a huge amount of time and space (memory resources) for the algorithm to determine that the problem has no solution. The issue here is that even though we avoid circuits during search, there are still too many nodes to be expanded. Not knowing at what depth to expect the solution, the algorithm searches deeper and deeper without success.

You probably have an idea of how to tackle this problem: iterative deepening. First write a function aStar3 that is just like aStar2 but receives a fifth input maxDepth :: Int that indicates the maximum depth for the search. The idea is to not create any new nodes whose depth is greater than maxDepth. Now test the new function aStar3. Finally, write a function aStarID that receives the same inputs as aStar3 but performs iterative deepening. It starts with a maximum search depth of 0, and if it does not find a solution, it performs a search with maximum depth 1, then 2, 3, and so on. This continues until it finds a solution, which is then provided as its output, or it performs all searches until maxDepth without finding a solution, which results in the empty list as its output.

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

aStarID :: Eq a => a -> a -> (a->a->Int) -> (a->[a]) -> Int -> [a]
  where deepen depth
            | depth > maxDepthID = []
            | aStarResult /= [] = aStarResult
            | otherwise = deepen (depth + 1)
    where aStarResult = aStar3 start goal h' genStates depth
```
Now that we have created this powerful \texttt{aStarID} function, let us make use of its generality and apply it to a different problem. We will consider two puzzles: The Bottle World puzzle from Question 1 and the TopSpin puzzle that will be explained below. You only have to solve one of them, but please feel free to do both to collect bonus points. Note that \textbf{CS470 students} already receive full points if they only solve the Bottle World puzzle, whereas \textbf{CS670 students} need to solve at least the TopSpin problem to get the full score. Again, both groups get extra points if they solve both puzzles.

As to the \textbf{TopSpin puzzle}, please take a look at the next 5 pages of this assignment. The author describes the puzzle very nicely and also its relationship to the shifting puzzles such as our eight-puzzle. Please do not worry about all the mathematics in that article; you only need to understand the rules of TopSpin. Let us consider the possibility that there are not exactly 20 pieces but that any number of pieces greater than 4 is allowed. For example, a state for an 8-piece TopSpin could be described by a list of 8 numbers, such as \([4,8,3,1,5,2,6,7]\), and its goal state would be \([1,2,3,4,5,6,7,8]\). Write an evaluation function \(h'\) for TopSpin puzzles with any number of pieces. Note that writing a good evaluation function is important and tricky here. For example, \([5,6,7,8,1,2,3,4]\) is only one step away from the goal. Then write a \texttt{genStatesSpin} function that generates the possible states after the next move, again considering any number of pieces. For an \(n\)-piece puzzle there are always \(n\) possible moves: \((n - 1)\) ways to move the pieces around the track, plus spinning the first (or last, whatever you like) four pieces. Finally, use \texttt{aStarID} to compute \texttt{solutionSpin}, listing the states from the 8-piece start state \([4,8,3,1,5,2,6,7]\) to the goal state.

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 \([3,2,5,1,6,4]\) into \([1,2,3,4,5,6]\)) 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 \texttt{“TopSpin6.fr”} on the course homepage:

\[
\text{h'Spin} :: \text{[Int]} \to \text{[Int]} \rightarrow \text{Double} \\
\text{h'Spin state goal} \\
\quad | \text{state == goal} = 0 \\
\quad | \text{otherwise} = 1 + 0.2 \times \text{mismatches (last state)} \times \text{state} \\
\text{where mismatches _ [] = 0} \\
\text{mismatches} \times \text{[(y:ys)} \\
\quad | \text{y == (mod x $ length state) + 1 = mismatches y ys} \\
\quad | \text{otherwise} = 1.0 + \text{mismatches y ys}
\]

The \texttt{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:
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’:

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

aStarID’ :: Eq a => a -> a -> (a->a->Double) -> (a->[a]) -> Int -> [a]
aStarID’ start goal h’ genStates maxDepthID = deepen 0  
  where deepen depth  
        | depth > maxDepthID = []
        | aStarResult /= [] = aStarResult
        | otherwise = deepen (depth + 1)
    where aStarResult = aStar3’ start goal h’ genStates depth

With these functions, we can at least solve the six-piece problem that was circulated by email:

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

This is the solution we get:

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

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, ..., n]. Here is the complete code that solves our eight-piece problem:
solutionSpin = aStarID [4,8,3,1,5,2,6,7] h'Spin genStatesSpin 7

aStar3 :: [Int] -> ([Int]->Double) -> ([[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] -> Double
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.
For the **Bottle World puzzle**, represent a state as a list of letters such as [‘B’, ‘C’, ‘A’, ‘D’] (or “BCAD” if you use Haskell or the packed/unpacked functions in Frege. Write functions h’Bottle and genStatesBottle and then use aStarID to compute solutionBottle, which is a list of states indicating a solution of Question 1b. Just the list of states is sufficient; you do not have to indicate the operations being used.

The solution can be found on the course homepage in the file “BottleWorld.fr”. The h’Bottle function computes the length of the list of all positional mismatches between the current state and the goal state and multiplies the result by 0.5:

\[
\text{h'}\text{Bottle} :: \text{[Char]} \rightarrow \text{[Char]} \rightarrow \text{Double} \\
\text{h'}\text{Bottle} \text{ state goal} = 0.5* (\text{fromIntegral} \; \text{length} \; \text{filter} \; (\lambda (x, y) \rightarrow x /= y) \; \text{zip} \; \text{state} \; \text{goal})
\]

The genStateBottle function uses (non-exhaustive) pattern matching to create a list of the three potential outcomes of the swap operations. However, this list still contains those illegal states in which the ‘A’ is to the right of the ‘D’. To remove these illegal states, we filter the list with the isLegal function. It simply goes through the state from left to right and checks whether it first hits an ‘A’ or a ‘D’, leading to output ‘True’ or ‘False’, respectively:

\[
\text{genStatesBottle} :: \text{[Char]} \rightarrow \text{[[Char]]} \\
\text{genStatesBottle} \; (w:x:y:z:[]) = \text{filter} \; \text{isLegal} \; [x:w:y:z:[], \; w:y:x:z:[], \; w:z:y:x:[]] \\
\quad \text{where} \; \text{isLegal} \; (\text{'A'}:xs) = \text{True} \\
\quad \text{isLegal} \; (\text{'D'}:xs) = \text{False} \\
\quad \text{isLegal} \; (\_ :xs) = \text{isLegal} \; xs
\]

We can then solve the problem as follows:

\[
\text{solutionBottle} = \text{aStarID'} \; ['A', 'D', 'B', 'C'] \; ['C', 'B', 'A', 'D'] \; \text{h'}\text{Bottle} \\
\text{genStatesBottle} \; 10
\]

It is (fortunately) the same solution we obtained by using paper and pencil in Question 1b:

["ADBC", "ACBD", "CABD", "CBAD"]