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 left than bottle D. For example, DCBA, CDAB, or BCDA are possible states of our world, whereas ABCD, CBAD, and CADB 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 BCDA into CBDA. Only adjacent bottles can be swapped, which means that swap(1, 2), swap(2, 3), and swap(3, 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 CBDA, but you would like it to be in state DACB. 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 \( f'(n) \).
**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?
Question 3: Making the Eight-Puzzle less Puzzling with A*

Below is some basic Haskell code for solving the eight-puzzle using A* as we did in class. You can also download it as “EightPuzzle.hs” from the Software section of the course homepage.

```haskell
import Data.List (sortBy)
import Data.Function (on)

main = putStrLn $ concat $ map showState8 solution

solution :: [[Int]]
solution = aStar [2,8,3,1,6,4,7,0,5] [1,2,3,8,0,4,7,6,5] h'8 genStates8

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

h'8 :: [Int] -> [Int] -> Int
h'8 state goal = length $ filter (\(x, y) -> x /= 0 && x /= y) (zip state goal)

genStates8 :: [Int] -> [[Int]]
genStates8 state = map newState $ swapLists!!(length $ takeWhile (/= 0) state)
    where swapLists = [[1,3],[0,2,4],[1,5],[0,4,6],[1,3,5,7],[2,4,8],[3,7],[6,4,8],[5,7]]
        newState pos = map (swap $ state!!pos) state
        swap p n = if n == p then 0 else if n == 0 then p else n

showState8 :: [Int] -> String
showState8 state = "\n+---+---+---+
| " ++ piece 0 ++ " | " ++ piece 1 ++ " | " ++ piece 2 ++ " |\n+---+---+---+
| " ++ piece 3 ++ " | " ++ piece 4 ++ " | " ++ piece 5 ++ " |\n+---+---+---+
| " ++ piece 6 ++ " | " ++ piece 7 ++ " | " ++ piece 8 ++ " |\n+---+---+---+
"
    where piece pos = if state!!pos == 0 then " " else show $ state!!pos
```

In this code, an eight-puzzle state is represented by a list of nine integers, which indicate the current content of each of the nine squares in reading direction, i.e., from the top row to the bottom row, and from left to right within each row. A zero stands for the empty square. At the top you can see the definitions of the start and goal states for the example we discussed in class. At the bottom, the `showState8` function turns a given state into a printable, artistic rendering of the eight-puzzle.

The `h'` function, estimating (optimistically) how many moves a given state is away from the goal, is implemented as `h'8` for the eight-puzzle. It goes through all pairs of corresponding board positions and counts how many pieces in the current state (except the empty piece) differ from those in the goal state.

The function `genStates8` generates a list of all states that can be reached from the given one within a single move. It uses mysterious `swapLists` to accomplish this, which contain, for any given empty position, a list of those positions from where pieces could be moved into it. For example, if the first (top-left) position is empty, we need to look at the first list in `swapLists`, which is `[1,3]`. Positions are indexed starting at 0, and thus positions 1 and 3 are the top-row-center and middle-row-left ones, respectively – exactly the ones from where you could move a piece into the empty position at the top-left. If the center position (4) is empty, we find the list `[1,3,5,7]`, indicating four possible moves.

To obtain the index of the empty position, in the first line of `genState` we first determine the position in the list where the 0 occurs, and then use the `!!` operator to obtain the corresponding element from `swapLists`. For each element in that list, we have to create a new state in which the indicated move
(which is a swap of the empty piece and a neighboring one) has been performed. The function \texttt{newState} achieves this by applying the function \texttt{swap} to each element in the current state. And \texttt{swap} turns an empty piece into the one it is to be swapped with and vice versa, while it leaves all other pieces unchanged. That is all we need to do.

Finally, there is the function \texttt{aStar} that implements the A* algorithm. In order to increase the flexibility and reusability of our code, the \texttt{aStar} function is implemented in its most general form, i.e., we can use this exact code to apply A* to \textit{any given problem}. Well, to be precise, right now we expect each move to have the same cost and \( h' \) to always yield integer values. And later you will see that there is a significant imperfection – but the good part about this is that \textit{you will fix it}!

Anyway, to keep the \texttt{aStar} function general, it expects from us a function \( h'(p) \) that receives an eight-puzzle state and returns an (optimistic!) estimate of how many moves we are away from the goal state. Furthermore, \texttt{aStar} requires a function \texttt{genStates} that generates all states that can be reached within a single move from the current (given) one. Fortunately, we already prepared the functions \texttt{h'8} and \texttt{genStates8} to turn \texttt{aStar} into an eight-puzzle solver. The only other inputs to \texttt{aStar} are the start and goal states, and its output is an optimal solution, i.e., the shortest sequence of consecutive states leading from the start to the goal.

The function \texttt{aStar} uses a helper function \texttt{expand} that takes as its input a list of all currently open nodes, where each node \( p \) is specified by a pair consisting of its score \( f'(p) \) and the path from \( p \) to the start state (the root of the search tree). This means that the first element in the path is the actual state at \( p \). We need the score to decide which node to expand next and the path to keep track of how we got to \( p \) so that once we reach the goal, we can provide the solution. The algorithm starts off with the start state being the only open node. We then examine the first node \( p \) in the list, and if it is the goal state, we return its path as the solution, but in reverse order to show the steps from the start state to the goal state. If we have not reached the goal yet, we need to search further by expanding node \( p \). The resulting new nodes are then added to the list of open nodes, and this list is sorted by its scores in ascending order so that the most promising node comes first. The function \texttt{sortBy (compare `on` fst)} simply sorts the pairs representing the open nodes by their first element, which is their score. Then \texttt{expand} calls itself with the updated node list.

As you can see in the last line of this function, \texttt{genStates} is used to create the new nodes. The input to \texttt{genStates} is the current state, i.e., the first element in \texttt{path}, and we exclude all states that are already in \texttt{path} in order to prevent circuits in our search tree (which would otherwise become a much less convenient search graph). For each new node \( p \), its score is computed as \( f'(p) = g'(p) + h'(p) \), where \( g'(p) \) is simply the length of the current path and \( h'(p) \) is the \( h' \) function provided to \texttt{aStar}, applied to the current state and the goal state. Then the new state for \( p \) is prepended to its parent’s path. That is all there is. This function looks a bit dense, but if you make the effort to go through every line of code with the help of these explanations, you will see that it is actually possible for humans to fully understand it.

The value of \texttt{solution} is obtained by applying \texttt{aStar} to the problem we discussed in class, and the \texttt{main} function applies \texttt{showState8} to its elements in order to print the solution on the console. This is what you should see on your screen when running the program:
This is exactly the solution we found in class. I tried to make it look nice for console or file output (you are welcome!) to keep things simple. However, in case you would like to implement better graphics, there are excellent Haskell bindings to libraries such as OpenGL.

At any rate, you can now play around with this code and create even more difficult eight-puzzle tasks to be solved. The code will solve all solvable problems efficiently, but unfortunately there are unsolvable problems, i.e., start and goal states that cannot be transformed into each other. And that is where the code is imperfect: It will not tell you that there is no solution, but it will inevitably crash. Your first task will be to fix this annoying problem.
(a) The imperfection of the code is 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.

(b) Unfortunately, you will notice 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.

(c) Now that we have created this powerful `aStarID` function, let us make use of its generality and apply it to a different problem - the Bottle World puzzle from Question 1. Represent a state as a list of letters such as “BCDA”. Write functions `h’Bottle` and `genStateBottle` 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.

(d) **100% bonus question for CS470, 50% bonus question for CS670**: Being an expert of applying A* to different problems now, find yet another problem that can be solved by A* and write Haskell code to solve it. It could be any kind of problem such as a puzzle; you can look online at problems people solve using A* for inspiration.