

CS 470/670 – Artificial Intelligence – Spring 2020

Instructor: Marc Pomplun

Assignment #2

Sample Solutions

Question 1: Shuffling Letters

Let us look at a similar but much simpler problem than the 8-puzzle that we discussed in class. It is the four-letter-word-shuffle puzzle, which is about words containing each of the four letters D, L, O, and U exactly once. The words can be manipulated by the schema `shuffle(x)`, which puts the first letter (position 1) of the current word into position `x`, and all letters that were previously in positions 2 to `x` move one position to the left. Since `shuffle(1)` would not change the word at all, `shuffle(2)`, `shuffle(3)`, and `shuffle(4)` are the only legitimate operators.

For example, if my current word is OLDU, then:

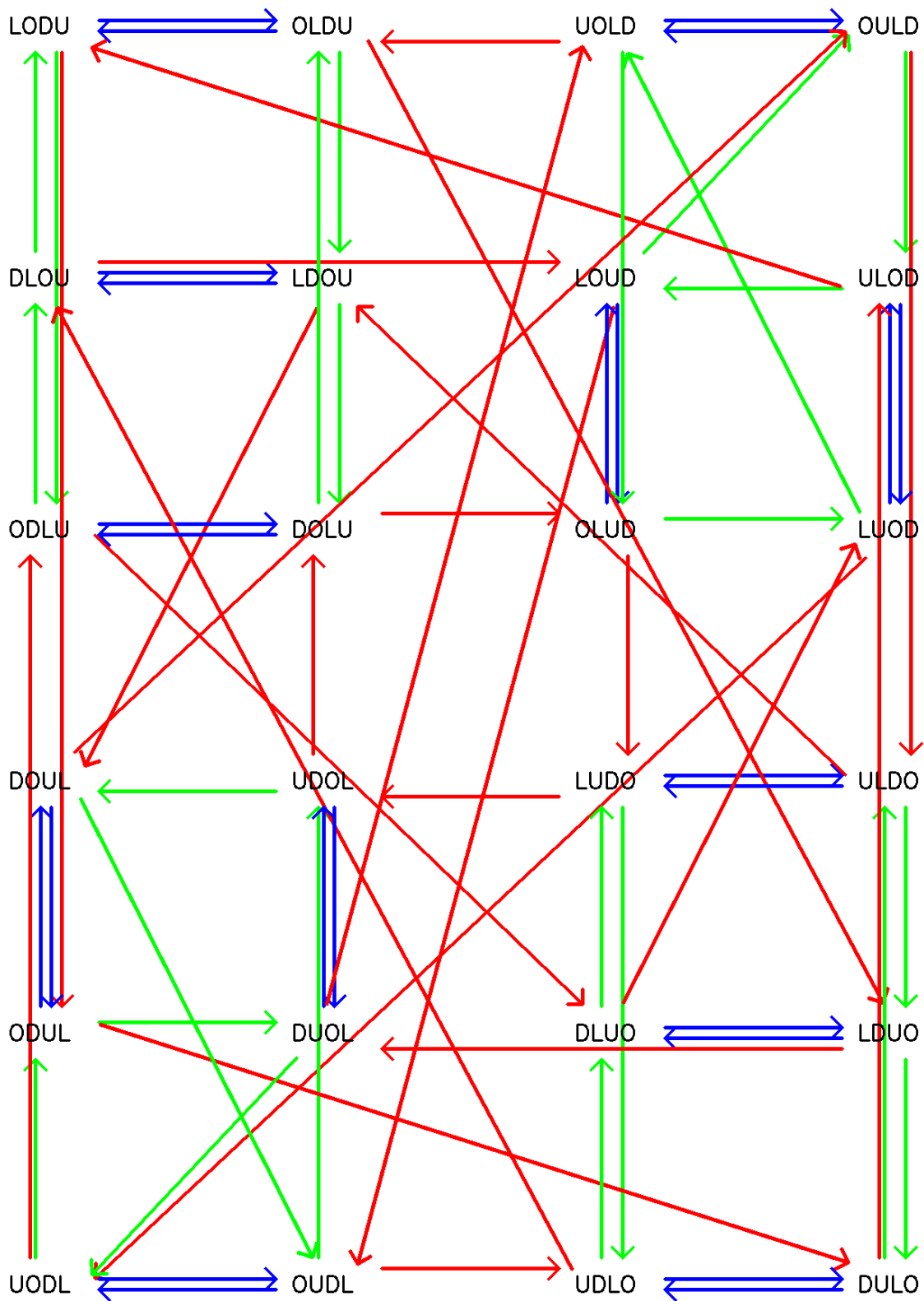
- `shuffle(2)` results in LODU,
 - `shuffle(3)` results in LDOU,
 - `shuffle(4)` results in LDUO.
- a) Draw the state-space graph of this world. This may be a bit too ambitious, given the number of states and transitions. You can give it a try, but if it becomes too tedious, write a transition table instead. To do that, list all possible states (i.e., words) and give each of them an index. Then indicate for each word, to which other word (identified by their index) we would transition for each of the three operators.

The following page shows a sample table, and the page after that shows a state-space graph. I thought that this question would be a great opportunity to show you an example of how to simulate evolutionary processes to tackle optimization problems.

As I understand, none of you drew the state-space graph, because it would be difficult to do and look very cluttered, because there are 24 states and 72 transitions. Well, an interesting question is: To what extent you can optimize the placement of the 24 states in the graph so that the state transitions (directed edges) are, in their sum, of minimal length in order to make the graph as clear as possible? Let us assume that we want to place the states on a 4×6 grid. To solve this problem in an evolutionary fashion, I wrote the program `optimize_state_space_graph.py` in Python (just because Python is well-suited for quick experiments) and uploaded it to the “General Resources” section on Piazza. It runs 1,000 generations with a population of 2,000 solutions, and usually (but not always) it finds the optimal sum of transition lengths of 123.43 units. Afterwards it writes an image showing the visualization of the state-space graph, an example of which is shown on page 3.

		shuffle(2)	shuffle(3)	shuffle(4)
(1)	LOUD	7	9	10
(2)	LODU	8	11	12
(3)	LUOD	13	15	16
(4)	LUDO	14	17	18
(5)	LDOU	19	21	22
(6)	LDUO	20	23	24
(7)	OLUD	1	3	4
(8)	OLDU	2	5	6
(9)	OULD	15	13	14
(10)	ODUL	16	18	17
(11)	ODLU	21	19	20
(12)	ODUL	22	24	23
(13)	ULOD	3	1	2
(14)	ULDO	4	6	5
(15)	UOLD	9	7	8
(16)	UODL	10	12	11
(17)	UDLO	23	20	19
(18)	UDOL	24	22	21
(19)	DLOU	5	2	1
(20)	DLUO	6	4	3
(21)	DOLU	11	8	7
(22)	DOUL	12	10	9
(23)	DULO	17	14	13
(24)	DUOL	18	16	15

shuffle(2) →
 shuffle(3) →
 shuffle(4) →






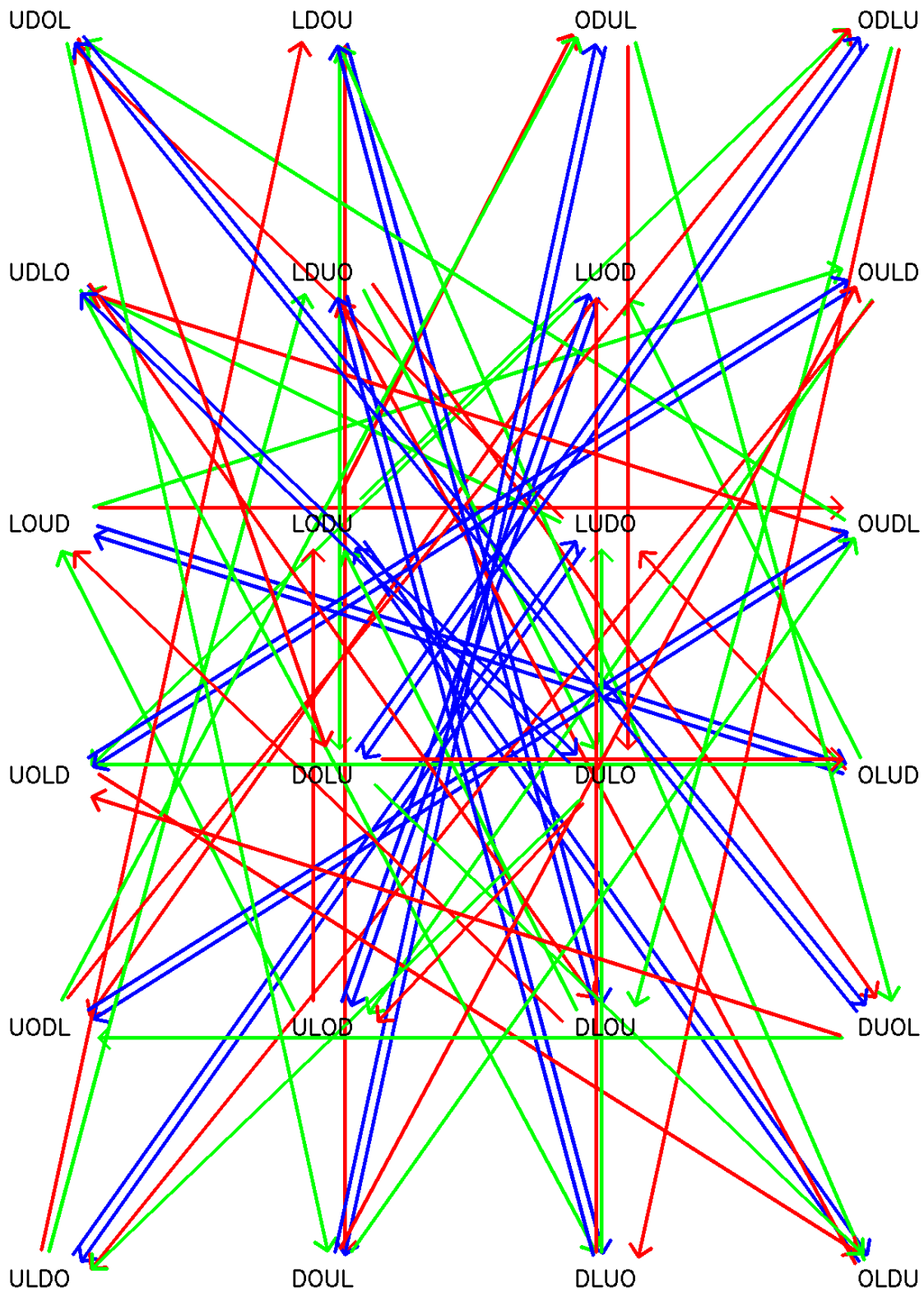
OK, so the state-space graph does not look extremely cluttered once the placement of states has been optimized. We can follow every single transition here. Then another - though less useful – question is: What is the worst possible solution that is the least readable?

We can easily modify the algorithm to answer this question, and this time the sum of transition distances is above 254. The resulting state-space graph is shown on the next page, and it is not easy to read indeed.

The code I uploaded is certainly no masterpiece of software engineering because I could not invest much time in writing it, but I added comments so that it should be easy for you to understand and modify it to play around with the idea of evolutionary optimization. Note that I commented out two other distance functions that you can give a try as well, and it is interesting to see how they affect the resulting graph as compared to using the Euclidean distance.

Please let me know if you have any questions regarding this code, and please experiment with it for a bit; it should really give you a good understanding of the idea of using evolutionary processes for solving difficult problems.

shuffle(2) 
shuffle(3) 
shuffle(4) 



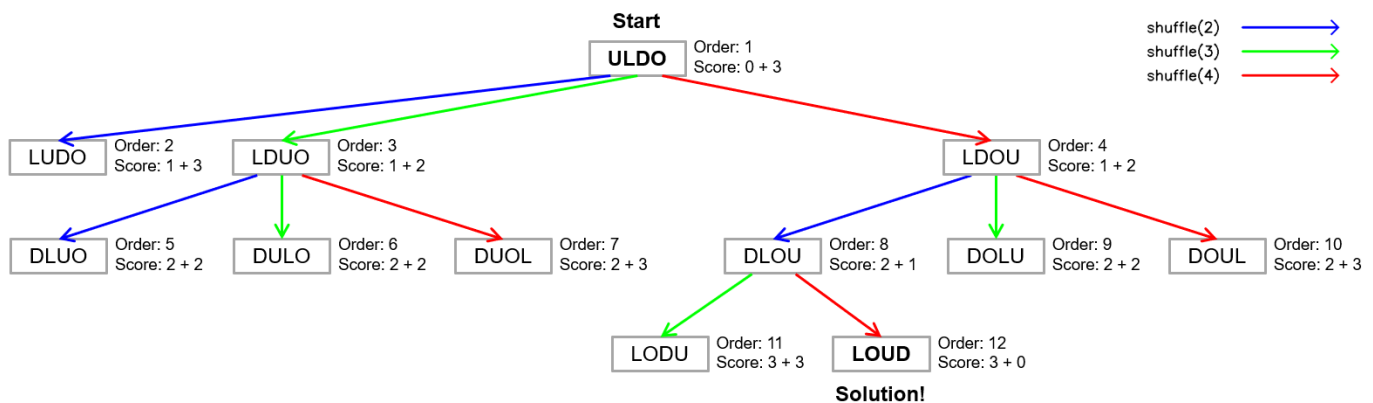
- b) Assume that your start state is the word ULDO but you would like it to become LOUD. Use the A* algorithm to find a solution that requires a minimum number of operations. To do this, first define an appropriate estimation function $f'(n) = g'(n) + h'(n)$. Then write down the resulting search tree, indicate the order in which nodes were created, and for each node n give the value of $f'(n)$.

In order to be guaranteed to find an optimal solution, we have to come up with an optimistic estimation function, i.e., a function that never overestimates the number of required operations to reach the goal.

For the letters currently in positions 1 to 4, let us call their goal (target) positions $t(1)$ to $t(4)$. For instance, at the start state ULDO, given the goal state LOUD, we have $t(1) = 3$ (the letter U has to move into the third position), $t(2) = 1$, $t(3) = 4$, $t(4) = 2$. One thing that is for sure is that the letters in positions 2, 3, and 4 can at most move one step to the left with each operation. For example, since we have $t(4) = 2$, we know that it will take us at least two steps to reach a solution. Things can get even worse if we have to move letters to the right. Given that $t(3) = 4$, we first have to move the letter D into the first position and then have it jump to position 4, so we need three steps to achieve it, which means that it is impossible to find a solution for getting from ULDO to LOUD requiring fewer than three steps. Therefore, determining the letter that needs the most steps gives us an estimation function $h'(n)$ that is never pessimistic. For a state n whose letters need to be moved into positions $p(1)$ to $p(4)$, we can define this function as follows:

$$h'(n) = \max_{p=1..4}[s(p)] \quad \text{with} \quad s(p) = \begin{cases} p - t(p), & \text{if } p \geq t(p) \\ p, & \text{otherwise} \end{cases}$$

Using it in the A* algorithm to compute $f'(n) = g'(n) + h'(n)$ gives us the following search tree (note that when we expand node #8, we do not check shuffle(2) because it would result in a state that we already encountered in node #4):

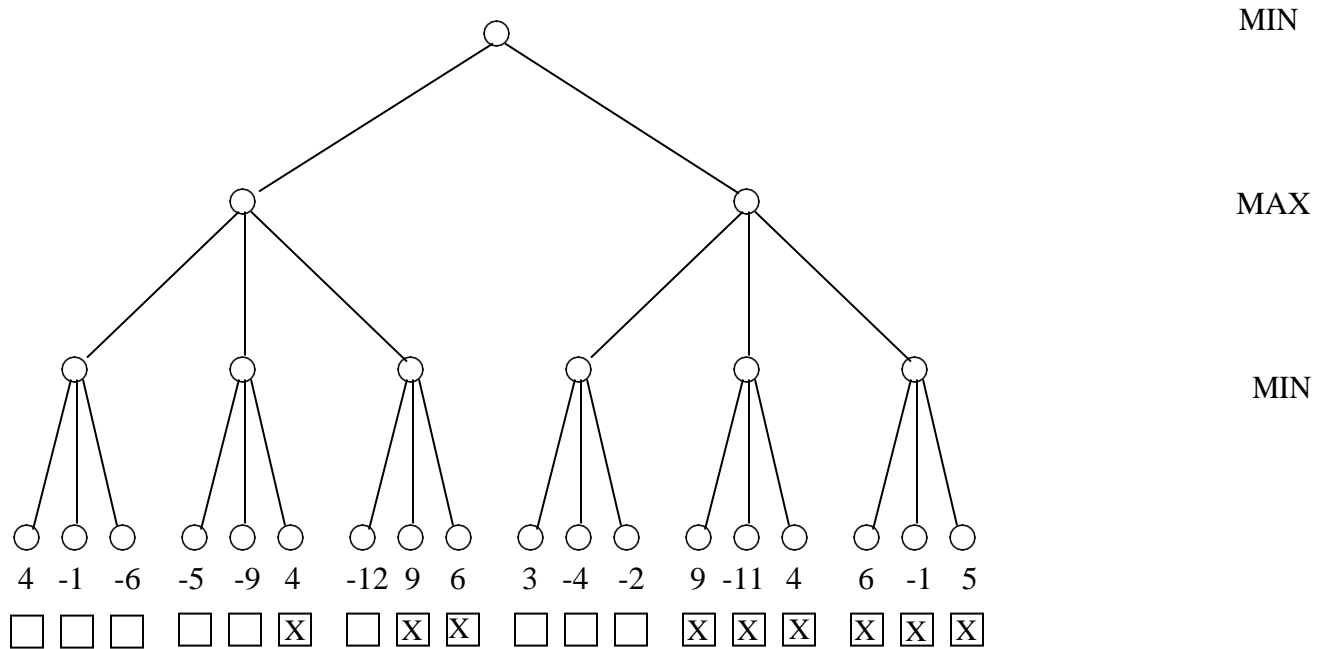


So the solution can be reached in three steps through shuffle(4), shuffle(2), and shuffle(4).

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 move on the left, because then MIN is guaranteed a score of -6 or better, i.e., less, if MAX does not play optimally in terms of the chosen static evaluation function. If MIN makes the move on the right and MAX plays optimally, the score after two further moves would be -4, which for MIN is worse than -6.

Question 3: Recycle the Haskell Backtrack Function

Download the NQueens.hs code from Piazza and modify it to solve the following problem instead of the n-queens one:

Given the numbers 7, 9, 15, 17, 18, 22, and 24, pick exactly five of them (repeats are allowed) so that their sum is 100. For example, you could pick $15 + 17 + 17 + 7 + 22$, but unfortunately their sum is only 78.

Use the **backtrack** function without any changes, but write a new sanity check function

```
fSum :: Int -> [Int] -> Bool
```

for the new problem and a

```
solveSum :: [Int]
```

function that calls the **backtrack** function (with **fSum** as one of its inputs) to find a solution, which is simply a list of the numbers being picked.

Possible (highly hard-coded) solution:

```
fSum :: Int -> [Int] -> Bool
fSum nextChoice plan = (length plan < 4 && newSum <= 93) || (length plan == 4 && newSum == 100)
    where newSum = nextChoice + sum plan

backtrack :: (a -> [a] -> Bool) -> [[a]] -> [a]
backtrack fCheck levels = bt [] levels
    where bt plan []      = plan
          bt plan ([]:_)  = []
          bt plan ((b:bs):ls) = if not (fCheck b plan) || null result
                                then bt plan (bs:ls)
                                else result
          where result = bt (b:plan) ls

solveSum :: [Int]
solveSum = backtrack fSum $ take 5 (repeat [7, 9, 15, 17, 18, 22, 24])

-- >>> solveSum
-- [24,24,22,15,15]
--
```