Solving a River-Crossing Problem

Rocky K. C. Chang
27 April 2022

I. The Problem

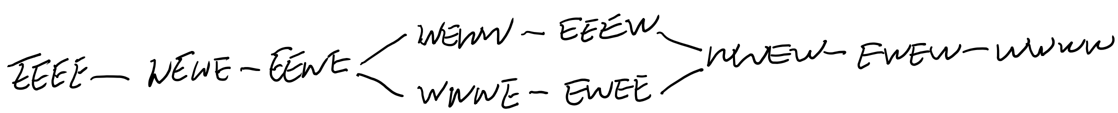
The problem is to bring a man, a cabbage, a goat and a wolf from the east side of a river to the west side in a smallest number of boat trips. There are several details about this problem:
(C1) The boat must be operated by the man.
(C2) Besides the man, at most one more entity can ride in the boat.
(C3) The cabbage cannot stay with the goat without the man.
(C4) The goat cannot stay with the wolf without the man.

II. Data Abstraction

System States The goal of this data abstraction step is to transform the human-readable problem to a machine-readable form. We define the state of this system by a 4-tuple of Boolean values—(E)ast and (W)est—which indicates the current locations of the man, cabbage, goat and wolf (MCGW). There are therefore a total of 16 states. For example, (E,W,W,E) is the man, the boat and the wolf on the East side and the rest on the West

Nodes and links We model each state by a node and connect a pair of nodes by a link, if the two states can reach to one another directly. As described below, there are two kinds of illegal states which violate C3 and C4:
1. By C3, states, such as (E,W,W,*), are illegal, where * means "don't care."
2. By C4, states, such as (E,*,W,W), are illegal.
Two states can reach each other directly only if C2 is not violated. For example, (E,E,E,E) is not linked to (W,W,W,W).

A shortest-path problem By modeling the legal states as nodes and linking two states which can reach each other directly, we come up a graph for this problem.



Therefore, the problem becomes the well-known shortest-path problem—finding a shortest path in terms of the number of nodes from (E,E,E,E) to (W,W,W,W). Clear there are two equally good paths.

EEEE→WEWE→EEWE→WEWW→EEEW→WWEW→EWEW→WWWW
EEEE→WEWE→EEWE→WWWE→EWEE→WWEW→EWEW→WWWW

Abstract data types From the discussions above, we identify the following abstract data types (ADTs) in our data model. These ADTs are abstract, because at this stage we do not concern about the specific implementations of these data types.
1. A Boolean value for the location
2. A list of four Boolean values for each state
3. A graph for the relationship among all the legal states
4. A list of legal states as a solution to the problem

III. Algorithm Design

We need a shortest-path algorithm to solve the shortest-path problem. The inputs to this algorithm are (E,E,E,E,), (W,W,W,W), and the graph. The output of this algorithm is a list of nodes starting from (E,E,E,E) and ending at (W,W,W,W) which any one of the shortest paths. Since the focus of this project is not on algorithm design, we will employ any shortest-path algorithm to solve the problem.

IV. Program Design

The goal of this section is to design a modular program by breaking the program into a set of functions. Following the best practice, each function will perform only a single task. Table 1 summarizes the functions with their signatures.

Functions	Inputs	Outputs
Solver()	None	Print out the solution to the MCGW problem.
genStates()	None	Return a set of all possible states
genGraph()	A set of all possible states	Return a graph consisting of only legal states
findShortestPath()	A graph, a source node and a destination node	Return a list of nodes that is a shortest path from the source node to the destination node
printPath()	A list of nodes	Print out the solution to the MCGW problem
isAStateLegal()	A state	Return True if the state is legal, False otherwise
isNeighbor()	Two states	Return True if the two states can reach each other directly, False otherwise

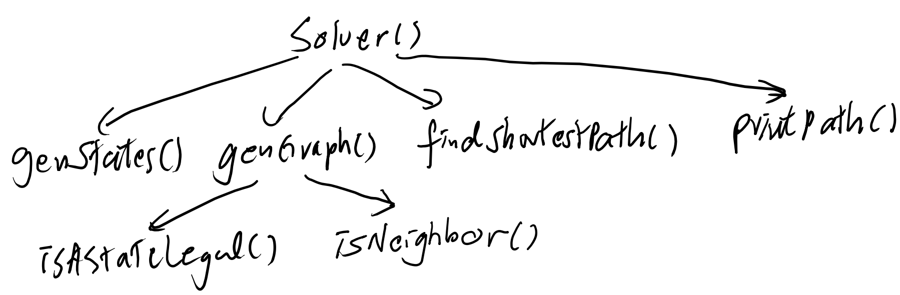


Figure 1. The relationship among the seven functions.

Moreover, we also employ a library for graph in which some functions are used to generate the graph. The functions in this library are defined below.

Functions	Inputs	Outputs
getNodalDeg()	A graph and a node in the graph	Return the node's nodal degree
getNodes()	A graph	Return the set of nodes in the graph
getLinks()	A graph	Return the set of links in the graph
getNeighbors()	A graph and a node in the graph	Return a set of other nodes in G to which this node is connected
isLinked()	A graph and two nodes in the graph	Return True if there is a link between the two nodes; False, otherwise
addNodes()	A graph and a node which may or may not be in the graph	If the node is already in the graph, return True. Else, the graph will be updated by including this node and return True.
addLinks()	A graph and two nodes (n1 and n2) which may or may not be in the graph	If any node is not in the graph, add the node(s) to the graph. The graph will be updated by including a undirected link from n1 to n2 and return True.

delNode()	A graph and a node which may or may not be in the graph	If the node is in the graph, the graph will be updated by removing the node and its links and return True. Else, print an error message and return False.
delLink()	A graph and two nodes which may or may not be in the graph	If the two nodes are in the graph, the graph will be updated with the undirected link removed from the graph and return True. Otherwise, print an error message and return False.

V. A PYTHON IMPLEMENTATION

As discussed in section II, there are four ADTs in our data model.

- We implement the Boolean by strings “E” and “W” for its readability (0/1 is another possibility but it is less readable, therefore not adopted in our implementation).
- We implement a state as a string of four characters, each is either “E” or “W”. Tuple is another equally good choice.
- We implement a list of legal states using Python list. Since we need to append new states, tuple cannot be used.
- As for the graph ADT, we use Python dictionary in which the keys are the states (implemented as 4-character strings) and the values are the set of neighboring states. Therefore, each dictionary item implements a set of unidirectional links from the state in the key to the set of its neighboring states in the values. We prefer this adjacency list approach over the adjacency matrix, because we anticipate the graph will be sparse for which the adjacency list is more space efficient. Moreover, we do not use Python list for the neighboring states, because there is no need to order them.

The source code for solving the MCGW problem and the library for the graph could be found in `MCGW_rocky.py` and `graph.py`, respectively.