# CS 108 Midterm Project (v0.5)

Fall 2024

Rocky K. C. Chang
October 18, 2024

**1. The learning outcomes**

You will learn from this project:

1. How to identify data (and their types) in a problem
2. How to implement the data using the most suitable Python data types
3. How to transform a human-readable problem to a computer-readable problem
4. How to model a problem using states
5. How to classify the states into legal and illegal states
6. How to implement a graph using Python dictionary
7. How to use a shortest-path algorithm to solve a problem
8. How to decompose the problem into several subproblems
9. How to design functions to solve a problem
10. How to format strings
11. How to document functions using docstrings and best practices
12. How to make your program readable

**2. The man, cabbage, goat and wolf (MCGW) 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. In the beginning, they are all on the east side. There are several constraints to 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.

**3. Steps to solve the MCGW project using computational thinking**

After understanding the problem, there are three main steps to solving the MCGW problem before coding: data abstraction, algorithm design and problem decomposition (or program design).

1. The goal of data abstraction is to transform a human-readable problem to a computer-readable problem.
2. The goal of the algorithm design is to design an efficient algorithm to solve the computer-readable problem obtained from the data abstraction.
3. The problem decomposition concerns how to decompose the problem into a set of “smaller” problems, each of which could be tackled independently from each other.

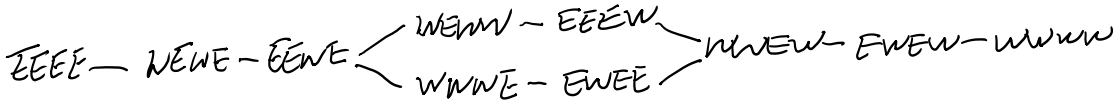
**4. Data abstraction**

**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 (and the boat), 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 (E,W,W,*) and (W,E,E,*) are illegal, where * means "don't care."
2. By C4, states (E,*,W,W) and (W,*,E,E) are illegal.

Two legal states can reach each other directly only if C1 and C2 are 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). Clearly 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 the four locations
3. A graph for the relationship among all the legal states
4. A list of legal states as a solution to the problem

How to implement the four ADTs?
1. We implement the Boolean by strings "E" and "W" primarily for its readability (0/1 is another possibility but it is much less readable, therefore not adopted in our implementation).
2. We implement a state as a string of four characters, each is either "E" or "W". Tuple is another equally good choice but not adopted because string sufficiently serves our purpose.
3. We implement a list of legal states using Python list. Since we need to append new states, tuple cannot be used.
4. 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 lists of neighboring states. Therefore, each dictionary item implements a set of unidirectional links from the state in the key to the list of its neighboring states in the values. We prefer this adjacency list approach over the adjacency matrix, because we anticipate that the graph will be sparse for which the adjacency list is more space efficient.

**5. 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 are two lists of nodes starting from (E,E,E,E) and ending at (W,W,W,W) both of which are shortest paths. In this course, we will not go into the algorithm design. CS students will take CS 112 and CS 212 on this topic. Therefore, you don't need to design and implement the shortest-path algorithm. The code will be provided for you.

**6. Problem design**

You are given the following functions to solve the MCGW problem, two of which are already implemented.

```
def main():
    """Solve the MCGW problem

    Print all solutions to the MCGW problem.

    Parameters
    ----------
    None

    Returns
    -------
    None
    """
    pass

    # Task 1: Generate a set of all possible states
                                        # Each state consists of 4 characters, each of
                                        # which could be E(ast)/W(est)
                                        # The four characters representing
                                        # - leftmost: the location (E/W) of the man
                                        # - left-middle: the location (E/W) of the cabbage
                                        # - right-middle: the location (E/W) of the goat
                                        # - rightmost: the location (E/W) of the wolf

    # Task 2: Generate a graph for the MCGW problem

    # Task 3: Find all shortest paths from EEEE to WWWW

    # Task 4: Print all solutions in a reading-friendly format

def genStates():
    """Generate a tuple of all states for the MCGW problem

    Parameters
    ----------
    None

    Returns
    -------
    Tuple of strings
        All possible states
```

```
    """
    pass

def genGraph(S):
    """Generate a graph for the MCGW problem

    Parameters
    ----------
    S : a tuple of strings

    Returns
    -------
    Dictionary (key : string, value : a list of strings)
        A graph for the MCGW problem
    """
    pass

def isAStateLegal(state):
    """Determine whether a state is legal

    Parameters
    ----------
    state : string
        A state

    Returns
    -------
    Boolean
        True if the state is legal; False, otherwise
    """
    pass

def explore_paths(graph, current, target, path, all_paths, shortest_length):
    """Find all shortest paths from start to end on a graph

    Parameters
    ----------
    graph           : dictionary (key : string, value : a list of strings)
        A graph model for the MCGW problem
    current         : string
        The current node that this function is processing
    target          : string
        A target node in the graph
    path            : list of strings
        The path obtained so far
    all_paths :     : list of list of strings
        The list of all shortest paths
    shortest_length : A list of an integer
        The current shortest length

    Returns
    -------
    None
    """
```

```
    if current == target:
        if len(path) < shortest_length[0]:
            shortest_length[0] = len(path)
            all_paths.clear()
        if len(path) == shortest_length[0]:
            all_paths.append(list(path))
        return

    # Explore all neighbors
    for neighbor in graph.get(current):
        if neighbor not in path:
            path.append(neighbor)
            explore_paths(graph, neighbor, target, path, all_paths, shortest_length)
            path.pop()


def find_all_shortest_paths(graph, start, target):
    """Find all shortest paths from start to end on a graph

    Parameters
    ----------
    graph  : dictionary (key : string, value : a list of strings)
        A graph model for the MCGW problem
    start  : string
        A starting node in the graph
    target : string
        A target node in the graph
    path   : list of strings
        The shortest path obtained so far

    Returns
    -------
    List of lists of strings
        A list of all shortest paths from the starting node to the target node
    """

    all_paths = []
    shortest_length = [float('inf')]
    explore_paths(graph, start, target, [start], all_paths, shortest_length)

    return all_paths


def printPath(list_paths):
    """Print out a solution to the MCGW problem in a human-readable format

    Parameters
    ----------
    list_paths : list of strings
        A list of states (nodes) in the graph model

    Returns
    -------
    None
```

```
    Side effects:
    - Print out all solutions to the MCGW problem
    """
    pass
```

**7. What are required of you?**

Your task in this project is to implement the functions provided to you, so that the `main()` function will produce the two solutions to the MCGW problem. The outputs must exactly match the ones below. As discussed earlier, there are two equally good solutions. Besides the given functions, you will design and implement **at least one more new function** to help you implement `genGraph()`.

**Important reminders and restrictions**

Points will be deducted if violating these restrictions.

- This is an individual project.
- You are expected to write all the code yourself.
- No import statements allowed.
- You have to use all the functions given.
- You cannot change the doctrings of all functions and their names.
- You cannot change the code in `explore_paths()` and `find_all_shortest_paths()`.
- You cannot use any built-in functions except `print()`, `len()` and type conversion functions.
- You cannot use list comprehensions.

```
Path 1:
1. The man takes the goat from the east to the west.
2. The man takes only himself from the west to the east.
3. The man takes the wolf from the east to the west.
4. The man takes the goat from the west to the east.
5. The man takes the cabbage from the east to the west.
6. The man takes only himself from the west to the east.
7. The man takes the goat from the east to the west.

Path 2:
1. The man takes the goat from the east to the west.
2. The man takes only himself from the west to the east.
3. The man takes the cabbage from the east to the west.
4. The man takes the goat from the west to the east.
5. The man takes the wolf from the east to the west.
6. The man takes only himself from the west to the east.
7. The man takes the goat from the east to the west.
```

**8. Assessments and due dates**

- `main()`: 5%
- `genStates()`: 15%
- `genGraph()` and other new functions: 50%

- `isAStateLegal()`: 15%
- `printPath()`: 15%

Due at 23:59 on November 5 in zyBooks. There will be several project assignments in zyBooks to evaluate different functions. The late submission policy will apply to this project. **There will be no deadline extension.**