Section A: Solving the Farmer Crosses River Puzzle using State Space Search

To solve this problem using state space search, we represent each state of the problem as a node in a search tree or graph. Each node represents a configuration of farmers, wolves, goats, and cabbages on either side of the river. Edges between nodes represent valid transitions between states, which in this case are actions that a farmer can take to transport items across the river.

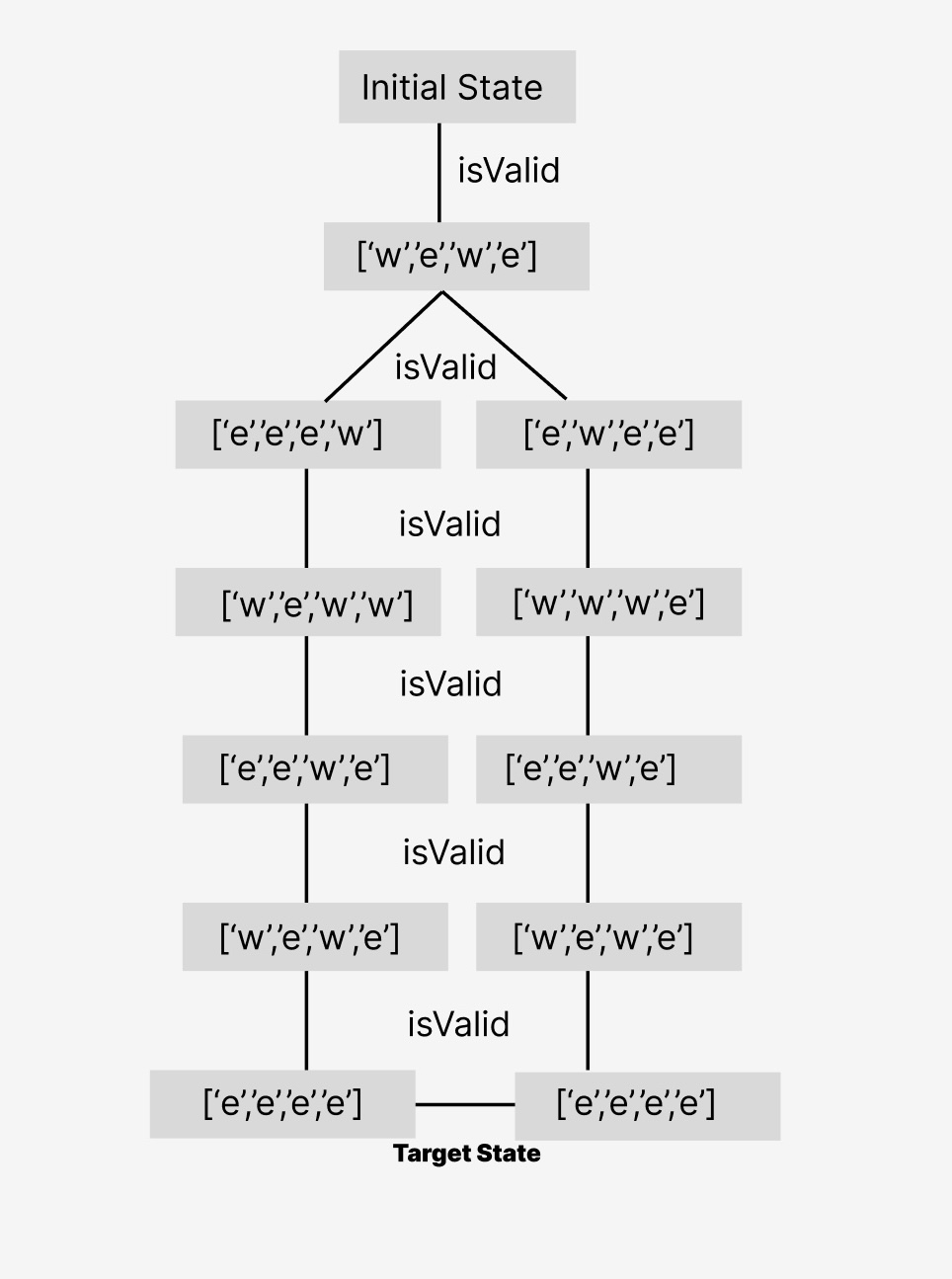
There are 4 steps to show how the algorithm works:

1. As for initialization, starting from an initial state, the farmer, Wolf, sheep, and cabbage are all west of the river。
2. As for expansion, generate all possible valid next states from the current state by applying valid operations. In this case, the valid behavior is to carry an item across the river.
3. As for goal test, check if any resulting state is a target state where all items are on opposite sides of the river. If the target state is found, the solution is complete.
4. As for iteration, repeat steps 2 and 3 for each generated state until the target state is found or all states have been explored.

In general, we can systematically explore the state space, starting from an initial state and generating all possible successor states. At each step, we check whether the generated state is valid and has not been accessed before. If a valid successor state is found, we add it to the list of visited states and continue the search. This process continues until we reach the target state where all entities are safely transported to the other side of the river.

This process forms a search graph, where each node represents a different state and edges represent valid transitions between states. We can find the shortest path from the initial state to the goal state, which represents the sequence of actions the farmer must take to solve the problem.

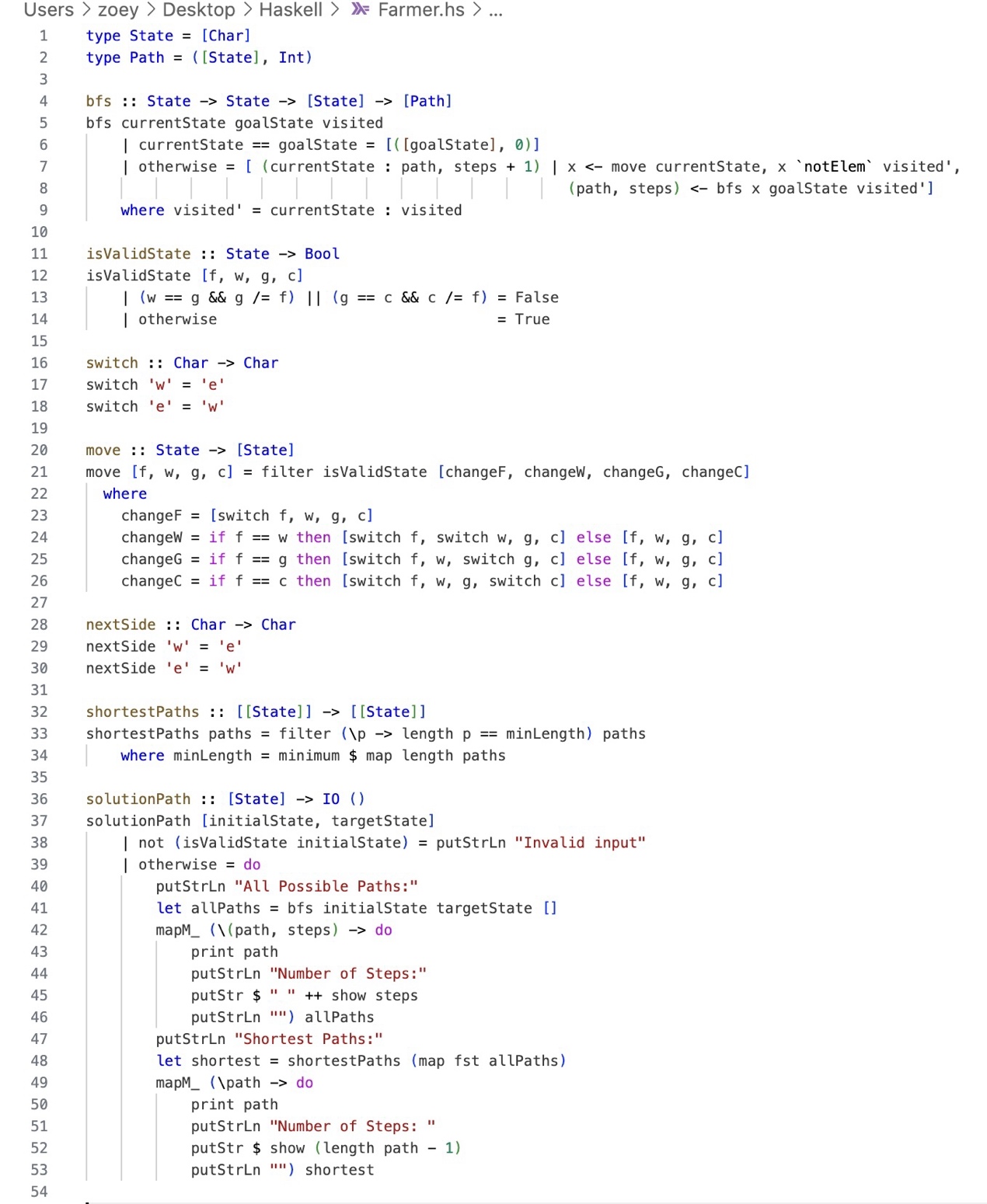
Below is search algorithm traverses either a tree or a graph:



In this example, each node represents a state and edges represent valid transitions between states. The search algorithm systematically explores these states until it finds a goal state in which all items are across the river, indicating a successful solution to the problem.

In summary, solving the farmer's river-crossing problem using state-space search requires systematically exploring all possible states and transitions to find a series of intersections that safely transport all entities across the river. Search algorithms utilize a graph or tree representation of the state space, with nodes representing states and edges representing transitions, to efficiently navigate the problem space and find solutions.

Section B: Haskell Source Code



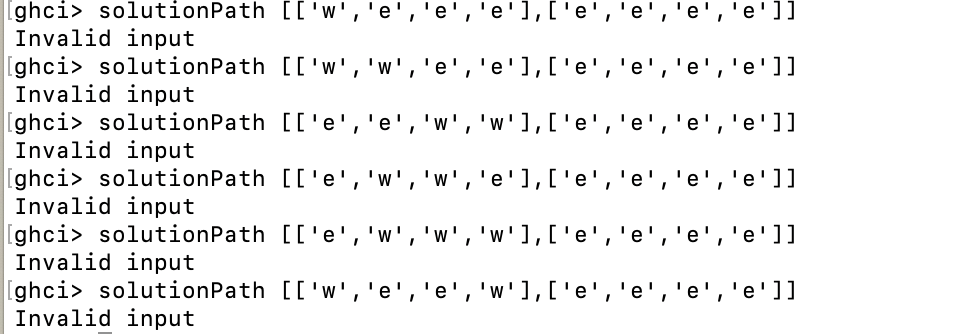
Section C: Experimentation with the Program (Input-Output Sessions)

1. This picture of Haskell code running includes all possible paths from west side to east side. In addition to all possible paths, it also includes finding the shortest path by comparing the length of steps. Moreover, in this case, there are two shortest paths. Finally, below each path, I have indicated the number of steps required to complete the path.

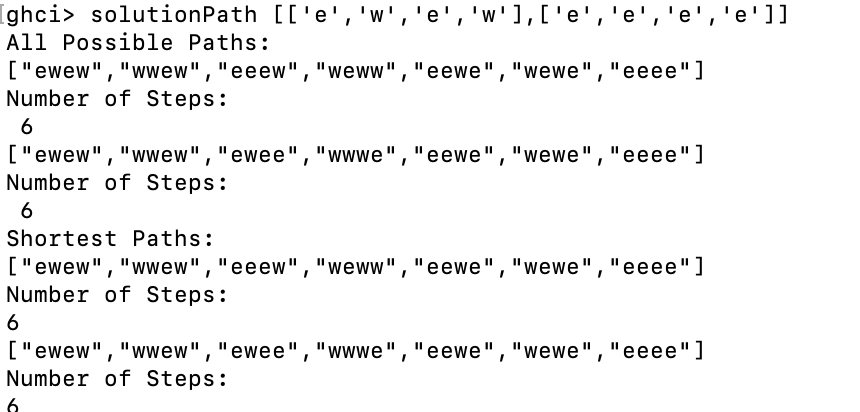
文本

描述已自动生成

1. This picture of Haskell code running contains some cases where an invalid input is displayed when the wolf is ever left alone with the goat, an invalid input is displayed when the goat is ever left alone with the cabbage and an invalid input is displayed when the farmer is alone on one side of the river, the wolf, the goat and the cabbage are on the other side.



1. This picture of Haskell code running, showing how the wolf and the cabbage on the west side are transported to the east side. Then, I also showed you the number of steps required for each path and compared them to find the shortest path.

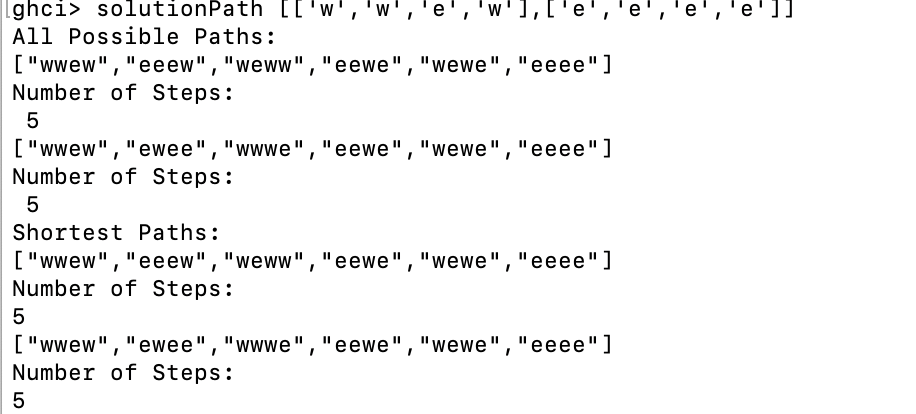


1. This picture of Haskell code running shows how the wolf alone on the west side is moving from west side to east side. Then, I also showed you the number of steps required for each path and compared them to find the shortest path.

图形用户界面, 文本

描述已自动生成

1. This picture of Haskell code running, showing how the farmer, the wolf and the cabbage on the west side are transported to the east side. Then, I also showed you the number of steps required for each path and compared them to find the shortest path.



In addition, my Haskell code contains many paths to specific initial states, and you can find the path you want by running different states. I've just listed a few examples.

Section D: Discussion on Solving the Same Problem in OOP Way

In Java, the problem of farmers crossing the river can be solved using the OOP method, which requires setting the farmer, wolf, goat, and cabbage involved as objects and defining their behavior and interaction through classes and methods.

In Entity Classes, define a class for each entity involved in the problem, such as "Farmer", "Wolf", "Goat", and "Cabbage". Each class represents an entity and contains properties that represent its state, such as the current direction of a river, and methods to perform actions, such as crossing a river.

In interaction methods, methods are implemented in each entity class to simulate interactions between entities. For example, the “crossRiver()” method in Farmer will move the farmer and any entities to the other side of the river, and also ensure that no wolf and goat or goat and cabbage are left alone on either side of the river during the move.

In the main method, create a main method to coordinate the process of solving the problem. In the main method, define the sequence of actions required to solve the puzzle while adhering to the rules, for example a wolf and a sheep, or a sheep and a cabbage cannot be left alone.

In the iterative approach, we adopt an iterative approach using breadth-first search to explore different motion sequences and find the solution. Each iteration involves an operation with a valid state and checking whether the resulting state is valid.

In general, OOP in java provides a structured and modular approach to solving problems by following a set of rules.

In terms of programming style:

As for inheritance, in Java, inheritance can be used to model relationships between entities. For example, a superclass "Entity" can have subclasses "Farmer", "Wolf", "Goat" and "Cabbage", each of which overrides common methods like "move". While Haskell does not have inherent inheritance support like OOP languages. Instead, it relies on data types and type classes to model relationships between entities.

As for polymorphism, in java polymorphism allows different types of objects to be treated as objects of a common super class. This makes the code flexible and extensible. In a puzzle context, polymorphism enables a generic "move" method to be defined in a superclass and overridden in a subclass to specify the movement behavior of each entity. Haskell implements polymorphism through type classes and parametric polymorphism. A type class defines a set of functions that operate on a type as an instance of the class, providing flexibility and abstraction similar to polymorphism in OOP.

As for function overloading, in Java, function overloading allows you to define multiple functions with the same name but different parameters. This allows methods with similar behavior to be grouped under one name. In a puzzle context, function overloading can be used to define methods for different types of movement, such as "move (Farmer f)". Haskell does not support function overloading like Java does. Instead, functions with similar behavior are often distinguished by pattern matching or different function names.

As for the imperative and declarative nature of the paradigm, Java is imperative in nature and the focus is on defining step-by-step instructions for the computer to execute. This is reflected in the imperative programming style, where code is organized around statements that change the state of the program. Haskell, on the other hand, is more declarative, emphasizing what needs to be done rather than the definition of how to do it. Functional programming in Haskell involves composing functions and expressing computations as a series of transformations on immutable data structures.

Essentially, while both languages ​​provide mechanisms for abstraction, encapsulation, and code organization, they handle these concepts differently. Java's OOP focuses more on organizing code around objects with state and behavior, while Haskell's functional programming emphasizes composing functions and expressing calculations declaratively.