

Cannibals and Missionaries Design Document

This cannibals and missionaries program is the first AI problem that used modern techniques of search trees. This design document describes the implementation made by Chris Lattner.

Main Classes in the C&M Project:

`CannMiss` - The main class that provides either an application interface (text only) or an applet interface (nice animating GUI). Only the application interface will be described in this document.

`State` - The `State` class holds one possible state of the problem, including the locations of the people and the location of the boat.

`StateList` - Maintains a linked list of states.

`Solver` - The `Solver` class uses a breadth first search technique to figure out how to get from state A to state B (which are two arbitrary state parameters to the `FindSolution` function).

Main Functions in the `CannMiss` class:

`main(String Args[])` - Main program, that starts everything else. It creates a `Solver` object, solves the standard version of the problem, then prints the solution.

Main Functions in the `State` class:

Constructor - Create a state, initialized to values the parameters indicate.

`Print()` - Print the state to standard output, including the states were used to get to the state being printed.

`InvalidState()` - Return true if the state is invalid, i.e. the number of cannibals is greater than the number of missionaries.

`Equals(State)` - Returns true if two states are equivalent.

`GetID()` - Converts a state to a numeric identifier that represents the state. No two states that are "Equal" to each other may have different identifiers.

`GetMaxID()` - Returns the maximum value that may be returned by `GetID()`.

Main Functions in the `StateList` class:

Constructor - Create an empty list.

`AddToTail(State)` - Append a state to the end of the list.

`GetHead()` - Returns the state at the head of the list.

`RemoveHead()` - Removes the state at the end of the list.

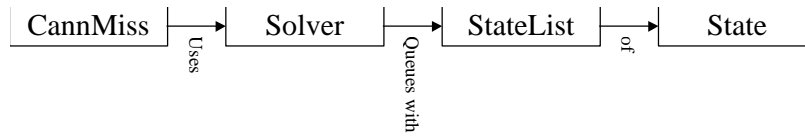
`IsEmpty()` - Returns true if the list is empty.

Main Functions in the `Solver` class:

Constructor - Create a "Solver" for a specific number of people. The number of people is equal to the number of cannibals involved in the scenario. The number of missionaries is equal to the number of cannibals.

`FindSolution(State StartState, State EndState)` - Returns a state "Equal" to `EndState`, but with the move history filled in. The move history keeps track of how to get from `StartState` to `EndState`. If no solution can be found, a null value is returned.

Design Diagram:



Example Run:

Found the following solution:

```

3M/3C < 0M/0C
3M/1C > 0M/2C
3M/2C < 0M/1C
3M/0C > 0M/3C
3M/1C < 0M/2C
1M/1C > 2M/2C
2M/2C < 1M/1C
0M/2C > 3M/1C
0M/3C < 3M/0C
0M/1C > 3M/2C
0M/2C < 3M/1C
0M/0C > 3M/3C
    
```

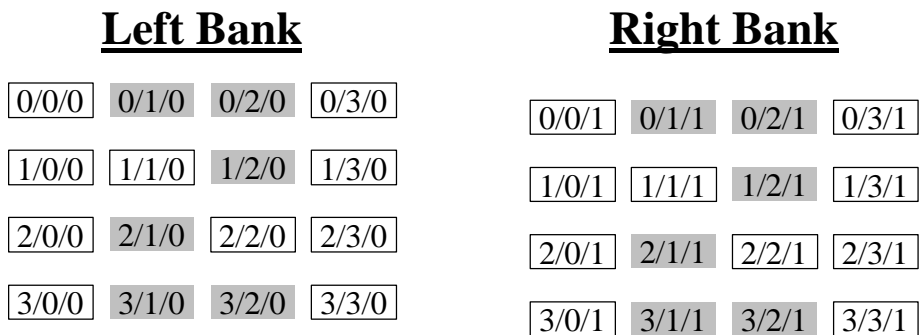
Note: the < and > characters represent the side of the river that the boat is on.

Questions from the Book:

Is it a good idea to check for repeated states?

Yes! It is crucial to check for repeated states. If you do not check for repeated states... and there is no solution to the problem, the program will loop infinitely. In practice, as a performance optimization, checking for states imposes little overhead and yields little gain. Note that there are only 32 states in the “classic” version of the problem.

Draw a diagram of the complete state space:



[#Cann/#Miss/Bank#]

shaded states reflect missionary soup