Oleg Perchyk
CS335 – Search

Search Homework

1. Cats and Dogs Problem

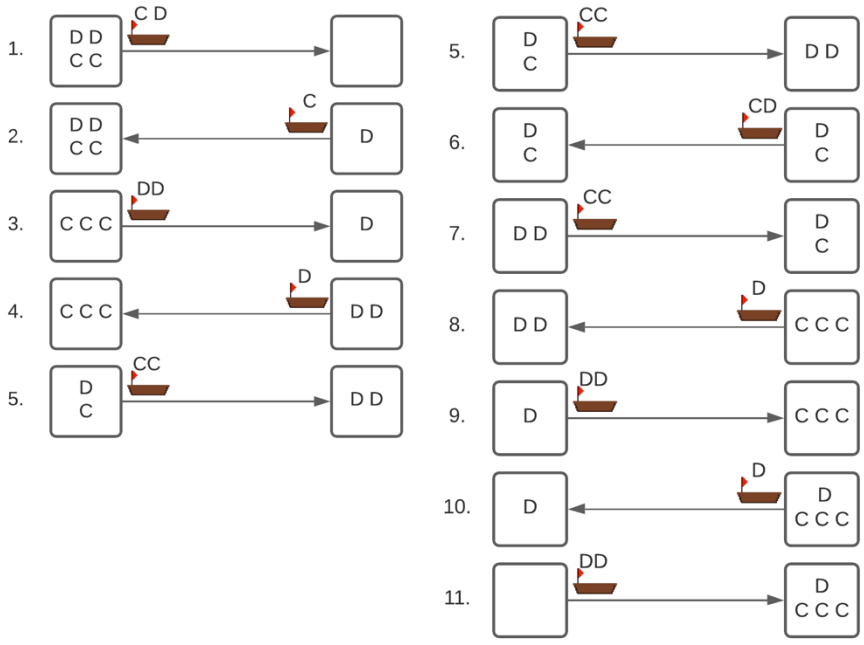
States: The States for this problem include the numbers of cats and dogs on each side of the river and the current location of the raft. The initial state is one where all cats, dogs and the raft are on the same side of the river.

Actions: The action step involves at least one animal crossing the river. There are 5 possible action steps (so long as there are enough animals on that side):

- 1 cat
- 1 dog
- 2 cats
- 2 dogs
- 1 cat 1 dog

Transition model: The transition model returns the result of each possible action given the parameters of the problem. A successful result means that after each action, 1 or 2 animals crossing the river, neither side has dogs outnumbering cats. A failure results when dogs outnumber cats after said action.

A diagram of the complete **state space** may look something like this:



A possible implementation for this problem using a breadth first search:

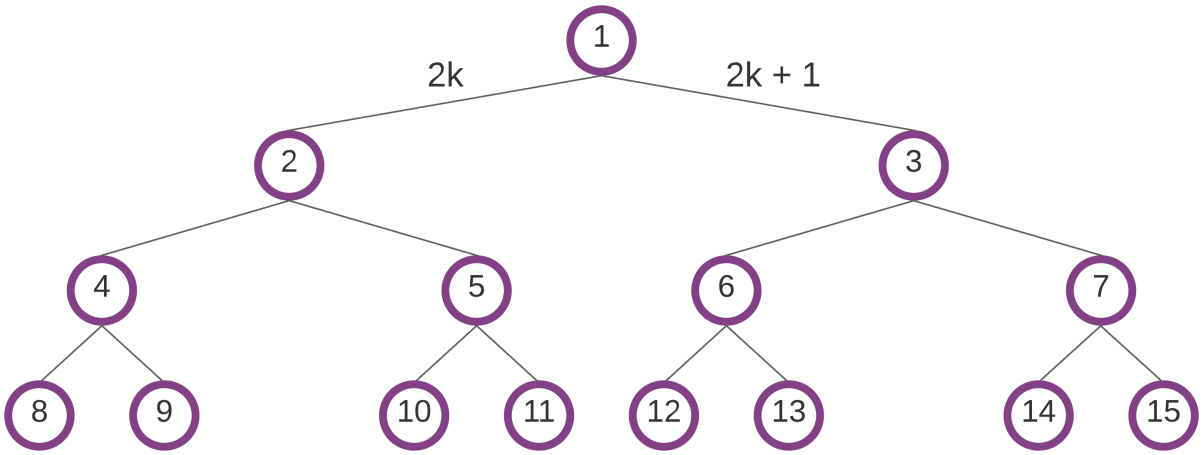
```
ACTIONS = list of possible movement combinations, eg. [C, D, CC, DD, CD]
INITIAL_STATE = { left: [CCCDDD], right:[ ] , boat: left }

function allCrossed(state):
	if (state == 3 cats and 3 dogs across):
		return true
	else:
		return false

function CrossingTheRiver(ACTIONS, INITIAL_STATE):

	node = INITIAL STATE (state of cats, dogs, and boat)
	frontier = queue of nodes with possible action branches
	explored = list of explored states

	if (allCrossed(node.state)):
		return success
	loop do:
		if (frontier is empty):
			return failure

		node = next queued up node in frontier
		explored.Add(node)

		foreach action in ACTIONS:
			child = new node from resulting action step
			if (child not in explored or frontier):
				if allCrossed(child.state):
					return success
				frontier.Add(child)


// root node – INITIAL STATE
node.state = < left: [CCCDDD], right: [], boat: left >
frontier = { level 1 nodes }
explored = { empty list }
ACTIONS = [CD, CC, DD, C, D]

// level 1
node.state = < left: [CCDD], right: [CD], boat: right >
frontier = { level 2 nodes }
explored = { root node}
ACTIONS = [C, D, CD]

//level 2
node.state = < left: [CCCDD], right: [D], boat: left >
frontier = { level 3 nodes}
explored = { level 1 nodes }
ACTIONS = [DD, CD, C, CC]
```

```
//level 3
node.state = < left: [CCC], right: [DDD], boat: right >
frontier = { level 4 nodes }
explored = { all level 1,2 nodes }
ACTIONS = [D, DD]

//level 4
node.state = < left: [CCCD], right: [DD], boat: left >
frontier = { level 5 nodes }
explored = { all level 1,2,3 nodes }
ACTIONS = [CC, C, D, CD]

//level 5
node.state = < left: [CD], right: [CCDD], boat: right >
frontier = { level 6 nodes }
explored = { all level 1,2,3,4 nodes }
ACTIONS = [CD, C]

//level 6
node.state = < left: [CCDD], right: [CD], boat: left >
frontier = { level 7 nodes }
explored = { all level 1,2,3,4,5 nodes }
ACTIONS = [CC, CD]

//level 7
node.state = < left: [DD], right: [CCCD], boat: right >
frontier = { level 8 nodes }
explored = { all level 1-6 nodes }
ACTIONS = [D, CC]

//level 8
node.state = < left: [DDD], right: [CCC], boat: left >
frontier = { level 9 nodes }
explored = { all level 1-7 nodes }
ACTIONS = [DD, D]

//level 9
node.state = < left: [D], right: [DDCCC], boat: right >
frontier = { level 10 nodes }
explored = { all level 1-8 nodes }
ACTIONS = [D, DD, C, CC]

//level 10
node.state = < left: [DD], right: [DCCC], boat: left >
frontier = { level 11 nodes }
explored = { all level 1-9 nodes }
ACTIONS = [DD, D]

//level 11 – SOLUTION condition met
node.state = < left: [], right: [DDDCCC], boat: right >
frontier = { level 12 nodes }
explored = { all level 1-10 nodes }
ACTIONS = []
```

People may have difficulty solving this kind of problem mentally because this is a memory intensive problem. A computer with sufficient memory may have less difficulty keeping track of the explored conditions, however people may need a visual aid or some pen and paper to keep track of each visited condition as to not repeat the same back and forth sequence and get caught in a loop.
Additionally, there is something counter-intuitive about moving an animal back over the river in order the aid the others, or to prevent too many dogs being left with cats when the goal is to get everyone to the other side of the river. We naturally gravitate towards trying to get as many animals over the river as possible, and once we hit a failure condition, then do we begin backtracking and trying something new.

2. State Space Problem

State Space Tree for states 1 to 15



Breadth First Search (BFS)
Goal state = 11
Order visited: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, **11**

Depth First Search (DF) limit 3
Goal state = 11
Order visited: 1, 2, 4, 8, 9, 5, 10, **11**

Oleg Perchyk
CS335 – Search

Algorithm:
When looking for a path from 1 to n we can consider the goal state as the starting point in our algorithm. We know that 2k always produces even numbers and 2k + 1 always produces odd numbers, and we can use this information to understand our current position and work backwards to get a stack of directions taken from the goal state back to the root, 1. Moving down the returned path stack will give us the directions from 1 to n.

```
GOAL = 11

function isEven(n):
	return n % 2 == 0

function FindPath(goal):

	current = GOAL
	path = stack of directions

	while current not 1:
		if isEven(current):
			path.Add(“left”)
			current = current / 2
		 else:
			path.Add(“right”)
			current = (current – 1) / 2

	return path
```