CS 3520—Forth Missionaries and Cannibals

Write a solver for the missionaries and cannibals problem in Forth. For review, here is the problem:

Suppose there are three missionaries and three cannibals who need to cross to the far side of a river using a single boat that can carry one or two people at a time. Both groups will cooperate and can paddle back and forth freely, but old habits will lead the cannibals to eat the missionaries if the missionaries are ever outnumbered on either side of the river.

The problem is to find a way to get all of the missionaries and all of the cannibals safely across the river.

You may write it in a different way than what is suggested below, but it must implement a depth-first or breadth-first search to find the answer.

General tips

Carefully test each word you write before moving on. The design suggested here makes it straightforward to test most words in isolation, but some require a bit more work to test.

If a word does not work, carefully track its use of the stack. Break each and every word onto its own line, and document the state of the stack after that line. For example, you could rewrite this word:

```forth
\ check if a number is a member of the used set
: isused ( n -- bool )
  \ loop through all set elements
  usedcounter @ 0 ?do
    \ compare n with elt i
    dup used i cells + @
    \ return true if its a match
    = if drop -1 unloop exit then
  loop
  \ return false
  drop 0
;```

to be like this:

```forth
\ check if a number is a member of the used set
: isused ( n -- bool )
  \ loop through all set elements
  usedcounter @ 0 ?do
    \ compare n with elt i
    dup used i cells + @
    \ return true if its a match
    = if drop -1 unloop exit then
  loop
  \ return false
  drop 0
;```

If you are meticulous about documenting how the stack is used, it is much easier to catch low-level mistakes.
It can make it harder to read and follow the higher-level flow of the word, however, so use this technique judiciously. For example, you might write a word this way, test it, and then convert it into the first version (shown above) once you are confident that it is correct.

**Overview**

I suggest representing states in two ways:

- Store three values on the stack for a single state: *near*, *m*, and *c*.*near* is true if the boat is on the near side of the river. *m* and *c* are the number of missionaries and cannibals on the near side of the river, respectively.

- Store all three values in a single integer. This is the “packed” format. While less convenient to work with, this format makes it easier to store states in stacks and sets.

The overall flow of the program will follow this pseudo-code:

- push the start state on the candidate stack
- search:
  - print the candidate stack
  - pop a candidate state off the candidate stack
  - push a copy on the bread-crumb trail stack
  - if it is the goal state
    - print out the contents of the bread-crumb trail in order. this is the solution to the puzzle.
  - else
    - generate a list of successor states (there should be exactly 5)
    - push the valid, legal, fresh successors on the candidate stack
    - for each successor generated in this step:
      - call search recursively
    - pop the state off the bread-crumb trail stack

I suggest writing lots of helper words. Avoid complexity in the search word, since it will already be the most difficult part to test. What follows is one suggested implementation. You are free to write it a different way if you would prefer.

**Implementation**

Start by writing some helper words. The first few are described adequately by their names and stack-effect comments:

```
; 3dup ( x y z -- x y z x y z ) ... ;
; 3drop ( x y z -- ) ... ;
; pack ( near m c -- packedstate ) ... ;
; unpack ( packedstate -- near m c ) ... ;
; printstate ( side m c -- ) ... ;
```

Write each of these and test them thoroughly. To load code from a source file, give the name of the source file when loading gforth:

```
gforth missionaries.fs
```

Next, implement basic stack and set data structures:

```
\ test if n is in the used set
: isused ( n -- bool ) ... ;

\ add n to the used set
: addused ( n -- ) ... ;

\ push a value on the candidate stack
: pushcandidate ( n -- ) ... ;

\ pop a value off the candidate stack
: popcandidate ( -- n ) ... ;

\ push a value on the bread crumb trail stack
: pushcrumb ( n -- ) ... ;
```
For debugging, I suggest writing words to print out the entire contents of each of these data structures:

\pop	a	value	off	the	bread	crumb	trail	stack
\popcrumb	(...)
\printcontents
to	print	out	the	entire	contents	of	each	of	these
data	structures:

\printcontents	of	the	used	set	in	order
\printcontents	of	the
candidate	stack	in	order
\printcontents	of	the	bread	crumb	trail	in	order

Next, start the words to work with states:

\push	the	starting	state	on	the	stack
\startstate	(...)
\isgoal	the	state	is	the	goal	state
\isvalid	the	state	is	valid	and	legal
\addcandidate	requires	a	bit	more	explanation. It	should:

\addcandidate
to	the
candidate	stack	if	it	is	valid	and	new
\addcandidate	invalid,	repeat,	or	fresh
\addcandidate
\successors	find	all	successor	candidates	for	the
given	state,	push	them	on	stack
\successors

The next two generate, record, and report on potential next moves from a given state:

\addcandidate
to	the
candidate	stack	if	it	is	valid	and	new
\addcandidate	invalid,	repeat,	or	fresh
\addcandidate
\successors

addcandidate	requires	a	bit	more	explanation. It	should:

• check	if	the	(packed)	state	is	valid	(reject	it	and	print	a	message	if
not)
• check	if	the	(packed)	state	is	already	used	(reject	it	and	print	a	message	if
not)
• otherwise:
  • print	a	message	indicating	that	it	is	a	fresh	state
  • add	the	packed	state
to	the	used	list
  • add	the	packed	state
to	the	candidate	stack

addcandidate	is	a	helper	for	successors, which	generates	all	(5)	possible	successors	for	a
given	state,
handing
each	one
to	addcandidate, which	filters	some	of	them	out	and	adds	the	rest	to	the	candidate	stack.

Then comes	the	main	search	loop:

\search	assumes	that	a	state	is	already	on	the	candidate	stack,	and	it	uses	the	words	already
described
to	implement	the	pseudo-code
given
earlier. For	a	word
to
call	itself	recursively,	it	uses	the	special	word	recurse
instead	of	the	normal	word	name\(\search\)\, in\this\ case).

Finally,
write	a	start	word	that	resets	the	stacks	and	the	used	set,	puts	the	start	state	on	the
candidate	stack,
adds	it
to	the	used	set,	and
calls\(\search\).

Example

Here	is	the	output	of	my	solution:

\$ gforth	missionaries.fs
redefined	search	Gforth	0.7.2,	Copyright	(C)	1995-2008	Free	Software	Foundation,	Inc.
Gforth	comes	with	ABSOLUTELY	NO	WARRANTY; for
details	type	`license'
Type	`bye'	to	exit
start
candidates:
[ near 3 3 ]
fresh [ far 3 2 ]
fresh [ far 3 1 ]
fresh [ far 2 2 ]
invalid [ far 2 3 ]
invalid [ far 1 3 ]

candidates:
[ far 3 2 ]
[ far 3 1 ]
[ far 2 2 ]
invalid [ near 2 3 ]
invalid [ near 2 4 ]
repeat [ near 3 3 ]
fresh [ near 3 2 ]
invalid [ near 4 2 ]

candidates:
[ far 3 2 ]
[ far 3 1 ]
[ near 3 2 ]
repeat [ far 3 1 ]
fresh [ far 3 0 ]
invalid [ far 2 1 ]
repeat [ far 2 2 ]
invalid [ far 1 2 ]

candidates:
[ far 3 2 ]
[ far 3 1 ]
[ far 3 0 ]
fresh [ near 3 1 ]
repeat [ near 3 2 ]
invalid [ near 4 1 ]
invalid [ near 4 0 ]
invalid [ near 5 0 ]

candidates:
[ far 3 2 ]
[ far 3 1 ]
[ near 3 1 ]
repeat [ far 3 0 ]
invalid [ far 3 -1 ]
invalid [ far 2 0 ]
invalid [ far 2 1 ]
fresh [ far 1 1 ]

candidates:
[ far 3 2 ]
[ far 3 1 ]
[ far 1 1 ]
invalid [ near 1 2 ]
invalid [ near 1 3 ]
fresh [ near 2 2 ]
invalid [ near 2 1 ]
repeat [ near 3 1 ]

candidates:
[ far 3 2 ]
[ far 3 1 ]
[ near 2 2 ]
invalid [ far 2 1 ]
invalid [ far 2 0 ]
repeat [ far 1 1 ]
invalid [ far 1 2 ]
fresh [ far 0 2 ]

candidates:
[ far 3 2 ]
[ far 3 1 ]
[ far 0 2 ]
fresh [ near 0 3 ]
invalid [ near 0 4 ]
invalid [ near 1 3 ]
invalid [ near 1 2 ]
repeat [ near 2 2 ]

candidates:
[ far 3 2 ]
[ far 3 1 ]
[ near 0 3 ]
repeat [ far 0 2 ]
fresh [ far 0 1 ]
invalid [ far -1 2 ]
invalid [ far -1 3 ]
invalid [ far -2 3 ]

candidates:
[ far 3 2 ]
[ far 3 1 ]
[ far 0 1 ]
fresh [ near 0 2 ]
repeat [ near 0 3 ]
invalid [ near 1 2 ]
fresh [ near 1 1 ]
invalid [ near 2 1 ]

candidates:
[ far 3 2 ]
[ far 3 1 ]
[ near 0 2 ]
[ near 1 1 ]
invalid [ far 1 0 ]
invalid [ far 1 -1 ]
fresh [ far 0 0 ]
repeat [ far 0 1 ]
invalid [ far -1 1 ]

candidates:
[ far 3 2 ]
[ far 3 1 ]
[ near 0 2 ]
[ far 0 0 ]
solution found

-------------
[ near 3 3 ]
[ far 2 2 ]
[ near 3 2 ]
[ far 3 0 ]
[ near 3 1 ]
[ far 1 1 ]
[ near 2 2 ]
[ far 0 2 ]
[ near 0 3 ]
[ far 0 1 ]
[ near 1 1 ]
[ far 0 0 ]
backtracking
backtracking

candidates:
[ far 3 2 ]
[ far 3 1 ]
[ near 0 2 ]
repeat [ far 0 1 ]
repeat [ far 0 0 ]
invalid [ far 0 0 ]
invalid [ far -1 1 ]
invalid [ far -1 2 ]
invalid [ far -2 2 ]
backtracking
backtracking
backtracking
backtracking
backtracking
backtracking
backtracking
backtracking
backtracking
candidates:
[ far 3 2 ]
[ far 3 1 ]
repeat  [ near 3 2 ]
repeat  [ near 3 3 ]
invalid [ near 4 2 ]
invalid [ near 4 1 ]
invalid [ near 5 1 ]
backtracking
candidates:
[ far 3 2 ]
repeat  [ near 3 3 ]
invalid [ near 3 4 ]
invalid [ near 4 3 ]
invalid [ near 4 2 ]
invalid [ near 5 2 ]
backtracking
backtracking
ok