

## Midterm 1 – Answers

Name: Max Pettersson

By submitting my midterm exam, I Max Pettersson guarantee that the solutions uploaded here are my own, and that I have not in any way obtained or given improper assistance.

### Task a)

I chose the task: *Autonomous vehicles and aircraft*

The task environment is:

1. Partially observable, because the environment where an autonomous vehicle or aircraft operates is too complex to be able to observe the full state. Since the autonomous vehicle uses sensors to map the environment, it creates an abstraction or representation of the environment that will have lesser detail, thus it will always be a partial observation in comparison to the actual environment (at least with the sensors that exist today).
2. Multiagent, because autonomous vehicles and aircraft will operate in an environment with other autonomous vehicles and aircraft. These agents can function both in competition, like in military applications, or co-operation, like in a connected city traffic network.
3. Stochastic, because, as stated in 1., the environment is only partially observable and too complex to be able to guarantee that you can predict future states based on the current state. Although this may be a computational limitation since even in a stochastic or chaotic system you could predict future states, given enough computational power so that you can read enough variables in the current state.
4. Sequential, because the environment (real life) that the autonomous vehicles and aircraft operate in is sequential by nature, cause and effect apply.
5. Dynamic, because the environment changes regardless of what the agent is doing. For autonomous vehicles and aircraft that would also mean that performance measure is affected by time.
6. Continuous, because the environment is dynamic and sequential. Since time affects performance measures, autonomous vehicles should, ideally, read the environment in real time. This is best done continuously.

### Task b)

I found the paper:

Y. Ma, H. Liu, Y. Zhang, Q. He and Z. Xu, "Intelligent decision making for UAV based on Monte Carlo Simulation," *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, Singapore, 2018, pp. 521-525.  
doi: 10.1109/ICARCV.2018.8581333

<http://ieeexplore.ieee.org.proxy.library.ju.se/stamp/stamp.jsp?tp=&arnumber=8581333&isnumber=8580630>

My search methodology was: I went to <http://www.conferenceranks.com> and searched for "robotics" and then choose the A-ranked conference "International Conference on Control, Automation, Robotics and Vision (ICARCV)" to search for a paper. I browsed through the 2018 program papers and found several interesting ones about autonomous vehicles. I took the title of these and searched for them in IEEE Xplore (they weren't

archived on the ICARCV 2018 web page). I read the abstract for the four or so papers I picked out and chose “Intelligent decision making for UAV based on Monte Carlo Simulation” because it seemed relevant and interesting.

**Task c)**

The paper’s problem statement/objective is to study how to develop an autonomous maneuvering strategy for an UAV against a hostile UAV. They use Monte Carlo simulation to predict an opposing UAV’s movements, and a multidimensional cloud model for situation assessment.

The methodology used is: Simulation. Matlab is used to simulate the two UAVs and a monte carlo simulation is run 1000 times for one-time, two-time, three-time and four-time simulation. The results of the confrontation simulation are compared across these four types of simulations.

The main conclusions of the paper are: a viable autonomous maneuvering strategy can be developed using monte carlo simulation and cloud a multidimensional cloud model. In the case of two agents acting as competitors (one UAV tries to catch the other).

2)

The rest of the code is included as an appendix at the end of this document.

a)

## GPS

```
1.  int generalProblemSolver(List* SOLUTION)
2.  {
3.      Problem problem;
4.      Node N;
5.      twoJugsProblem(&problem);
6.      //manWolfGoatCabbageProblem(&problem);
7.
8.      List OPEN;
9.      listInit(&OPEN);
10.
11.     /*
12.     * 1. Put the initial state S in an empty list called OPEN.
13.     * OPEN contains nodes for expansion.
14.     */
15.     listAppend(&OPEN, problem.initialState);
16.
17.
18.     /*
19.     * 2. Create an empty list CLOSED which
20.     * contains already expanded nodes.
21.     */
22.     List CLOSED;
23.     listInit(&CLOSED);
24.
25.     while(1)
26.     {
27.         /*
28.         * 3. If OPEN is empty, finish with failure.
29.         */
30.         if(listIsEmpty(&OPEN))
31.         {
32.             int failed = 0;
33.             return failed;
34.         }
35.
36.
37.         /*
38.         * 4. Select the first node N in OPEN. Remove N from OPEN and place it in CLOSED.
39.         */
40.         N = listPopFront(&OPEN); //Breadth first
41.         //N = listPopBack(&OPEN); //Depth first
42.         listAppend(&CLOSED, N);
43.
44.
45.         /*
46.         * 5. If N represents a goal state, finish with success.
47.         * (We've found a way from S to N)
48.         */
49.         if(nodeIsGoal(problem, N))
50.         {
51.             getSolutionPath(SOLUTION, &CLOSED, problem, N);
52.             int success = 1;
53.             return success;
54.         }
55.
56.
57.         /*
58.         * 6. Expand N, i.e. use the rules to find a set, NEW, of nodes that we can reach from N.
59.         * Add the nodes in NEW that are not already in OPEN or CLOSED, to OPEN.
60.         */
```

```

61.     for(int ruleIterator = 1; ruleIterator <= problem.nRules; ruleIterator++)
62.     {
63.         // Iterate through the rules and return a viable expanded node
64.         Node NEW = problem.rules(N, ruleIterator);
65.
66.         // -1 means that the node could not be expanded for the specific rule it checked
67.         if(NEW.element[0] != -1)
68.         {
69.             //check if expanded node exists in the lists
70.             if(!listContains(&OPEN, NEW) && !listContains(&CLOSED, NEW))
71.             {
72.                 addParent(&N, &NEW);
73.                 listAppend(&OPEN, NEW);
74.             }
75.         }
76.     }
77.     /*
78.     * 7. GoTo 3
79.     */
80. }
81. }

```

## Two jugs specific rules

```

1. Node rulesTwoJugs(Node N, int iCase)
2. {
3.     switch(iCase)
4.     {
5.         case 1:
6.             if(N.element[0] < 4 )
7.             {
8.                 N.element[0] = 4;
9.                 return N;
10.            }
11.            break;
12.
13.         case 2:
14.             if(N.element[1] < 3 )
15.             {
16.                 N.element[1] = 3;
17.                 return N;
18.            }
19.            break;
20.
21.         case 3:
22.             if(N.element[0] > 0 )
23.             {
24.                 N.element[0] = 0;
25.                 return N;
26.            }
27.            break;
28.
29.         case 4:
30.             if(N.element[1] > 0 )
31.             {
32.                 N.element[1] = 0;
33.                 return N;
34.            }
35.            break;
36.
37.         case 5:
38.             if(((N.element[0] + N.element[1]) >= 4) && N.element[1] > 0)
39.             {
40.                 N.element[0] = 4;
41.                 N.element[1] = N.element[1] - (4 - N.element[0]);
42.                 return N;

```

```

43.     }
44.     break;
45.
46.     case 6:
47.         if(((N.element[0] + N.element[1]) >= 3) && N.element[0] > 0)
48.         {
49.             N.element[0] = N.element[0] - (3 - N.element[1]);
50.             N.element[1] = 3;
51.             return N;
52.         }
53.         break;
54.
55.     case 7:
56.         if(((N.element[0] + N.element[1]) <= 4) && N.element[1] > 0)
57.         {
58.             N.element[0] = N.element[0] + N.element[1];
59.             N.element[1] = 0;
60.             return N;
61.         }
62.         break;
63.
64.     case 8:
65.         if(((N.element[0] + N.element[1]) <= 3) && N.element[0] > 0)
66.         {
67.             N.element[1] = N.element[0] + N.element[1];
68.             N.element[0] = 0;
69.
70.             return N;
71.         }
72.         break;
73.     }
74.
75.     N.element[0] = -1;
76.     return N;
77. }
78.
79. void twoJugsProblem(Problem *problem)
80. {
81.     problem->initialState.nUsedElements = 2;
82.     problem->initialState.element[0] = 0;
83.     problem->initialState.element[1] = 0;
84.
85.     problem->goalState.element[0] = 2;
86.     problem->goalState.element[1] = 0;
87.     problem->nRules = 8;
88.     problem->rules = rulesTwoJugs;
89.
90. }

```

## Structures

```

1.  typedef struct {
2.      int element[10];
3.      int parentElement[10];
4.      int nUsedElements;
5.  }Node;
6.
7.  typedef struct List{
8.      Node nodeBuff[BUFSIZE];
9.      Node *pBuffBegin;
10.     Node *pBuffEnd;
11.     int size;
12. }List;
13.
14. typedef struct{
15.     Node initialState;

```

```

16. Node goalState;
17. int nRules;
18. Node (*rules)(Node N, int iCase);
19. }Problem;

```

## Two jugs solution print-out

### Breadth first

```

Terminal
Serial COM4 (2019-09-14 17:13)
Solution found!
(0,0)
(4,0)
(1,3)
(1,0)
(0,1)
(4,1)
(2,3)
(2,0)

```

### Depth first

```

Serial COM4 (2019-09-14 17:13)
Solution found!
(0,0)
(4,0)
(1,3)
(1,0)
(0,1)
(4,1)
(2,3)
(2,0)

```

I also had the search and solution presented with LEDs for the jugs on a microcontroller, however I could not find a way to attach an animated gif to this word document.

b)

### Man, wolf, goat and cabbage specific rules and initiation:

```

1. Node rulesManWolfGoatCabbage(Node N, int iCase)
2. {
3.     // The order goes (Man, wolf, goat, cabbage)
4.     // 0 = starting side of the river
5.     // 1 = opposite side of the river
6.     switch(iCase)
7.     {
8.     case 1:
9.         if((N.element[1] != N.element[2]) && // Move man alone
10.            (N.element[2] != N.element[3]))
11.         {
12.             N.element[0] = !N.element[0];
13.             return N;
14.         }
15.         break;
16.
17.     case 2:
18.         if(N.element[2] != N.element[3] && // Move wolf
19.            N.element[0] == N.element[1])
20.         {
21.             N.element[0] = !N.element[0];
22.             N.element[1] = !N.element[1];
23.
24.             return N;
25.         }
26.         break;
27.
28.     case 3:
29.         if(N.element[0] == N.element[2]) // Move goat
30.         {
31.             N.element[0] = !N.element[0];
32.             N.element[2] = !N.element[2];
33.

```

```

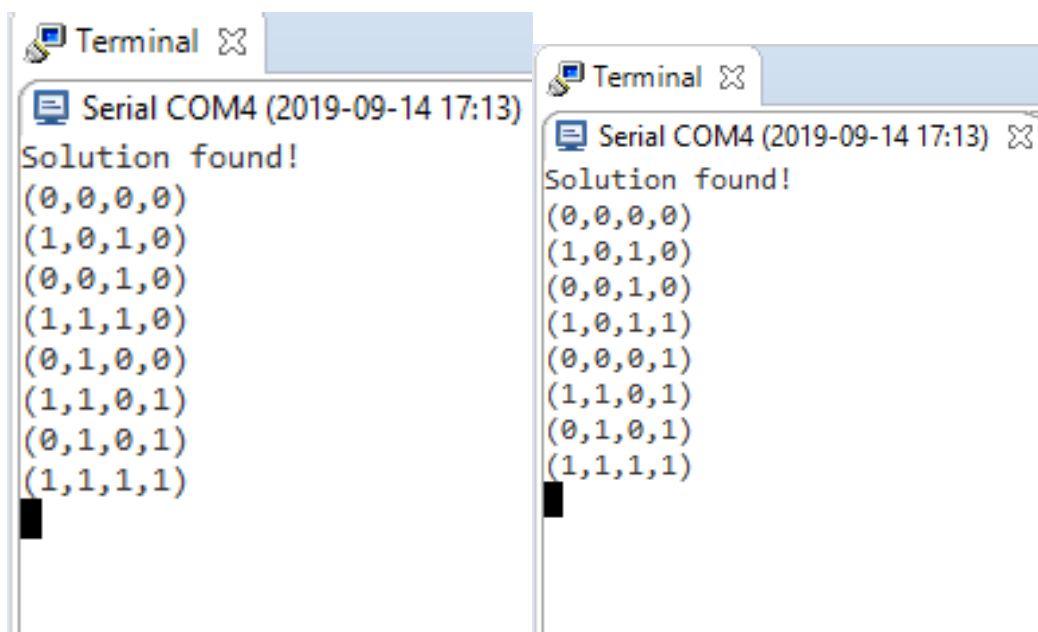
34.     return N;
35. }
36. break;
37.
38. case 4:
39.     if(N.element[1] != N.element[2] && // Move cabbage
40.        N.element[0] == N.element[3])
41.     {
42.         N.element[0] = !N.element[0];
43.         N.element[3] = !N.element[3];
44.
45.         return N;
46.     }
47.     break;
48. }
49.
50. N.element[0] = -1;
51. return N;
52. }
53.
54. void manWolfGoatCabbageProblem(Problem *problem)
55. {
56.     problem->initialState.nUsedElements = 4;
57.     problem->initialState.element[0] = 0;
58.     problem->initialState.element[1] = 0;
59.     problem->initialState.element[2] = 0;
60.     problem->initialState.element[3] = 0;
61.
62.     problem->goalState.element[0] = 1;
63.     problem->goalState.element[1] = 1;
64.     problem->goalState.element[2] = 1;
65.     problem->goalState.element[3] = 1;
66.     problem->nRules = 4;
67.     problem->rules = rulesManWolfGoatCabbage;
68.
69. }

```

**Man, wolf, goat and cabbage solution print-out. (Man, Wolf, Goat, Cabbage).**

**breadth first**

**depth first right.**



```

Terminal Serial COM4 (2019-09-14 17:13)
Solution found!
(0,0,0,0)
(1,0,1,0)
(0,0,1,0)
(1,1,1,0)
(0,1,0,0)
(1,1,0,1)
(0,1,0,1)
(1,1,1,1)

Terminal Serial COM4 (2019-09-14 17:13)
Solution found!
(0,0,0,0)
(1,0,1,0)
(0,0,1,0)
(1,0,1,1)
(0,0,0,1)
(1,1,0,1)
(0,1,0,1)
(1,1,1,1)

```

3)

a)

The state is defined as (A, B), where A = Alan's current city, and B = Bob's current city. Initial state would then be (Kiruna, Malmö). Goal state would be any state where the first and second element are equal, as in (A, A).

Operations/rules:

If  $\text{nextTo}(A/B, X/Y)$  is true, where  $X/Y$  is any city on the map that is not  $A/B$ , move to the  $X/Y$  with the least associated cost. Then if  $\text{nextTo}(A, G)$  and  $\text{nextTo}(B, G)$  are both true, as in Alan and Bob found the same adjacent city (G), always move to city G. Step cost would be calculated as the sum of the travel cost of both Alan and Bob to one city.

b)

**Solution with above rules:**

(Kiruna, Malmö)  $14+34 \rightarrow$  (Luleå, Halmstad)  $13+26 \rightarrow$  (Umeå, Göteborg)  $7+55 \rightarrow$   
(Falun, Borås)  $8+18 \rightarrow$  (Örebro, Jönköping)  $11+12 \rightarrow$  (Linköping, Linköping) = 198 cost.

This is not the minimum cost solution for the whole problem though.

c)

i.  $h(x,y)$  would be admissible, because the straight line between two cities is always the shortest and ideal path. Because of this,  $h(x,y)$  would never overestimate, and thus be admissible.



4)

4.1

```
a) killer(butch).
```

```
b)
marriage(mia, marsellus).
married(X, Y) :- marriage(X,Y).
married(X, Y) :- marriage(Y,X). % This way the marriage goes both ways
```

```
c) dead(zed).
```

```
d)
kills(marsellus, X) :-
    footmassage(X, mia).
```

```
e)
loves(mia, X) :-
    dancer(X),
    good(dancing, X).
```

```
f)
eats(jules, X) :-
    nutritious(X);
    tasty(X).
```

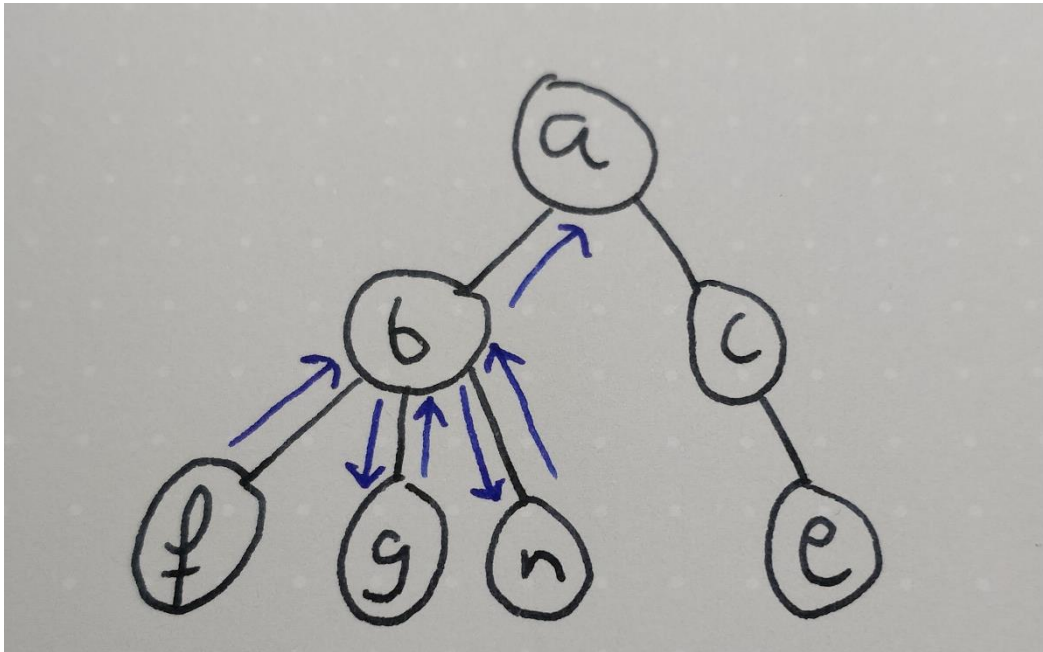
4.2

```
directly_in(irina, natasha).
directly_in(natasha, olga).
directly_in(olga, katarina).

is_inside(X, Y) :-
    directly_in(X, Y).
is_inside(X, Y) :-
    directly_in(X, Z),
    is_inside(Z, Y).
```

4.3

The query fails because “g” is not defined in the program. However, if “g” was to be defined the same as “e”, “f” and “n” it would succeed and use backtracking, and the diagram would be as follows:



The blue arrows are where the program backtracks.

4.4

a)

```
combine1([], A, A).  
combine1([H|T], [H2|T2], [H,H2|Result]) :-  
    combine1(T, T2, Result).
```

b)

## Appendix, all code written for the GPS.

### Main.c

```
1.  while (1)
2.  {
3.
4.  if(generalProblemSolver(&solution))
5.  {
6.      Node tempNode;
7.      int listSize = solution.size;
8.      HAL_UART_Transmit(&huart2, "Solution found!\r\n", 17, 500);
9.      for(int i = 0; i < listSize; i++)
10.     {
11.         tempNode = listPopBack(&solution);
12.         sprintf(uartSend, "(%d,%d)", tempNode.element[0], tempNode.element[1]);
13.         //sprintf(uartSend, "(%d,%d,%d,%d)", tempNode.element[0], tempNode.element[1], tempNode.element[2], tempNode.element[3]);
14.         HAL_UART_Transmit(&huart2, uartSend, 5, 500);
15.         HAL_UART_Transmit(&huart2, "\r\n", 2, 500);
16.
17.     }
18. }
19. else
20.     HAL_UART_Transmit(&huart2, "FAILED!", 7, 500);
21.
22. HAL_Delay(10000);
23. /* USER CODE END WHILE */
24.
25. /* USER CODE BEGIN 3 */
26. }
27. /* USER CODE END 3 */
```

### GPS.h:

```
1.  /*
2.  * GPS.h
3.  *
4.  * Created on: 9 sep. 2019
5.  * Author: Max
6.  */
7.
8.  #ifndef GPS_H_
9.  #define GPS_H_
10. #define BUFFSIZE 200
11. #define NODE_ELEMENTS 10
12. #define BREADTH_FIRST 1
13.
14.
15. #include <stdio.h>
16. #include "datatypes.h"
17.
18.
19.
20. typedef struct {
21.     int element[10];
22.     int parentElement[10];
```

```

23.     int nUsedElements;
24. }Node;
25.
26. typedef struct List{
27.     Node nodeBuff[BUFFSIZE];
28.     Node *pBuffBegin;
29.     Node *pBuffEnd;
30.     int size;
31. }List;
32.
33. typedef struct{
34.     Node initialState;
35.     Node goalState;
36.     int nRules;
37.     Node (*rules)(Node N, int iCase);
38. }Problem;
39.
40. // ***** //
41. void listInit(List *list);
42. void listAppend(List *list, Node node);
43. Node listPopFront(List *list);
44. Node listPopBack(List *list);
45. bool listIsEmpty(List* list);
46. Node listElementAt(List *list, int element);
47. bool listContains(List *list, Node comparisonNode);
48. Node listCompareAndRead(List *list, Node comparisonNode);
49.
50. void copyNode(Node* original, Node* copy);
51. Node expandNode(Problem problem, Node open, Node closed, Node New);
52. bool nodeIsGoal(Problem problem, Node N);
53. void addParent(Node *parent, Node *child);
54. Node getParent(Node child);
55. bool nodeIsSame(Node original, Node comparisonNode);
56. void getSolutionPath(List *SOLUTION, List *CLOSED, Problem problem, Node N);
57.
58. Node rulesTwoJugs(Node N, int iCase);
59. Node rulesManWolfGoatCabbage(Node N, int iCase);
60. void manWolfGoatCabbageProblem(Problem *problem);
61. void twoJugsProblem(Problem *problem);
62. int generalProblemSolver(List *SOLUTION);
63. #endif /* GPS_H_ */

```

## GPS.c

```

1.  /*
2.   * GPS.c
3.   *
4.   * Created on: 9 sep. 2019
5.   * Author: Max
6.   */
7.
8. #include "GPS.h"
9.
10. Node rulesTwoJugs(Node N, int iCase)
11. {
12.     switch(iCase)
13.     {
14.     case 1:
15.         if(N.element[0] < 4 )
16.         {
17.             N.element[0] = 4;
18.             return N;
19.         }
20.         break;
21.
22.     case 2:

```

```

23.         if(N.element[1] < 3 )
24.         {
25.             N.element[1] = 3;
26.             return N;
27.         }
28.         break;
29.
30.     case 3:
31.         if(N.element[0] > 0 )
32.         {
33.             N.element[0] = 0;
34.             return N;
35.         }
36.         break;
37.
38.     case 4:
39.         if(N.element[1] > 0 )
40.         {
41.             N.element[1] = 0;
42.             return N;
43.         }
44.         break;
45.
46.     case 5:
47.         if(((N.element[0] + N.element[1]) >= 4) && N.element[1] > 0)
48.         {
49.             N.element[0] = 4;
50.             N.element[1] = N.element[1] - (4 - N.element[0]);
51.             return N;
52.         }
53.         break;
54.
55.     case 6:
56.         if(((N.element[0] + N.element[1]) >= 3) && N.element[0] > 0)
57.         {
58.             N.element[0] = N.element[0] - (3 - N.element[1]);
59.             N.element[1] = 3;
60.             return N;
61.         }
62.         break;
63.
64.     case 7:
65.         if(((N.element[0] + N.element[1]) <= 4) && N.element[1] > 0)
66.         {
67.             N.element[0] = N.element[0] + N.element[1];
68.             N.element[1] = 0;
69.             return N;
70.         }
71.         break;
72.
73.     case 8:
74.         if(((N.element[0] + N.element[1]) <= 3) && N.element[0] > 0)
75.         {
76.             N.element[1] = N.element[0] + N.element[1];
77.             N.element[0] = 0;
78.
79.             return N;
80.         }
81.         break;
82.     }
83.
84.     N.element[0] = -1;
85.     return N;
86. }
87.
88. void twoJugsProblem(Problem *problem)

```

```

89. {
90.     problem->initialState.nUsedElements = 2;
91.     problem->initialState.element[0] = 0;
92.     problem->initialState.element[1] = 0;
93.
94.     problem->goalState.element[0] = 2;
95.     problem->goalState.element[1] = 0;
96.     problem->nRules = 8;
97.     problem->rules = rulesTwoJugs;
98.
99. }
100.
101.
102.
103. Node rulesManWolfGoatCabbage(Node N, int iCase)
104. {
105.     // The order goes (Man, wolf, goat, cabbage)
106.     // 0 = starting side of the river
107.     // 1 = opposite side of the river
108.     switch(iCase)
109.     {
110.     case 1:
111.         if((N.element[1] != N.element[2]) && // Move man alone
112.            (N.element[2] != N.element[3]))
113.         {
114.             N.element[0] = !N.element[0];
115.             return N;
116.         }
117.         break;
118.
119.     case 2:
120.         if(N.element[2] != N.element[3] && // Move wolf
121.            N.element[0] == N.element[1])
122.         {
123.             N.element[0] = !N.element[0];
124.             N.element[1] = !N.element[1];
125.
126.             return N;
127.         }
128.         break;
129.
130.     case 3:
131.         if(N.element[0] == N.element[2]) // Move goat
132.         {
133.             N.element[0] = !N.element[0];
134.             N.element[2] = !N.element[2];
135.
136.             return N;
137.         }
138.         break;
139.
140.     case 4:
141.         if(N.element[1] != N.element[2] && // Move cabbage
142.            N.element[0] == N.element[3])
143.         {
144.             N.element[0] = !N.element[0];
145.             N.element[3] = !N.element[3];
146.
147.             return N;
148.         }
149.         break;
150.     }
151.
152.     N.element[0] = -1;
153.     return N;
154. }

```

```

155. void manWolfGoatCabbageProblem(Problem *problem)
156. {
157.     problem->initialState.nUsedElements = 4;
158.     problem->initialState.element[0] = 0;
159.     problem->initialState.element[1] = 0;
160.     problem->initialState.element[2] = 0;
161.     problem->initialState.element[3] = 0;
162.
163.     problem->goalState.element[0] = 1;
164.     problem->goalState.element[1] = 1;
165.     problem->goalState.element[2] = 1;
166.     problem->goalState.element[3] = 1;
167.     problem->nRules = 4;
168.     problem->rules = rulesManWolfGoatCabbage;
169.
170. }
171.
172. void listInit(List *list)
173. {
174.     list->pBuffBegin = list->nodeBuff;
175.     list->pBuffEnd = list->nodeBuff;
176.     list->size = 0;
177. }
178.
179. void listAppend(List *list, Node node)
180. {
181.     if(list->pBuffEnd > &list->nodeBuff[BUFFSIZE-1])
182.     {
183.         list->pBuffEnd = &list->nodeBuff[0];
184.     }
185.     *list->pBuffEnd = node;
186.     list->pBuffEnd++;
187.     list->size++;
188. }
189.
190. Node listPopFront(List *list)
191. {
192.     Node temp;
193.     temp.element[0] = -1;
194.     if(list->pBuffBegin != list->pBuffEnd)
195.     {
196.         if(list->pBuffBegin > &list->nodeBuff[BUFFSIZE-1])
197.         {
198.             list->pBuffBegin = &list->nodeBuff[0];
199.         }
200.         temp = *list->pBuffBegin;
201.         list->pBuffBegin++;
202.         list->size--;
203.     }
204.
205.     return temp;
206. }
207.
208. Node listPopBack(List *list)
209. {
210.     Node temp;
211.     temp.element[0] = -1;
212.     if(list->pBuffBegin != list->pBuffEnd)
213.     {
214.         list->pBuffEnd--;
215.         temp = *list->pBuffEnd;
216.         list->size--;
217.     }
218.
219.     return temp;
220. }

```

```

221.
222.
223. bool listIsEmpty(List* list)
224. {
225.     if(list->pBuffBegin == list->pBuffEnd)
226.     {
227.         return true;
228.     }
229.     else
230.     {
231.         return false;
232.     }
233. }
234.
235. Node listElementAt(List *list, int element)
236. {
237.     Node temp;
238.     Node *pTemp;
239.     pTemp = list->pBuffBegin;
240.     temp.element[0] = -1;
241.     if(list->pBuffBegin != list->pBuffEnd)
242.     {
243.         for(int i = 0; i < element; i++)
244.         {
245.             list->pBuffBegin++;
246.             if(list->pBuffBegin > &list->nodeBuff[BUFSIZE-1])
247.             {
248.                 list->pBuffBegin = &list->nodeBuff[0];
249.             }
250.         }
251.         temp = *list->pBuffBegin;
252.         list->pBuffBegin = pTemp;
253.     }
254.
255.     return temp;
256. }
257.
258. Node listCompareAndRead(List *list, Node comparisonNode)
259. {
260.     for(int iList = 0; iList < list->size; iList++)
261.     {
262.         Node listTempNode = listElementAt(list, iList);
263.         int correct = 0;
264.
265.         for(int i = 0; i < comparisonNode.nUsedElements; i++)
266.         {
267.             if(comparisonNode.element[i] == listTempNode.element[i])
268.             {
269.                 correct++;
270.             }
271.         }
272.         if(correct == comparisonNode.nUsedElements)
273.         {
274.             return listTempNode;
275.         }
276.     }
277. }
278. }
279.
280. bool listContains(List *list, Node comparisonNode)
281. {
282.     for(int iList = 0; iList < list->size; iList++)
283.     {
284.         Node listTempNode = listElementAt(list, iList);
285.         int correct = 0;
286.

```



```

287.     for(int i = 0; i < comparisonNode.nUsedElements; i++)
288.     {
289.         if(comparisonNode.element[i] == listTempNode.element[i])
290.         {
291.             correct++;
292.         }
293.     }
294.     if(correct == comparisonNode.nUsedElements)
295.         return true;
296. }
297.
298. return false;
299.}
300.
301.bool nodeIsGoal(Problem problem, Node N)
302.{
303.    int correct = 0;
304.    for(int i = 0; i < N.nUsedElements; i++)
305.    {
306.        if(N.element[i] == problem.goalState.element[i])
307.        {
308.            correct++;
309.        }
310.    }
311.
312.    if(correct == N.nUsedElements)
313.        return true;
314.    else
315.        return false;
316.}
317.}
318.
319.
320.void addParent(Node *parent, Node*child)
321.{
322.    for(int i = 0; i < parent->nUsedElements; i++)
323.    {
324.        child->parentElement[i] = parent->element[i];
325.    }
326.}
327.
328.Node getParent(Node child)
329.{
330.    Node temp;
331.    temp.nUsedElements = child.nUsedElements;
332.    for(int i = 0; i < child.nUsedElements; i++)
333.    {
334.        temp.element[i] = child.parentElement[i];
335.    }
336.
337.    return temp;
338.}
339.
340.bool nodeIsSame(Node original, Node comparisonNode)
341.{
342.    int correct = 0;
343.    for(int i = 0; i < original.nUsedElements; i++)
344.    {
345.        if(original.element[i] == comparisonNode.element[i])
346.        {
347.            correct++;
348.        }
349.    }
350.
351.    if(correct == original.nUsedElements)
352.        return true;

```

```

353.     else
354.         return false;
355. }
356.
357. void getSolutionPath(List *SOLUTION, List *CLOSED, Problem problem, Node N)
358. {
359.     Node solutionPathNode;
360.     solutionPathNode = N;
361.     listAppend(SOLUTION, solutionPathNode);
362.
363.     while(!nodeIsSame(solutionPathNode, problem.initialState))
364.     {
365.         solutionPathNode = getParent(solutionPathNode);
366.         if(listContains(CLOSED, solutionPathNode))
367.         {
368.             solutionPathNode = listCompareAndRead(CLOSED, solutionPathNode);
369.             listAppend(SOLUTION, solutionPathNode);
370.         }
371.     }
372.
373. }
374.
375.
376. int generalProblemSolver(List* SOLUTION)
377. {
378.     Problem problem;
379.     Node N;
380.     twoJugsProblem(&problem);
381.     //manWolfGoatCabbageProblem(&problem);
382.
383.     List OPEN;
384.     listInit(&OPEN);
385.     List NEW;
386.     listInit(&NEW);
387.
388.     /*
389.      * 1. Put the initial state S in an empty list called OPEN.
390.      * OPEN contains nodes for expansion.
391.      */
392.     listAppend(&OPEN, problem.initialState);
393.
394.
395.     /*
396.      * 2. Create an empty list CLOSED which
397.      * contains already expanded nodes.
398.      */
399.     List CLOSED;
400.     listInit(&CLOSED);
401.
402.     while(1)
403.     {
404.         /*
405.          * 3. If OPEN is empty, finish with failure.
406.          */
407.         if(listIsEmpty(&OPEN))
408.         {
409.             int failed = 0;
410.             return failed;
411.         }
412.
413.
414.         /*
415.          * 4. Select the first node N in OPEN. Remove N from OPEN and place it
in CLOSED.
416.          */
417.         //N = listPopFront(&OPEN); //Breadth first

```

```

418.     N = listPopBack(&OPEN); //Depth first
419.     listAppend(&CLOSED, N);
420.
421.
422.     /*
423.      * 5. If N represents a goal state, finish with success.
424.      * (We've found a way from S to N)
425.      */
426.     if(nodeIsGoal(problem, N))
427.     {
428.         getSolutionPath(SOLUTION, &CLOSED, problem, N);
429.         int success = 1;
430.         return success;
431.     }
432.
433.
434.     /*
435.      * 6. Expand N, i.e. use the rules to find a set, NEW, of nodes that we
      can reach from N.
436.      * Add the nodes in NEW that are not already in OPEN or CLOSED, to O
      PEN.
437.      */
438.     for(int ruleIterator = 1; ruleIterator <= problem.nRules; ruleIterator+
      +)
439.     {
440.         // Iterate through the rules and return a viable expanded node
441.         Node NEW = problem.rules(N, ruleIterator);
442.
443.         // -
      1 means that the node could not be expanded for the specific rule it checked
444.         if(NEW.element[0] != -1)
445.         {
446.             //check if expanded node exists in the lists
447.             if(!listContains(&OPEN, NEW) && !listContains(&CLOSED, NEW))
448.             {
449.                 addParent(&N, &NEW);
450.                 listAppend(&OPEN, NEW);
451.             }
452.         }
453.     }
454.     /*
455.      * 7. GoTo 3
456.      */
457. }
458. }

```