

Puzzle Game Engine

Assignment for Advanced Programming

Marius Mikučionis and Simonas Šaltenis

May 20, 2019

1 Introduction

Many puzzles and real world problems can be modeled as a state-transition system, which is a special case of a graph, where nodes represent system states and edges denote transitions between them. Solving a puzzle, then, involves describing it in terms of configurations (states) and changes between them (transitions), exploring the space of configurations, finding a desired configuration and extracting the sequence of changes leading to the desired configuration.

We present three classical puzzles which can easily be modeled as state-transition systems and can be used to demonstrate the method. The puzzle-modeling C++ code is appended.

1.1 Leaping Frogs

Green and brown frogs sit on rocks opposing each other with an unoccupied rock in between them. The frogs can only jump in one direction to the next unoccupied rock or over one other frog. The goal is to find a sequence of correct moves so that frogs pass each other and swap places. The initial state of the puzzle can be described by a



Figure 1: Leaping frogs from <http://www.studentfreestuff.com/frog leap.php>.

configuration “GGG–BBB” where “G” means green frog, “B” means brown frog, dash means unoccupied rock, and “BBB–GGG” is the desired goal state. There are four possible moves from the initial state: two options for greens and two for browns, therefore four transitions to the following successor states: “GG–G BBB”, “G–GG BBB”, “GGGB–BB” and “GGGBB–B”.

1.2 Wolf, Goat and Cabbage

A farmer went to a fair and bought a cabbage, a goat and a wolf. On the way home he has to cross a river using a boat, however only one item fits on the boat with the farmer. Moreover, the goat cannot be left alone with the cabbage, and the wolf cannot be trusted alone with the goat. The goal is to find a *shortest* sequence of river crossings so that all items are transferred to another side of the river. The puzzle state can be modeled by the positions of each



(a) Wolf, goat and cabbage.



(b) Japanese version <https://www.pedagonet.com/Fun/flashgame185.htm>.

Figure 2: River crossing puzzles.

of the three items. If we denote the shore on one side of the river as 1, the shore on the other side as 2 and the boat as $-$, the initial state is “111” and the desired goal is “222”. There are three possible successors from the initial state: “-11” (the wolf is on the boat, the goat is on the shore 1, the cabbage is on the shore 1), “1-1”, “11-”, however the states “-11” and “11-” are not valid due to conflicts among items. Further the state “1-1” has two successors: “111” and “121”. Obviously the first one brings us back to the state we have already seen, and we need to detect that to avoid exploring it again.

1.3 Family Crossing

A mother, a father, two daughters, two sons, policeman and a prisoner need to cross a river using a raft which can hold only two persons at a time. Only the mother, the father and the policeman know how to operate the raft. The prisoner cannot be left alone with a family without the policeman. The daughters cannot be left alone with the father without the mother, and the sons cannot be left alone with the mother without the father. The puzzle can be modeled the same way like the wolf, goat and cabbage problem, except we have many more solutions due to symmetries among the children. Suppose the sons get bored on the shore 1 and the longer they stay the more noise they make. Can you find a crossing sequence with the least noise?

1.4 Method

Once a puzzle is modeled using states and transitions, the problem can be solved using a graph search algorithm. One can easily develop a recursive procedure to follow the transitions, however the recursion is limited to depth-first search order and may exhaust the stack if the state space is deep. Alternatively, Algorithm 1 explores the graph and checks the states on the fly using passed and waiting lists of states. It remembers the states it has explored in the `passed` list and stores unexplored states in the `waiting` list. The `popstate` method can be customized to pick the first state (resulting in breadth-first order), the last state (resulting in depth-first order), or the lowest-cost state (cost-guided order). Also, on line 6, if the same state has already been visited, the state with the least amount of transitions (or any other cost function) can be recorded in the `passed` list. For most puzzles, the newly generated states on line 8 should be checked against the puzzle-invariant predicate to discard the invalid states (e.g., the wolf left alone with the goat).

Input : Initial state `state0`, property `goal`

Output: true if there exists a state satisfying `goal`

```

1 waiting := {state0};
2 passed := ∅;
3 while waiting ≠ ∅ do
4   state := waiting.popstate();
5   if goal(state) == true then return true ;
6   if state ∉ passed then
7     passed := passed ∪ {state};
8     foreach state' such that state → state' do
9       if state' ∉ waiting then
10        | waiting := waiting ∪ {state'}
11      end
12    end
13  end
14 end
15 return false;
```

Algorithm 1: Search using passed-waiting lists.

2 Requirements

The goal of the assignment is to develop a header-only library for solving puzzles which can be reduced to reachability questions in transition systems. The usage of the library should be demonstrated on the puzzles above. The library implementation should support the following features:

1. Extend `std::hash` function for an arbitrary (iterable) container of basic types.
2. Create a generic successor generator function out of a transition generator function. A transition generator function generates functions that change a state. Each such function corresponds to a transition. A successor generator function gets a state and generates a set of its successor states.
3. Find a state satisfying the goal predicate when given the initial state and successor generating function.
4. Print the trace of a state sequence from the initial state to the found goal state.
5. Support various search orders: breadth-first search, depth-first search (the leaping frogs have different solutions).
6. Support a given invariant predicate (state validation function, like in river crossing puzzles).
7. Support custom cost function over states (like noise in Japanese river crossing puzzle).
8. The implementation should work on any (iterable) containers.
9. The implementation should be generic and applicable to all puzzles using the same library templates. If the search order or cost are not specified, the library should use reasonable defaults.
10. User friendly to use and fail with a message if the user supplies arguments of wrong types.

A Puzzle Modeling Code

The following code listings are just suggested examples, one may choose any other data structures, provided that the requirements are fulfilled.

Listing 1: frogs.cpp

```
1  /**
2   * Solution to a frog leap puzzle:
3   * http://arcade.modemhelp.net/play-4863.html
4   * Author: Marius Mikucionis <marius@cs.aau.dk>
5   * Compile and run:
6   * g++ -std=c++17 -pedantic -Wall -DNDEBUG -O3 -o frogs frogs.cpp && ./frogs
7   */
8  #include "reachability.hpp" // your header-only library solution
9
10 #include <iostream>
11 #include <list>
12 #include <functional> // std::function
13
14 enum class frog_t { empty, green, brown };
15 using stones_t = std::vector<frog_t>;
16
17 std::list<std::function<void(stones_t&)>> transitions(const stones_t& stones) {
18     auto res = std::list<std::function<void(stones_t&)>>{};
19     if (stones.size() < 2)
20         return res;
21     auto i = 0u;
22     while (i < stones.size() && stones[i] != frog_t::empty) ++i; // find empty stone
23     if (i == stones.size())
24         return res; // did not find empty stone
25     // explore moves to fill the empty from left to right (only green can do that):
26     if (i > 0 && stones[i-1] == frog_t::green)
27         res.push_back([i](stones_t& s){ // green jump to next
28             s[i-1] = frog_t::empty;
29             s[i] = frog_t::green;
30         });
31     if (i > 1 && stones[i-2] == frog_t::green)
32         res.push_back([i](stones_t& s){ // green jump over 1
33             s[i-2] = frog_t::empty;
34             s[i] = frog_t::green;
35         });
36     // explore moves to fill the empty from right to left (only brown can do that):
37     if (i < stones.size()-1 && stones[i+1] == frog_t::brown) {
38         res.push_back([i](stones_t& s){ // brown jump to next
39             s[i+1] = frog_t::empty;
40             s[i] = frog_t::brown;
41         });
42     }
43     if (i < stones.size()-2 && stones[i+2] == frog_t::brown) {
44         res.push_back([i](stones_t& s){ // brown jump over 1
45             s[i+2] = frog_t::empty;
46             s[i] = frog_t::brown;
47         });
48     }
49     return res;
50 }
51
52 std::ostream& operator<<(std::ostream& os, const stones_t& stones) {
53     for (auto&& stone: stones)
54         switch (stone) {
55             case frog_t::green: os << "G"; break;
56             case frog_t::empty: os << "_"; break;
57             case frog_t::brown: os << "B"; break;
```

```

58         default: os << "?"; break; // something went terribly wrong
59     }
60     return os;
61 }
62
63 std::ostream& operator<<(std::ostream& os, const std::list<const stones_t*>& trace) {
64     for (auto stones: trace)
65         os << "State of " << stones->size() << " stones: " << *stones << '\n';
66     return os;
67 }
68
69 void show_successors(const stones_t& state, const size_t level=0) {
70     // Caution: this function uses recursion, which is not suitable for solving puzzles!!
71     // 1) some state spaces can be deeper than stack allows.
72     // 2) it can only perform depth-first search
73     // 3) it cannot perform breadth-first search, cheapest-first, greatest-first etc.
74     auto trans = transitions(state); // compute the transitions
75     std::cout << std::string(level*2, ' ')
76         << "state " << state << " has " << trans.size() << " transitions";
77     if (trans.empty())
78         std::cout << '\n';
79     else
80         std::cout << ", leading to:\n";
81     for (auto& t: trans) {
82         auto succ = state; // copy the original state
83         t(succ); // apply the transition on the state to compute successor
84         show_successors(succ, level+1);
85     }
86 }
87
88 void explain(){
89     const auto start = stones_t{{ frog_t::green, frog_t::green, frog_t::empty,
90                                     frog_t::brown, frog_t::brown }};
91     std::cout << "Leaping frog puzzle start: " << start << '\n';
92     show_successors(start);
93     const auto finish = stones_t{{ frog_t::brown, frog_t::brown, frog_t::empty,
94                                     frog_t::green, frog_t::green }};
95     std::cout << "Leaping frog puzzle start: " << start << ", finish: " << finish << '\n';
96     auto space = state_space_t(start, successors<stones_t>(transitions)); // define state space
97     // explore the state space and find the solutions satisfying goal:
98     std::cout << "--- Solve with default (breadth-first) search: ---\n";
99     auto solutions = space.check([&finish](const stones_t& state){ return state==finish; });
100     for (auto& trace: solutions) { // iterate through solutions:
101         std::cout << "Solution: a trace of " << trace.size() << " states\n";
102         std::cout << trace; // print solution
103     }
104 }
105
106 void solve(size_t frogs, search_order_t order = search_order_t::breadth_first){
107     const auto stones = frogs*2+1; // frogs on either side and 1 empty in the middle
108     auto start = stones_t(stones, frog_t::empty); // initially all empty
109     auto finish = stones_t(stones, frog_t::empty); // initially all empty
110     while (frogs-->0) { // count down from frogs-1 to 0 and put frogs into positions:
111         start[frogs] = frog_t::green; // green on left
112         start[start.size()-frogs-1] = frog_t::brown; // brown on right
113         finish[frogs] = frog_t::brown; // brown on left
114         finish[finish.size()-frogs-1] = frog_t::green; // green on right
115     }
116     std::cout << "Leaping frog puzzle start: " << start << ", finish: " << finish << '\n';
117     auto space = state_space_t(std::move(start), successors<stones_t>(transitions));
118     auto solutions = space.check(
119         [finish=std::move(finish)](const stones_t& state){ return state==finish; },
120         order);

```

```

121     for (auto&& trace: solutions) {
122         std::cout << "Solution: trace of " << trace.size() << " states\n";
123         std::cout << trace;
124     }
125 }
126
127 int main(){
128     explain();
129     std::cout << "--- Solve with depth-first search: ---\n";
130     solve(2, search_order_t::depth_first);
131     solve(4); // 20 frogs may take >5.8GB of memory
132 }
133
134 /** Sample output:
135 Leaping frog puzzle start: GG_BB
136 state GG_BB has 4 transitions, leading to:
137 state G_GBB has 2 transitions, leading to:
138 state _GGBB has 0 transitions
139 state GBG_B has 2 transitions, leading to:
140 state _BGBB has 1 transitions, leading to:
141 state B_GGB has 0 transitions
142 state GBBG_ has 1 transitions, leading to:
143 state GBB_G has 0 transitions
144 state GBGB_ has 1 transitions, leading to:
145 state GB_BG has 2 transitions, leading to:
146 state _BGBG has 1 transitions, leading to:
147 state B_GBG has 1 transitions, leading to:
148 state BBG_G has 1 transitions, leading to:
149 state BB_GG has 0 transitions
150 state GBB_G has 0 transitions
151 state _GGBB has 0 transitions
152 state GGB_B has 2 transitions, leading to:
153 state G_BGB has 2 transitions, leading to:
154 state _GBGB has 1 transitions, leading to:
155 state BG_GB has 2 transitions, leading to:
156 state B_GGB has 0 transitions
157 state BGBG_ has 1 transitions, leading to:
158 state BGB_G has 1 transitions, leading to:
159 state B_BGG has 1 transitions, leading to:
160 state BB_GG has 0 transitions
161 state GB_GB has 2 transitions, leading to:
162 state _BGBB has 1 transitions, leading to:
163 state B_GGB has 0 transitions
164 state GBBG_ has 1 transitions, leading to:
165 state GBB_G has 0 transitions
166 state GGBB_ has 0 transitions
167 state GGBB_ has 0 transitions
168 Leaping frog puzzle start: GG_BB, finish: BB_GG
169 --- Solve with default (breadth-first) search: ---
170 Solution: a trace of 9 states
171 State of 5 stones: GG_BB
172 State of 5 stones: G_GBB
173 State of 5 stones: GBG_B
174 State of 5 stones: GBGB_
175 State of 5 stones: GB_BG
176 State of 5 stones: _BGBG
177 State of 5 stones: B_GBG
178 State of 5 stones: BBG_G
179 State of 5 stones: BB_GG
180 --- Solve with depth-first search: ---
181 Leaping frog puzzle start: GG_BB, finish: BB_GG
182 Solution: trace of 9 states
183 State of 5 stones: GG_BB

```

```

184 State of 5 stones: GGB_B
185 State of 5 stones: G_BGB
186 State of 5 stones: _GBGB
187 State of 5 stones: BG_GB
188 State of 5 stones: BGBG_
189 State of 5 stones: BGB_G
190 State of 5 stones: B_BGG
191 State of 5 stones: BB_GG
192 Leaping frog puzzle start: GGGG_BBBB, finish: BBBB_GGGG
193 Solution: trace of 25 states
194 State of 9 stones: GGGG_BBBB
195 State of 9 stones: GGG_GB BBB
196 State of 9 stones: GGBG_BBB
197 State of 9 stones: GGBGB_BB
198 State of 9 stones: GGBB_BGBB
199 State of 9 stones: GG_BGBGBB
200 State of 9 stones: G_GBGBGBB
201 State of 9 stones: GBG_GBGBB
202 State of 9 stones: GBGBG_GBB
203 State of 9 stones: GBGBGBG_B
204 State of 9 stones: GBGBGBGB_
205 State of 9 stones: GBGBGB_BG
206 State of 9 stones: GBGB_BGBG
207 State of 9 stones: GB_BGBGBG
208 State of 9 stones: _BGBGBGBG
209 State of 9 stones: B_GBGBGBG
210 State of 9 stones: BBG_GBGBG
211 State of 9 stones: BBGBG_GBGB
212 State of 9 stones: BBGBGBG_G
213 State of 9 stones: BBGBGB_GG
214 State of 9 stones: BBGB_BGGG
215 State of 9 stones: BB_BGBGGG
216 State of 9 stones: BBB_GBGGG
217 State of 9 stones: BBBBG_GGG
218 State of 9 stones: BBBB_GGGG
219
220 */

```

Listing 2: crossing.cpp

```

1 /**
2  * Solution to river crossing puzzle with a goat, a cabbage and a wolf.
3  * Author: Marius Mikucionis <marius@cs.aau.dk>
4  * Compile and run:
5  * g++ -std=c++17 -pedantic -Wall -DNDEBUG -O3 -o crossing crossing.cpp && ./crossing
6  */
7 #include "reachability.hpp" // your header-only library solution
8
9 #include <functional> // std::function
10 #include <list>
11 #include <array>
12 #include <iostream>
13
14 enum actor_t { cabbage, goat, wolf }; // names of the actors
15 enum class pos_t { shore1, travel, shore2 }; // names of the actor positions
16 using actors_t = std::array<pos_t,3>; // positions of the actors
17
18 auto transitions(const actors_t& actors) {
19     auto res = std::list<std::function<void(actors_t&)>>{};
20     for (auto i=0u; i<actors.size(); ++i)
21         switch(actors[i]) {
22             case pos_t::shore1:
23                 res.push_back([i](actors_t& actors){ actors[i] = pos_t::travel; });

```

```

24         break;
25     case pos_t::travel:
26         res.push_back([i](actors_t& actors){ actors[i] = pos_t::shore1; });
27         res.push_back([i](actors_t& actors){ actors[i] = pos_t::shore2; });
28         break;
29     case pos_t::shore2:
30         res.push_back([i](actors_t& actors){ actors[i] = pos_t::travel; });
31         break;
32     }
33     return res;
34 }
35
36 bool is_valid(const actors_t& actors) {
37     // only one passenger:
38     if (std::count(std::begin(actors), std::end(actors), pos_t::travel)>1)
39         return false;
40     // goat cannot be left alone with wolf, as wolf will eat the goat:
41     if (actors[actor_t::goat]==actors[actor_t::wolf] && actors[actor_t::cabbage]==pos_t::travel)
42         return false;
43     // goat cannot be left alone with cabbage, as goat will eat the cabbage:
44     if (actors[actor_t::goat]==actors[actor_t::cabbage] && actors[actor_t::wolf]==pos_t::travel)
45         return false;
46     return true;
47 }
48
49 std::ostream& operator<<(std::ostream& os, const pos_t& pos) {
50     switch(pos) {
51     case pos_t::shore1: os << "1"; break;
52     case pos_t::travel: os << "~"; break;
53     case pos_t::shore2: os << "2"; break;
54     default: os << "?"; break; // something went terribly wrong
55     }
56     return os;
57 }
58
59 std::ostream& operator<<(std::ostream& os, const actors_t& actors) {
60     return os << actors[actor_t::cabbage]
61         << actors[actor_t::goat]
62         << actors[actor_t::wolf];
63 }
64
65 std::ostream& operator<<(std::ostream& os, std::list<const actors_t*>& trace) {
66     auto step = 0u;
67     for (auto* actors: trace)
68         os << step++ << ": " << *actors << '\n';
69     return os;
70 }
71
72 void solve(){
73     auto state_space = state_space_t(
74         actors_t{}, // initial state
75         successors<actors_t>(transitions), // successor generator
76         &is_valid); // invariant over all states
77     auto solution = state_space.check(
78         [](const actors_t& actors){ // all actors should be on the shore2:
79             return std::count(std::begin(actors), std::end(actors), pos_t::shore2)==actors.size();
80         });
81     for (auto&& trace: solution)
82         std::cout << "# CGW\n" << trace;
83 }
84
85 int main(){
86     solve();

```



```

87 }
88
89 /** Sample output:
90 # CGW
91 0: 111
92 1: 1~1
93 2: 121
94 3: ~21
95 4: 221
96 5: 2~1
97 6: 211
98 7: 21~
99 8: 212
100 9: 2~2
101 10: 222
102 */

```

Listing 3: family.cpp

```

1 /**
2  * Reachability algorithm implementation for river-crossing puzzle:
3  * https://www.funzug.com/index.php/flash-games/japanese-river-crossing-puzzle-game.html
4  * Author: Marius Mikucionis <marius@cs.aau.dk>
5  * Compile using:
6  * g++ -std=c++17 -pedantic -Wall -DNDEBUG -O3 -o family family.cpp && ./family
7  * Inspect the solution (only the traveling part):
8  * ./family | grep trv | grep '~~~'
9  */
10
11 #include "reachability.hpp" // your header-only library solution
12
13 #include <iostream>
14 #include <vector>
15 #include <list>
16 #include <array>
17 #include <functional> // std::function
18 #include <algorithm> // all_of
19
20 /** Model of the river crossing: persons and a boat */
21 struct person_t {
22     enum { shore1, onboard, shore2 } pos = shore1;
23     enum { mother, father, daughter1, daughter2, son1, son2, policeman, prisoner };
24 };
25
26 struct boat_t {
27     enum { shore1, travel, shore2 } pos = shore1;
28     uint16_t capacity{2};
29     uint16_t passengers{0};
30 };
31 struct state_t {
32     boat_t boat;
33     std::array<person_t,8> persons;
34 };
35
36 /** less-than operators for std::map */
37 bool operator<(const person_t& p1, const person_t& p2) {
38     if (p1.pos < p2.pos)
39         return true;
40     else if (p2.pos < p1.pos)
41         return false; // p2 < p1
42     return false; // equal
43 }
44

```



```

45 bool operator<(const boat_t& b1, const boat_t& b2) {
46     if (b1.pos < b2.pos)
47         return true;
48     else if (b2.pos < b1.pos)
49         return false;
50     if (b1.passengers < b2.passengers)
51         return true;
52     else if (b2.passengers < b1.passengers)
53         return false;
54     if (b1.capacity < b2.capacity)
55         return true;
56     else if (b2.capacity < b1.capacity)
57         return false;
58     return false;
59 }
60
61 bool operator<(const state_t& s1, const state_t& s2) {
62     if (s1.boat < s2.boat)
63         return true;
64     if (s2.boat < s1.boat)
65         return false; // s2 < s1
66     for (auto i=0u; i<s1.persons.size(); ++i)
67         if (s1.persons[i] < s2.persons[i])
68             return true;
69         else if (s2.persons[i] < s1.persons[i])
70             return false;
71     return false; // s2 == s1
72 }
73
74 /** equality operations for std::unordered_map */
75 bool operator==(const person_t& p1, const person_t& p2) {
76     return (p1.pos == p2.pos);
77 }
78
79 bool operator==(const boat_t& b1, const boat_t& b2) {
80     return (b1.pos == b2.pos) &&
81         (b1.capacity == b2.capacity) &&
82         (b1.passengers == b2.passengers);
83 }
84
85 bool operator==(const state_t& s1, const state_t& s2) {
86     return (s1.boat == s2.boat) && (s1.persons == s2.persons);
87 }
88
89 /** hash operations for std::unordered_map */
90 namespace std {
91     template <>
92     struct hash<person_t> {
93         std::size_t operator()(const person_t& key) const {
94             return std::hash<decltype(key.pos)>{}(key.pos);
95         }
96     };
97     template <>
98     struct hash<boat_t> {
99         std::size_t operator()(const boat_t& key) const {
100             auto h_pos = std::hash<decltype(key.pos)>{};
101             auto h_int = std::hash<decltype(key.capacity)>{};
102             return (((h_pos(key.pos) << 1) ^
103                 h_int(key.capacity)) << 1) ^
104                 h_int(key.passengers));
105         }
106     };
107

```

```

108     template <>
109     struct hash<state_t> {
110         std::size_t operator()(const state_t& key) const {
111             return (std::hash<boat_t>{}(key.boat) << 1) ^
112                 std::hash<decltype(key.persons)>{}(key.persons); // assumes hash over container
113         }
114     };
115 }
116
117 std::ostream& operator<<(std::ostream& os, const person_t& p) {
118     os << '{';
119     switch (p.pos) {
120     case person_t::shore1: os << "sh1"; break;
121     case person_t::onboard: os << "~~~"; break;
122     case person_t::shore2: os << "SH2"; break;
123     default: os << "???" ; break; // something went terribly wrong
124     }
125     return os << '}';
126 }
127
128 std::ostream& operator<<(std::ostream& os, const boat_t& b) {
129     os << '{';
130     switch (b.pos) {
131     case boat_t::shore1: os << "sh1"; break;
132     case boat_t::travel: os << "trv"; break;
133     case boat_t::shore2: os << "SH2"; break;
134     default: os << "???" ; break; // something went terribly wrong
135     }
136     return os << ',' << b.passengers << ',' << b.capacity << '}';
137 }
138
139
140 std::ostream& operator<<(std::ostream& os, const state_t& s){
141     return os << s.boat << ','
142         << s.persons[person_t::mother] << ','
143         << s.persons[person_t::father] << ','
144         << s.persons[person_t::daughter1] << ','
145         << s.persons[person_t::daughter2] << ','
146         << s.persons[person_t::son1] << ','
147         << s.persons[person_t::son2] << ','
148         << s.persons[person_t::policeman] << ','
149         << s.persons[person_t::prisoner];
150 }
151
152 /**
153  * Returns a list of transitions applicable on a given state.
154  * transition is a function modifying a state
155  */
156 std::list<std::function<void(state_t&)>>
157 transitions(const state_t& s) {
158     auto res = std::list<std::function<void(state_t&)>>{};
159     switch (s.boat.pos) {
160     case boat_t::shore1:
161     case boat_t::shore2:
162         if (s.boat.passengers>0) // start traveling
163             res.push_back([](state_t& state){ state.boat.pos = boat_t::travel; });
164         break;
165     case boat_t::travel:
166         res.emplace_back([](state_t& state){ // arrive to shore1
167             state.boat.pos = boat_t::shore1;
168             state.boat.passengers = 0;
169             for (auto& p: state.persons)
170                 if (p.pos == person_t::onboard)

```

```

171         p.pos = person_t::shore1;
172     });
173     res.emplace_back([](state_t& state){ // arrive to shore2
174         state.boat.pos = boat_t::shore2;
175         state.boat.passengers = 0;
176         for (auto& p: state.persons)
177             if (p.pos == person_t::onboard)
178                 p.pos = person_t::shore2;
179     });
180     break;
181 }
182 for (auto i=0u; i<s.persons.size(); ++i) {
183     switch (s.persons[i].pos) {
184     case person_t::shore1: // board the boat on shore1:
185         if (s.boat.pos == boat_t::shore1)
186             res.push_back([](state_t& state){
187                 state.persons[i].pos = person_t::onboard;
188                 state.boat.passengers++;
189             });
190         break;
191     case person_t::shore2: // board the boat on shore2:
192         if (s.boat.pos == boat_t::shore2)
193             res.push_back([](state_t& state){
194                 state.persons[i].pos = person_t::onboard;
195                 state.boat.passengers++;
196             });
197         break;
198     case person_t::onboard:
199         if (s.boat.pos == boat_t::shore1) // leave the boat to shore1
200             res.push_back([](state_t& state){
201                 state.persons[i].pos = person_t::shore1;
202                 state.boat.passengers--;
203             });
204         else if (s.boat.pos == boat_t::shore2) // leave the boat to shore2
205             res.push_back([](state_t& state){
206                 state.persons[i].pos = person_t::shore2;
207                 state.boat.passengers--;
208             });
209         break;
210     }
211 }
212 return res;
213 }
214
215 bool river_crossing_valid(const state_t& s) {
216     if (s.boat.passengers > s.boat.capacity) {
217         log(" boat overload\n");
218         return false;
219     }
220     if (s.boat.pos == boat_t::travel) {
221         if (s.persons[person_t::daughter1].pos == person_t::onboard) {
222             if (s.boat.passengers==1 ||
223                 (s.persons[person_t::daughter2].pos == person_t::onboard) ||
224                 (s.persons[person_t::son1].pos == person_t::onboard) ||
225                 (s.persons[person_t::son2].pos == person_t::onboard) ||
226                 (s.persons[person_t::prisoner].pos == person_t::onboard)) {
227                 log(" d1 travel alone\n");
228                 return false;
229             }
230         } else if (s.persons[person_t::daughter2].pos == person_t::onboard) {
231             if (s.boat.passengers==1 ||
232                 (s.persons[person_t::daughter1].pos == person_t::onboard) ||
233                 (s.persons[person_t::son1].pos == person_t::onboard) ||

```

```

234         (s.persons[person_t::son2].pos == person_t::onboard) ||
235         (s.persons[person_t::prisoner].pos == person_t::onboard)) {
236             log(" d2 travel alone\n");
237             return false;
238         }
239     } else if (s.persons[person_t::son1].pos == person_t::onboard) {
240         if (s.boat.passengers==1 ||
241             (s.persons[person_t::daughter1].pos == person_t::onboard) ||
242             (s.persons[person_t::daughter2].pos == person_t::onboard) ||
243             (s.persons[person_t::son2].pos == person_t::onboard) ||
244             (s.persons[person_t::prisoner].pos == person_t::onboard)) {
245             log(" s1 travel alone\n");
246             return false;
247         }
248     } else if (s.persons[person_t::son2].pos == person_t::onboard) {
249         if (s.boat.passengers==1 ||
250             (s.persons[person_t::daughter1].pos == person_t::onboard) ||
251             (s.persons[person_t::daughter2].pos == person_t::onboard) ||
252             (s.persons[person_t::son1].pos == person_t::onboard) ||
253             (s.persons[person_t::prisoner].pos == person_t::onboard)) {
254             log(" s2 travel alone\n");
255             return false;
256         }
257     }
258     if (s.persons[person_t::prisoner].pos != s.persons[person_t::policeman].pos) {
259         auto prisoner_pos = s.persons[person_t::prisoner].pos;
260         if ((s.persons[person_t::daughter1].pos == prisoner_pos) ||
261             (s.persons[person_t::daughter2].pos == prisoner_pos) ||
262             (s.persons[person_t::son1].pos == prisoner_pos) ||
263             (s.persons[person_t::son2].pos == prisoner_pos) ||
264             (s.persons[person_t::mother].pos == prisoner_pos) ||
265             (s.persons[person_t::father].pos == prisoner_pos)) {
266             log(" pr with family\n");
267             return false;
268         }
269     }
270     if (s.persons[person_t::prisoner].pos == person_t::onboard && s.boat.passengers<2) {
271         log(" pr on boat\n");
272         return false;
273     }
274 }
275 if ((s.persons[person_t::daughter1].pos == s.persons[person_t::father].pos) &&
276     (s.persons[person_t::daughter1].pos != s.persons[person_t::mother].pos)) {
277     log(" d1 with f\n");
278     return false;
279 } else if ((s.persons[person_t::daughter2].pos == s.persons[person_t::father].pos) &&
280     (s.persons[person_t::daughter2].pos != s.persons[person_t::mother].pos)) {
281     log(" d2 with f\n");
282     return false;
283 } else if ((s.persons[person_t::son1].pos == s.persons[person_t::mother].pos) &&
284     (s.persons[person_t::son1].pos != s.persons[person_t::father].pos)) {
285     log(" s1 with m\n");
286     return false;
287 } else if ((s.persons[person_t::son2].pos == s.persons[person_t::mother].pos) &&
288     (s.persons[person_t::son2].pos != s.persons[person_t::father].pos)) {
289     log(" s2 with m\n");
290     return false;
291 }
292 log(" OK\n");
293 return true;
294 }
295
296 struct cost_t {

```

```

297     size_t depth{0}; // counts the number of transitions
298     size_t noise{0}; // kids get bored on shore1 and start making noise there
299     bool operator<(const cost_t& other) const {
300         if (depth < other.depth)
301             return true;
302         if (other.depth < depth)
303             return false;
304         return noise < other.noise;
305     }
306 };
307
308 bool goal(const state_t& s){
309     return std::all_of(std::begin(s.persons), std::end(s.persons),
310         [](const person_t& p) { return p.pos == person_t::shore2; });
311 }
312
313
314 template <typename CostFn>
315 void solve(CostFn&& cost) { // no type checking: OK hack here, but not good for a library.
316     // Overall there are 4*3*2*1/2 solutions to the puzzle
317     // (children form 2 symmetric groups and thus result in 2 out of 4 permutations).
318     // However the search algorithm may collapse symmetric solutions, thus only one is reported.
319     // By changing the cost function we can express a preference and
320     // then the algorithm should report different solutions
321     auto states = state_space_t{
322         state_t{}, cost_t{}, // initial state and cost
323         successors<state_t>(transitions), // successor generator
324         &river_crossing_valid, // invariant over states
325         std::forward<CostFn>(cost)); // cost over states
326     auto solutions = states.check(&goal);
327     if (solutions.empty()) {
328         std::cout << "No solution\n";
329     } else {
330         for (auto&& trace: solutions) {
331             std::cout << "Solution:\n";
332             std::cout << "Boat,      Mothr,Fathr,Daug1,Daug2,Son1, Son2, Polic,Prisn\n";
333             for (auto&& state: trace)
334                 std::cout << *state << '\n';
335         }
336     }
337 }
338
339 int main() {
340     std::cout << "-- Solve using depth as a cost: ---\n";
341     solve([](const state_t& state, const cost_t& prev_cost){
342         return cost_t{ prev_cost.depth+1, prev_cost.noise };
343     }); // it is likely that daughters will get to shore2 first
344     std::cout << "-- Solve using noise as a cost: ---\n";
345     solve([](const state_t& state, const cost_t& prev_cost){
346         auto noise = prev_cost.noise;
347         if (state.persons[person_t::son1].pos == person_t::shore1)
348             noise += 2; // older son is more naughty, prefer him first
349         if (state.persons[person_t::son2].pos == person_t::shore1)
350             noise += 1;
351         return cost_t{ prev_cost.depth, noise };
352     }); // son1 should get to shore2 first
353     std::cout << "-- Solve using different noise as a cost: ---\n";
354     solve([](const state_t& state, const cost_t& prev_cost){
355         auto noise = prev_cost.noise;
356         if (state.persons[person_t::son1].pos == person_t::shore1)
357             noise += 1;
358         if (state.persons[person_t::son2].pos == person_t::shore1)
359             noise += 2; // younger son is more distressed, prefer him first

```

```

360         return cost_t{ prev_cost.depth, noise };
361     }); // son2 should get to the shore2 first
362 }
363 /** Example solutions (shows only the states with travel):
364 --- Solve using depth as a cost: ---
365 Boat,    Mothr,Fathr,Daug1,Daug2,Son1, Son2, Polic,Prisn
366 {trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{~~~}
367 {trv,1,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{SH2}
368 {trv,2,2},{sh1},{sh1},{~~~},{sh1},{sh1},{sh1},{~~~},{SH2}
369 {trv,2,2},{sh1},{sh1},{SH2},{sh1},{sh1},{sh1},{~~~},{~~~}
370 {trv,2,2},{~~~},{sh1},{SH2},{~~~},{sh1},{sh1},{sh1},{sh1}
371 {trv,1,2},{~~~},{sh1},{SH2},{SH2},{sh1},{sh1},{sh1},{sh1}
372 {trv,2,2},{~~~},{~~~},{SH2},{SH2},{sh1},{sh1},{sh1},{sh1}
373 {trv,1,2},{SH2},{~~~},{SH2},{SH2},{sh1},{sh1},{sh1},{sh1}
374 {trv,2,2},{SH2},{sh1},{SH2},{SH2},{sh1},{sh1},{~~~},{~~~}
375 {trv,1,2},{~~~},{sh1},{SH2},{SH2},{sh1},{sh1},{SH2},{SH2}
376 {trv,2,2},{~~~},{~~~},{SH2},{SH2},{sh1},{sh1},{SH2},{SH2}
377 {trv,1,2},{SH2},{~~~},{SH2},{SH2},{sh1},{sh1},{SH2},{SH2}
378 {trv,2,2},{SH2},{~~~},{SH2},{SH2},{sh1},{sh1},{SH2},{SH2}
379 {trv,2,2},{SH2},{SH2},{SH2},{SH2},{SH2},{sh1},{~~~},{~~~}
380 {trv,2,2},{SH2},{SH2},{SH2},{SH2},{SH2},{sh1},{~~~},{sh1}
381 {trv,1,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~~},{sh1}
382 {trv,2,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~~},{~~~}
383 --- Solve using noise as a cost: ---
384 Boat,    Mothr,Fathr,Daug1,Daug2,Son1, Son2, Polic,Prisn
385 {trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{~~~}
386 {trv,1,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{SH2}
387 {trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{SH2}
388 {trv,2,2},{sh1},{sh1},{sh1},{sh1},{SH2},{sh1},{~~~},{~~~}
389 {trv,2,2},{sh1},{~~~},{sh1},{sh1},{SH2},{sh1},{sh1},{sh1}
390 {trv,1,2},{sh1},{~~~},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
391 {trv,2,2},{~~~},{~~~},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
392 {trv,1,2},{~~~},{SH2},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
393 {trv,2,2},{sh1},{SH2},{sh1},{sh1},{SH2},{SH2},{~~~},{~~~}
394 {trv,1,2},{sh1},{~~~},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
395 {trv,2,2},{~~~},{~~~},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
396 {trv,1,2},{~~~},{SH2},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
397 {trv,2,2},{~~~},{SH2},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
398 {trv,2,2},{SH2},{SH2},{SH2},{sh1},{SH2},{SH2},{~~~},{~~~}
399 {trv,2,2},{SH2},{SH2},{SH2},{sh1},{SH2},{SH2},{sh1},{sh1}
400 {trv,1,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{sh1},{sh1}
401 {trv,2,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{sh1},{sh1}
402 -- Solve using different noise as a cost: ---
403 Boat,    Mothr,Fathr,Daug1,Daug2,Son1, Son2, Polic,Prisn
404 {trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{~~~}
405 {trv,1,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{SH2}
406 {trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{SH2}
407 {trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{SH2},{sh1},{sh1}
408 {trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{SH2},{sh1},{sh1}
409 {trv,1,2},{sh1},{sh1},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
410 {trv,2,2},{sh1},{sh1},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
411 {trv,1,2},{sh1},{SH2},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
412 {trv,2,2},{sh1},{SH2},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
413 {trv,1,2},{sh1},{sh1},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
414 {trv,2,2},{sh1},{sh1},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
415 {trv,1,2},{sh1},{SH2},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
416 {trv,2,2},{sh1},{SH2},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
417 {trv,2,2},{SH2},{SH2},{SH2},{sh1},{SH2},{SH2},{sh1},{sh1}
418 {trv,2,2},{SH2},{SH2},{SH2},{sh1},{SH2},{SH2},{sh1},{sh1}
419 {trv,1,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{sh1},{sh1}
420 {trv,2,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{sh1},{sh1}
421 */

```

Listing 4: CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.10)
2 project(PuzzleEngine CXX)
3
4 set(CMAKE_CXX_STANDARD 17)
5 set(CMAKE_CXX_STANDARD_REQUIRED ON)
6 set(CMAKE_CXX_EXTENSIONS OFF)
7
8 set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -fsanitize=undefined -fsanitize=address")
9 set(CMAKE_LINK_FLAGS_DEBUG "${CMAKE_LINK_FLAGS_DEBUG} -fsanitize=undefined -fsanitize=address")
10
11 add_executable(frogs frogs.cpp)
12 add_executable(crossing crossing.cpp)
13 add_executable(family family.cpp)
```