

Puzzle Engine Library AP Exam

Christopher Hansen Nielsen

June 10, 2019

Listing 1: 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)
```

Listing 2: reachability.hpp

```
1 /**
2  * The reachability header file library.
3  * Author: Christopher Hansen Nielsen
4  * Mail: chni15@student.aau.dk, Student Number: 20154154
5  *
6  * Written and compiled on a windows machine.
7  * g++ -std=c++17 -pedantic -Wall -DNDEBUG -O3 -o frogs frogs.cpp
8  * g++ -std=c++17 -pedantic -Wall -DNDEBUG -O3 -o crossing crossing.cpp
9  * g++ -std=c++17 -pedantic -Wall -DNDEBUG -O3 -o family family.cpp
10 */
11
12 #ifndef PUZZLEENGINE_REACHABILITY_HPP
13 #define PUZZLEENGINE_REACHABILITY_HPP
14
15 #include <vector>
16 #include <list>
17 #include <functional>
18 #include <iostream>
19 #include <algorithm>
20 #include <typeinfo>
21
22 // This enum is used to handle the support for different search orders except for cost order. It ↗
23 // is implemented
24 // as part of requirement 5.
25 enum search_order_t {
26     breadth_first, depth_first
27 };
28
29 // This struct is the basis for keeping track of the solution when traversing the states. When ↗
30 // it is used, it holds a
31 // pointer to the parent node as well as a copy of the state.
32 template<class StateTypeT>
33 struct trace_node {
```

```

32     trace_node *parentState;
33     StateTypeT selfState;
34 };
35
36 // This function is used to pass on the transition generator function from the respective
37 // puzzles. It is implemented
38 // as part of requirement 2.
39 template<class StateTypeT>
40 std::function<std::list<std::function<void(StateTypeT &)>>(StateTypeT &)>
41 successors(std::list<std::function<void(StateTypeT &)>> transitions(const StateTypeT &state)) {
42     return transitions;
43 }
44
45 // This class holds all the information about a given state space. It utilizes two template
46 // types StateTypeT and
47 // CostTypeT. These are the basis of the generic implementation as part of requirements 8 and 9.
48 template<class StateTypeT, class CostTypeT = std::nullptr_t>
49 class state_space_t {
50 private:
51     StateTypeT _startState;
52     CostTypeT _initialCost;
53     std::function<std::list<std::function<void(StateTypeT &)>>(StateTypeT &)> _transitionFunctions;
54     std::function<bool(const StateTypeT &)> _invariantFunction;
55     std::function<CostTypeT(const StateTypeT &state, const CostTypeT &cost)> _costFunction;
56     bool _isCostEnabled; // used explicitly to determine whether or not a cost have been specified.
57
58     template<class ValidationFunction>
59     std::list<StateTypeT> solveOrder(ValidationFunction isGoalState, search_order_t order);
60
61     template<class ValidationFunction>
62     std::list<StateTypeT> solveCost(ValidationFunction isGoalState);
63
64 public:
65     // This is the first constructor for the class, which handles calls from the frogs.cpp
66     // and crossing.cpp instantiation.
67     state_space_t(const StateTypeT startInputState,
68                  std::function<std::list<std::function<void(StateTypeT &)>>(StateTypeT &)>
69                  transFunctions,
70                  bool invariantFunc(const StateTypeT &) = [] (
71                      const StateTypeT &state) { return true; },
72                  const StateTypeT &startInputState,
73                  std::function<std::list<std::function<void(StateTypeT &)>>(StateTypeT &)>
74                  _transitionFunctions(
75                      transFunctions),
76                  std::function<bool(const StateTypeT &)> _invariantFunction(invariantFunc),
77                  bool _isCostEnabled(false)){
78     }
79
80     // This the second and overloaded constructor for the class. This handles calls from the
81     // family.cpp.
82     state_space_t(const StateTypeT startInputState, const CostTypeT costInput,
83                  std::function<std::list<std::function<void(StateTypeT &)>>(StateTypeT &)>
84                  transFunctions,
85                  bool invariantFunc(const StateTypeT &) = [] (
86                      const StateTypeT &state) { return true; },
87                  std::function<CostTypeT(const StateTypeT &state, const CostTypeT &cost)>
88                  costFunc = [] (
89                      const StateTypeT &state, const CostTypeT &cost)
90                      { return CostTypeT{0, 0}; }) : _startState(
91                      startInputState),
92                  _initialCost(
93                      costInput),
94                  _transitionFunctions(

```

```

87         transFunctions),
88         _invariantFunction(
89             invariantFunc),
90         _costFunction(
91             costFunc),
92         _isCostEnabled(
93             true) {
94     }
95
96     template<class ValidationFunction>
97     std::list<StateTypeT> check(ValidationFunction isGoalState, search_order_t order = ↵
↵search_order_t::breadth_first);
98 };
99
100 // This function is called from the different puzzle files and returns a solution if found. It ↵
↵introduces a new template
101 // ValidationFunction that handles the goal predicate function. It also takes an order, which is ↵
↵defaulted to
102 // breadth first if nothing else is specified.
103 // It returns a list of states.
104 template<class StateTypeT, class CostTypeT>
105 template<class ValidationFunction>
106 std::list<StateTypeT> state_space_t<StateTypeT, CostTypeT>::check(ValidationFunction ↵
↵isGoalState, search_order_t order) {
107     std::list<StateTypeT> solution;
108
109     if (_isCostEnabled) { // Here we check if the cost method is specified, and calls the ↵
↵solveCost if true.
110         solution = solveCost(isGoalState);
111     } else { // Otherwise we call the solveOrder method with the order provided.
112         solution = solveOrder(isGoalState, order);
113     }
114
115     // Returns the list of states. Implemented as part of requirement 4.
116     return solution;
117 }
118
119 // The method is used when solving the state space based on a given cost. It takes in ↵
↵isGoalState which is a predicate
120 // that is used to determine whether a solution have been found.
121 // It returns a list of states. It is implemented as part of requirement 7.
122 template<class StateTypeT, class CostTypeT>
123 template<class ValidationFunction>
124 std::list<StateTypeT> state_space_t<StateTypeT, CostTypeT>::solveCost(ValidationFunction ↵
↵isGoalState) {
125     StateTypeT currentState;
126     CostTypeT itCost{_initialCost}, newCost;
127     trace_node<StateTypeT> *traceState {};
128     std::list<StateTypeT> passed, solution;
129     std::list<std::pair<CostTypeT, trace_node<StateTypeT> *>> waiting;
130
131     // The method utilizes a waiting list that contains a pair of cost and a trace_node. The ↵
↵cost is used to determine
132     // which element should be popped from the list next.
133     waiting.push_back(std::make_pair(itCost, new trace_node<StateTypeT>{nullptr, _startState}));
134
135     while (!waiting.empty()) {
136         // Here we take the first element from the list and then pop it.
137         currentState = waiting.front().second->selfState;
138         traceState = waiting.front().second;
139         itCost = waiting.front().first;

```

```

140     waiting.pop_front();
141
142     // Here we check if the goal state has been reached. This is implemented as part of ↵
143     ↪requirement 3.
144     if (isGoalState(currentState)) {
145         // If a goal is found, we use the traceState to traverse back up through the tree of ↵
146         ↪trace_nodes
147         // until the parentState is NULL. For each node we push the state to the solution list.
148         while (traceState->parentState != NULL) {
149             solution.push_front(traceState->selfState);
150             traceState = traceState->parentState;
151         }
152         solution.push_front(traceState->selfState); // Adds the start state to the solution ↵
153         ↪trace.
154         return solution;
155     }
156
157     // Here we check if the currentState is part of the passed list.
158     if (!std::find(passed.begin(), passed.end(), currentState) != passed.end()) {
159         // If it is not, we push it to the list, and generate the transitions via the ↵
160         ↪_transitionFunctions which
161         // is a member of the state_space_t class.
162         passed.push_back(currentState);
163         auto transitions = _transitionFunctions(currentState);
164
165         for (auto transition: transitions) {
166             // For each transition, we then generate the successor
167             auto successor{currentState};
168             transition(successor);
169
170             // Prevents invalid states being added to waiting via an invariant predicate. ↵
171             ↪This is implemented as
172             // part of requirement 6.
173             if (!_invariantFunction(successor)) {
174                 continue;
175             }
176             newCost = _costFunction(successor, itCost);
177             waiting.push_back(std::make_pair(newCost, new trace_node<StateTypeT>{traceState, ↵
178             ↪successor}));
179         }
180         if (!transitions.empty()) { // Prevents sorting if no new transitions have been found.
181             // Here we sort the waiting list based on the cost for each pair in the list.
182             waiting.sort([](const std::pair<CostTypeT, trace_node<StateTypeT> *> &a,
183             ↪std::pair<CostTypeT, trace_node<StateTypeT> *> &b) { return ↵
184             ↪a.first < b.first; });
185         }
186     }
187
188     return solution;
189 }
190
191 // The method is used when solving the state space based on a given order. It takes in ↵
192 ↪isGoalState which is a predicate
193 // that is used to determine whether a solution have been found. It also takes an order, which ↵
194 ↪specifies how the
195 // solution should be found. The method is very similar to solveCost in functionality.
196 // It returns a list of states.
197 template<class StateTypeT, class CostTypeT>
198 template<class ValidationFunction>

```

```

192 std::list<StateTypeT>
193 state_space_t<StateTypeT, CostTypeT>::solveOrder(ValidationFunction isGoalState, search_order_t ↵
↪order) {
194     StateTypeT currentState;
195     trace_node<StateTypeT> *traceState {};
196     std::list<StateTypeT> passed, solution;
197     std::list<trace_node<StateTypeT> *> waiting;
198
199     // As solveOrder does not utilize a cost, waiting is just a list of trace_nodes.
200     waiting.push_back(new trace_node<StateTypeT>{nullptr, _startState});
201
202     while (!waiting.empty()) {
203         switch (order) { // We switch on the order to determine what element should be accessed ↵
↪and popped from waiting.
204             case breadth_first:
205                 currentState = waiting.front()->selfState;
206                 traceState = waiting.front();
207                 waiting.pop_front();
208                 break;
209             case depth_first:
210                 currentState = waiting.back()->selfState;
211                 traceState = waiting.back();
212                 waiting.pop_back();
213                 break;
214             default:
215                 std::cout << "Order not supported" << std::endl;
216                 break;
217         }
218         if (isGoalState(currentState)) {
219             while (traceState->parentState != NULL) {
220                 solution.push_front(traceState->selfState);
221                 traceState = traceState->parentState;
222             }
223
224             solution.push_front(traceState->selfState); // Adds the start state to the solution ↵
↪trace.
225             return solution;
226         }
227         if (!(std::find(passed.begin(), passed.end(), currentState) != passed.end())) {
228             passed.push_back(currentState);
229             auto transitions = _transitionFunctions(currentState);
230
231             for (auto transition: transitions) {
232                 auto successor{currentState};
233                 transition(successor);
234
235                 if (!_invariantFunction(successor)) { // Prevents invalid states being added to ↵
↪waiting.
236                     continue;
237                 }
238                 waiting.push_back(new trace_node<StateTypeT>{traceState, successor});
239             }
240         }
241     }
242
243     return solution;
244 }
245
246 // The following hash override is implemented in order to compile family.cpp. It is, however, ↵
↪never used and should
247 // not be used at all. Gives a warning when compiling the program.

```

```

248 template<class StateType, size_t typeSize>
249 struct std::hash<std::array<StateType, typeSize>> {
250     std::size_t operator()(const array<StateType, typeSize> &key) const {
251         return NULL;
252     }
253 };
254
255 #endif //PUZZLEENGINE_REACHABILITY_HPP

```

Listing 3: 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     // Added type specification to the template.
97     auto space = state_space_t<stones_t>(start, successors<stones_t>(transitions)); // define state space
98     // explore the state space and find the solutions satisfying goal:
99     std::cout << "--- Solve with default (breadth-first) search: ---\n";
100    auto solutions = space.check([&finish](const stones_t& state){ return state==finish; });
101    for (auto& trace: solutions) { // iterate through solutions:
102        std::cout << "Solution: a trace of " << trace.size() << " states\n";
103        std::cout << trace; // print solution
104    }
105 }
106
107 void solve(size_t frogs, search_order_t order = search_order_t::breadth_first){
108     const auto stones = frogs*2+1; // frogs on either side and 1 empty in the middle
109     auto start = stones_t(stones, frog_t::empty); // initially all empty
110     auto finish = stones_t(stones, frog_t::empty); // initially all empty
111     while (frogs-->0) { // count down from frogs-1 to 0 and put frogs into positions:

```

```

112     start[frogs] = frog_t::green;           // green on left
113     start[start.size()-frogs-1] = frog_t::brown; // brown on right
114     finish[frogs] = frog_t::brown;         // brown on left
115     finish[finish.size()-frogs-1] = frog_t::green; // green on right
116 }
117 std::cout << "Leaping frog puzzle start: " << start << ", finish: " << finish << '\n';
118 // Added type specification to the template.
119 auto space = state_space_t<stones_t>(std::move(start), successors<stones_t>(transitions));
120 auto solutions = space.check(
121     [finish=std::move(finish)](const stones_t& state){ return state==finish; },
122     order);
123 // Introduced a change in print to adhere to my solution
124 std::cout << "Solution: trace of " << solutions.size() << " states\n";
125 for (auto&& trace: solutions) {
126     std::cout << trace << std::endl;
127 }
128 }
129
130 int main(){
131     //explain();
132     std::cout << "--- Solve with depth-first search: ---\n";
133     solve(2, search_order_t::depth_first);
134     std::cout << "--- Solve with breadth-first search: ---\n";
135     solve(2); // 20 frogs may take >5.8GB of memory
136 }
137
138 /** Sample output:
139 Leaping frog puzzle start: GG_BB
140 state GG_BB has 4 transitions, leading to:
141     state G_GBB has 2 transitions, leading to:
142         state _GGBB has 0 transitions
143         state GBG_B has 2 transitions, leading to:
144             state GB_GB has 2 transitions, leading to:
145                 state _BGGB has 1 transitions, leading to:
146                     state B_GGB has 0 transitions
147                     state GBBG_ has 1 transitions, leading to:
148                         state GBB_G has 0 transitions
149                     state GBGB_ has 1 transitions, leading to:
150                         state GB_BG has 2 transitions, leading to:
151                             state _GBBG has 1 transitions, leading to:
152                                 state B_GBG has 1 transitions, leading to:
153                                     state BBG_G has 1 transitions, leading to:
154                                         state BB_GG has 0 transitions
155                                         state GBB_G has 0 transitions
156                                     state _GGBB has 0 transitions
157                                 state GGBB_ has 2 transitions, leading to:
158                                     state G_BGB has 2 transitions, leading to:
159                                         state _GBGB has 1 transitions, leading to:
160                                             state BG_GB has 2 transitions, leading to:
161                                                 state B_GGB has 0 transitions
162                                                 state BGBG_ has 1 transitions, leading to:
163                                                     state BGB_G has 1 transitions, leading to:
164                                                         state B_BGG has 1 transitions, leading to:
165                                                             state BB_GG has 0 transitions
166                                                         state GB_GB has 2 transitions, leading to:
167                                                             state _BGGB has 1 transitions, leading to:
168                                                                 state B_GGB has 0 transitions
169                                                                 state GBBG_ has 1 transitions, leading to:
170                                                                     state GBB_G has 0 transitions
171                                                                     state GGBB_ has 0 transitions
172 Leaping frog puzzle start: GG_BB, finish: BB_GG

```



```

173  --- Solve with default (breadth-first) search: ---
174  Solution: a trace of 9 states
175  State of 5 stones: GG_BB
176  State of 5 stones: G_GBB
177  State of 5 stones: GBG_B
178  State of 5 stones: GBGB_
179  State of 5 stones: GB_BG
180  State of 5 stones: _GBBG
181  State of 5 stones: B_GBG
182  State of 5 stones: BBG_G
183  State of 5 stones: BB_GG
184  --- Solve with depth-first search: ---
185  Leaping frog puzzle start: GG_BB, finish: BB_GG
186  Solution: trace of 9 states
187  State of 5 stones: GG_BB
188  State of 5 stones: GGB_B
189  State of 5 stones: G_BGB
190  State of 5 stones: _GBGB
191  State of 5 stones: BG_GB
192  State of 5 stones: BGBG_
193  State of 5 stones: BGB_G
194  State of 5 stones: B_BGG
195  State of 5 stones: BB_GG
196  Leaping frog puzzle start: GGGG_BBBB, finish: BBBB_GGGG
197  Solution: trace of 25 states
198  State of 9 stones: GGGG_BBBB
199  State of 9 stones: GGG_GBBBB
200  State of 9 stones: GGGBG_BBB
201  State of 9 stones: GGGBGB_BB
202  State of 9 stones: GGGB_BGBB
203  State of 9 stones: GG_BGBGBB
204  State of 9 stones: G_GBGBGBB
205  State of 9 stones: GBG_GBGBB
206  State of 9 stones: GBGBG_GBB
207  State of 9 stones: GBGBGBG_B
208  State of 9 stones: GBGBGBGB_
209  State of 9 stones: GBGBGB_BG
210  State of 9 stones: GBGB_BGBG
211  State of 9 stones: GB_BGBGBG
212  State of 9 stones: _BGBGBGBG
213  State of 9 stones: B_GBGBGBG
214  State of 9 stones: BBG_GBGBG
215  State of 9 stones: BBGBG_GBG
216  State of 9 stones: BBGBGBG_G
217  State of 9 stones: BBGBGB_GG
218  State of 9 stones: BBGB_BGGG
219  State of 9 stones: BB_BGBGGG
220  State of 9 stones: BBB_GBGGG
221  State of 9 stones: BBBBG_GGG
222  State of 9 stones: BBBB_GGGG
223
224  */

```

Listing 4: 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<actors_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     // Introduced a change in print to adhere to my solution
82     std::cout << "# CGW" << std::endl;
83     int it = 0;
84     for (auto&& trace: solution)
85         std::cout << it << ": " << trace << std::endl;
86 }
87
88 int main(){
89     solve();
90 }
91
92 /** Sample output:
93 # CGW
94 0: 111
95 1: 1~1
96 2: 121
97 3: ~21
98 4: 221
99 5: 2~1
100 6: 211
101 7: 21~
102 8: 212
103 9: 2~2
104 10: 222
105 */

```

Listing 5: 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([i](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([i](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([i](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([i](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     }
259     if (s.persons[person_t::prisoner].pos != s.persons[person_t::policeman].pos) {
260         auto prisoner_pos = s.persons[person_t::prisoner].pos;
261         if ((s.persons[person_t::daughter1].pos == prisoner_pos) ||
262             (s.persons[person_t::daughter2].pos == prisoner_pos) ||
263             (s.persons[person_t::son1].pos == prisoner_pos) ||
264             (s.persons[person_t::son2].pos == prisoner_pos) ||
265             (s.persons[person_t::mother].pos == prisoner_pos) ||
266             (s.persons[person_t::father].pos == prisoner_pos)) {
267             // 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<state_t, cost_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()) {

```



```

389 {trv,2,2},{sh1},{~~~},{sh1},{sh1},{SH2},{~~~},{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},{SH2},{SH2},{SH2},{SH2}
398 {trv,2,2},{SH2},{SH2},{SH2},{sh1},{SH2},{SH2},{~~~},{~~~}
399 {trv,2,2},{SH2},{SH2},{SH2},{~~~},{SH2},{SH2},{~~~},{sh1}
400 {trv,1,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~~},{sh1}
401 {trv,2,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~~},{~~~}
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},{~~~},{SH2}
406 {trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{~~~},{SH2}
407 {trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{SH2},{~~~},{~~~}
408 {trv,2,2},{sh1},{~~~},{sh1},{sh1},{~~~},{SH2},{sh1},{sh1}
409 {trv,1,2},{sh1},{~~~},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
410 {trv,2,2},{~~~},{~~~},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
411 {trv,1,2},{~~~},{SH2},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
412 {trv,2,2},{sh1},{SH2},{sh1},{sh1},{SH2},{SH2},{~~~},{~~~}
413 {trv,1,2},{sh1},{~~~},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
414 {trv,2,2},{~~~},{~~~},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
415 {trv,1,2},{~~~},{SH2},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
416 {trv,2,2},{~~~},{SH2},{~~~},{sh1},{SH2},{SH2},{SH2},{SH2}
417 {trv,2,2},{SH2},{SH2},{SH2},{sh1},{SH2},{SH2},{~~~},{~~~}
418 {trv,2,2},{SH2},{SH2},{SH2},{~~~},{SH2},{SH2},{~~~},{sh1}
419 {trv,1,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~~},{sh1}
420 {trv,2,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~~},{~~~}
421 */

```