

AP Puzzle Mini Project

Christian Møllnitz

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 set(CMAKE_C_COMPILER "gcc")
8 set(CMAKE_CXX_COMPILER "g++")
9
10 set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -fsanitize=undefined -fsanitize=address")
11 set(CMAKE_LINK_FLAGS_DEBUG "${CMAKE_LINK_FLAGS_DEBUG} -fsanitize=undefined -fsanitize=address")
12
13 add_executable(frogs frogs.cpp)
14 add_executable(crossing crossing.cpp)
15 add_executable(family family.cpp)
```

Listing 2: reachability.hpp

```
1 #ifndef REACHABILITY
2
3 #define REACHABILITY
4
5 #define REACHABILITY_DEBUG false
6
7 #include <ostream>
8 #include <list>
9 #include <functional>
10 #include <iostream>
11 #include <string>
12 #include <utility>
13 #include <memory> // This is memory_resource in windows.
14 #include <iterator>
15 #include <initializer_list>
16 #include <set>
17 #include <queue>
18 #include <type_traits>
19 #include <algorithm>
20
21 using namespace std;
22
23 //Function was missing in code provided, so I made my own. - It's kinda lame because it only ↗
↪takes a string.
24 /*Paints out a string, ends the line*/
25 void log( string input) {
26 #if REACHABILITY_DEBUG
27     cout << input << endl;
28 #endif // REACHABILITY_DEBUG
29 };
30
```

```

31
32  /* Part of: 5. Support various search orders: breadth-first search, depth-first search (the ↵
↵leaping frogs have different solutions). */
33  /*Details the order, depth_first or breadth_first, notice that this library uses breadth_first ↵
↵as its default order */
34  enum class search_order_t
35  {
36      depth_first,
37      breadth_first
38  };
39
40  /*
41  Performs a search with a priority queue.
42  state: The initial state.
43  cost: The initial cost.
44  cost_generator: A function that generates the next cost, using the previous cost, and previous ↵
↵state.
45  successor_generator: A function that generates all possible successor states to the current state.
46  valid_state_func: A function that checks if a state is a valid state.
47  accept_state_func A function that checks if a state is an acceptable state.
48  comparison_func: the comparison function used by the priority queue itself, bfs or other costs = ↵
↵std::greater && dfs = std::less
49  */
50  template <class State, class Cost>
51  list<list<shared_ptr<State>>> priority_queue_search(State state, Cost cost, ↵
↵function<vector<State>(State&)> successor_generator, function<Cost(State&, Cost&)> ↵
↵cost_generator, function<bool(const State&)> valid_state_func, function<bool(const State&)> ↵
↵accept_state_func, function<bool(const pair<Cost, State>&, const pair<Cost, State>&)> ↵
↵comparison_func)
52  {
53      using pair_type = pair<Cost, State>;
54      using comp_type = function<bool(const pair_type&, const pair_type&)>;
55
56      // a set that holds all seen states up to this point.
57      set<State> seen_set{};
58
59      auto result = list<list<shared_ptr<State>>>{};
60      unordered_map<State, State> child_to_parent_map = unordered_map<State, State>{};
61
62      //Priority maps are sorted by the first element in a pair structure, that is why it's ↵
↵Pair<cost,state>.
63      auto pqs_queue = priority_queue<pair_type, vector<pair_type>, comp_type> {comparison_func};
64      //Eemplace smartly picks constructor.
65      pqs_queue.emplace(cost, state);
66
67      //Infinite loop protection
68      seen_set.insert(state);
69
70      while (!pqs_queue.empty())
71      {
72          pair_type parent_item = pqs_queue.top();
73          pqs_queue.pop();
74
75          if (accept_state_func(parent_item.second))
76          {
77              result.push_back(map_ancestry_shared(child_to_parent_map, parent_item.second));
78              log("Added a result");
79          }
80          else if (valid_state_func(parent_item.second)) {
81              auto successors = successor_generator(parent_item.second);
82              for (auto&& child_item : successors)

```

```

83         {
84             //Keeps track of seen items, avoid infinite loops
85             if (seen_set.count(child_item) == 0) //Set is faster than vector (lg(n))
86             {
87                 seen_set.insert(child_item);
88                 child_to_parent_map[child_item] = parent_item.second;
89                 pqs_queue.push(make_pair(cost_generator(parent_item.second,
83 →parent_item.first), child_item));
90             }
91         }
92     }
93 }
94 return result; //RVO - ITEM 25
95 }
96
97 /*Gets a route from the acceptable state child (or any child really) though the chain of
97 →parents, to the root node*/
98 template <class State>
99 list<shared_ptr<State>> map_ancestry_shared(unordered_map<State, State> c_to_p_map, State child) {
100     auto ret = list<shared_ptr<State>>{};
101     ret.push_back(make_shared<State>(child));
102     // https://en.cppreference.com/w/cpp/container/unordered_map/find - if it doesn't find the
102 →element, it returns end iterator.
103     auto c_to_p_pair = c_to_p_map.find(child);
104     while (c_to_p_pair != c_to_p_map.end())
105     {
106         ret.push_back(make_shared<State>(c_to_p_pair->second));
107         c_to_p_pair = c_to_p_map.find(c_to_p_pair->second);
108     }
109     return ret;
110 }
111
112
113
114
115 //9. The implementation should be generic and applicable to all puzzles using the same library
115 →templates. If the search order or cost are not specified, the library should use reasonable
115 →defaults.
116 /*Mimics 'cost_t' struct from family.cpp, and lets this facimily make BFS and DFS search for any
116 →type*/
117
118 /*Default cost structure, simply annotates costs in the conventional BFS / DFS manner, as nodes
118 →are discovered, chronologically*/
119 struct default_cost_t {
120     int id;
121     static int total;
122     default_cost_t()
123     {
124         id = total++;
125     }
126
127     bool operator<(const default_cost_t& other) const {
128         return id < other.id;
129     }
130 };
131
132 //9. The implementation should be generic and applicable to all puzzles using the same library
132 →templates. If the search order or cost are not specified, the library should use reasonable
132 →defaults.
133 /*Cheap cost_gen, that merely annotates costs in the conventional BFS and DFS manner,
133 →chronologically*/

```

```

134 template <class State, class default_cost_t>
135 default_cost_t cost_gen(const State& prev_state, const default_cost_t& prev_cost) {
136     return default_cost_t{}; //Use default constructor.
137 }
138
139 int default_cost_t::total = 0;
140
141
142 //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.
143 /*Cheap way to ensure all states are valid*/
144 template <class State>
145 bool return_true(const State& state) {
146     return true;
147 }
148
149
150 template <class State, class Cost = default_cost_t, class CostFn = function<Cost(const State&, ↗
    ↗const Cost&>>>
151 class state_space_t
152 {
153 public:
154
155     state_space_t() = delete; //No default constructor - library should not be used like this.
156     ~state_space_t() = default; //Default deleter - rule of zero.
157
158
159     /* 3. Find a state satisfying the goal predicate when given the initial state and successor ↗
    ↗generating function. */
160     state_space_t(State state, function<vector<State>(State&)> successor_generator) : ↗
    ↗state_space_t(state, successor_generator, return_true<State>) { } ;
161
162     /* 6. Support a given invariant predicate (state validation function, like in river crossing ↗
    ↗puzzles). */
163     /* 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. */
164     state_space_t(State state, function<vector<State>(State&)> successor_generator, bool ↗
    ↗(*val_gen) (const State&)) : state_space_t(state, Cost{}, successor_generator, val_gen, ↗
    ↗cost_gen<State, Cost>) {
165         default_cost_t::total = 0; // reset node index inbetween runs. - only needed for this ↗
    ↗constructor, and the one above.
166     };
167
168     /* 7. Support custom cost function over states (like noise in Japanese river crossing ↗
    ↗puzzle). */
169     state_space_t(State state, Cost init_cost, function<vector<State>(State&)> succ_gen, ↗
    ↗bool(*val_gen) (const State&), CostFn cost_succ_gen) : cost_generator(cost_succ_gen) {
170         //Generic type static asserts.
171
172         /* 10. User friendly to use and fail with a message if the user supplies arguments of ↗
    ↗wrong types. */
173         static_assert(is_class<State>::value, "Template argument state must be struct (or class)");
174         static_assert(is_class<Cost>::value, "Template argument init_cost must be struct (or ↗
    ↗class)");
175         static_assert(is_convertible<CostFn, function<Cost (const State&, const Cost&>>>::value, ↗
    ↗"Template argument cost_succ_gen must be function or lambda, that is convertible to function<U ↗
    ↗(const T&, const U&>>>");
176
177         //Assignments - note the cost_generator assignment is above to avoid constructing a lambda.

```

```

178     this->first_state = state;
179     this->successor_generator = succ_gen;
180     this->valid_state_func = val_gen;
181     this->first_cost = init_cost;
182 };
183
184 //Check with sensible default = breadth first search.
185 //http://mishadoff.com/images/dfs/binary_tree_search.png - visual aid
186 /* 5. Support various search orders: breadth-first search, depth-first search (the leaping
→frogs have different solutions) */
187 auto check(function<bool(const State&)> accept_func, search_order_t order =
→search_order_t::breadth_first) {
188     list<list<shared_ptr<State>>> search; //Result
189     if (order == search_order_t::breadth_first)
190     {
191         search = priority_queue_search<State, Cost>(first_state, first_cost,
→successor_generator, cost_generator, valid_state_func, accept_func, greater<pair<Cost, State>>());
192     }
193     else
194     {
195         search = priority_queue_search<State, Cost>(first_state, first_cost,
→successor_generator, cost_generator, valid_state_func, accept_func, less<pair<Cost, State>>());
196     }
197     //No need to use move here due to RVO
198     return search;
199 }
200
201 private:
202
203     State first_state;
204     Cost first_cost;
205
206     //This could say ' function<Cost (const State&, const Cost&)>' as type.
207     CostFn cost_generator;
208
209     function<vector<State>(State&)> successor_generator;
210     function<bool(const State&)> valid_state_func;
211
212 };
213
214 //1. Extend hash function for an arbitrary (iterable) container of basic types.
215 namespace std {
216
217     /* FULL DISCLOSURE, THIS WAS WRITTEN BY Marius Mikucionis
→http://people.cs.aau.dk/~marius/Teaching/AP2019/lecture9.html#/7/4 */
218     template <typename... Ts>
219     using void_t = void; // eats all valid types
220     template <typename T, typename = void> // primary declaration
221     struct is_container: std::false_type {}; // computes false
222     template <typename C> // specialization
223     struct is_container<C, // type c examination
224         void_t< // the following must evaluate to types:
225             typename C::iterator, // check subtype
226             typename C::const_iterator, // check subtype
227             //is_array<C>,
228             decltype(std::begin(std::declval<C&>())), // check iteration:
229             decltype(std::end(std::declval<C&>())) // create C() and call
230         > // finished checks
231     : std::true_type {}; // computes "true"
232     template <typename C> // *_t type alias

```

```

234     using is_container_t = typename is_container<C>::type;
235     template <typename C> // *_v value
236     constexpr auto is_container_v = is_container<C>::value;
237
238     template<template<typename, typename...> class Cont, typename T, typename... N>
239     struct hash<Cont<T, N...>> {
240         enable_if_t<is_container_v<Cont<T, N...>>, size_t> //enable_if_t<is_container_v<Cont<T,
↪N...>> || is_array<Cont<T, N...>>::value, size_t>
241         operator()(const Cont<T, N...> &container) const {
242
243             hash<T> hashT;
244             auto res = 0;
245             for (auto&& val : container)
246             {
247                 res = (res << 1) ^ hashT(val);
248             }
249             return res;
250         }
251     };
252
253     template<class T, size_t U>
254     struct hash<array<T,U>> {
255         auto operator() (const array<T,U>& data) const {
256             hash<T> hashT;
257             size_t res = 0;
258             for (auto&& val : data) // I love these loops
259             {
260                 res = (res << 1) ^ hashT(val);
261             }
262             return res;
263         }
264     };
265
266     /* // Use this to compile with Clang / MSVS
267     template<typename T>
268     struct hash <vector<T>> {
269         auto operator() (const vector<T>& data) const {
270             hash<T> hashT;
271             size_t res = 0;
272             for (auto&& val : data)
273             {
274                 res = (res << 1) ^ hashT(val);
275             }
276             return res;
277         }
278     }; */
279 };
280
281
282 /* 4. Print the trace of a state sequence from the initial state to the found goal state. */
283 template <class State>
284 ostream& operator <<(ostream& os, const list<shared_ptr<State>>& lst)
285 {
286     int i = 0;
287     //Print backwards - avoids having to shuffle list.
288     for (auto end = lst.rbegin(), front = lst.rend(); end != front; end++, i++)
289     {
290         os << i << " : " << **end << endl;
291     }
292 #if REACHABILITY_DEBUG
293     cin >> i; //Pause here because it's nice to see results - a real library obviously shouldn't

```

```

→do this.
294 #endif // REACHABILITY_DEBUG
295 return os;
296 }
297
298 /*
299 2. Create a generic successor generator function out of a transition generator function.
300 A transition generator function generates functions that change a state.
301 Each such function corresponds to a transition.
302 A successor generator function gets a state and generates a set of its successor states.
303 */
304 //Would have used auto here, but it narrowed it out and gave me 60 compiler errors, no fun.
305
306 /*Turns a transition generator function (transformer) into a successor generator function*/
307 template <class State>
308 function<vector<State>(State&)> successors(function<list<function<void(State&)>>(const State&)>
→transformer)
309 {
310     return [transformer](const State& data){
311         vector<State> res{};
312         for (auto&& func: transformer(data)) //For each function in the resulting list.
313         {
314             State copy = data;
315             func(copy);
316             res.push_back(copy);
317         }
318         return res;
319     };
320 }
321 #endif // !REACHABILITY

```

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     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
117
118
119     std::cout << "Leaping frog puzzle start: " << start << ", finish: " << finish << '\n';
120     auto space = state_space_t(std::move(start), successors<stones_t>(transitions));
121     auto solutions = space.check(
122         [finish=std::move(finish)](const stones_t& state){ return state==finish; },
123         order);
124     for (auto&& trace: solutions) {
125         std::cout << "Solution: trace of " << trace.size() << " states\n";
126         std::cout << trace;
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     solve(4); // 20 frogs may take >5.8GB of memory
135 }
136
137 /** Sample output:
138 Leaping frog puzzle start: GG_BB
139 state GG_BB has 4 transitions, leading to:
140   state G_GBB has 2 transitions, leading to:
141     state _GGBB has 0 transitions
142     state GBG_B has 2 transitions, leading to:
143       state GB_GB has 2 transitions, leading to:
144         state _BGGB has 1 transitions, leading to:
145           state B_GGB has 0 transitions
146           state GBBG_ has 1 transitions, leading to:
147             state GBB_G has 0 transitions
148             state GBGB_ has 1 transitions, leading to:
149               state _GBBG has 1 transitions, leading to:
150                 state B_GBG has 1 transitions, leading to:
151                   state BBG_G has 1 transitions, leading to:

```

```

152         state BB_GG has 0 transitions
153     state GBB_G has 0 transitions
154 state _GGBB has 0 transitions
155 state GGB_B has 2 transitions, leading to:
156     state G_BGB has 2 transitions, leading to:
157         state _GBGB has 1 transitions, leading to:
158             state BG_GB has 2 transitions, leading to:
159                 state B_GGB has 0 transitions
160                 state BGBG_ has 1 transitions, leading to:
161                     state BGB_G has 1 transitions, leading to:
162                         state B_BGG has 1 transitions, leading to:
163                             state BB_GG has 0 transitions
164             state GB_GB has 2 transitions, leading to:
165                 state _BGGB has 1 transitions, leading to:
166                     state B_GGB has 0 transitions
167                 state GBBG_ has 1 transitions, leading to:
168                     state GBB_G has 0 transitions
169             state GGBB_ has 0 transitions
170             state GGBB_ has 0 transitions
171 Leaping frog puzzle start: GG_BB, finish: BB_GG
172 --- Solve with default (breadth-first) search: ---
173 Solution: a trace of 9 states
174 State of 5 stones: GG_BB
175 State of 5 stones: G_GBB
176 State of 5 stones: GBG_B
177 State of 5 stones: GBGB_
178 State of 5 stones: GB_BG
179 State of 5 stones: _BGBG
180 State of 5 stones: B_GBG
181 State of 5 stones: BBG_G
182 State of 5 stones: BB_GG
183 --- Solve with depth-first search: ---
184 Leaping frog puzzle start: GG_BB, finish: BB_GG
185 Solution: trace of 9 states
186 State of 5 stones: GG_BB
187 State of 5 stones: GGB_B
188 State of 5 stones: G_BGB
189 State of 5 stones: _GBGB
190 State of 5 stones: BG_GB
191 State of 5 stones: BGBG_
192 State of 5 stones: BGB_G
193 State of 5 stones: B_BGG
194 State of 5 stones: BB_GG
195 Leaping frog puzzle start: GGGG_BBBB, finish: BBBB_GGGG
196 Solution: trace of 25 states
197 State of 9 stones: GGGG_BBBB
198 State of 9 stones: GGG_GB BBB
199 State of 9 stones: GGGBG_BBB
200 State of 9 stones: GGGBGB_BB
201 State of 9 stones: GGGB_BGBB
202 State of 9 stones: GG_BGBGBB
203 State of 9 stones: G_GBGBGBB
204 State of 9 stones: GBG_GBGBB
205 State of 9 stones: GBGBG_GBB
206 State of 9 stones: GBGBGBG_B
207 State of 9 stones: GBGBGBGB_
208 State of 9 stones: GBGBGB_BG
209 State of 9 stones: GBGB_BGBG
210 State of 9 stones: GB_BGBGBG
211 State of 9 stones: _BGBGBGBG
212 State of 9 stones: B_GBGBGBG

```

```

213 State of 9 stones: BBG_GBGBG
214 State of 9 stones: BBGBG_GBGB
215 State of 9 stones: BBGBGBG_G
216 State of 9 stones: BBGBGB_GG
217 State of 9 stones: BBGB_BGGG
218 State of 9 stones: BB_BGBGGG
219 State of 9 stones: BBB_GBGGG
220 State of 9 stones: BBBBG_GGG
221 State of 9 stones: BBBB_GGGG
222
223 */

```

Listing 4: 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
340 int main() {
341     std::cout << "-- Solve using depth as a cost: ---\n";
342     solve([](const state_t& state, const cost_t& prev_cost){
343         return cost_t{ prev_cost.depth+1, prev_cost.noise };
344     }); // it is likely that daughters will get to shore2 first
345     std::cout << "-- Solve using noise as a cost: ---\n";
346     solve([](const state_t& state, const cost_t& prev_cost){
347         auto noise = prev_cost.noise;
348         if (state.persons[person_t::son1].pos == person_t::shore1)
349             noise += 2; // older son is more naughty, prefer him first
350         if (state.persons[person_t::son2].pos == person_t::shore1)
351             noise += 1;
352         return cost_t{ prev_cost.depth, noise };
353     }); // son1 should get to shore2 first

```



```

415 {trv,2,2},{~~},{~~},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
416 {trv,1,2},{~~},{SH2},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
417 {trv,2,2},{~~},{SH2},{~~},{sh1},{SH2},{SH2},{SH2},{SH2}
418 {trv,2,2},{SH2},{SH2},{SH2},{sh1},{SH2},{SH2},{~~},{~~}
419 {trv,2,2},{SH2},{SH2},{SH2},{~~},{SH2},{SH2},{~~},{sh1}
420 {trv,1,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~},{sh1}
421 {trv,2,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~},{~~}
422 */

```

Listing 5: 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 */

```
