Puzzle Engine Library AP Exam

Christopher Hansen Nielsen

June 10, 2019

Listing 1: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(PuzzleEngine CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -fsanitize=undefined -fsanitize=address")
set(CMAKE_LINK_FLAGS_DEBUG "${CMAKE_LINK_FLAGS_DEBUG} -fsanitize=undefined -fsanitize=address")

add_executable(frogs frogs.cpp)
add_executable(crossing crossing.cpp)
add_executable(family family.cpp)
```

Listing 2: reachability.hpp

```
/**
 * The reachability header file library.
 * Author: Christopher Hansen Nielsen
 * Mail: chni15@student.aau.dk, Student Number: 20154154
 *
 * Written and compiled on a windows machine.
 * g++ -std=c++17 -pedantic -Wall -DNDEBUG -O3 -o frogs frogs.cpp
 * g++ -std=c++17 -pedantic -Wall -DNDEBUG -O3 -o crossing crossing.cpp
 * g++ -std=c++17 -pedantic -Wall -DNDEBUG -O3 -o family family.cpp
 */

#ifndef PUZZLEENGINE_REACHABILITY_HPP
#define PUZZLEENGINE_REACHABILITY_HPP

#include <vector>
#include <list>
#include <functional>
#include <iostream>
#include <algorithm>
#include <typeinfo>

// This enum is used to handle the support for different search orders except for cost order. It is implemented
// as part of requirement 5.
enum search_order_t {
    breadth_first, depth_first
};

// This struct is the basis for keeping track of the solution when traversing the states. When it is used, it holds a
// pointer to the parent node as well as a copy of the state.
template<class StateTypeT>
struct trace_node {
```

```
    trace_node *parentState;
    StateTypeT selfState;
};

// This function is used to pass on the transition generator function from the respective puzzles. It is implemented
// as part of requirement 2.
template<class StateTypeT>
std::function<std::list<std::function<void(StateTypeT &)>>(StateTypeT &)>
successors(std::list<std::function<void(StateTypeT &)>> transitions(const StateTypeT &state)) {
    return transitions;
}

// This class holds all the information about a given state space. It utilizes two template types StateTypeT and
// CostTypeT. These are the basis of the generic implementation as part of requirements 8 and 9.
template<class StateTypeT, class CostTypeT = std::nullptr_t>
class state_space_t {
private:
    StateTypeT _startState;
    CostTypeT _initialCost;
    std::function<std::list<std::function<void(StateTypeT &)>>(StateTypeT &)> _transitionFunctions;
    std::function<bool(const StateTypeT &)> _invariantFunction;
    std::function<CostTypeT(const StateTypeT &state, const CostTypeT &cost)> _costFunction;
    bool _isCostEnabled; // used explicitly to determine whether or not a cost have been specified.

    template<class ValidationFunction>
    std::list<StateTypeT> solveOrder(ValidationFunction isGoalState, search_order_t order);

    template<class ValidationFunction>
    std::list<StateTypeT> solveCost(ValidationFunction isGoalState);

public:
    // This is the first constructor for the class, which handles calls from the frogs.cpp
    // and crossing.cpp instantiation.
    state_space_t(const StateTypeT startInputState,
                  std::function<std::list<std::function<void(StateTypeT &)>>(StateTypeT &)> transFunctions,
                  bool invariantFunc(const StateTypeT &) = [](
                          const StateTypeT &state) { return true; }) : _startState(startInputState),
                                                                       _transitionFunctions(
                                                                               transFunctions),
                                                                       _invariantFunction(invariantFunc),
                                                                       _isCostEnabled(false){
    }

    // This the second and overloaded constructor for the class. This handles calls from the family.cpp.
    state_space_t(const StateTypeT startInputState, const CostTypeT costInput,
                  std::function<std::list<std::function<void(StateTypeT &)>>(StateTypeT &)> transFunctions,
                  bool invariantFunc(const StateTypeT &) = [](
                          const StateTypeT &state) { return true; },
                  std::function<CostTypeT(const StateTypeT &state, const CostTypeT &cost)> costFunc = [](
                          const StateTypeT &state, const CostTypeT &cost)
                                  { return CostTypeT{0, 0}; }) : _startState(
                                                                       startInputState),
                                                               _initialCost(
                                                                       costInput),
                                                               _transitionFunctions(
```

```
                                                                              transFunctions),
                                                                      _invariantFunction(
                                                                              invariantFunc),
                                                                      _costFunction(
                                                                              costFunc),
                                                                      _isCostEnabled(
                                                                              true) {
    }

    template<class ValidationFunction>
    std::list<StateTypeT> check(ValidationFunction isGoalState, search_order_t order = search_order_t::breadth_first);
};

// This function is called from the different puzzle files and returns a solution if found. It introduces a new template
// ValidationFunction that handles the goal predicate function. It also takes an order, which is defaulted to
// breadth first if nothing else is specified.
// It returns a list of states.
template<class StateTypeT, class CostTypeT>
template<class ValidationFunction>
std::list<StateTypeT> state_space_t<StateTypeT, CostTypeT>::check(ValidationFunction isGoalState, search_order_t order) {
    std::list<StateTypeT> solution;

    if (_isCostEnabled) { // Here we check if the cost method is specified, and calls the solveCost if true.
        solution = solveCost(isGoalState);
    } else { // Otherwise we call the solveOrder method with the order provided.
        solution = solveOrder(isGoalState, order);
    }

    // Returns the list of states. Implemented as part of requirement 4.
    return solution;
}

// The method is used when solving the state space based on a given cost. It takes in isGoalState which is a predicate
// that is used to determine whether a solution have been found.
// It returns a list of states. It is implemented as part of requirement 7.
template<class StateTypeT, class CostTypeT>
template<class ValidationFunction>
std::list<StateTypeT> state_space_t<StateTypeT, CostTypeT>::solveCost(ValidationFunction isGoalState) {
    StateTypeT currentState;
    CostTypeT itCost{_initialCost}, newCost;
    trace_node<StateTypeT> *traceState {};
    std::list<StateTypeT> passed, solution;
    std::list<std::pair<CostTypeT, trace_node<StateTypeT> *>> waiting;

    // The method utilizes a waiting list that contains a pair of cost and a trace_node. The cost is used to determine
    // which element should be popped from the list next.
    waiting.push_back(std::make_pair(itCost, new trace_node<StateTypeT>{nullptr, _startState}));

    while (!waiting.empty()) {
        // Here we take the first element from the list and then pop it.
        currentState = waiting.front().second->selfState;
        traceState = waiting.front().second;
        itCost = waiting.front().first;
```

```
        waiting.pop_front();

        // Here we check if the goal state has been reached. This is implemented as part of requirement 3.
        if (isGoalState(currentState)) {
            // If a goal is found, we use the traceState to traverse back up through the tree of trace_nodes
            // until the parentState is NULL. For each node we push the state to the solution list.
            while (traceState->parentState != NULL) {
                solution.push_front(traceState->selfState);
                traceState = traceState->parentState;
            }

            solution.push_front(traceState->selfState); // Adds the start state to the solution trace.
            return solution;
        }

        // Here we check if the currentState is part of the passed list.
        if (!(std::find(passed.begin(), passed.end(), currentState) != passed.end())) {
            // If it is not, we push it to the list, and generate the transitions via the _transitionFunctions which
            // is a member of the state_space_t class.
            passed.push_back(currentState);
            auto transitions = _transitionFunctions(currentState);

            for (auto transition: transitions) {
                // For each transition, we then generate the successor
                auto successor{currentState};
                transition(successor);

                // Prevents invalid states being added to waiting via an invariant predicate. This is implemented as
                // part of requirement 6.
                if (!_invariantFunction(successor)) {
                    continue;
                }
                newCost = _costFunction(successor, itCost);
                waiting.push_back(std::make_pair(newCost, new trace_node<StateTypeT>{traceState, successor}));
            }
            if (!transitions.empty()) { // Prevents sorting if no new transitions have been found.
                // Here we sort the waiting list based on the cost for each pair in the list.
                waiting.sort([](const std::pair<CostTypeT, trace_node<StateTypeT> *> &a,
                                std::pair<CostTypeT, trace_node<StateTypeT> *> &b) { return a.first < b.first; });
            }
        }
    }

    return solution;
}

// The method is used when solving the state space based on a given order. It takes in isGoalState which is a predicate
// that is used to determine whether a solution have been found. It also takes an order, which specifies how the
// solution should be found. The method is very similar to solveCost in functionality.
// It returns a list of states.
template<class StateTypeT, class CostTypeT>
template<class ValidationFunction>
```

```cpp
std::list<StateTypeT>
state_space_t<StateTypeT, CostTypeT>::solveOrder(ValidationFunction isGoalState, search_order_t order) {
    StateTypeT currentState;
    trace_node<StateTypeT> *traceState {};
    std::list<StateTypeT> passed, solution;
    std::list<trace_node<StateTypeT> *> waiting;

    // As solveOrder does not utilize a cost, waiting is just a list of trace_nodes.
    waiting.push_back(new trace_node<StateTypeT>{nullptr, _startState});

    while (!waiting.empty()) {
        switch (order) { // We switch on the order to determine what element should be accessed and popped from waiting.
            case breadth_first:
                currentState = waiting.front()->selfState;
                traceState = waiting.front();
                waiting.pop_front();
                break;
            case depth_first:
                currentState = waiting.back()->selfState;
                traceState = waiting.back();
                waiting.pop_back();
                break;
            default:
                std::cout << "Order not supported" << std::endl;
                break;
        }
        if (isGoalState(currentState)) {
            while (traceState->parentState != NULL) {
                solution.push_front(traceState->selfState);
                traceState = traceState->parentState;
            }

            solution.push_front(traceState->selfState); // Adds the start state to the solution trace.
            return solution;
        }
        if (!(std::find(passed.begin(), passed.end(), currentState) != passed.end())) {
            passed.push_back(currentState);
            auto transitions = _transitionFunctions(currentState);

            for (auto transition: transitions) {
                auto successor{currentState};
                transition(successor);

                if (!_invariantFunction(successor)) { // Prevents invalid states being added to waiting.
                    continue;
                }
                waiting.push_back(new trace_node<StateTypeT>{traceState, successor});
            }
        }
    }

    return solution;
}

// The following hash override is implemented in order to compile family.cpp. It is, however, never used and should
// not be used at all. Gives a warning when compiling the program.
```

```
template<class StateType, size_t typeSize>
struct std::hash<std::array<StateType, typeSize>> {
    std::size_t operator()(const array<StateType, typeSize> &key) const {
        return NULL;
    }
};

#endif //PUZZLEENGINE_REACHABILITY_HPP
```

Listing 3: frogs.cpp

```
/**
 * Solution to a frog leap puzzle:
 * http://arcade.modemhelp.net/play-4863.html
 * Author: Marius Mikucionis <marius@cs.aau.dk>
 * Compile and run:
 * g++ -std=c++17 -pedantic -Wall -DNDEBUG -O3 -o frogs frogs.cpp && ./frogs
 */
#include "reachability.hpp" // your header-only library solution

#include <iostream>
#include <list>
//#include <functional> // std::function

enum class frog_t { empty, green, brown };
using stones_t = std::vector<frog_t>;

std::list<std::function<void(stones_t&)>> transitions(const stones_t& stones) {
    auto res = std::list<std::function<void(stones_t&)>>{};
    if (stones.size()<2)
        return res;
    auto i=0u;
    while (i < stones.size() && stones[i]!=frog_t::empty) ++i; // find empty stone
    if (i==stones.size())
        return res;  // did not find empty stone
    // explore moves to fill the empty from left to right (only green can do that):
    if (i > 0 && stones[i-1]==frog_t::green)
        res.push_back([i](stones_t& s){ // green jump to next
                          s[i-1] = frog_t::empty;
                          s[i]   = frog_t::green;
                      });
    if (i > 1 && stones[i-2]==frog_t::green)
        res.push_back([i](stones_t& s){ // green jump over 1
                          s[i-2] = frog_t::empty;
                          s[i]   = frog_t::green;
                      });
    // explore moves to fill the empty from right to left (only brown can do that):
    if (i < stones.size()-1 && stones[i+1]==frog_t::brown) {
        res.push_back([i](stones_t& s){ // brown jump to next
                          s[i+1] = frog_t::empty;
                          s[i]   = frog_t::brown;
                      });
    }
    if (i < stones.size()-2 && stones[i+2]==frog_t::brown) {
        res.push_back([i](stones_t& s){ // brown jump over 1
                          s[i+2]=frog_t::empty;
                          s[i]=frog_t::brown;
                      });
    }
    return res;
}

```

```
std::ostream& operator<<(std::ostream& os, const stones_t& stones) {
    for (auto&& stone: stones)
        switch (stone) {
        case frog_t::green: os << "G"; break;
        case frog_t::empty: os << "_"; break;
        case frog_t::brown: os << "B"; break;
        default: os << "?"; break; // something went terribly wrong
        }
    return os;
}

std::ostream& operator<<(std::ostream& os, const std::list<const stones_t*>& trace) {
    for (auto stones: trace)
        os << "State of " << stones->size() << " stones: " << *stones << '\n';
    return os;
}

void show_successors(const stones_t& state, const size_t level=0) {
    // Caution: this function uses recursion, which is not suitable for solving puzzles!!
    // 1) some state spaces can be deeper than stack allows.
    // 2) it can only perform depth-first search
    // 3) it cannot perform breadth-first search, cheapest-first, greatest-first etc.
    auto trans = transitions(state); // compute the transitions
    std::cout << std::string(level*2, ' ')
              << "state " << state << " has " << trans.size() << " transitions";
    if (trans.empty())
        std::cout << '\n';
    else
        std::cout << ", leading to:\n";
    for (auto& t: trans) {
        auto succ = state; // copy the original state
        t(succ); // apply the transition on the state to compute successor
        show_successors(succ, level+1);
    }
}

void explain(){
    const auto start = stones_t{{ frog_t::green, frog_t::green, frog_t::empty,
                                  frog_t::brown, frog_t::brown }};
    std::cout << "Leaping frog puzzle start: " << start << '\n';
    show_successors(start);
    const auto finish = stones_t{{ frog_t::brown, frog_t::brown, frog_t::empty,
                                   frog_t::green, frog_t::green }};
    std::cout << "Leaping frog puzzle start: " << start << ", finish: " << finish << '\n';
    // Added type specification to the template.
    auto space = state_space_t<stones_t >(start, successors<stones_t>(transitions));// define state space
    // explore the state space and find the solutions satisfying goal:
    std::cout << "--- Solve with default (breadth-first) search: ---\n";
    auto solutions = space.check([&finish](const stones_t& state){ return state==finish; });
    for (auto&& trace: solutions) { // iterate through solutions:
        std::cout << "Solution: a trace of " << trace.size() << " states\n";
        std::cout << trace; // print solution
    }
}

void solve(size_t frogs, search_order_t order = search_order_t::breadth_first){
    const auto stones = frogs*2+1; // frogs on either side and 1 empty in the middle
    auto start = stones_t(stones, frog_t::empty);  // initially all empty
    auto finish = stones_t(stones, frog_t::empty); // initially all empty
    while (frogs-->0) { // count down from frogs-1 to 0 and put frogs into positions:
```

```
        start[frogs] = frog_t::green;                  // green on left
        start[start.size()-frogs-1] = frog_t::brown;   // brown on right
        finish[frogs] = frog_t::brown;                 // brown on left
        finish[finish.size()-frogs-1] = frog_t::green; // green on right
    }
    std::cout << "Leaping frog puzzle start: " << start << ", finish: " << finish << '\n';
    // Added type specification to the template.
    auto space = state_space_t<stones_t >(std::move(start), successors<stones_t>(transitions));
    auto solutions = space.check(
        [finish=std::move(finish)](const stones_t& state){ return state==finish; },
        order);
    // Introduced a change in print to adhere to my solution
    std::cout << "Solution: trace of " << solutions.size() << " states\n";
    for (auto&& trace: solutions) {
        std::cout << trace << std::endl;
    }
}

int main(){
    //explain();
    std::cout << "--- Solve with depth-first search: ---\n";
    solve(2, search_order_t::depth_first);
    std::cout << "--- Solve with breadth-first search: ---\n";
    solve(2); // 20 frogs may take >5.8GB of memory
}
/** Sample output:
Leaping frog puzzle start: GG_BB
state GG_BB has 4 transitions, leading to:
  state G_GBB has 2 transitions, leading to:
    state _GGBB has 0 transitions
    state GBG_B has 2 transitions, leading to:
      state GB_GB has 2 transitions, leading to:
        state _BGGB has 1 transitions, leading to:
          state B_GGB has 0 transitions
        state GBBG_ has 1 transitions, leading to:
          state GBB_G has 0 transitions
      state GBGB_ has 1 transitions, leading to:
        state GB_BG has 2 transitions, leading to:
          state _BGBG has 1 transitions, leading to:
            state B_GBG has 1 transitions, leading to:
              state BBG_G has 1 transitions, leading to:
                state BB_GG has 0 transitions
          state GBB_G has 0 transitions
  state _GGBB has 0 transitions
  state GGB_B has 2 transitions, leading to:
    state G_BGB has 2 transitions, leading to:
      state _GBGB has 1 transitions, leading to:
        state BG_GB has 2 transitions, leading to:
          state B_GGB has 0 transitions
          state BGBG_ has 1 transitions, leading to:
            state BGB_G has 1 transitions, leading to:
              state B_BGG has 1 transitions, leading to:
                state BB_GG has 0 transitions
      state GB_GB has 2 transitions, leading to:
        state _BGGB has 1 transitions, leading to:
          state B_GGB has 0 transitions
        state GBBG_ has 1 transitions, leading to:
          state GBB_G has 0 transitions
    state GGBB_ has 0 transitions
  state GGBB_ has 0 transitions
Leaping frog puzzle start: GG_BB, finish: BB_GG
```

```
--- Solve with default (breadth-first) search: ---
Solution: a trace of 9 states
State of 5 stones: GG_BB
State of 5 stones: G_GBB
State of 5 stones: GBG_B
State of 5 stones: GBGB_
State of 5 stones: GB_BG
State of 5 stones: _BGBG
State of 5 stones: B_GBG
State of 5 stones: BBG_G
State of 5 stones: BB_GG
--- Solve with depth-first search: ---
Leaping frog puzzle start: GG_BB, finish: BB_GG
Solution: trace of 9 states
State of 5 stones: GG_BB
State of 5 stones: GGB_B
State of 5 stones: G_BGB
State of 5 stones: _GBGB
State of 5 stones: BG_GB
State of 5 stones: BGBG_
State of 5 stones: BGB_G
State of 5 stones: B_BGG
State of 5 stones: BB_GG
Leaping frog puzzle start: GGGG_BBBB, finish: BBBB_GGGG
Solution: trace of 25 states
State of 9 stones: GGGG_BBBB
State of 9 stones: GGG_GBBBB
State of 9 stones: GGGBG_BBB
State of 9 stones: GGGBGB_BB
State of 9 stones: GGGB_BGBB
State of 9 stones: GG_BGBGBB
State of 9 stones: G_GBGBGBB
State of 9 stones: GBG_GBGBB
State of 9 stones: GBGBG_GBB
State of 9 stones: GBGBGBG_B
State of 9 stones: GBGBGBGB_
State of 9 stones: GBGBGB_BG
State of 9 stones: GBGB_BGBG
State of 9 stones: GB_BGBGBG
State of 9 stones: _BGBGBGBG
State of 9 stones: B_GBGBGBG
State of 9 stones: BBG_GBGBG
State of 9 stones: BBGBG_GBG
State of 9 stones: BBGBGBG_G
State of 9 stones: BBGBGB_GG
State of 9 stones: BBGB_BGGG
State of 9 stones: BB_BGBGGG
State of 9 stones: BBB_GBGGG
State of 9 stones: BBBBG_GGG
State of 9 stones: BBBB_GGGG

*/
```

Listing 4: crossing.cpp

```
/**
 * Solution to river crossing puzzle with a goat, a cabbage and a wolf.
 * Author: Marius Mikucionis <marius@cs.aau.dk>
 * Compile and run:
 * g++ -std=c++17 -pedantic -Wall -DNDEBUG -O3 -o crossing crossing.cpp && ./crossing
 */
#include "reachability.hpp" // your header-only library solution
```

```
#include <functional> // std::function
#include <list>
#include <array>
#include <iostream>

enum actor_t { cabbage, goat, wolf }; // names of the actors
enum class pos_t { shore1, travel, shore2}; // names of the actor positions
using actors_t = std::array<pos_t,3>; // positions of the actors

auto transitions(const actors_t& actors) {
    auto res = std::list<std::function<void(actors_t&)>>{};
    for (auto i=0u; i<actors.size(); ++i)
        switch(actors[i]) {
        case pos_t::shore1:
            res.push_back([i](actors_t& actors){ actors[i] = pos_t::travel; });
            break;
        case pos_t::travel:
            res.push_back([i](actors_t& actors){ actors[i] = pos_t::shore1; });
            res.push_back([i](actors_t& actors){ actors[i] = pos_t::shore2; });
            break;
        case pos_t::shore2:
            res.push_back([i](actors_t& actors){ actors[i] = pos_t::travel; });
            break;
        }
    return res;
}

bool is_valid(const actors_t& actors) {
    // only one passenger:
    if (std::count(std::begin(actors), std::end(actors), pos_t::travel)>1)
        return false;
    // goat cannot be left alone with wolf, as wolf will eat the goat:
    if (actors[actor_t::goat]==actors[actor_t::wolf] && actors[actor_t::cabbage]==pos_t::travel)
        return false;
    // goat cannot be left alone with cabbage, as goat will eat the cabbage:
    if (actors[actor_t::goat]==actors[actor_t::cabbage] && actors[actor_t::wolf]==pos_t::travel)
        return false;
    return true;
}

std::ostream& operator<<(std::ostream& os, const pos_t& pos) {
    switch(pos) {
    case pos_t::shore1: os << "1"; break;
    case pos_t::travel: os << "~"; break;
    case pos_t::shore2: os << "2"; break;
    default: os << "?"; break; // something went terribly wrong
    }
    return os;
}

std::ostream& operator<<(std::ostream& os, const actors_t& actors) {
    return os << actors[actor_t::cabbage]
              << actors[actor_t::goat]
              << actors[actor_t::wolf];
}

std::ostream& operator<<(std::ostream& os, std::list<const actors_t*>& trace) {
    auto step = 0u;
    for (auto* actors: trace)
        os << step++ << ": " << *actors << '\n';
```

```
    return os;
}

void solve(){
    auto state_space = state_space_t<actors_t>(
        actors_t{},                        // initial state
        successors<actors_t>(transitions), // successor generator
        &is_valid);                        // invariant over all states
    auto solution = state_space.check(
        [](const actors_t& actors){ // all actors should be on the shore2:
            return std::count(std::begin(actors), std::end(actors), pos_t::shore2)==actors.size();
        });
    // Introduced a change in print to adhere to my solution
    std::cout << "#  CGW" << std::endl;
    int it = 0;
    for (auto&& trace: solution)
        std::cout << it << ": " << trace << std::endl;
}

int main(){
    solve();
}

/** Sample output:
#  CGW
0: 111
1: 1~1
2: 121
3: ~21
4: 221
5: 2~1
6: 211
7: 21~
8: 212
9: 2~2
10: 222
*/
```

Listing 5: family.cpp

```
/**
 * Reachability algorithm implementation for river-crossing puzzle:
 * https://www.funzug.com/index.php/flash-games/japanese-river-crossing-puzzle-game.html
 * Author: Marius Mikucionis <marius@cs.aau.dk>
 * Compile using:
 * g++ -std=c++17 -pedantic -Wall -DNDEBUG -O3 -o family family.cpp && ./family
 * Inspect the solution (only the traveling part):
 * ./family | grep trv | grep '~~~'
 */

#include "reachability.hpp" // your header-only library solution

#include <iostream>
#include <vector>
#include <list>
#include <array>
#include <functional> // std::function
#include <algorithm>  // all_of

/** Model of the river crossing: persons and a boat */
struct person_t {
    enum { shore1, onboard, shore2 } pos = shore1;
```

```
    enum { mother, father, daughter1, daughter2, son1, son2, policeman, prisoner };
};

struct boat_t {
    enum { shore1, travel, shore2 } pos = shore1;
    uint16_t capacity{2};
    uint16_t passengers{0};
};
struct state_t {
    boat_t boat;
    std::array<person_t,8> persons;
};

/** less-than operators for std::map */
bool operator<(const person_t& p1, const person_t& p2) {
    if (p1.pos < p2.pos)
        return true;
    else if (p2.pos < p1.pos)
        return false; // p2 < p1
    return false; // equal
}

bool operator<(const boat_t& b1, const boat_t& b2) {
    if (b1.pos < b2.pos)
        return true;
    else if (b2.pos < b1.pos)
        return false;
    if (b1.passengers < b2.passengers)
        return true;
    else if (b2.passengers < b1.passengers)
        return false;
    if (b1.capacity < b2.capacity)
        return true;
    else if (b2.capacity < b1.capacity)
        return false;
    return false;
}

bool operator<(const state_t& s1, const state_t& s2) {
    if (s1.boat < s2.boat)
        return true;
    if (s2.boat < s1.boat)
        return false; // s2 < s1
    for (auto i=0u; i<s1.persons.size(); ++i)
        if (s1.persons[i] < s2.persons[i])
            return true;
        else if (s2.persons[i] < s1.persons[i])
            return false;
    return false; // s2 == s1
}

/** equality operations for std::unordered_map */
bool operator==(const person_t& p1, const person_t& p2) {
    return (p1.pos == p2.pos);
}

bool operator==(const boat_t& b1, const boat_t& b2) {
    return (b1.pos == b2.pos) &&
        (b1.capacity == b2.capacity) &&
        (b1.passengers == b2.passengers);
}
```

```
bool operator==(const state_t& s1, const state_t& s2) {
    return (s1.boat == s2.boat) && (s1.persons == s2.persons);
}

/** hash operations for std::unordered_map */
namespace std {
    template <>
    struct hash<person_t> {
        std::size_t operator()(const person_t& key) const {
            return std::hash<decltype(key.pos)>{}(key.pos);
        }
    };
    template <>
    struct hash<boat_t> {
        std::size_t operator()(const boat_t& key) const {
            auto h_pos = std::hash<decltype(key.pos)>{};
            auto h_int = std::hash<decltype(key.capacity)>{};
            return (((h_pos(key.pos) << 1) ^
                      h_int(key.capacity)) << 1) ^
                    h_int(key.passengers));
        }
    };

    template <>
    struct hash<state_t> {
        std::size_t operator()(const state_t& key) const {
            return (std::hash<boat_t>{}(key.boat) << 1) ^
                std::hash<decltype(key.persons)>{}(key.persons); // assumes hash over container
        }
    };
}

std::ostream& operator<<(std::ostream& os, const person_t& p) {
    os << '{';
    switch (p.pos) {
    case person_t::shore1: os << "sh1"; break;
    case person_t::onboard: os << "~~~"; break;
    case person_t::shore2: os << "SH2"; break;
    default: os << "???" ; break; // something went terribly wrong
    }
    return os << '}';
}

std::ostream& operator<<(std::ostream& os, const boat_t& b) {
    os << '{';
    switch (b.pos) {
    case boat_t::shore1: os << "sh1"; break;
    case boat_t::travel: os << "trv"; break;
    case boat_t::shore2: os << "SH2"; break;
    default: os << "???" ; break; // something went terribly wrong
    }
    return os << ',' << b.passengers << ',' << b.capacity << '}';
}


std::ostream& operator<<(std::ostream& os, const state_t& s){
    return os << s.boat << ','
              << s.persons[person_t::mother] << ','
              << s.persons[person_t::father] << ','
              << s.persons[person_t::daughter1] << ','
```

```
              << s.persons[person_t::daughter2] << ','
              << s.persons[person_t::son1] << ','
              << s.persons[person_t::son2] << ','
              << s.persons[person_t::policeman] << ','
              << s.persons[person_t::prisoner];
}

/**
 * Returns a list of transitions applicable on a given state.
 * transition is a function modifying a state
 */
std::list<std::function<void(state_t&)>>
transitions(const state_t& s) {
    auto res = std::list<std::function<void(state_t&)>>{};
    switch (s.boat.pos) {
    case boat_t::shore1:
    case boat_t::shore2:
        if (s.boat.passengers>0) // start traveling
            res.push_back([](state_t& state){ state.boat.pos = boat_t::travel; });
        break;
    case boat_t::travel:
        res.emplace_back([](state_t& state){ // arrive to shore1
                             state.boat.pos = boat_t::shore1;
                             state.boat.passengers = 0;
                             for (auto& p: state.persons)
                                 if (p.pos == person_t::onboard)
                                     p.pos = person_t::shore1;
                         });
        res.emplace_back([](state_t& state){    // arrive to shore2
                             state.boat.pos = boat_t::shore2;
                             state.boat.passengers = 0;
                             for (auto& p: state.persons)
                                 if (p.pos == person_t::onboard)
                                     p.pos = person_t::shore2;
                         });
        break;
    }
    for (auto i=0u; i<s.persons.size(); ++i) {
        switch (s.persons[i].pos) {
        case person_t::shore1:  // board the boat on shore1:
            if (s.boat.pos == boat_t::shore1)
                res.push_back([i](state_t& state){
                                  state.persons[i].pos = person_t::onboard;
                                  state.boat.passengers++;
                              });
            break;
        case person_t::shore2: // board the boat on shore2:
            if (s.boat.pos == boat_t::shore2)
                res.push_back([i](state_t& state){
                                  state.persons[i].pos = person_t::onboard;
                                  state.boat.passengers++;
                              });
            break;
        case person_t::onboard:
            if (s.boat.pos == boat_t::shore1) // leave the boat to shore1
                res.push_back([i](state_t& state){
                                  state.persons[i].pos = person_t::shore1;
                                  state.boat.passengers--;
                              });
            else if (s.boat.pos == boat_t::shore2) // leave the boat to shore2
                res.push_back([i](state_t& state){
```

```
                                state.persons[i].pos = person_t::shore2;
                                state.boat.passengers--;
                            });
            break;
        }
    }
    return res;
}

bool river_crossing_valid(const state_t& s) {
    if (s.boat.passengers > s.boat.capacity) {
//      log(" boat overload\n");
        return false;
    }
    if (s.boat.pos == boat_t::travel) {
        if (s.persons[person_t::daughter1].pos == person_t::onboard) {
            if (s.boat.passengers==1 ||
                (s.persons[person_t::daughter2].pos == person_t::onboard) ||
                (s.persons[person_t::son1].pos == person_t::onboard) ||
                (s.persons[person_t::son2].pos == person_t::onboard) ||
                (s.persons[person_t::prisoner].pos == person_t::onboard)) {
//              log(" d1 travel alone\n");
                return false;
            }
        } else if (s.persons[person_t::daughter2].pos == person_t::onboard) {
            if (s.boat.passengers==1 ||
                (s.persons[person_t::daughter1].pos == person_t::onboard) ||
                (s.persons[person_t::son1].pos == person_t::onboard) ||
                (s.persons[person_t::son2].pos == person_t::onboard) ||
                (s.persons[person_t::prisoner].pos == person_t::onboard)) {
//              log(" d2 travel alone\n");
                return false;
            }
        } else if (s.persons[person_t::son1].pos == person_t::onboard) {
            if (s.boat.passengers==1 ||
                (s.persons[person_t::daughter1].pos == person_t::onboard) ||
                (s.persons[person_t::daughter2].pos == person_t::onboard) ||
                (s.persons[person_t::son2].pos == person_t::onboard) ||
                (s.persons[person_t::prisoner].pos == person_t::onboard)) {
//              log(" s1 travel alone\n");
                return false;
            }
        } else if (s.persons[person_t::son2].pos == person_t::onboard) {
            if (s.boat.passengers==1 ||
                (s.persons[person_t::daughter1].pos == person_t::onboard) ||
                (s.persons[person_t::daughter2].pos == person_t::onboard) ||
                (s.persons[person_t::son1].pos == person_t::onboard) ||
                (s.persons[person_t::prisoner].pos == person_t::onboard)) {
//              log(" s2 travel alone\n");
                return false;
            }
        }
        if (s.persons[person_t::prisoner].pos != s.persons[person_t::policeman].pos) {
            auto prisoner_pos = s.persons[person_t::prisoner].pos;
            if ((s.persons[person_t::daughter1].pos == prisoner_pos) ||
                (s.persons[person_t::daughter2].pos == prisoner_pos) ||
                (s.persons[person_t::son1].pos == prisoner_pos) ||
                (s.persons[person_t::son2].pos == prisoner_pos) ||
                (s.persons[person_t::mother].pos == prisoner_pos) ||
                (s.persons[person_t::father].pos == prisoner_pos)) {
//              log(" pr with family\n");
```

```
                return false;
            }
        }
        if (s.persons[person_t::prisoner].pos == person_t::onboard && s.boat.passengers<2) {
//          log(" pr on boat\n");
            return false;
        }
    }
    if ((s.persons[person_t::daughter1].pos == s.persons[person_t::father].pos) &&
        (s.persons[person_t::daughter1].pos != s.persons[person_t::mother].pos)) {
//      log(" d1 with f\n");
        return false;
    } else if ((s.persons[person_t::daughter2].pos == s.persons[person_t::father].pos) &&
               (s.persons[person_t::daughter2].pos != s.persons[person_t::mother].pos)) {
//      log(" d2 with f\n");
        return false;
    } else if ((s.persons[person_t::son1].pos == s.persons[person_t::mother].pos) &&
               (s.persons[person_t::son1].pos != s.persons[person_t::father].pos)) {
//      log(" s1 with m\n");
        return false;
    } else if ((s.persons[person_t::son2].pos == s.persons[person_t::mother].pos) &&
               (s.persons[person_t::son2].pos != s.persons[person_t::father].pos)) {
//      log(" s2 with m\n");
        return false;
    }
//  log(" OK\n");
    return true;
}

struct cost_t {
    size_t depth{0}; // counts the number of transitions
    size_t noise{0}; // kids get bored on shore1 and start making noise there
    bool operator<(const cost_t& other) const {
        if (depth < other.depth)
            return true;
        if (other.depth < depth)
            return false;
        return noise < other.noise;
    }
};

bool goal(const state_t& s){
    return std::all_of(std::begin(s.persons), std::end(s.persons),
                       [](const person_t& p) { return p.pos == person_t::shore2; });
}


template <typename CostFn>
void solve(CostFn&& cost) { // no type checking: OK hack here, but not good for a library.
    // Overall there are 4*3*2*1/2 solutions to the puzzle
    // (children form 2 symmetric groups and thus result in 2 out of 4 permutations).
    // However the search algorithm may collapse symmetric solutions, thus only one is reported.
    // By changing the cost function we can express a preference and
    // then the algorithm should report different solutions
    auto states = state_space_t<state_t, cost_t>{
        state_t{}, cost_t{},              // initial state and cost
        successors<state_t>(transitions), // successor generator
        &river_crossing_valid,            // invariant over states
        std::forward<CostFn>(cost)};      // cost over states
    auto solutions = states.check(&goal);
    if (solutions.empty()) {
```

```
        std::cout << "No solution\n";
    } else {
        // Introduced a change in print to adhere to my solution
        std::cout << "Solution:\n";
        std::cout << "Boat,     Mothr,Fathr,Daug1,Daug2,Son1, Son2, Polic,Prisn\n";
        for (auto&& trace: solutions) {
            std::cout << trace << '\n';
        }
    }
}

int main() {
    std::cout << "-- Solve using depth as a cost: ---\n";
    solve([](const state_t& state, const cost_t& prev_cost){
              return cost_t{ prev_cost.depth+1, prev_cost.noise };
          }); // it is likely that daughters will get to shore2 first
    std::cout << "-- Solve using noise as a cost: ---\n";
    solve([](const state_t& state, const cost_t& prev_cost){
              auto noise = prev_cost.noise;
              if (state.persons[person_t::son1].pos == person_t::shore1)
                  noise += 2; // older son is more noughty, prefer him first
              if (state.persons[person_t::son2].pos == person_t::shore1)
                  noise += 1;
              return cost_t{ prev_cost.depth, noise };
          }); // son1 should get to shore2 first
    std::cout << "-- Solve using different noise as a cost: ---\n";
    solve([](const state_t& state, const cost_t& prev_cost){
              auto noise = prev_cost.noise;
              if (state.persons[person_t::son1].pos == person_t::shore1)
                  noise += 1;
              if (state.persons[person_t::son2].pos == person_t::shore1)
                  noise += 2; // younger son is more distressed, prefer him first
              return cost_t{ prev_cost.depth, noise };
          }); // son2 should get to the shore2 first
}
/** Example solutions (shows only the states with travel):
--- Solve using depth as a cost: ---
Boat,     Mothr,Fathr,Daug1,Daug2,Son1, Son2, Polic,Prisn
{trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{~~~}
{trv,1,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{SH2}
{trv,2,2},{sh1},{sh1},{~~~},{sh1},{sh1},{sh1},{~~~},{SH2}
{trv,2,2},{sh1},{sh1},{SH2},{sh1},{sh1},{sh1},{~~~},{~~~}
{trv,2,2},{~~~},{sh1},{SH2},{~~~},{sh1},{sh1},{sh1},{sh1}
{trv,1,2},{~~~},{sh1},{SH2},{SH2},{sh1},{sh1},{sh1},{sh1}
{trv,2,2},{~~~},{~~~},{SH2},{SH2},{sh1},{sh1},{sh1},{sh1}
{trv,1,2},{SH2},{~~~},{SH2},{SH2},{sh1},{sh1},{sh1},{sh1}
{trv,2,2},{SH2},{sh1},{SH2},{SH2},{sh1},{sh1},{~~~},{~~~}
{trv,1,2},{~~~},{sh1},{SH2},{SH2},{sh1},{sh1},{SH2},{SH2}
{trv,2,2},{~~~},{~~~},{SH2},{SH2},{sh1},{sh1},{SH2},{SH2}
{trv,1,2},{SH2},{~~~},{SH2},{SH2},{sh1},{sh1},{SH2},{SH2}
{trv,2,2},{SH2},{~~~},{SH2},{SH2},{~~~},{sh1},{SH2},{SH2}
{trv,2,2},{SH2},{SH2},{SH2},{SH2},{SH2},{sh1},{~~~},{~~~}
{trv,2,2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~~},{~~~},{sh1}
{trv,1,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~~},{sh1}
{trv,2,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~~},{~~~}
--- Solve using noise as a cost: ---
Boat,     Mothr,Fathr,Daug1,Daug2,Son1, Son2, Polic,Prisn
{trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{~~~}
{trv,1,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{SH2}
{trv,2,2},{sh1},{sh1},{sh1},{sh1},{~~~},{sh1},{~~~},{SH2}
{trv,2,2},{sh1},{sh1},{sh1},{sh1},{SH2},{sh1},{~~~},{~~~}
```

```
{trv,2,2},{sh1},{~~~},{sh1},{sh1},{SH2},{~~~},{sh1},{sh1}
{trv,1,2},{sh1},{~~~},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
{trv,2,2},{~~~},{~~~},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
{trv,1,2},{~~~},{SH2},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
{trv,2,2},{sh1},{SH2},{sh1},{sh1},{SH2},{SH2},{~~~},{~~~}
{trv,1,2},{sh1},{~~~},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
{trv,2,2},{~~~},{~~~},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
{trv,1,2},{~~~},{SH2},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
{trv,2,2},{~~~},{SH2},{~~~},{sh1},{SH2},{SH2},{SH2},{SH2}
{trv,2,2},{SH2},{SH2},{SH2},{sh1},{SH2},{SH2},{~~~},{~~~}
{trv,2,2},{SH2},{SH2},{SH2},{~~~},{SH2},{SH2},{~~~},{sh1}
{trv,1,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~~},{sh1}
{trv,2,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~~},{~~~}
-- Solve using different noise as a cost: ---
Boat,     Mothr,Fathr,Daug1,Daug2,Son1, Son2, Polic,Prisn
{trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{~~~}
{trv,1,2},{sh1},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{SH2}
{trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{~~~},{~~~},{SH2}
{trv,2,2},{sh1},{sh1},{sh1},{sh1},{sh1},{SH2},{~~~},{~~~}
{trv,2,2},{sh1},{~~~},{sh1},{sh1},{~~~},{SH2},{sh1},{sh1}
{trv,1,2},{sh1},{~~~},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
{trv,2,2},{~~~},{~~~},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
{trv,1,2},{~~~},{SH2},{sh1},{sh1},{SH2},{SH2},{sh1},{sh1}
{trv,2,2},{sh1},{SH2},{sh1},{sh1},{SH2},{SH2},{~~~},{~~~}
{trv,1,2},{sh1},{~~~},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
{trv,2,2},{~~~},{~~~},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
{trv,1,2},{~~~},{SH2},{sh1},{sh1},{SH2},{SH2},{SH2},{SH2}
{trv,2,2},{~~~},{SH2},{~~~},{sh1},{SH2},{SH2},{SH2},{SH2}
{trv,2,2},{SH2},{SH2},{SH2},{sh1},{SH2},{SH2},{~~~},{~~~}
{trv,2,2},{SH2},{SH2},{SH2},{~~~},{SH2},{SH2},{~~~},{sh1}
{trv,1,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~~},{sh1}
{trv,2,2},{SH2},{SH2},{SH2},{SH2},{SH2},{SH2},{~~~},{~~~}
 */
```