AP Puzzle Mini Project

Christian Møllnitz

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_C_COMPILER "gcc")
set(CMAKE_CXX_COMPILER "g++")

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

```
   #ifndef REACHABILITY

#define REACHABILITY

#define REACHABILITY_DEBUG false

#include <ostream>
#include <list>
#include <functional>
#include <iostream>
#include <string>
#include <utility>
#include <memory> // This is memory_resource in windows.
#include <iterator>
#include <initializer_list>
#include <set>
#include <queue>
#include <type_traits>
#include <algorithm>

using namespace std;

//Function was missing in code provided, so I made my own. - It's kinda lame because it only takes a string.
/*Paints out a string, ends the line*/
void log( string input) {
#if REACHABILITY_DEBUG
    cout << input << endl;
#endif // REACHBILITYDEBUG
};

```

```

/* Part of: 5. Support various search orders: breadth-first search, depth-first search (the leaping frogs have different solutions). */
/*Details the order, depth_first or breadth_first, notice that this library uses breadth_first as its default order */
enum class search_order_t
{
    depth_first,
    breadth_first
};

/*
Performs a search with a priority queue.
state: The initial state.
cost: The initial cost.
cost_generator: A function that generates the next cost, using the previous cost, and previous state.
successor_generator: A function that generates all possible successor states to the current state.
valid_state_func: A function that checks if a state is a valid state.
accept_state_func A function that checks if a state is an acceptable state.
comparison_func: the comparison function used by the priority queue itself, bfs or other costs = std::greater && dfs = std::less
*/
template <class State, class Cost>
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)
{
    using pair_type = pair<Cost, State>;
    using comp_type = function<bool(const pair_type&, const pair_type&)>;

    // a set that holds all seen states up to this point.
    set<State> seen_set{};

    auto result = list<list<shared_ptr<State>>>{};
    unordered_map<State, State> child_to_parent_map = unordered_map<State, State>{};

    //Priority maps are sorted by the first element in a pair structure, that is why it’s Pair<cost,state>.
    auto pqs_queue = priority_queue<pair_type, vector<pair_type>, comp_type> {comparison_func};
    //Emplace smartly picks constructor.
    pqs_queue.emplace(cost, state);

    //Infinite loop protection
    seen_set.insert(state);

    while (!pqs_queue.empty())
    {
        pair_type parent_item = pqs_queue.top();
        pqs_queue.pop();

        if (accept_state_func(parent_item.second))
        {
            result.push_back(map_ancestry_shared(child_to_parent_map, parent_item.second));
            log("Added a result");
        }
        else if (valid_state_func(parent_item.second)) {
            auto successors = successor_generator(parent_item.second);
            for (auto&& child_item : successors)
            {
                //Keeps track of seen items, avoid infinite loops
                if (seen_set.count(child_item) == 0) //Set is faster than vector (lg(n))
                {
                    seen_set.insert(child_item);
                    child_to_parent_map[child_item] = parent_item.second;
                    pqs_queue.push(make_pair(cost_generator(parent_item.second, parent_item.first), child_item));
                }
            }
        }
    }
    return result; //RVO - ITEM 25
}

/*Gets a route from the acceptable state child (or any child really) though the chain of parents, to the root node*/
template <class State>
list<shared_ptr<State>> map_ancestry_shared(unordered_map<State, State> c_to_p_map, State child) {
    auto ret = list<shared_ptr<State>>{};
    ret.push_back(make_shared<State>(child));
    // https://en.cppreference.com/w/cpp/container/unordered_map/find - if it doesn’t find the element, it returns end iterator.
    auto c_to_p_pair = c_to_p_map.find(child);
    while (c_to_p_pair != c_to_p_map.end())
    {
        ret.push_back(make_shared<State>(c_to_p_pair->second));
        c_to_p_pair = c_to_p_map.find(c_to_p_pair->second);
    }
    return ret;
}




//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.
/*Mimics ‘cost_t‘ struct from family.cpp, and lets this facimily make BFS and DFS search for any type*/

/*Default cost structure, simply annotates costs in the conventional BFS / DFS manner, as nodes are discovered, chronologically*/
struct default_cost_t {
    int id;
    static int total;
    default_cost_t()
    {
        id = total++;
    }

    bool operator<(const default_cost_t& other) const {
        return id < other.id;
    }
};

//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.
/*Cheap cost_gen, that merely annotates costs in the conventional BFS and DFS manner, chronologically*/
```

```cpp
template <class State, class default_cost_t>
default_cost_t cost_gen(const State& prev_state, const default_cost_t& prev_cost) {
    return default_cost_t{}; //Use default constructor.
}

int default_cost_t::total = 0;


//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.
/*Cheap way to ensure all states are valid*/
template <class State>
bool return_true(const State& state) {
    return true;
}


template <class State, class Cost = default_cost_t, class CostFn = function<Cost(const State&, const Cost&)>>
class state_space_t
{
public:

    state_space_t() = delete; //No default constructor - library should not be used like this.
    ~state_space_t() = default; //Default deleter - rule of zero.


    /* 3. Find a state satisfying the goal predicate when given the initial state and successor generating function. */
    state_space_t(State state, function<vector<State>(State&)> successor_generator) : state_space_t(state, successor_generator, return_true<State>) { } ;

    /* 6. Support a given invariant predicate (state validation function, like in river crossing puzzles). */
    /* 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. */
    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>) {
        default_cost_t::total = 0; // reset node index inbetween runs. - only needed for this constructor, and the one above.
    };

    /* 7. Support custom cost function over states (like noise in Japanese river crossing puzzle).  */
    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) {
        //Generic type static asserts.

        /* 10. User friendly to use and fail with a message if the user supplies arguments of wrong types. */
        static_assert(is_class<State>::value, "Template argument state must be struct (or class)");
        static_assert(is_class<Cost>::value, "Template argument init_cost must be struct (or class)");
        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&)>>");

        //Assignments - note the cost_generator assignment is above to avoid constructing a lambda.
        this->first_state = state;
        this->successor_generator = succ_gen;
        this->valid_state_func = val_gen;
        this->first_cost = init_cost;
    };

    //Check with sensible default = breadth first search.
    //http://mishadoff.com/images/dfs/binary_tree_search.png - visual aid
    /* 5. Support various search orders: breadth-first search, depth-first search (the leaping frogs have different solutions) */
    auto check(function<bool(const State&)>  accept_func, search_order_t order = search_order_t::breadth_first) {
        list<list<shared_ptr<State>>> search; //Result
        if (order == search_order_t::breadth_first)
        {
            search = priority_queue_search<State, Cost>(first_state, first_cost, successor_generator, cost_generator, valid_state_func, accept_func, greater<pair<Cost, State>>());
        }
        else
        {
            search = priority_queue_search<State, Cost>(first_state, first_cost, successor_generator, cost_generator, valid_state_func, accept_func, less<pair<Cost, State>>());
        }
        //No need to use move here due to RVO
        return search;
    }

private:

    State first_state;
    Cost first_cost;

    //This could say ’ function<Cost (const State&, const Cost&)>’ as type.
    CostFn cost_generator;

    function<vector<State>(State&)> successor_generator;
    function<bool(const State&)> valid_state_func;

};


//1. Extend  hash function for an arbitrary (iterable) container of basic types.
namespace std {

    /* FULL DISCLOSURE, THIS WAS WRITTEN BY Marius Mikucionis http://people.cs.aau.dk/~marius/Teaching/AP2019/lecture9.html#/7/4 */
    template <typename... Ts>
    using void_t = void; // eats all valid types
    template <typename T, typename = void> // primary declaration
    struct is_container: std::false_type {};  // computes false
    template <typename C>  // specialization
    struct is_container<C, // type c examination
            void_t< // the following must evaluate to types:
                    typename C::iterator,          // check subtype
                    typename C::const_iterator,    // check subtype
                    //is_array<C>,
                    decltype(std::begin(std::declval<C&>())), // check iteration:
                    decltype(std::end(std::declval<C&>()))    // create C() and call
            > // finished checks
    > : std::true_type {}; // computes "true"
    template <typename C>  // *_t type alias
    using is_container_t = typename is_container<C>::type;
    template <typename C>  // *_v value
    constexpr auto is_container_v = is_container<C>::value;

    template<template<typename, typename...> class Cont, typename T, typename... N>
    struct hash<Cont<T, N...>> {
        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>
        operator()(const Cont<T, N...> &container) const {

            hash<T> hashT;
            auto res = 0;
            for (auto&& val : container)
            {
                res = (res << 1) ^ hashT(val);
            }
            return res;
        }
    };

    template<class T, size_t U>
    struct hash<array<T,U>> {
        auto operator() (const array<T,U>& data) const {
            hash<T> hashT;
            size_t res = 0;
            for (auto&& val : data) // I love these loops
            {
                res = (res << 1) ^ hashT(val);
            }
            return res;
        }
    };

    /* // Use this to compile with Clang / MSVS
    template<typename T>
    struct hash <vector<T>> {
        auto operator() (const vector<T>& data) const {
            hash<T> hashT;
            size_t res = 0;
            for (auto&& val : data)
            {
                res = (res << 1) ^ hashT(val);
            }
            return res;
        }
    }; */
};


/* 4. Print the trace of a state sequence from the initial state to the found goal state. */
template <class State>
ostream& operator <<(ostream& os, const list<shared_ptr<State>>& lst)
{
    int i = 0;
    //Print backwards - avoids having to shuffle list.
    for (auto end = lst.rbegin(), front = lst.rend(); end != front; end++, i++)
    {
        os << i << " : " << **end <<  endl;
    }
#if REACHABILITY_DEBUG
    cin >> i; //Pause here because it's nice to see results - a real library obviously shouldn't
```

```
do this.
#endif // REACHABILITY_DEBUG
return os;
}

/*
2. Create a generic successor generator function out of a transition generator function.
    A transition generator function generates functions that change a state.
    Each such function corresponds to a transition.
    A successor generator function gets a state and generates a set of its successor states.
*/
//Would have used auto here, but it narrowed it out and gave me 60 compiler errors, no fun.

/*Turns a transition generator function (transformer) into a successor generator function*/
template <class State>
function<vector<State>(State&)> successors(function<list<function<void(State&)>>(const State&)> transformer)
{
    return [transformer](const State& data){
        vector<State> res{};
        for (auto&& func: transformer(data)) //For each function in the resulting list.
        {
            State copy = data;
            func(copy);
            res.push_back(copy);
        }
        return res;
    };
}
#endif // !REACHABILITY
```

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 }};
```

```cpp
    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';
    auto space = state_space_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';
    auto space = state_space_t(std::move(start), successors<stones_t>(transitions));
    auto solutions = space.check(
        [finish=std::move(finish)](const stones_t& state){ return state==finish; },
        order);
    for (auto&& trace: solutions) {
        std::cout << "Solution: trace of " << trace.size() << " states\n";
        std::cout << trace;
    }
}

int main(){
    explain();
    std::cout << "--- Solve with depth-first search: ---\n";
    solve(2, search_order_t::depth_first);
    solve(4); // 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: family.cpp

```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> {
```

```cpp
        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{},              // 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 {
        for (auto&& trace: solutions) {
            std::cout << "Solution:\n";
            std::cout << "Boat,     Mothr,Fathr,Daug1,Daug2,Son1, Son2, Polic,Prisn\n";
            for (auto&& state: trace)
                std::cout << *state << '\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},{~~~},{~~~}
 */
```

Listing 5: 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{},                        // 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();
        });
    for (auto&& trace: solution)
        std::cout << "#  CGW\n" << trace;
}

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
*/
```