SP Exam Project

Mathias Andresen

May 12, 2021

Listing 1: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.17)
   project(sp_exam_project)
   set(CMAKE_CXX_STANDARD 20)
   add_library(
       stochastic-simulation
       library/simulation.cpp
       library/simulation.cpp
       library/SymbolTable.h
10
       library/simulation\_monitor.h
       library/data.h
       library/data.cpp
13
       library/my-thread-pool.h
14
   )
15
   add_executable(sp_exam_project main.cpp vessels.h)
17
   target_link_libraries(sp_exam_project PRIVATE stochastic-simulation)
```

Listing 2: main.cpp

```
#include <iostream>
   #include "library/simulation.h"
   #include <chrono>
   #include "vessels.h"
   using namespace StochasticSimulation;
   class hospitalized_monitor: public simulation_monitor {
       double_t hospitalized_acc{0.0};
10
       double_t last_time{0.0};
11
       size_t max_hospitalized{0};
13
       void monitor(SimulationState &state) override {
           auto currently_hospitalized = state.reactants.get("H").amount;
           if (currently_hospitalized > max_hospitalized) {
               max_hospitalized = currently_hospitalized;
           }
           hospitalized_acc += (currently_hospitalized * (state.time - last_time));
           last_time = state.time;
25
26
       double_t get_mean_hospitalized() const {
```

```
return (hospitalized_acc / last_time);
       }
   };
30
31
   void simulate_covid() {
32
       std::cout << "Simulating covid19 example with hospitalized monitor" << std::endl;</pre>
33
       Vessel covid_vessel = seihr(10000);
34
       std::cout << covid_vessel << std::endl;</pre>
        covid_vessel.visualize_reactions("covid_graph.png");
38
       std::cout << "reaction graph can be seen at: covid_graph.png" << std::endl;
39
       hospitalized_monitor monitor{};
41
42
       auto trajectory = covid_vessel.do_simulation(120, monitor);
       std::cout << "Simulation done" << std::endl;</pre>
45
       std::cout << "Max hospitalized: " << monitor.max_hospitalized << std::endl;</pre>
46
       std::cout << "Mean hospitalized: " << monitor.get_mean_hospitalized() << std::endl;</pre>
47
       std::cout << "Writing trajectory to csv file at covid_output.csv" << std::endl;</pre>
49
       trajectory->write_csv("covid_output.csv");
50
       std::cout << "Turn it into a graph using python ./draw_graph.py covid release" << std::endl;
51
52
   }
53
   void simulate_covid_multiple() {
54
       std::cout << "Simulating covid19 example 30 times and calculating mean" << std::endl;</pre>
       Vessel covid_vessel = seihr(10000);
56
57
       auto trajectories = covid_vessel.do_multiple_simulations(110, 100);
       std::cout << "Simulations done" << std::endl << "Computing mean trajectory" << std::endl;</pre>
60
61
       auto mean = SimulationTrajectory::compute_mean_trajectory(trajectories);
62
       std::cout << "Writing mean trajectory to csv file at covid_output_multiple.csv" << std::endl;</pre>
64
       mean.write_csv("covid_output_multiple.csv");
       std::cout << "Turn it into a graph using python ./draw_graph.py covid
 ⇔covid_output_multiple.csv" << std::endl;
67
68
   void simulate_introduction() {
69
       std::cout << "Simulating introduction example" << std::endl;</pre>
       Vessel introduction_vessel = introduction(25, 50, 1, 0.001);
71
       std::cout << introduction_vessel << std::endl:</pre>
       introduction_vessel.visualize_reactions("intro_graph.png");
75
       auto trajectory = introduction_vessel.do_simulation(400);
76
       trajectory->write_csv("intro_output.csv");
78
   }
79
80
   void simulate_circadian() {
82
       std::cout << "Simulating circadian rhythm example..." << std::endl;</pre>
83
       Vessel oscillator = circadian_oscillator():
84
       std::cout << oscillator << std::endl;</pre>
       oscillator.visualize_reactions("cir_graph.png");
```

```
auto trajectory = oscillator.do_simulation(110);
90
        std::cout << "Writing csv file..." << std::endl;</pre>
91
        trajectory->write_csv("circadian_output.csv");
92
    }
93
94
    void simulate_circadian2() {
95
        std::cout << "Simulating circadian rhythm alternative example..." << std::endl;</pre>
        Vessel oscillator = circadian_oscillator2();
97
98
        std::cout << oscillator << std::endl;</pre>
99
        oscillator.visualize_reactions("cir2_graph.png");
100
101
        auto trajectory = oscillator.do_simulation(110);
102
        std::cout << "Writing csv file..." << std::endl;</pre>
104
        trajectory->write_csv("circadian2_output.csv");
105
    }
106
107
    void benchmark() {
108
        std::cout << "Benchmarking with circadian rhythm example (max_time=100)" << std::endl;</pre>
109
110
        auto runs{30};
        Vessel oscillator = circadian_oscillator();
113
114
        unsigned long time_acc1{0};
115
         for (int i = 0; i < runs; ++i) {
116
             auto t0 = std::chrono::high_resolution_clock::now();
117
             oscillator.do_simulation(100);
             auto t1 = std::chrono::high_resolution_clock::now();
120
             time_acc1 += std::chrono::duration_cast<std::chrono::nanoseconds>(t1-t0).count();
121
122
        }
        auto mean_time1 = time_acc1 / runs;
        std::cout << "Simulation 1 mean time (nanoseconds): " << mean_time1 << std::endl;</pre>
124
125
        unsigned long time_acc2{0};
126
        for (int i = 0; i < runs; ++i) {</pre>
             auto t0 = std::chrono::high_resolution_clock::now();
128
             oscillator.do_simulation2(100);
129
             auto t1 = std::chrono::high_resolution_clock::now();
130
131
             time_acc2 += std::chrono::duration_cast<std::chrono::nanoseconds>(t1-t0).count();
132
        }
133
        auto mean_time2 = time_acc2 / runs;
        std::cout << "Simulation 2 mean time (nanoseconds): " << mean_time2 << std::endl;</pre>
135
    }
136
137
    int main() {
138
        simulate_covid();
139
          simulate_covid_multiple();
    //
140
141
          simulate_introduction();
    //
          simulate_circadian();
143
    //
          simulate_circadian2();
144
145
          benchmark();
146
   //
    }
147
```

```
//
   // Created by Mathias on 12-05-2021.
   #ifndef SP_EXAM_PROJECT_VESSELS_H
   #define SP_EXAM_PROJECT_VESSELS_H
   #include "library/simulation.h"
   using namespace StochasticSimulation;
10
11
   Vessel seihr(uint32_t N)
12
   {
13
       auto v = Vessel{};
14
       const auto eps = 0.0009; // initial fraction of infectious
15
       const auto I0 = size_t(std::round(eps*N)); // initial infectious
16
       const auto E0 = size_t(std::round(eps*N*15)); // initial exposed
       const auto S0 = N-I0-E0; // initial susceptible
18
       const auto R0 = 2.4; // basic reproductive number (initial, without lockdown etc)
19
       const auto alpha = 1.0 / 5.1; // incubation rate (E -> I) ~5.1 days
20
       const auto gamma = 1.0 / 3.1; // recovery rate (I \rightarrow R) ~3.1 days
21
       const auto beta = R0 * gamma; // infection/generation rate (S+I -> E+I)
22
       const auto P_H = 0.9e-3; // probability of hospitalization
23
       const auto kappa = gamma * P_H*(1.0-P_H); // hospitalization rate (I -> H)
       const auto tau = 1.0/10.12; // recovery/death rate in hospital (H -> R) ~10.12 days
26
       // Reactants
27
       auto S = v("S", S0); // susceptible
28
       auto E = v("E", E0); // exposed
       auto I = v("I", I0); // infectious
30
       auto H = v("H", 0); // hospitalized
31
       auto R = v("R", 0); // removed/immune (recovered + dead)
33
       // Reactions
34
       v(S >= E, I, beta/N);
35
       v(E >>= I, alpha);
       v(I >>= R, gamma);
37
       v(I >>= H, kappa);
38
       v(H >>= R, tau);
39
       return v;
41
   }
42
43
   Vessel introduction(uint32_t A_start, uint32_t B_Start, uint32_t D_amount, double_t lambda) {
       auto v = Vessel{};
45
       // Reactants
46
       auto A = v("A", A_start);
       auto B = v("B", B_Start);
       auto C = v("C", 0);
49
       auto D = v("D", D_amount);
50
       // Reactions
51
       v(A + B * 2 >>= C, D, lambda);
53
       return v;
54
   }
56
   /** direct encoding */
57
   Vessel circadian_oscillator()
58
   {
59
       auto alphaA = 50.0;
```

```
auto alpha_A = 500.0;
        auto alphaR = 0.01;
        auto alpha_R = 50.0;
63
        auto betaA = 50.0:
64
        auto betaR = 5.0;
65
        auto gammaA = 1.0;
66
        auto gammaR = 1.0;
67
        auto gammaC = 2.0;
        auto deltaA = 1.0;
        auto deltaR = 0.2;
70
        auto deltaMA = 10.0;
71
        auto deltaMR = 0.5;
72
        auto thetaA = 50.0;
        auto thetaR = 100.0;
74
        auto v = Vessel{};
75
        auto env = v.environment();
76
        auto DA = v("DA", 1);
77
        auto D_A = v("D_A", 0);
78
        auto DR = v("DR", 1);
79
        auto D_R = v("D_R", 0);
80
        auto MA = v("MA", 0);
        auto MR = v("MR", 0);
82
        auto A = v("A", 0);
83
        auto R = v("R", 0);
        auto C = v("C", 0);
        v(A + DA >>= D_A, gammaA);
86
        v(D_A >>= DA + A, thetaA);
87
        v(A + DR >>= D_R, gammaR);
        v(D_R >>= DR + A, thetaR);
89
        v(D_A \gg MA + D_A, alpha_A);
90
        v(DA >>= MA + DA, alphaA);
91
        v(D_R \gg MR + D_R, alpha_R);
        v(DR >>= MR + DR, alphaR);
93
        v(MA >>= MA + A, betaA);
94
        v(MR >>= MR + R, betaR);
95
        v(A + R >>= C, gammaC);
        v(C >>= R, deltaA);
97
        v(A >>= env, deltaA);
98
        v(R >>= env, deltaR);
99
        v(MA >>= env, deltaMA);
        v(MR >>= env, deltaMR);
101
        return v;
102
    }
103
104
    /** alternative encoding using catalysts */
105
    Vessel circadian_oscillator2()
106
107
        auto alphaA = 50.0;
108
        auto alpha_A = 500.0;
109
        auto alphaR = 0.01;
110
        auto alpha_R = 50.0;
111
        auto betaA = 50.0;
112
        auto betaR = 5.0;
113
        auto gammaA = 1.0;
114
        auto gammaR = 1.0;
        auto gammaC = 2.0;
116
        auto deltaA = 1.0;
117
        auto deltaR = 0.2;
118
        auto deltaMA = 10.0;
119
        auto deltaMR = 0.5;
120
        auto thetaA = 50.0;
121
```

```
auto thetaR = 100.0;
122
        auto v = Vessel{};
        auto env = v.environment();
124
        auto DA = v("DA", 1);
125
        auto D_A = v("D_A", 0);
126
        auto DR = v("DR", 1);
127
        auto D_R = v("D_R", 0);
128
        auto MA = v("MA", 0);
129
        auto MR = v("MR", 0);
130
        auto A = v("A", 0);
131
        auto R = v("R", 0);
132
        auto C = v("C", 0);
133
        v(A + DA >>= D_A, gammaA);
134
        v(D_A >>= DA + A, thetaA);
135
        v(DR + A >>= D_R, gammaR);
136
        v(D_R >>= DR + A, thetaR);
137
        v(env >>= MA, D_A, alpha_A);
138
        v(env >>= MA, DA, alphaA);
139
        v(env >>= MR, D_R, alpha_R);
140
        v(env >>= MR, DR, alphaR);
141
        v(env >>= A, MA, betaA);
        v(env >>= R, MR, betaR);
143
        v(A + R >>= C, gammaC);
144
        v(C >>= R, deltaA);
145
        v(A >>= env, deltaA);
146
        v(R >>= env, deltaR);
147
        v(MA >>= env, deltaMA);
148
        v(MR >>= env, deltaMR);
149
        return v;
150
    }
151
152
    #endif //SP_EXAM_PROJECT_VESSELS_H
```

Listing 4: simulation.h

```
// Created by Mathias on 09-05-2021.
   #ifndef SP_EXAM_PROJECT_SIMULATION_H
   #define SP_EXAM_PROJECT_SIMULATION_H
   #include <string>
   #include <utility>
   #include <vector>
   #include <set>
   #include <optional>
   #include <map>
   #include <numeric>
   #include <random>
   #include <algorithm>
   #include <functional>
   #include <sstream>
   #include <fstream>
   #include <chrono>
   #include <thread>
   #include <future>
   #include <ranges>
   #include "SymbolTable.h"
   #include "simulation_monitor.h"
   #include "data.h"
26
27
```

```
namespace StochasticSimulation {
       using map_type = std::map<double_t, SimulationState>;
30
       class SimulationTrajectory: public map_type {
31
       private:
32
           double_t largest_time{-1};
33
            static double_t compute_interpolated_value(
34
                    const std::string& key,
                    SimulationState& s0,
                    SimulationState& s1,
37
                    double_t x);
38
       public:
39
           using map_type::map;
41
           SimulationTrajectory(const SimulationTrajectory& val): map_type(val) {
42
                largest_time = val.largest_time;
           }
45
           SimulationTrajectory(SimulationTrajectory&& rval): map_type(std::move(rval)) {
46
47
                largest_time = std::move(rval.largest_time);
           };
49
           SimulationTrajectory& operator=(const SimulationTrajectory & val) {
                map_type::operator=(val);
                largest_time = val.largest_time;
           };
53
54
           SimulationTrajectory& operator=(SimulationTrajectory&& rval) {
                map_type::operator=(std::move(rval));
56
                largest_time = std::move(rval.largest_time);
57
           };
            static SimulationTrajectory
 →compute_mean_trajectory(std::vector<std::shared_ptr<SimulationTrajectory>>& trajectories);
61
           void insert(SimulationState state) {
                if (state.time > largest_time) {
63
                    largest_time = state.time;
                }
                map_type::insert({state.time, std::move(state)});
           }
68
69
           void write_csv(const std::string& path);
           double_t get_max_time() {
72
                return largest_time;
       };
75
76
       class Vessel {
       private:
           std::vector<Reaction> reactions{};
           SymbolTable<Reactant> reactants;
       public:
82
           Vessel() = default;
83
84
           Vessel(const Vessel &val) {
                reactions = val.reactions;
86
                reactants = val.reactants;
```

```
}
            Vessel (Vessel&& rval) {
90
                reactions = std::move(rval.reactions);
91
                reactants = std::move(rval.reactants);
            };
            Reactant& operator()(std::string name, size_t initial_amount) {
                Reactant newReactant{std::move(name), initial_amount};
                reactants.put(newReactant.name, newReactant);
98
99
                return reactants.get(newReactant.name);
100
            }
101
102
            Reaction operator()(Reaction& reaction, double_t rate) {
                 reaction.rate = rate;
104
105
                reactions.push_back(reaction);
106
107
                return reaction;
            }
109
110
            Reaction operator()(Reaction&& reaction, std::initializer_list<Reactant> catalysts,
  →double rate) {
                reaction.rate = rate;
112
                reaction.catalysts = catalysts;
113
                // Add to vessel reactions
115
                reactions.push_back(reaction);
116
                 return reaction;
            }
119
120
            Reaction operator()(Reaction& reaction, Reactant catalyst, double_t rate) {
121
                reaction.rate = rate;
                reaction.catalysts = {catalyst};
123
124
                // Add to vessel reactions
                reactions.push_back(reaction);
127
                return reaction;
128
            }
129
130
131
            Reactant& environment() {
132
                if (reactants.contains("__env__")) {
                    return reactants.get("__env__");
134
135
136
                auto newReactant = Reactant("__env__", 0, 0);
138
                reactants.put(newReactant.name, newReactant);
139
                 return reactants.get(newReactant.name);
            }
142
143
            void visualize_reactions(const std::string& filename);
144
145
            std::shared_ptr<SimulationTrajectory> do_simulation2(double_t end_time,
  →simulation_monitor& monitor = EMPTY_SIMULATION_MONITOR);
```

```
147
            std::shared_ptr<SimulationTrajectory> do_simulation(double_t end_time,
  →simulation_monitor& monitor = EMPTY_SIMULATION_MONITOR);
149
            std::vector<std::shared_ptr<SimulationTrajectory>> do_multiple_simulations(double_t 🗸
  →end_time, size_t simulations_to_run);
151
            friend std::ostream& operator<<(std::ostream& s, const Vessel& vessel);</pre>
152
153
        };
154
155
156
    }
157
158
    #endif //SP_EXAM_PROJECT_SIMULATION_H
159
```

Listing 5: simulation.cpp

```
11
   // Created by Mathias on 09-05-2021.
   #include <iostream>
   #include <utility>
   #include "simulation.h"
   namespace StochasticSimulation {
9
10
11
       std::ostream &operator<<(std::ostream &s, const Vessel &vessel) {</pre>
12
            s << "{" << std::endl;
            for (const auto& reaction: vessel.reactions) {
14
                s << "\t" << reaction;
15
                if (&reaction != &vessel.reactions.back()) {
                    s << ",";
18
                s << std::endl;
19
20
            }
            return s << "}";
       }
23
       void Vessel::visualize_reactions(const std::string& filename) {
            std::stringstream str;
            SymbolTable<std::string> node_map{};
26
27
            str << "digraph {" << std::endl;</pre>
28
            auto i = 0;
30
            for (auto& reactant: reactants.getMap()) {
31
                if (reactant.second.name != "__env__") {
                    node_map.put(reactant.second.name, "s" + std::to_string(i));
33
34
                    str << node_map.get(reactant.second.name)</pre>
35
                         << "[label=\"" << reactant.second.name <<</pre>
 \rightarrow"\",shape=\"box\",style=\"filled\",fillcolor=\"cyan\"];" << std::endl;
                    i++;
37
                }
            }
            i = 0;
41
            for (auto& reaction: reactions) {
42
                std::string reaction_node{"r" + std::to_string(i)};
```

```
44
                str << reaction_node << "[label=\"" << reaction.rate << ∠
 \rightarrow "\", shape=\"oval\", style=\"filled\", fillcolor=\"yellow\"];" << std::endl;
                if (reaction.catalysts.has_value()) {
46
                    for (auto& catalyst: reaction.catalysts.value()) {
47
                        str << node_map.get(catalyst.name) << " -> " << reaction_node << "</pre>
 →[arrowhead=\"tee\"];" << std::endl;</pre>
49
                    }
50
                }
                for (auto& reactant: reaction.from) {
51
                    if (reactant.name != "__env__") {
52
                        str << node_map.get(reactant.name) << " -> " << reaction_node << ";" <<</pre>
 →std::endl;
54
                }
55
                for (auto& product: reaction.to) {
56
                    if (product.name != "__env__") {
57
                         str << reaction_node << " -> " << node_map.get(product.name) << ";" <<
 →std::endl;
59
                    }
                }
61
                i++:
62
            }
            str << "}";
65
66
            std::ofstream dotfile;
            dotfile.open(filename + ".dot");
68
            dotfile << str.str();</pre>
69
            dotfile.close();
70
            std::stringstream command_builder;
72
            command_builder << "dot -Tpng -o " << filename << " " << filename << ".dot";</pre>
73
            system(command_builder.str().c_str());
74
       }
76
       std::shared_ptr<SimulationTrajectory> Vessel::do_simulation2(double_t end_time,
  ⇒simulation_monitor &monitor) {
            SimulationTrajectory trajectory{};
78
            double_t t{0};
79
80
            auto thread_id = std::this_thread::get_id();
81
            auto epoch = std::chrono::system_clock::now().time_since_epoch().count();
            std::default_random_engine engine(epoch * (std::hash<std::thread::id>{}(thread_id)));
83
            // Insert initial state
            trajectory.insert(SimulationState{reactants, t});
86
           while (t <= end_time) {</pre>
88
                for (Reaction& reaction: reactions) {
                    // New: using new compute delay function
                    reaction.compute_delay2(trajectory.at(t), engine);
                }
                auto r = reactions.front();
94
95
                // Select Reaction with min delay which is not -1
96
                for (auto& reaction: reactions) {
                    if (reaction.delay == -1) {
98
                        continue;
99
```

```
} else if (reaction.delay < r.delay) {</pre>
                         r = reaction;
101
                     } else if (r.delay == -1) {
102
                         r = reaction;
103
                     }
                 }
105
106
                 // Stop if we have no reactions to do, thus r.delay == -1
                 if (r.delay == -1) {
                     break;
109
                 }
110
111
                 auto& last_state = trajectory.at(t);
112
113
                 t += r.delay;
114
                 SimulationState state{last_state.reactants, t};
116
117
                 if (
118
                         std::all_of(r.from.begin(), r.from.end(), [&state](const Reactant& e){return
  →state.reactants.get(e.name).amount >= e.required;}) &&
120
                                  !r.catalysts.has_value() ||
121
                                  std::all_of(r.catalysts.value().begin(), r.catalysts.value().end(),
122
  →[&state](const Reactant& e){return state.reactants.get(e.name).amount >= e.required;})
                         )
123
                         ) {
124
                     for (auto& reactant: r.from) {
125
                         state.reactants.get(reactant.name).amount -= reactant.required;
126
127
                     for (auto& reactant: r.to) {
128
                         state.reactants.get(reactant.name).amount += reactant.required;
130
                 }
131
132
                 trajectory.insert(std::move(state));
133
134
                 monitor.monitor(trajectory.at(t));
135
            }
136
             return std::make_shared<SimulationTrajectory>(std::move(trajectory));
138
        }
139
140
        std::shared_ptr<SimulationTrajectory> Vessel::do_simulation(double_t end_time,
   →simulation_monitor &monitor) {
            SimulationTrajectory trajectory{};
142
            double_t t{0};
144
            auto thread_id = std::this_thread::get_id();
145
            auto epoch = std::chrono::system_clock::now().time_since_epoch().count();
146
            std::default_random_engine engine(epoch * (std::hash<std::thread::id>{}(thread_id)));
148
            // Insert initial state
149
            trajectory.insert(SimulationState{reactants, t});
            while (t <= end_time) {</pre>
152
                 for (Reaction& reaction: reactions) {
153
                     reaction.compute_delay(trajectory.at(t), engine);
154
                 }
156
                 auto r = reactions.front();
157
```

```
// Select Reaction with min delay which is not -1
159
                 for (auto& reaction: reactions) {
160
                     if (reaction.delay == -1) {
161
                         continue;
162
                     } else if (reaction.delay < r.delay) {</pre>
163
                         r = reaction;
164
                     } else if (r.delay == -1 && reaction.delay != -1) {
                         r = reaction;
                     }
167
                 }
168
169
                 // Stop if we have no reactions to do, thus r.delay == -1
                 if (r.delay == -1) {
171
                     break:
172
                 }
                 auto& last_state = trajectory.at(t);
175
176
177
                 t += r.delay;
                 SimulationState state{last_state.reactants, t};
179
180
                 if (
                         std::all_of(r.from.begin(), r.from.end(), [&state](const Reactant& e){return
   ⇒state.reactants.get(e.name).amount >= e.reguired;}) &&
183
                                  !r.catalysts.has_value() ||
184
                                  std::all_of(r.catalysts.value().begin(), r.catalysts.value().end(),
  →[&state](const Reactant& e){return state.reactants.get(e.name).amount >= e.required;})
                         )
186
                         ) {
187
                     for (auto& reactant: r.from) {
188
                         state.reactants.get(reactant.name).amount -= reactant.required;
189
190
                     }
                     for (auto& reactant: r.to) {
                         state.reactants.get(reactant.name).amount += reactant.required;
192
                     }
193
                 }
                 trajectory.insert(std::move(state));
196
197
                 monitor.monitor(trajectory.at(t));
198
            }
199
200
            return std::make_shared<SimulationTrajectory>(std::move(trajectory));
201
        }
203
        std::vector<std::shared_ptr<SimulationTrajectory>>
204
        Vessel::do_multiple_simulations(double_t end_time, size_t simulations_to_run) {
205
            std::vector<std::shared_ptr<SimulationTrajectory>> result{};
            result.reserve(simulations_to_run);
207
208
            auto cores = std::thread::hardware_concurrency();
209
            int jobs = std::min(simulations_to_run, (cores - 1));
210
            auto simulations_per_job = simulations_to_run / jobs;
211
212
            auto futures =
213
  →std::vector<std::future<std::vector<std::shared_ptr<SimulationTrajectory>>>>{};
214
            auto lambda = [&vessel = *this, &end_time](size_t to_run){
215
```

```
auto simulations = std::vector<std::shared_ptr<SimulationTrajectory>>{};
                 simulations.reserve(to_run);
218
                 auto new_vessel = Vessel(vessel);
219
220
                 for (int i = 0; i < to_run; ++i) {</pre>
221
                     simulations.push_back(new_vessel.do_simulation(end_time));
222
                 }
                 return simulations;
225
            };
226
227
            for (int i = 0; i < jobs; ++i) {
228
                 futures.push_back(std::async(std::launch::async, lambda, simulations_per_job));
229
230
            auto missing_simulations = simulations_to_run - (simulations_per_job * jobs);
            if (missing_simulations != 0) {
232
                 futures.push_back(std::async(std::launch::async, lambda, missing_simulations));
233
234
235
            for (auto& future: futures) {
236
                 auto future_result = future.get();
237
                 for (auto& res: future_result) {
238
                     result.push_back(std::move(res));
240
            }
241
242
            return result;
243
        }
244
245
        double_t SimulationTrajectory::compute_interpolated_value(const std::string& key,
  →SimulationState& s0, SimulationState& s1, double_t x) {
            return
^{247}
                 s0.reactants.get(key).amount
248
249
                 + ((
                        ( (double_t) s1.reactants.get(key).amount - (double_t)
   →s0.reactants.get(key).amount) /
                        ( s1.time - s0.time )
251
                    ) * (x - s0.time));
252
        }
254
        SimulationTrajectory
255
  →SimulationTrajectory::compute_mean_trajectory(std::vector<std::shared_ptr<SimulationTrajectory>>&
  →trajectories) {
            auto average_delay = trajectories.front()->get_max_time() / trajectories.front()->size();
256
257
            // Get a list of all keys
            std::vector<std::string> reactant_keys{};
259
            for (auto& reactant: trajectories.front()->begin()->second.reactants) {
260
                 reactant_keys.push_back(reactant.second.name);
261
            }
263
            // Find upper bound for mean trajectory
264
            double_t upper_bound{-1.0};
            for (auto& trajectory: trajectories) {
266
                 if (upper_bound == -1.0 || trajectory->get_max_time() < upper_bound) {</pre>
267
                     upper_bound = trajectory->get_max_time();
268
                 }
269
            }
270
271
            SimulationTrajectory mean_trajectory{};
272
```

```
for (auto& trajectory: trajectories) {
                 auto iterator = trajectory->begin();
275
                 SimulationState& s0 = iterator->second;
276
                 iterator++;
277
                 SimulationState& s1 = iterator->second;
278
279
                 double_t t{0};
                 while((t + average_delay) <= upper_bound) {</pre>
282
                      if (t >= s0.time) {
283
                          if (t <= s1.time) {</pre>
284
                              if (!mean_trajectory.contains(t)) {
                                   mean_trajectory.insert((SimulationState{{}, t}));
286
                              }
287
                               for (auto& key: reactant_keys) {
289
                                   auto interpolated_value =
290
  →SimulationTrajectory::compute_interpolated_value(key, s0, s1, t);
291
                                   auto& table = mean_trajectory.at(t).reactants;
292
293
                                   if (!table.contains(key)) {
294
                                       Reactant reactant{key, 0.0};
                                       table.put(key, reactant);
296
                                   }
297
298
                                   table.get(key).amount += interpolated_value;
                              }
300
301
                               t += average_delay;
302
                          } else {
                               s0 = s1;
304
                              iterator++;
305
306
                               if (iterator != trajectory->end()) {
                                   s1 = iterator->second;
308
                              } else {
309
                                   break;
310
                              }
                          }
312
                     }
313
                 }
314
             }
315
316
             for (double_t i = 0; (i + average_delay) <= upper_bound; i += average_delay) {</pre>
317
                 for (auto& key: reactant_keys) {
                      mean_trajectory.at(i).reactants.get(key).amount /= trajectories.size();
319
320
             }
321
322
             return std::move(mean_trajectory);
323
        }
324
325
        void SimulationTrajectory::write_csv(const std::string &path) {
             std::ofstream csv_file;
327
             csv_file.open(path);
328
329
             auto reactants = at(0).reactants;
330
331
             for (auto& reactant : reactants) {
332
```

```
csv_file << reactant.second.name << ",";</pre>
             }
334
             csv_file << "time" << std::endl;</pre>
335
336
             for (auto& state : *this) {
337
                  for (auto& reactant: reactants) {
338
                       csv_file << state.second.reactants.get(reactant.second.name).amount << ",";</pre>
339
                  csv_file << state.second.time << std::endl;</pre>
             }
342
343
             csv_file.close();
344
         }
    }
346
```

Listing 6: data.h

```
//
   // Created by Mathias on 11-05-2021.
   //
   #ifndef SP_EXAM_PROJECT_DATA_H
   #define SP_EXAM_PROJECT_DATA_H
   namespace StochasticSimulation {
       class SimulationState;
9
       struct Reaction;
10
       class ReactantCollection;
12
       struct Reactant {
13
           std::string name;
14
           double_t amount; // double to allow for mean values
15
           size_t required{1};
16
           Reactant(std::string name, size_t initial_amount):
                    name(std::move(name)),
19
                    amount(initial_amount)
20
           {}
21
           Reactant(std::string name, double_t initial_amount):
                    name(std::move(name)),
                    amount(initial_amount)
           {}
           Reactant(std::string name, size_t initial_amount, size_t required):
28
                    name(std::move(name)),
29
                    amount(initial_amount),
30
                    required(required)
31
           {}
32
           ~Reactant() = default;
35
           Reactant(const Reactant& a) {
36
                name = a.name;
                amount = a.amount;
38
                required = a.required;
39
           }
           Reactant(Reactant&& a) {
                name = std::move(a.name);
43
                amount = std::move(a.amount);
44
                required = std::move(a.required);
45
```

```
}
46
            Reactant& operator=(Reactant&& a) {
48
                 name = std::move(a.name);
49
                 amount = std::move(a.amount);
                 required = std::move(a.required);
                 return *this;
            }
55
            Reactant& operator=(const Reactant& a) {
56
                 name = a.name;
57
                 amount = a.amount;
                 required = a.required;
59
60
                 return *this;
            }
63
            Reaction operator>>=(Reactant other);
64
65
            Reaction operator>>=(ReactantCollection other);
            ReactantCollection operator+(const Reactant& other);
            bool operator<(const Reactant& other) const;</pre>
71
            Reactant operator*(size_t req) {
72
                 required = req;
                 return *this;
            }
75
        };
78
        class ReactantCollection: public std::set<Reactant> {
79
        public:
80
            using std::set<Reactant>::set;
            Reaction operator>>=(Reactant other);
82
            Reaction operator>>=(ReactantCollection other);
        };
        class Reaction {
86
        public:
87
            std::set<Reactant> from;
            std::set<Reactant> to;
            std::optional<std::vector<Reactant>> catalysts;
90
            double_t rate{};
91
            double_t delay{-1};
            Reaction(std::set<Reactant> from, std::set<Reactant> to):
94
                     from(from),
95
                     to(to)
            {}
97
98
            Reaction(std::set<Reactant> from, std::set<Reactant> to, std::initializer_list<Reactant>
  →catalysts, double rate):
                     from(from),
100
                     to(to),
101
                     catalysts(catalysts),
102
                     rate(rate)
            {}
104
```

105

```
Reaction(std::set<Reactant> from, std::set<Reactant> to, double rate):
                     from(from),
                     to(to),
108
                     catalysts{},
109
                     rate(rate)
            {}
112
            void compute_delay(SimulationState& state, std::default_random_engine& engine);
            void compute_delay2(SimulationState& state, std::default_random_engine& engine);
115
            friend std::ostream &operator<<(std::ostream &s, const Reaction &reaction);</pre>
116
        };
117
118
        struct SimulationState {
119
        public:
120
            SymbolTable<Reactant> reactants;
            double_t time;
122
123
            SimulationState(SymbolTable<Reactant> reactants, double_t time):
124
125
                 reactants{reactants},
                 time{time}
            {};
127
128
            SimulationState(const SimulationState&) = default;
            SimulationState(SimulationState&&) = default;
130
131
            SimulationState& operator=(const SimulationState &) = default;
132
            SimulationState& operator=(SimulationState&&) = default;
134
            ~SimulationState() = default;
135
136
             friend std::ostream &operator<<(std::ostream &, const SimulationState &);</pre>
        };
138
139
140
    }
    #endif //SP_EXAM_PROJECT_DATA_H
142
```

Listing 7: data.cpp

```
//
   // Created by Mathias on 11-05-2021.
   #include <iostream>
   #include <utility>
   #include "simulation.h"
   namespace StochasticSimulation {
9
       Reaction Reactant::operator>>=(StochasticSimulation::Reactant other) {
10
           return Reaction{{*this}, {std::move(other)}};
       }
12
13
       Reaction Reactant::operator>>=(ReactantCollection other) {
           return Reaction{{*this}, std::move(other)};
15
16
       ReactantCollection Reactant::operator+(const Reactant& other) {
           return ReactantCollection{*this, other};
       }
20
21
       bool Reactant::operator<(const Reactant& other) const {</pre>
22
```

```
return name < other.name;</pre>
23
       }
25
       Reaction ReactantCollection::operator>>=(Reactant other) {
26
            return Reaction{*this, {std::move(other)}};
27
28
29
       Reaction ReactantCollection::operator>>=(ReactantCollection other) {
            return Reaction{*this, std::move(other)};
32
33
       std::ostream &operator<<(std::ostream &s, const Reaction &reaction) {</pre>
34
           s << "{ ";
           for (const auto& reactant: reaction.from) {
36
                s << reactant.name << "+";
37
           s << "\b" << " >>= ";
           if (reaction.catalysts.has_value()) {
40
                s << "(";
41
42
                for (const auto& catalyst: reaction.catalysts.value()) {
                    s << catalyst.name << "+";
                }
                s << "\b" << ") ";
           }
           for (const auto& reactant: reaction.to) {
                s << reactant.name << "+";
48
           }
49
           s << "\b";
51
            return s << " - " << reaction.rate << " }";</pre>
52
       }
53
       void Reaction::compute_delay2(SimulationState& state, std::default_random_engine &engine) {
55
           size_t reactant_amount{1};
56
           size_t catalyst_amount{1};
57
           for (const Reactant& reactant: from) {
59
                auto amount = reactant.name == "__env__" ? 1 :
 →state.reactants.get(reactant.name).amount;
61
                reactant_amount *= amount;
           }
62
           // New: check if amount is 0 already
63
           if (reactant_amount == 0) {
64
                delay = -1;
                return;
66
           }
67
           if (catalysts.has_value()) {
69
                for (auto& catalyst: catalysts.value()) {
70
                    catalyst_amount *= state.reactants.get(catalyst.name).amount;
71
                }
           }
           double_t rate_k = rate * reactant_amount * catalyst_amount;
           if (rate_k > 0) {
                delay = std::exponential_distribution<double_t>(rate_k)(engine);
78
           } else {
79
                delay = -1;
           }
81
       }
82
```

```
void Reaction::compute_delay(SimulationState& state, std::default_random_engine &engine) {
            size_t reactant_amount{1};
85
            size_t catalyst_amount{1};
86
            for (const Reactant& reactant: from) {
                 auto amount = reactant.name == "__env__" ? 1 :
  →state.reactants.get(reactant.name).amount;
90
                 reactant_amount *= amount;
91
            if (catalysts.has_value()) {
92
                 for (auto& catalyst: catalysts.value()) {
93
                     catalyst_amount *= state.reactants.get(catalyst.name).amount;
                 }
95
            }
            double_t rate_k = rate * reactant_amount * catalyst_amount;
99
            if (rate_k > 0) {
100
                 delay = std::exponential_distribution<double_t>(rate_k)(engine);
101
            } else {
                 delay = -1;
103
            }
104
        }
106
        std::ostream &operator<<(std::ostream &s, const SimulationState& state) {</pre>
107
            s << "{" << std::endl
108
                << "time: " << state.time << "," << std::endl
                 << "reactants: {" << std::endl;</pre>
110
            for(auto& pair: state.reactants) {
111
                 s << pair.first << ": " << pair.second.amount << "," << std::endl;
114
            s << "}";
115
            return s;
116
        }
    }
118
```

Listing 8: simulation monitor.h

```
//
   // Created by Mathias on 11-05-2021.
   #ifndef SP_EXAM_PROJECT_SIMULATION_MONITOR_H
   #define SP_EXAM_PROJECT_SIMULATION_MONITOR_H
   #include <functional>
   #include "data.h"
9
10
   namespace StochasticSimulation {
11
       class simulation_monitor {
12
       public:
13
           virtual void monitor(SimulationState& state) = 0;
       };
15
16
       class empty_simulation_monitor: public simulation_monitor {
           void monitor(SimulationState &state) override {
              return;
20
           };
21
       };
```

```
static auto EMPTY_SIMULATION_MONITOR = empty_simulation_monitor{};
25
       class basic_simulation_monitor: public simulation_monitor {
26
       private:
27
           const std::function<void(SimulationState&)> monitor_function;
28
       public:
29
           basic_simulation_monitor(const std::function<void(SimulationState&)>& monitor_func):
                simulation_monitor{},
                monitor_function{monitor_func}
32
           {}
33
34
           void monitor(SimulationState& state) override {
                monitor_function(state);
36
           }
37
       };
39
40
   #endif //SP_EXAM_PROJECT_SIMULATION_MONITOR_H
```

Listing 9: SymbolTable.h

```
// Created by Mathias on 10-05-2021.
   #ifndef SP_EXAM_PROJECT_SYMBOLTABLE_H
   #define SP_EXAM_PROJECT_SYMBOLTABLE_H
   #include <unordered_map>
   #include <string>
   #include <stdexcept>
   #include <utility>
11
   #include <iterator>
12
   namespace StochasticSimulation {
14
15
       struct SymbolTableException : public std::exception
16
           std::string message;
18
       public:
19
           explicit SymbolTableException(std::string message): message(std::move(message))
22
           [[nodiscard]] const char* what() const override
23
24
                return message.c_str();
           }
26
       };
27
       template<typename T>
29
       class SymbolTable {
30
           using map_type = std::unordered_map<std::string, T>;
31
       private:
           map_type map{};
33
       public:
34
           using iterator = typename map_type::iterator;
           using const_iterator = typename map_type::const_iterator;
           SymbolTable<T>() = default;
38
39
           SymbolTable<T>(const SymbolTable<T>& a) {
```

```
map = a.map;
            };
43
            SymbolTable<T>(SymbolTable<T>&& a) {
44
                 map = std::move(a.map);
45
46
            ~SymbolTable() = default;
            SymbolTable<T>& operator=(const SymbolTable& a) {
50
                 map = a.map;
51
                 return *this;
52
            };
54
            SymbolTable<T>& operator=(SymbolTable&& a) {
55
                 map = std::move(a.map);
                 return *this;
            };
58
59
60
            void put(const std::string& key, T value) {
                 if (!map.contains(key)) {
                     map.insert_or_assign(key, value);
62
                 } else {
                     throw SymbolTableException("Key " + key + " already used");
                 }
            }
66
67
            T& get(const std::string& key) {
                 try {
69
                     return map.at(key);
70
                 } catch (std::out_of_range& e) {
                     throw SymbolTableException("Key " + key + " was not found");
73
            }
74
75
             const T& get(const std::string& key) const {
                try {
                     return map.at(key);
                 } catch (std::out_of_range& e) {
                     throw SymbolTableException("Key " + key + " was not found");
                 }
81
            }
82
83
            bool contains(const std::string& key) {
                 return map.contains(key);
85
            }
86
            std::unordered_map<std::string, T> getMap() {
88
                 return map;
89
            }
90
            iterator begin() {
                 return map.begin();
93
            }
            iterator end() {
96
                 return map.end();
97
            }
98
             const_iterator begin() const {
100
                 return map.begin();
101
```

Listing 10: Results

```
Pretty-print Reactions:
Introduction:
     \{ A+B >>= (D) C - 0.001 \}
Covid:
{
          \{ S >>= (I) E - 7.74194e - 05 \},
          \{E >>= I - 0.196078 \},
          \{ I >>= R - 0.322581 \},
          \{ I >>= H - 0.000290061 \},
          \{ H >>= R - 0.0988142 \}
}
Carcadian Rythm:
          \{ A+DA >>= D A - 1 \},
          \{ D A >>= A + DA - 50 \},
          \{ A+DR >>= D_R - 1 \},
           D R >>= A + DR - 100 
           D_A >>= D_A + MA - 500  },
            DA >>= DA + MA - 50 \},
           D_R >>= D_R + MR - 50  },
          \{ DR >>= DR + MR - 0.01 \},
          \{ MA >>= A + MA - 50 \},
           MR >>= MR + R - 5  },
            A+R >>= C - 2 \},
            C >>= R - 1 \},
            A>>= \_\_env\_\_-1 \ \} \,,
            R >>= \underline{-env}_{-} - 0.2 \},
           MA >>= \underline{-env}_- - 10  },
          \{ MR >>= __{env}_{-} - 0.5 \}
}
Carcadian Rythm alternative:
            A+DA >>= D A - 1 },
           D_A >>= A + DA - 50 },
            A+DR >>= D_R - 1 },
           D_R >>= A + DR - 100 },
            _{-}env_{-} >>= (D_A) MA - 500 },
            \_{\rm env}\_\_>>=~({\rm DA})~{\rm MA}-~50~\}\,,
            _{-}env_{-} >>= (D_R) MR - 50 },
            _{-\text{env}} > > = (DR) MR - 0.01),
            \_\_env\_\_ >>= (MA) \ A - 50 \ \} \,,
              \underline{\quad} = \text{env}_{\underline{\quad}} >>= \text{(MR)} \quad \text{R} - 5 \quad \},
            A+R >>= C - 2 \},
```

```
 \left\{ \begin{array}{l} C>>=R-1 \end{array} \right\}, \\ \left\{ \begin{array}{l} A>>=\_\operatorname{env}\_-1 \end{array} \right\}, \\ \left\{ \begin{array}{l} R>>=\_\operatorname{env}\_-0.2 \end{array} \right\}, \\ \left\{ \begin{array}{l} MA>>=\_\operatorname{env}\_-10 \end{array} \right\}, \\ \left\{ \begin{array}{l} MR>>=\_\operatorname{env}\_-0.5 \end{array} \right\} \\ \end{array}
```

Example output from monitoring hospitalized (not the one on the graph)

Max hospitalized: 3

Mean hospitalized: 0.551814

Benchmarks:

Benchmarking with circadian rhythm example $(max_time=100)$ (30 times each)

Simulation 1 mean time (nanoseconds): 117913398 Simulation 2 mean time (nanoseconds): 17683541

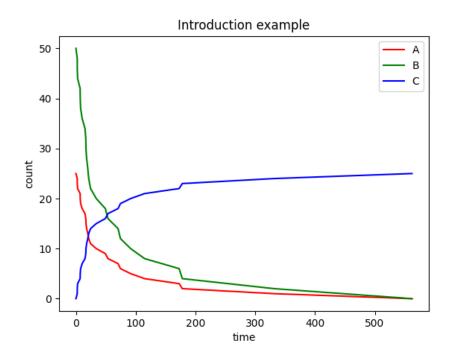


Figure 1: A(0)=25, B(0)=50, D=1

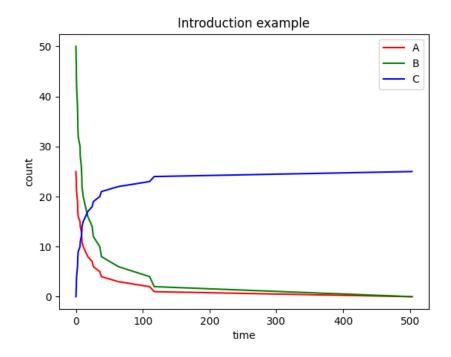


Figure 2: A(0)=25, B(0)=50, D=2

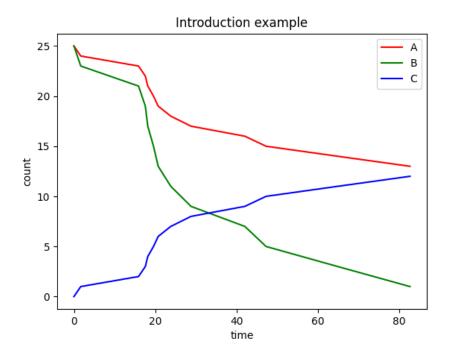


Figure 3: A(0)=25, B(0)=25, D=1

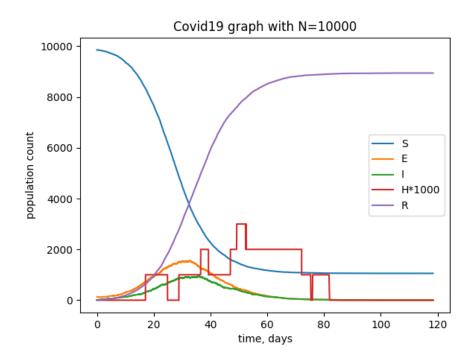


Figure 4: Sample covid trajectory

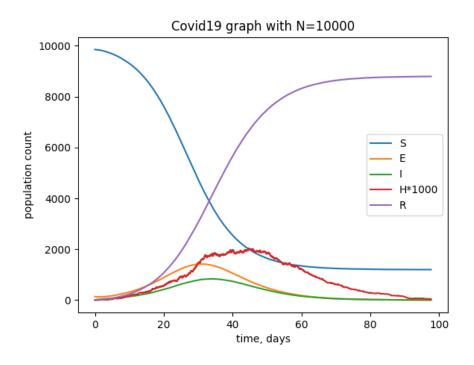


Figure 5: Mean covid trajectory of 30 simulations

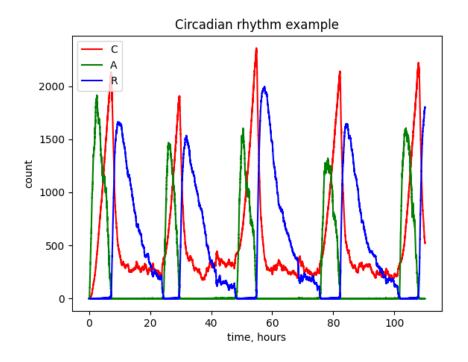


Figure 6: Sample circadian rythm trajectory

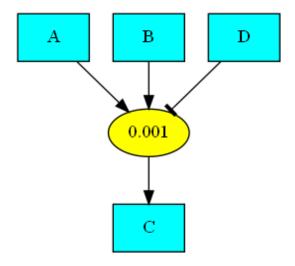


Figure 7: Intro reaction graph

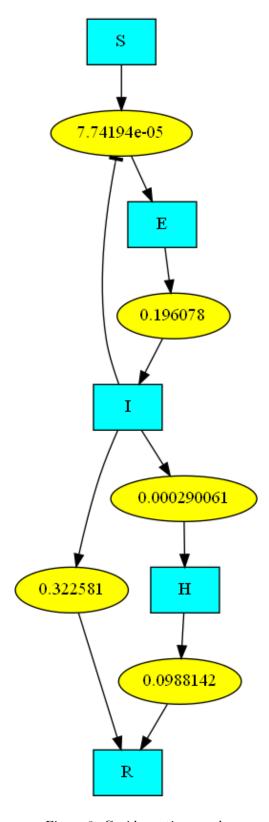


Figure 8: Covid reaction graph

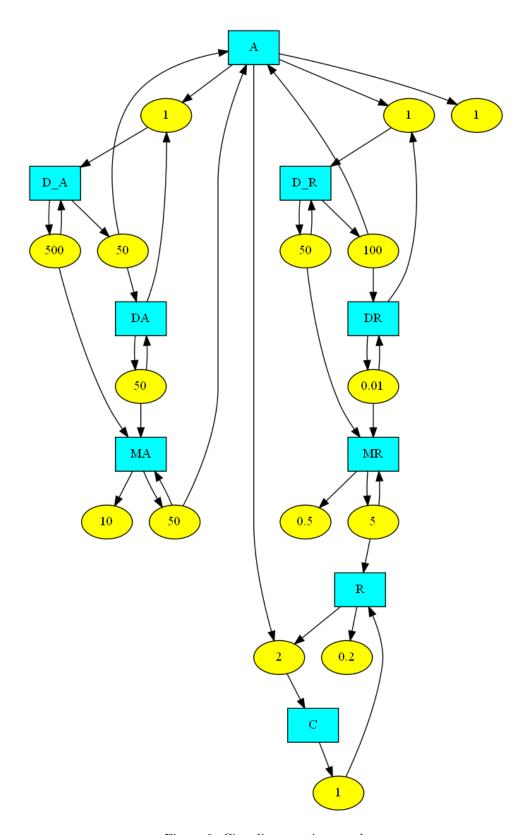


Figure 9: Circadian reaction graph

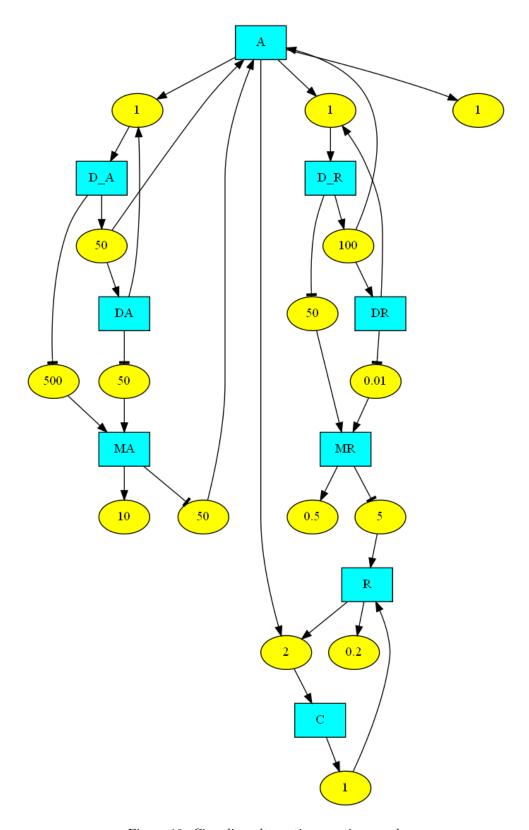


Figure 10: Circadian alternative reaction graph