CP 331 Term Project: Monte Carlo AIXI Approximation
Group 2

Hilts, Vaughan
hilts.vaughan@gmail.com, 120892740

Morouney, Robert
robert@morouney.com, 069001422

Rusu, David
davidrusu.me@gmail.com, 131920260

April 24^{th} 2016

Contents

1 AIXI **2**
- 1.1 Introduction . . . 2
- 1.2 A Monte-Carlo AIXI Approximation . . . 2

2 Context Tree Weighting (CTW) **6**
- 2.1 Introduction . . . 6
- 2.2 How We Use CTW . . . 6
- 2.3 How CTW Works . . . 6
- 2.4 Krichevsky-Trofimov Estimator . . . 8

3 Agent **10**
- 3.1 Introduction . . . 10

4 Environment **12**
- 4.1 Introduction . . . 12
- 4.2 Making C Object Oriented . . . 12
- 4.3 Coin-Flip Environment . . . 20

5 Full Code Listing **23**
- 5.1 Top level . . . 23
- 5.2 Object . . . 33
- 5.3 Agent . . . 35
- 5.4 Search . . . 41
- 5.5 Predict . . . 45
- 5.6 Environment . . . 56
- 5.7 Utility . . . 64

1 AIXI

1.1 Introduction

The AIXI agent is a mathematical definition of the perfect reinforcement learning agent. A reinforcement learning agent works by iteratively performing actions on an environment and receives a reward and observation after each action. Over time, the agent should learn to predict future states of the environment and use that to it's advantage to maximize the reward it receives. The AIXI algorithm solves this using a brute force approach, intuitively, AIXI receives observations and rewards of the environment as a series bits, it then generates all programs that would predict these bits, takes the simplest (shortest) of these programs and uses it to predict future reward observation pairs. It then chooses the action that resulted in the largest predicted reward and performs it onto the environment. This is turns out to be not computable, but given the generality of this approach it is pleasing to see that something so powerful can be expressed in one line of math.

TODO: include aixi equation here

1.2 A Monte-Carlo AIXI Approximation

Under the most ideal conditions, AIXI would be perfect as previously described. However, it is found that AIXI is not actually computable due to constraints of the Solomonoff induction. Due to this, the only realistic and viable way to implement AIXI is using an approximation. One of these approximations can be done a la "monte-carlo", that is using random sampling instead of completely brute-forcing the entire solution space. In the next following paragraphs, we will discus the procedure.

We begin with a sparse search tree beginning with a decision node at the root, with a few children. Each of these nodes represents some history from the simulation, it can be thought of as of some sort of memory for the simulation. If the history ended in an action, then it is a chance

node. Otherwise, it is a decision node. However, to create this history we must perform actual simulations. For that, there are four stages. In the next few sections, we will look at them and their applications, especially in the monte_sample context.

Selection Phase

In the selection phase, we traverse the tree moving down to the first leaf node that we can find that is a chance node. We traverse in such a way that follows a particular policy. The most important part of the policy is that only a maximum horizon of actions can be chosen.

```
} else if(tree->type == NODE_TYPE_CHANCE) {
    u32Tuple* tuple = Agent_generate_percept_and_update(agent);

    u32 observation = tuple->first;
    u32 random_reward = tuple->second;

    bool notInTreeYet = dict_find(tree->children, observation) == NULL;

    if(notInTreeYet) {
        MonteNode* newChild = monte_create_tree(NODE_TYPE_DECISION);
        dict_add(tree->children, observation, newChild);
    }

    // Grab a monte node that is a child of.
    MonteNode* child = dict_find(tree->children, observation);

    if(child == NULL) {
        exit(1);
    }

    // Recurse
    reward = random_reward + monte_sample(child, agent, horizon - 1);
```

Listing 1.1: "Monte Sample"

Expansion Phase

In the above code listing from the previous section, there is a portion which adds the decision node to the tree before traversing down more. This is exactly what the expansion stage is about – expanding the tree and creating yet more information about the simulation.

Simulation Phase / Back Propagation

In this stage, we sample some random path from the environment until we end up a certain point from the root, that is the agent horizon. In the sample function above, there is another branch:

```
1
2  else {
3          u32 action = _monte_select_action(tree, agent);
4          Agent_model_update_action(agent, action);
5
6          MonteNode* child = dict_find(tree->actions, action);
7          if(child == NULL) {
8              child = monte_create_tree(NODE_TYPE_CHANCE);
9            dict_add(tree->actions, action, child);
10         }
11
12         if(child == NULL) {
13             exit(1338);
14         }
15
16         reward = monte_sample(child, agent, horizon);
17
```

Listing 1.2: "Simulation Phase"

Specifically, the select action call from above is the one which does the interaction with the environment. After this phase, the back propagation begins and the node values are updated accordingly to how they now estimate in relative to the root.

Parallelization

Since the problem boils down to searching this sparse tree, the parallelization is obvious. You can invoke the sample function on the tree just like you would normally, following the four phases but while locking the interior nodes. However, there is another option as well when making copies of the CTW is cheap or affordable. That is, the memory-copy time is lower than the computation time to run in parallel with locking. If this is the case, we can make a copy of the tree and run seperately. The CTW is mutated, so while running in parallel without locking would be an issue.

As a proof of concept, we implemented this with copying of the CTW tree and saving it to a high speed I/O scratch disk where we could thaw it fast on each processor for each cycle. The communication framework used was MPI. So, on each non rank-zero processor, we perform the search in parallel:

```
1      int data;
2      MPI_Recv(&data, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
3
4      ctw_load(agent->context_tree, CTW_DATA_FILE);
5      double mean;
6      int action = Agent_search_mean(agent, &mean);
7
8      MPI_Send(&action, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
9      MPI_Send(&mean, 1, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
```

Listing 1.3: "Parallel Search"

Then, on the main processor we compare their results by retrieving them as they come in:

```
1 for (i = 1; i < P; i++) {
2   double mean;
3   MPI_Recv(&action, 1, MPI_INT, i, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
4     MPI_Recv(&mean, 1, MPI_DOUBLE, i, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
5     if (mean > best_mean) {
6       best_action = action;
7       best_mean = mean;
8     }
9   }
```

Since the Monte-Carlo Approximation on it's own has a sense of randomness to it while selection nodes, not every path chosen would be the same. Due to this, we have many options to choose for each path. We take advantage of this in the parallel implementation: we can pick the path each time from each iteration that gives us the best result. By choosing to do this, we get a wider breadth of search, enabling us to converge in our simulation quicker.

2 Context Tree Weighting (CTW)

2.1 Introduction

Context Tree Weighting (Willems; Shtarkov; Tjalkens (1995)) is an algorithm tha t can be used both for lossless compression and predicting bit-strings. We are using it's predictive power for our agent to build it's internal model of the environment.

2.2 How We Use CTW

The CTW is the agents predictive model of how it thinks the environment behaves. It is trained each iteration by taking the reward, observation pair, converting them to bit-strings and feeding those bits into the CTW, when we choose an action, we convert this action into a bit-string and feed that into the CTW as well. While we are deciding what actions to take, we temporarily feed a potential action bit-string into the CTW and see what reward, observation pairs it predicts. Afterwards, we revert the tree to the state before the action was taken. This hypothetical action taking process is governed by the MCTS algorithm.

2.3 How CTW Works

The CTW tree is a full binary tree up to some predefined depth. In practice allocating the full binary tree is not practical, so as we traverse the tree, if we find a node that is not yet allocated, we then allocate it and continue. Since most simple environments follow a well defined pattern, they will not get close to exploring an entire CTW of a decent depth.

Each node in the CTW has a One and Zero child, a probability, and a count of how many ones and zeroes this node has seen. The probabilities are stored as log probabilities to get rid of some

inefficient power calculations.

```
typedef struct ContextTreeNode {
  double log_kt;
  double log_probability;
  uint32_t ones_in_history;
  uint32_t zeroes_in_history;
  struct ContextTreeNode *zero_child;
  struct ContextTreeNode *one_child;
} ContextTreeNode;
```

The Context Tree itself stores a pointer to the root node of the tree and a bit-string of the entire history of the agents interaction with the enviroment. The Context Tree also stores the trace of the most recent path through the context tree, known as the context. This context is updated each time new data is fed to the CTW.

```
typedef struct ContextTree {
  uint32_t depth;
  ContextTreeNode *root;
  BitVector *history;
  CTWNodeList *context;
} ContextTree;
```

To update the context tree, we can either send in a bit-string or a single bit symbol, we will look at the single symbol implementation for simplicity. Here the probability calculations come from Krichevsky-Trofimov Estimator. We run through all the nodes in the current context and update each one from the leaf node to the parent.

```
void ctw_update_symbol(ContextTree *tree, bool symbol) {
  if (tree->history->size >= tree->depth) {
    ctw_update_context(tree);
    int64_t i;
    for (i = tree->depth-1; i >= 0; i--) {
      ctw_node_update(ctw_list_get(tree->context, i), symbol);
    }
  }
  bv_push(tree->history, symbol);
}

void ctw_node_update(ContextTreeNode *node, bool symbol) {
  node->log_kt += ctw_node_log_kt_multiplier(node, symbol);
  ctw_node_update_log_probability(node);
  if (symbol) {
    node->ones_in_history += 1;
  } else {
    node->zeroes_in_history += 1;
  }
}

double ctw_node_log_kt_multiplier(ContextTreeNode *node, bool symbol) {
```

```
23   uint32_t numerator;
24   if (symbol) {
25     numerator = node->ones_in_history;
26   } else {
27     numerator = node->zeroes_in_history;
28   }
29   uint32_t denominator = ctw_node_visits(node);
30   return log((numerator + 0.5) / (denominator + 1.0));
31 }
32
33 void ctw_node_update_log_probability(ContextTreeNode *node) {
34   if (ctw_node_is_leaf(node)) {
35     node->log_probability = node->log_kt;
36   } else {
37     double log_child_prob = 0.0;
38     if (node->zero_child != NULL) {
39       log_child_prob += node->zero_child->log_probability;
40     }
41     if (node->one_child != NULL) {
42       log_child_prob += node->one_child->log_probability;
43     }
44
45     double a, b;
46     if (node->log_kt >= log_child_prob) {
47       a = node->log_kt;
48       b = log_child_prob;
49     } else {
50       a = log_child_prob;
51       b = node->log_kt;
52     }
53     node->log_probability = log(0.5) + a + log1p(exp(b - a));
54   }
55 }
```

2.4 Krichevsky-Trofimov Estimator

CTW relies heavily on the Krichevsky-Trofimov Estimator. Given a bit-string s with a 0's and b 1's the estimator gives the probability of the next bit with the following recursive definition.

$$Pr_{kt}(0,0) = 1 \tag{2.1}$$

$$Pr_{kt}(a+1,b) = \frac{a+1/2}{a+b+1/2} Pr_{kt}(a,b) \tag{2.2}$$

$$Pr_{kt}(a,b+1) = \frac{b+1/2}{a+b+1/2} Pr_{kt}(a,b) \tag{2.3}$$

As you can see, all we need is the number of 1's and 0's in a string to give an estimate of the next bit, this helps us in CTW because we don't need to store the entire bit-string that led to a node in the tree. The recursive calculation happens in our code when the CTW is updated. Because of this simple recursive definition, it is simple to revert updates. We just decrement the count of that symbol and recalculate the probabilities with the new count.

```
void ctw_node_revert(ContextTreeNode *node, bool symbol) {
  // This is called in a loop from leaf to root, so we know that the
  // node's children have already been treated

  if (symbol && node->ones_in_history > 0) {
    // symbol is 1
    node->ones_in_history -= 1;
  } else if (!symbol && node->zeroes_in_history > 0) {
    // symbol is 0
    node->zeroes_in_history -= 1;
  }

  // need to remove redundant nodes, since this has already been called on
  // the node's children, they may have 0 visits now
  if (symbol) {
    if (node->one_child != NULL && ctw_node_visits(node->one_child) == 0) {
      free(node->one_child);
      node->one_child = NULL;
    }
  } else {
    if (node->zero_child != NULL && ctw_node_visits(node->zero_child) == 0) {
      free(node->zero_child);
      node->zero_child = NULL;
    }
  }

  node->log_kt -= ctw_node_log_kt_multiplier(node, symbol);
  ctw_node_update_log_probability(node);
}
```

3 Agent

3.1 Introduction

The agent in AIXI can be thought akin to something playing with the environment and performing actions against it while trying to learn more about how it works to develop and learn. AIXI does not specify how the agent must be implemented but as alluded to before in this paper, we had chosen to implement one using a CTW (Context Tree Weighting). Due to this, that means the agent does not know what it is actually performing against neccesairly – only that is performing in some abstract way. To handle this, the actual interaction with the world is handled internally to the environment. This is described in more detail in the later sections.

On each tick, the agent will receive an observation and reward. The observation is just that – an observation about the environment has been portrayed and reported back to the agent. It is given in the form of a bit-string and will vary depending on how the environment has decided to model it. It is not actually important what the bit string is or how it was encoded – only that it is consistent and of a fixed length. For a coin flip game, it could be the state the coin. For a maze, it could be a portion of the maze. 0 could be a tails and 1 could be a heads. However, once again, their roles could easily be reversed and encoded inversely.

The reward on the other hand is an indiciator of how well the agent had performed. The agent will get a "positive" reward to enforce good actions and it will get "negative" or "no reward" for ones that are not neccessairly needed to be rewarded. The reward is an indicator of just how well it is doing and factors into how it should act in the future.

The actions by the agent are also abstract but determined from the environment. The environment provides a list of actions to the agent and based on a probability distibution for it's current environment, it will pick accordingly. This is where the CTW portion comes in for the agent. Using the CTW, the sequence of bits that are being provided can be guessed at with a good degree of accuracy – so it serves as our probability distribution in this case. The agent will consider the full history (due to the CTW) when choosing its action. The actual process of searching for this is described below in code:

```
u32 Agent_search_mean(Agent* self, double *mean) {

  printf("start search\n");
  AgentUndo* undo = Agent_clone_into_temp(self);

  MonteNode* node = monte_create_tree(NODE_TYPE_DECISION);

  printf("start sampling\n");
  // 300 sims
  for(u32 i = 0; i < 50;i++ ) {
    monte_sample(node, self, self->horizon);
    Agent_model_revert(self, undo);
  }

  printf("done sampling\n");
  u32 best_action = Agent_generate_random_action(self);
  double best_mean = -1;

  for(u32 i = 0; i < self->environment->num_actions; i++) {
    u32 action =  self->environment->_valid_actions[i];
    MonteNode* searchNode = dict_find(node->actions, action);

    if(searchNode != NULL) {
      double mean = searchNode->mean + ((float)rand()/(float)(RAND_MAX)) * 0.0001;
      if(mean > best_mean) {
  best_mean = mean;
  best_action = action;
      }
    }
  }

  printf("done search\n");
  *mean = best_mean;
  return best_action;
}
```

4 Environment

4.1 Introduction

The Environment for AIXI is the actual place where AIXI interacts with the program. AIXI does not necessarily have a picture of the full environment. The agent sends actions to the environment in the form of bit-strings. The environment receives the bit-string and checks it against a list of acceptable actions. If the action is in the list the environment generates an observation and a reward. The reward lets the agent know if it's action was valid and/or beneficial. The observation is a bit-string which gives the agent a snap-shot of the environment. In the case of our coin-flip environment this means telling the agent if the coin was heads or tails.

Example For the coin-flip environment the agent would send an action, either heads or tails, to the environment. The environment would then receive this action and generate an observation. If the observation matches the action given by the agent then the environment sends the proper reward, which in this case is 'Win'.

4.2 Making C Object Oriented

The main concern we had while making the environment was creating a generic structure so that new environments could be swapped in and out. Because C generally not considered an 'Object Oriented' language, we decided to create our own class and inheritance structure. To accomplish this we used generic pointers and representation files. Our base object class is held in 'class.r':

```
1 #ifndef CLASS_R
2 #define CLASS_R
3
4 #include <stdarg.h>
5 #include "../_utils/types.h"
```

```
6
7  struct Class {
8      size_t  size;
9      void *  ( * __init__ )          ( void * self, va_list      args );
10     void *  ( * __delete__ )        ( void * self );
11     void *  ( * __copy__ )          ( const void * self );
12     void *  ( * __str__ )           ( const void * self );
13 };
14
15 #endif
```

Listing 4.1: class.r

paragraphOur base class handles construction, destruction, secure copy and string representation. Because these are just void function pointers they can easily be over written by the inheriting class. Though this does sacrifice some type security the abstraction decreases the lines of code needed significantly which makes the overall program more readable and very portable. We needed a generic way to create classes based on the above representation.

```
1  #include <assert.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include "class.h"
5  #include "class.r"
6  #include "../_utils/macros.h"
7
8  void * new ( const void * _class, ... )
9  {
10     const struct Class * class = _class;
11     void * mem = calloc( 1, class->size );
12
13     //TODO: Replace with better error handling.
14     assert ( mem );
15
16     * ( const struct Class ** ) mem = class;
17
18     // This handles and vars passed to the constructor.
19     if ( class->__init__ )
20     {
21         va_list args;
22         #ifdef DEBUG
23             TRACE("Class Created", "__init__",class->__str__);
24         #endif
25         va_start( args, _class );                  // intialize '...'
26       mem = class->__init__( mem, &args );       // call constructor
27         va_end( args );                            // clean
28     }
29
30     return mem;
31 }
32
33 void delete ( void * self )
34 {
35     const struct Class ** parent = self;
36
37     if ( self && * parent && ( * parent )->__delete__ ) {
38         self = ( * parent )->__delete__(self);
39         #ifdef DEBUG
```

```
40              TRACE("Class Destroyed","delete(...)",class->__str__);
41          #endif
42      }
43      free(self);
44  }
45
46  void * cpy ( const void * self )
47  {
48      const struct Class * const * parent = self;
49
50      assert ( self && parent && ( * parent )->__copy__);
51
52      #ifdef DEBUG
53          TRACE("Class Copied","cpy(..)",class->__str__);
54      #endif
55
56      return ( * parent )->__copy__(self);
57  }
58
59  char * print ( const void * self )
60  {
61      const struct Class * const * parent = self;
62      char * pstring = malloc ( 255 * (sizeof(char)));
63      sprintf(pstring,"%s",( * parent )->__str__(self));
64
65      #ifdef DEBUG
66          TRACE ( "Generated Print String: ", "print(...)", pstring );
67      #endif
68
69      return pstring;
70  }
```

Listing 4.2: class.c

Using the above functions we can now take any class child class of struct Class and call its constructor using $new(Class_Type, ...)$. The same goes for delete, copy and print.

Once we had a base class structure we were ready to start creating our environment class. To do this we turned again to a representation file:

```
1  #ifndef ENV_R
2      #define ENV_R
3      #include "../_utils/types.h"
4      #include <stdarg.h>
5      // Class Environment:
6      struct Environment
7      {
8          const void * class; // must be first
9          u32        _action;
10         u08        _is_finished;
11         u32        _observation;
12         va_list    _options;
13         u08        _reward;
14         u32        *_valid_actions;
15         u32        *_valid_observations;
16         u32        *_valid_rewards;
```

```
17          u32       num_actions;
18      };//----------------------------------------------------------
19
20      #define action(e) (((const struct Environment *)(e)) -> _action)
21      #define is_finished(e) ((const struct Environment *)(e)) -> _is_finished
22      #define observation(e) (((const struct Environment *)(e)) -> _observation)
23      #define reward(e) (((const struct Environment *)(e)) -> _reward)
24      #define valid_actions(e) (((const struct Environment *)(e)) -> _valid_actions)
25      #define valid_observations(e) (((const struct Environment *)(e)) ->
        _valid_observations)
26      #define valid_rewards(e) (((const struct Environment *)(e)) -> _valid_rewards)
27
28  #endif
```

Listing 4.3: environment.r

The above representation sets the environment up in such a way that the variables declared here will be available to all the inheriting classes. As you will see in a moment we decided to use regular functions instead of class methods for this object. We chose to do this because the functions declared in 'environment.c' need only ever be used on the Environment variables and do not need to be overwritten by the inheriting class. This representation provides some safety as well since the variables cannot be directly accessed by the inheriting class, instead the defined accessors (e.g action(e)) must be used instead. This eliminates most type safety issues.

```
1  #include <assert.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include "../_utils/types.h"
5  #include "../_utils/macros.h"
6  #include "class.h"
7  #include "class.r"
8  #include "environment.h"
9  #include "environment.r"
10
11 // def __init__():
12 static void * Environment_init ( void * _self, va_list * args )
13 {
14     struct Environment * self = _self;
15
16     va_copy ( self -> _options, *args );
17     self -> _is_finished    = 0x00;
18     self -> _reward         = 0x00;
19     self -> _action         = 0x00;
20
21     #ifdef DEBUG
22         TRACE("Environment initialized\n","Environment_init\n");
23     #endif
24
25     return self;
26 }//----------------------------------------------------------------------
27
28 // def __delete__():
29 static void * Environment_delete ( void * _self )
30 {
31     struct Environment * self = _self;
```

```
32
33      //free ( self->_options ),              self->_options = 0;
34      //free ( self->_observation )
35      //self->_observation = 0;
36      //free ( self->_valid_observations )//, self->_valid_observations = 0;
37      //free ( self->_valid_actions )//,      self->_valid_actions = 0;
38      free ( self );
39
40      #ifdef DEBUG
41          TRACE("Environment destroyed\n","Environment_delete\n");
42      #endif
43
44      return self;
45  }//-------------------------------------------------------------------
46
47  // def secure-copy():
48  static void * Environment_cpy ( const void * _self )
49  {
50      const struct Environment * self = _self;
51
52      #ifdef DEBUG
53          TRACE("Environment copied\n","Environment_cpy\n");
54      #endif
55
56      return new ( Environment , self->_options );
57  }//-------------------------------------------------------------------
58
59  // def ___str___():
60  static void * Environment_str ( const void * _self )
61  {
62      const struct Environment * self = _self;
63
64      // reserve 255 Characters for print string
65      char *  pstring = malloc(sizeof(char) * 0xFF);
66
67      sprintf ( pstring, "action = %x, observation = %x, reward = %x\n",
68                self->_action, self->_observation, self->_reward );
69      #ifdef DEBUG
70          TRACE("%s \n Environment_init\n", pstring);
71      #endif
72
73      return pstring;
74  }//-------------------------------------------------------------------
75
76  // This is kind of complicated but it basically uses the parent object class to
        define
77  // the enironment object pointer which holds all of the accessor methods.  This is
        all
78  // in the hopes that anyone can take this code and call Environment e = new(
        environment,...)
79  // then class methods can be called with e.method()
80  //
81  static const struct Class _Environment = {
82      sizeof(struct Environment),
83      Environment_init,                       // done
84      Environment_delete,                     // done
85      Environment_cpy,                        // done
86      Environment_str,                        // done
```

```
87 }; //-------------------------------------------------------------
88
89 const void * Environment = & _Environment;
90
91 // def action_bits():
92 u32  action_bits ( void * _self )
93 {
94     struct Environment * self = _self;
95     assert ( self->_valid_actions != NULL);
96
97     u32 max_action = 0;
98
99     foreach ( u32 const * action, self->_valid_actions )
100         max_action = *action && *action > max_action ? *action : max_action;
101
102     return LG2( max_action );
103
104 }//-------------------------------------------------------------------
105
106 // def observation_bits():
107 u32  observation_bits ( void * _self )
108 {
109     struct Environment * self = _self;
110     assert ( self->_valid_observations != NULL);
111
112     u32 max_observation = 0;
113
114     foreach ( u32 const * observation, self->_valid_observations )
115         max_observation = *observation && *observation > max_observation ? *
    observation : max_observation;
116
117     return LG2( max_observation );
118
119 }//-------------------------------------------------------------------
120
121 // def reward_bits():
122 u32  reward_bits ( void * _self )
123 {
124     struct Environment * self = _self;
125     assert ( self->_valid_rewards != NULL);
126
127     u32 max_reward = 0;
128
129     foreach ( u32 const * reward, self->_valid_rewards )
130         max_reward = *reward && *reward > max_reward ? *reward : max_reward;
131
132     return LG2( max_reward );
133
134 }//-------------------------------------------------------------------
135
136 // def perception_bits():
137 u32  percption_bits ( void * _self )
138 {
139     struct Environment * self = _self;
140     return reward_bits(self) + action_bits(self);
141
142 }//-------------------------------------------------------------------
143
```

```
144 // check if the action is valid
145 u08  is_valid_action ( void * _self, u32   action )
146 {
147
148     struct Environment * self = _self;
149
150     foreach( u32 const * a , self->_valid_actions)
151         if ( * a == action ) return TRUE;
152
153     return FALSE;
154
155 }//----------------------------------------------------------------------
156
157 // find out if the observation is a valid one
158 u08  is_valid_observation ( void * _self, u32   observation )
159 {
160
161     struct Environment * self = _self;
162
163     foreach( u32 const * o , self->_valid_observations)
164         if ( * o == observation ) return TRUE;
165
166     return FALSE;
167
168 }//----------------------------------------------------------------------
169
170 // check if the action is a valid action
171 u08  is_valid_reward ( void * _self, u32   reward )
172 {
173
174     struct Environment * self = _self;
175
176     foreach( u32 const * r , self->_valid_rewards)
177         if ( * r == reward ) return TRUE;
178
179     return FALSE;
180
181 }//----------------------------------------------------------------------
182
183 // Get maximum action
184 u32  maximum_action ( void * _self )
185 {
186     struct Environment * self = _self;
187     u16 idx = 0;
188
189     foreach ( u32 const * x , self->_valid_actions)
190         idx++;
191
192     return self->_valid_actions[idx];
193
194 }//----------------------------------------------------------------------
195
196 // Get maximum observation
197 u32  maximum_observation ( void * _self )
198 {
199     struct Environment * self = _self;
200     u16 idx = 0;
201
```

```
202     foreach ( u32 const * x , self->_valid_observations)
203       idx++;
204 
205     return self->_valid_observations[idx];
206 
207 }//------------------------------------------------------------------
208 
209 
210 // Get maximum reward
211 u32  maximum_reward ( void * _self )
212 {
213     struct Environment * self = _self;
214     u16 idx = 0;
215 
216     foreach ( u32 const * x , self->_valid_rewards)
217     {
218         if ( *x ) idx++;
219         else break;
220     }
221 
222     return self->_valid_rewards[idx];
223 
224 }//------------------------------------------------------------------
225 
226 // Get minimum action
227 u32  minimum_action ( void * _self )
228 {
229     struct Environment * self = _self;
230     return self->_valid_actions[0];
231 
232 }//------------------------------------------------------------------
233 
234 // Get minimum observation
235 u32  minimum_observation ( void * _self )
236 {
237     struct Environment * self = _self;
238     return self->_valid_observations[0];
239 
240 }//------------------------------------------------------------------
241 
242 // Get minimum reward
243 u32  minimum_reward ( void * _self )
244 {
245     struct Environment * self = _self;
246     return self->_valid_rewards[0];
247 
248 }//------------------------------------------------------------------
249 
250 u32 LG2 ( u32 x )
251 {
252     #define LT(n) n, n, n, n, n, n, n, n, n, n, n, n, n, n, n, n
253     const char LogTable256[256] =
254     {
255         -1, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3,
256         LT(4), LT(5), LT(5), LT(6), LT(6), LT(6), LT(6),
257         LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7)
258     };
259 
```

```
260     register u32 ret, t, tt; // temp var2
261
262     if ((tt = x >> 16))
263         ret = (t = tt >> 8) ? 24 + LogTable256[t] : 16 + LogTable256[tt];
264     else
265         ret = (t = x >> 8) ? 8 + LogTable256[t] : LogTable256[x];
266
267     return (u32) ret;
268 }//-------------------------------------------------------------------------
```

Listing 4.4: environment.c

The above code may seem like magic at first but the secret is all in the inheritance structure. The first few functions are not defined anywhere but are used to overwrite the functions outlined in the base class. This way all the base class functions can still be used but they will access the constructor for the inheriting class, in this case the environment. This provides the full frame work for our environements. Next we will show you an example of a working environment which uses the above structure.

4.3 Coin-Flip Environment

The idea of the coin-flip environment was outlined at the beginning of this chapter. The environment itself is trivial the real magic is in how few lines of code it can be accomplised on becuase of the above structure. As with any class in our structure the coin-flip environment must be started with a representation file:

```
1 #ifndef COIN_FLIP_R
2     #define COIN_FLIP_R
3     #include "../_utils/types.h"
4    struct Coin_Flip {
5         struct Environment _;
6         double probability;
7    };
8     #define probability(e) (((const struct Coin_Flip*)(e)) -> probability)
9
10    //typedef enum { Tails , Heads } _action_enum;
11    //typedef enum { Tails , Heads } _observation_enum;
12    //typedef enum { Loss , Win } _reward_enum;
13
14    static double default_probability = 7e-1;
15 #endif
```

Listing 4.5: coin$_f$*lip.r*

Once the class structure is understood the above file should be very easy to read. Since this is such a simple environment the only variable that needs to be monitored is the probability of the coin. This way we can insert a probability to create a biased coin which demonstrates that AIXI can converge to the bias of the coin by choosing optimal answers. The environment itself is listed below:

```
void * CF_init ( void * _self, va_list * args )
{
    struct Coin_Flip * self =
        ((const struct Class *) Environment) -> __init__( _self , args );

    self -> _ . num_actions             = 2;

    self -> _ . _valid_actions          = calloc (1, 2 * sizeof ( u32 ) );
    self -> _ . _valid_actions[0]       = 0;
    self -> _ . _valid_actions[1]       = 1;

    self -> _ . _valid_observations     = calloc (1, 2 * sizeof ( u32 ) );
    self -> _ . _valid_observations[0]  = 0;
    self -> _ . _valid_observations[1]  = 1;

    self -> _ . _valid_rewards          = calloc (1, 2 * sizeof ( u32 ) );
    self -> _ . _valid_rewards[0]       = 0;
    self -> _ . _valid_rewards[1]       = 1;

   double probability_t = va_arg ( * args , double );
   if ( probability_t <= 0.0001 || probability_t >= 1.0001 ) probability_t = 0.7;

    #ifdef DEBUG
        TRACE ( "Probability = %d\n", probability_t );
    #endif

   self -> probability = probability_t;

   srand(time(NULL));
   u32 random_index = rand() % 2;
   self->_._observation = self->_._valid_observations[random_index];

   //reward(self) = 0;
   return self;
}

double __rp() { return (double) rand() / (double)RAND_MAX; }

u32Tuple* perform_action ( void * _self, u32 action_t )
{
    struct Coin_Flip * self = _self;

    #ifdef DEBUG
        TRACE ( "Action = %d\n", action_t );
    #endif

    BLOCK_START
        u08 is_valid = 0x00;
        foreach ( u32 const * a , valid_actions(self) )
            if ( * a == action_t ) is_valid = !(is_valid);
        assert ( is_valid != 0x00 );
    BLOCK_END

    self -> _ . _action = action_t;

    u32 observation_t , reward_t;

```

```
58      if (__rp() > probability(self) ){
59          observation_t = 1;
60          reward_t = ( action_t == 0 ) ? 0 : 1;
61      } else {
62          observation_t = 0;
63          reward_t = ( action_t == 0 ) ? 1 : 0;
64      }
65
66      #ifdef DEBUG
67          TRACE ( "Observation = %d Reward = %d\n", observation_t, reward_t );
68      #endif
69
70      self -> _ . _observation    = observation_t;
71      self -> _ . _reward         = reward_t;
72
73      u32Tuple* tuple = calloc (1, sizeof(u32Tuple));
74      tuple -> first = observation_t;
75      tuple -> second = reward_t;
76
77      return tuple;
78  }
79
80  static void CF_print(void * _self)
81  {
82      struct Coin_Flip * self = _self;
83      printf ("Prediction = %x, Observation = %x, Reward = %x\n",
84              action(self),observation(self),reward(self));
85  }
86
87  void * CF_cpy ( void * _self )
88  {
89      struct Coin_Flip * self = _self;
90      return new ( Coin_Flip , probability(_self) );
91  }
92
93  static const struct Class _Coin_Flip = {
94      sizeof(struct Coin_Flip), CF_init, NULL, CF_cpy, NULL
95  };
96
97  const void * Coin_Flip = & _Coin_Flip;
```

Listing 4.6: coin$_f$*lip.c*

It should be obvious that using the above structure any problem that can be solved by AIXI can be written into an environment very easily and the environment can be swapped out by simply changing the include path. This makes testing multiple environments very easy and time effective.

5 Full Code Listing

5.1 Top level

```
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>
#include <time.h>
#include "environment/environment.r"
#include "environment/environment.h"
#include "environment/class.r"
#include "environment/class.h"
#include "environment/coin_flip.r"
#include "environment/coin_flip.h"
#include "agent/agent.h"
#include "_utils/macros.h"

#ifdef USE_MPI
    #include "mpi.h"
#endif

#define CTW_DATA_FILE "/scratch/drusu/ctw.dat"

typedef struct app_options {
    int agent;
    int agent_horizon;
    int ct_depth;
    int environment;
    float exploration;
    float explore_decay;
    int learning_period;
    int mc_simulations;
    bool profile;
    int terminate_age;
```

```
31      bool verbose;
32  } app_options;
33
34  app_options* _make_default_options() {
35      app_options* options = malloc(sizeof(app_options));
36      options->agent = 0;
37      options->agent_horizon = 5;
38      options->ct_depth = 30;
39      options->environment = 0;
40      options->exploration = 0.001f;
41      options->explore_decay = 1.0f;
42      options->learning_period = 0;
43      options->mc_simulations = 300;
44      options->profile = false;
45      options->terminate_age = 0;
46      options->verbose = false;
47
48      return options;
49  }
50
51  float _random_0_1() {
52      return (float)rand()/(float)(RAND_MAX/1);
53  }
54
55  #ifdef USE_MPI
56  void mpi_main(Agent* agent, struct Environment* environment, app_options* options,
        int argc, const char* argv[]) {
57    int rank, P;
58
59    if (MPI_Init(&argc, &argv) != MPI_SUCCESS) {
60      printf("Error\n");
61      return 1;
62    }
63
64
65    MPI_Comm_size(MPI_COMM_WORLD, &P);
66    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
67
68    if (rank == 0) {
69       printf("%d\n", agent->context_tree);
70      printf("desu\n");
71        agent_proc(P, agent, environment, options);
72    } else {
73      search_proc(rank, P, agent, environment, options);
74    }
75
76    MPI_Finalize();
77  }
78
79  void agent_proc(int P, Agent* agent, struct Environment* environment, app_options*
        options) {
80    printf("MC AIXI training warming up...\n");
81    srand(1337);
82
83    bool explore =  options->exploration > 0;
84    if(0.0f > options->exploration || 0.0f > options->explore_decay || options->
        explore_decay > 1.0f) {
85      printf("Some exploration parameter is invalid. Application force quitting.\n");
```

```
86      exit(1);
87    }
88
89    bool terminate_check = options->terminate_age > 0;
90    bool isEnvironmentFinished = false;
91    int cycle = 1;
92
93    while(!isEnvironmentFinished) {
94      int agent_age = agent->age;
95
96      if(terminate_check && agent_age > options->terminate_age) {
97        TRACE("Interaction looked broken; terminate age exceeded.\n", "main");
98        break;
99      }
100
101     long cycle_start = time(NULL);
102
103     u32 observation = environment->_observation;
104     u32 reward = environment->_reward;
105
106     if(options->learning_period > 0 && cycle > options->learning_period) {
107       explore = false;
108     }
109
110     Agent_model_update_percept(agent, observation, reward);
111
112     bool explored =  false;
113
114     u32 best_action = 0;
115     double best_mean = -1;
116     u32 action;
117     if (explore && _random_0_1() < options->exploration) {
118       explored =  true;
119       printf("Agent is trying action at random...\n");
120       action = Agent_generate_random_action(agent);
121     } else {
122       ctw_save(agent->context_tree, CTW_DATA_FILE);
123       int i;
124       for (i = 1; i < P; i++) {
125   int data = 1;
126   MPI_Send(&data, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
127       }
128       for (i = 1; i < P; i++) {
129   double mean;
130   MPI_Recv(&action, 1, MPI_INT, i, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
131   MPI_Recv(&mean, 1, MPI_DOUBLE, i, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
132   if (mean > best_mean) {
133     best_action = action;
134     best_mean = mean;
135   }
136       }
137     }
138
139
140     perform_action(environment, action);
141
142     Agent_model_update_action(agent, action);
143
```

```
144     long ticks_taken = time(NULL) - cycle_start;
145
146     if (cycle % 1 == 0) {
147       printf("%-12s%-12s%-12s%-12s%-12s%-12s%-12s%-12s%-12s\n", "Cycle", "
        Observe.", "Reward", "Action", "Explored", "Exp. Rate", "Tot. Reward", "Avg
        Reward", "Time", "Model Size");
148     }   // Just a large padded statement about what is going on in the world as we
        step through
149
150     printf("%-12d%-12u%-12u%-12u%-12d%-12f%-12f%-12f%-12lu%-12u\n", cycle,
        observation, reward, action, explored, options->exploration, agent->total_reward
        , Agent_average_reward(agent), ticks_taken, ctw_size(agent->context_tree));
151
152     if(explore) {
153       options->exploration *= options->explore_decay;
154     }
155
156     cycle++;
157
158     isEnvironmentFinished = environment->_is_finished;
159   }
160 }
161
162 void search_proc(int rank, int P, Agent* agent, struct Environment* environment,
        app_options* options) {
163   for (;;) {
164     int data;
165     MPI_Recv(&data, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
166
167     ctw_load(agent->context_tree, CTW_DATA_FILE);
168     double mean;
169     int action = Agent_search_mean(agent, &mean);
170
171     MPI_Send(&action, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
172     MPI_Send(&mean, 1, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
173   }
174 }
175 #endif
176 void _interaction_loop(Agent* agent, struct Environment* environment, app_options*
        options) {
177     printf("MC AIXI training warming up...\n");
178     srand(1337);
179
180     bool explore =  options->exploration > 0;
181     if(0.0f > options->exploration || 0.0f > options->explore_decay || options->
        explore_decay > 1.0f) {
182         printf("Some exploration parameter is invalid. Application force quitting.\n
        ");
183         exit(1);
184     }
185
186     bool terminate_check = options->terminate_age > 0;
187     bool isEnvironmentFinished = false;
188     int cycle = 1;
189
190
191     while(!isEnvironmentFinished) {
192         int agent_age = agent->age;
```

```
193
194         if(terminate_check && agent_age > options->terminate_age) {
195             TRACE("Interaction looked broken; terminate age exceeded.\n", "main");
196             break;
197         }
198
199         long cycle_start = time(NULL);
200
201   u32 observation = environment->_observation;
202         u32 reward = environment->_reward;
203
204         if(options->learning_period > 0 && cycle > options->learning_period) {
205             explore = false;
206         }
207
208         Agent_model_update_percept(agent, observation, reward);
209
210         bool explored =  false;
211
212       u32 action;
213         if (explore && _random_0_1() < options->exploration) {
214       explored =  true;
215       printf("Agent is trying action at random...\n");
216       action = Agent_generate_random_action(agent);
217   } else {
218       action = Agent_search(agent);
219   }
220
221         perform_action(environment, action);
222
223         Agent_model_update_action(agent, action);
224
225         long ticks_taken = time(NULL) - cycle_start;
226
227   if (cycle % 1 == 0) {
228     printf("%-12s%-12s%-12s%-12s%-12s%-12s%-12s%-12s%-12s%-12s\n", "Cycle", "Observe
      .", "Reward", "Action", "Explored", "Exp. Rate", "Tot. Reward", "Avg Reward", "
      Time", "Model Size");
229     }   // Just a large padded statement about what is going on in the world as we
      step through
230         //printf("%-15s%-15s%-15s%-15s%-15s%-15s%-15s%-15s%-15s%-15s\n", "Cycle", "
      Observation", "Reward", "Action", "Explored", "Explore Rate", "Total Reward", "
      Average Reward", "Time", "Model Size");
231         printf("%-12d%-12u%-12u%-12u%-12d%-12f%-12f%-12f%-12lu%-12u\n", cycle,
      environment->_observation, environment->_reward, action, explored, options->
      exploration, agent->total_reward, Agent_average_reward(agent), ticks_taken,
      ctw_size(agent->context_tree));
232
233         if(explore) {
234             options->exploration *= options->explore_decay;
235         }
236
237         cycle++;
238
239         isEnvironmentFinished = environment->_is_finished;
240     }
241
242 }
```

```
243
244 int main(int argc, const char* argv[]) {
245     app_options* appOptions = _make_default_options();
246
247     printf("Booting MC AIXI kernel...\n");
248
249     TRACE("Creating environment...\n", "desu");
250
251     struct Coin_Flip* environment = new (Coin_Flip, 0.9f);
252
253     TRACE("Creating agent... please be patient\n", "desu");
254
255     TRACE("allocate agent\n", "main");
256     Agent* agent = malloc(sizeof(Agent));
257     TRACE("init agent\n", "main");
258     agent = Agent_init(agent, environment, appOptions->learning_period);
259
260
261     if(appOptions->profile) {
262         printf("Profiling is not currently supported. Ignoring.\n");
263     }
264
265 #ifndef USE_MPI
266     _interaction_loop(agent, (struct Environment *) environment, appOptions);
267 #else
268     mpi_main(agent, environment, appOptions, argc, argv);
269 #endif
270 }
```

Listing 5.1: main.c

```
1  #include <stdint.h>
2  #include <stdbool.h>
3  #include <assert.h>
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include "bit_vector.h"
7
8  BitVector *bv_create() {
9    BitVector *bit_vector = (BitVector *) malloc(sizeof(BitVector));
10   bit_vector->size = 0;
11   bit_vector->capacity = DEFAULT_BIT_VECTOR_CAPACITY;
12   bool *bits = (bool *) malloc(bit_vector->capacity * sizeof(bool));
13   assert(bits != NULL);
14   bit_vector->bits = bits;
15   return bit_vector;
16 }
17
18 void bv_free(BitVector *bv) {
19   free(bv->bits);
20   free(bv);
21 }
22
23 BitVector *bv_from_char(char c) {
24   BitVector *bv = bv_create();
25   int64_t j;
26   for (j=sizeof(char) * 8 - 1; j >= 0; j--) {
27     bv_push(bv, (c >> j) % 2 == 1);
```

```
28    }
29    return bv;
30  }
31
32  BitVector *bv_from_uint32(uint32_t v) {
33    BitVector *bv = bv_create();
34    int64_t j;
35    for (j=sizeof(uint32_t) * 8 - 1; j >= 0; j--) {
36      bv_push(bv, (v >> j) % 2 == 1);
37    }
38    return bv;
39  }
40
41  BitVector *bv_from_uint64(uint64_t v) {
42    BitVector *bv = bv_create();
43    int64_t j;
44    for (j=sizeof(uint64_t) * 8 - 1; j >= 0; j--) {
45      bv_push(bv, (v >> j) % 2 == 1);
46    }
47    return bv;
48  }
49
50  BitVector *bv_from_double(double d) {
51    // interpret the double bits as an int
52    uint64_t v = *((uint64_t *) &d);
53    return bv_from_uint64(v);
54  }
55
56  BitVector *bv_from_float(float f) {
57    // interpret the float bits as an int
58    uint32_t v = *((uint32_t *) &f);
59    return bv_from_uint32(v);
60  }
61
62  uint32_t bv_peek_uint32(BitVector *bv) {
63    uint32_t result = 0;
64    int64_t i;
65    for (i = sizeof(uint32_t) * 8; i > 0; i--) {
66      result = result << 1;
67      if (bv_test(bv, bv->size - i)) {
68        result += 1;
69      }
70    }
71    return result;
72  }
73
74  uint64_t bv_peek_uint64(BitVector *bv) {
75    uint64_t result = 0;
76    int64_t i;
77    for (i = sizeof(uint64_t) * 8; i > 0; i--) {
78      result = result << 1;
79      if (bv_test(bv, bv->size - i)) {
80        result += 1;
81      }
82    }
83    return result;
84  }
85
```

```
86 void bv_append(BitVector *a, BitVector *b) {
87   uint64_t i;
88   for (i = 0; i < b->size; i++) {
89     bv_push(a, bv_test(b, i));
90   }
91 }
92
93 void __bv_check_bounds(BitVector *bv, uint64_t index) {
94   if (index >= bv->size) {
95     printf("BV, Index out of bounds, index: %llu size: %llu\n", index, bv->size);
96     assert(false);
97   }
98 }
99
100 void __bv_grow(BitVector *bv) {
101   uint64_t new_capacity = bv->capacity * BIT_VECTOR_GROW_RATE;
102   bool *new_bits = (bool *) malloc(new_capacity * sizeof(bool));
103   uint64_t i;
104   for (i = 0; i < bv->size; i++) {
105     new_bits[i] = bv->bits[i];
106   }
107
108   free(bv->bits);
109   bv->capacity = new_capacity;
110   bv->bits = new_bits;
111 }
112
113 bool bv_test(BitVector *bv, uint64_t index) {
114   __bv_check_bounds(bv, index);
115   return bv->bits[index];
116 }
117
118 void bv_set(BitVector *bv, uint64_t index, bool bit) {
119   __bv_check_bounds(bv, index);
120   bv->bits[index] = bit;
121 }
122
123 void bv_push(BitVector *bv, bool bit) {
124   if (bv->size == bv->capacity) {
125     __bv_grow(bv);
126   }
127   bv->bits[bv->size] = bit;
128   bv->size += 1;
129 }
130
131 bool bv_peek(BitVector *bv) {
132   assert(bv->size > 0);
133   return bv->bits[bv->size-1];
134 }
135
136 bool bv_pop(BitVector *bv) {
137   assert(bv->size > 0);
138   bool bit = bv->bits[bv->size-1];
139   bv->size -= 1;
140   return bit;
141 }
142
143 void bv_clear(BitVector *bv) {
```

```
144
145   bv->size = 0;
146 }
147
148 void bv_print(BitVector *bv) {
149   printf("Size: %llu\n", bv->size);
150   if (bv->size > 400) {
151     return;
152   }
153   uint64_t i;
154   for (i = 0; i < bv->size; i++) {
155     if (bv_test(bv, i)) {
156       printf("1");
157     } else {
158       printf("0");
159     }
160   }
161   printf("\n");
162 }
163
164 void bv_print_ascii(BitVector *bv) {
165   uint64_t i;
166   for (i = 0; i < bv->size - (bv->size % 8); i += 8) {
167     char c = 0;
168     uint64_t j;
169     for (j = 0; j < 8; j++) {
170       c = c << 1;
171       if (bv_test(bv, j+i)) {
172   c += 1;
173       }
174     }
175     printf("%c", c);
176   }
177   printf("\n");
178 }
179
180 void bv_save(BitVector *bv, FILE *fp) {
181   fwrite(&(bv->size), sizeof(uint64_t), 1, fp);
182   uint64_t i;
183   for (i = 0; i < bv->size; i++) {
184     bool bit = bv_test(bv, i);
185     fwrite(&bit, sizeof(bool), 1, fp);
186   }
187 }
188
189 void bv_load(BitVector *bv, FILE *fp) {
190   bv_clear(bv);
191   uint64_t size;
192   fread(&size, sizeof(uint64_t), 1, fp);
193   uint64_t i;
194   for (i = 0; i < size; i++) {
195     bool bit;
196     fread(&bit, sizeof(bool), 1, fp);
197     bv_push(bv, bit);
198   }
199 }
200
201 BitVector *bv_slice(BitVector *bv, uint64_t start, uint64_t end) {
```

```
202   BitVector *slice = bv_create();
203   for (uint64_t i = start; i < end; i++) {
204     bv_push(slice, bv_test(bv, i));
205   }
206   return slice;
207 }
```

Listing 5.2: bit$_v$*ector.c*

```
1  #ifndef _BIT_VECTOR_
2  #define _BIT_VECTOR_
3
4  #include <stdbool.h>
5  #include <stdio.h>
6  #include <stdint.h>
7
8  #define DEFAULT_BIT_VECTOR_CAPACITY 16
9  #define BIT_VECTOR_GROW_RATE 2
10
11 typedef struct BitVector {
12   uint64_t size;
13   uint64_t capacity;
14   bool *bits;
15 } BitVector;
16
17 BitVector *bv_create();
18
19 void bv_free(BitVector *);
20
21 BitVector *bv_from_char(char);
22
23 BitVector *bv_from_uint32(uint32_t);
24
25 BitVector *bv_from_uint64(uint64_t);
26
27 void bv_append(BitVector *, BitVector *);
28
29 uint32_t bv_peek_uint32(BitVector *);
30
31 uint64_t bv_peek_uint64(BitVector *);
32
33 void __bv_check_bounds(BitVector *, uint64_t);
34
35 void __bv_grow(BitVector *);
36
37 bool bv_test(BitVector *, uint64_t);
38
39 void bv_set(BitVector *, uint64_t, bool);
40
41 void bv_push(BitVector *, bool);
42
43 bool bv_peek(BitVector *);
44
45 bool bv_pop(BitVector *);
46
47 void bv_clear(BitVector *);
48
49 void bv_print(BitVector *);
```

```
50
51 void bv_print_ascii(BitVector *);
52
53 void bv_save(BitVector *, FILE *);
54
55 void bv_load(BitVector *, FILE *);
56
57 BitVector *bv_slice(BitVector *bv, uint64_t start, uint64_t end);
58
59 #endif
```

Listing 5.3: bit$_v$*ector.h*

5.2 Object

```
1 #ifndef NEW_H
2 #define NEW_H
3
4 void * new      ( const void * class, ... );
5 void delete     ( void * item );
6 void * cpy      ( const void * self );
7 char * print    ( const void * self );
8
9 #endif
```

Listing 5.4: class.h

```
1 #ifndef CLASS_R
2 #define CLASS_R
3
4 #include <stdarg.h>
5 #include "../_utils/types.h"
6
7 struct Class {
8     size_t  size;
9     void *  ( * __init__ )          ( void * self, va_list      args );
10    void *  ( * __delete__ )        ( void * self );
11    void *  ( * __copy__ )          ( const void * self );
12    void *  ( * __str__ )           ( const void * self );
13 };
14
15
16
17 #endif
```

Listing 5.5: class.r

```
1 #include <assert.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include "class.h"
5 #include "class.r"
6 #include "../_utils/macros.h"
7
```

```
8  void * new ( const void * _class, ... )
9  {
10     const struct Class * class = _class;
11     void * mem = calloc( 1, class->size );
12
13     assert ( mem );
14
15     * ( const struct Class ** ) mem = class;
16
17     // This handles and vars passed to the constructor.
18     if ( class->__init__ )
19     {
20         va_list args;
21         #ifdef DEBUG
22             TRACE("Class Created", "__init__",class->__str__);
23         #endif
24         va_start( args, _class );                    // intialize '...'
25       mem = class->__init__( mem, &args );         // call constructor
26         va_end( args );                              // clean
27     }
28
29     return mem;
30 }
31
32 void delete ( void * self )
33 {
34     const struct Class ** parent = self;
35
36     if ( self && * parent && ( * parent )->__delete__ ) {
37         self = ( * parent )->__delete__(self);
38         #ifdef DEBUG
39             TRACE("Class Destroyed","delete(...)",class->__str__);
40         #endif
41     }
42     free(self);
43 }
44
45 void * cpy ( const void * self )
46 {
47     const struct Class * const * parent = self;
48
49     assert ( self && parent && ( * parent )->__copy__);
50
51     #ifdef DEBUG
52         TRACE("Class Copied","cpy(..)",class->__str__);
53     #endif
54
55     return ( * parent )->__copy__(self);
56 }
57
58 char * print ( const void * self )
59 {
60     const struct Class * const * parent = self;
61     char * pstring = malloc ( 255 * (sizeof(char)));
62     sprintf(pstring,"%s",( * parent )->__str__(self));
63
64     #ifdef DEBUG
65         TRACE ( "Generated Print String: ", "print(...)", pstring );
```

```
66      #endif
67
68      return pstring;
69  }
```

Listing 5.6: class.c

5.3 Agent

```
1   #ifndef AGENT_H
2       #define AGENT_H
3       #include <stdarg.h>
4       #include "../_utils/types.h"
5       #include "../bit_vector.h"
6       #include "../predict/context_tree.h"
7
8       typedef struct Agent
9       {
10          struct Environment * environment;
11          va_list              _options;
12          double               total_reward;
13          update_enum          last_update;
14          u32                  age;
15          u32                  horizon;
16          u32                  learning_period;
17          ContextTree*         context_tree;
18      } Agent;
19
20      Agent* Agent_init ( Agent* self, void * _env, u32 learn  );
21      AgentUndo* Agent_clone_into_temp                    (Agent* self);
22
23      double  Agent_average_reward                        ( Agent* self);
24
25      u32  Agent_generate_random_action                   ( Agent* self);
26      u32  Agent_maximum_action                           ( Agent* self  );
27      u32  Agent_maximum_reward                           ( Agent* self  );
28      u32  Agent_model_size                               ( Agent* self  );
29
30      void  Agent_model_update_action                     ( Agent* self, u32 action );
31      BitVector * Agent_encode_action                     ( Agent* self , u32 action );
32
33      // decoding
34      u32 Agent_decode_action                             (Agent* self , BitVector*
        symbols);
35      u32 Agent_decode_observation                        (Agent* self , BitVector*
        symbols);
36      u32 Agent_decode_reward                             (Agent* self, BitVector*
        symbols);
37      u32Tuple* Agent_decode_percept                      (Agent* self , BitVector*
        symbols);
38
39      // generators
40      u32 Agent_generate_action                           (Agent* self);
41      u32Tuple * Agent_generate_percept                          (Agent* self );
42      u32Tuple * Agent_generate_percept_and_update               (Agent* self );
```

```
43
44      u32 Agent_history_size                                  (Agent* self );
45      double Agent_get_predicted_action_probability (Agent* self , u32 action);
46      u32 Agent_maximum_bits_needed                           (Agent* self );
47
48      void Agent_model_revert                                 (Agent* self , AgentUndo*
        undo);
49      void Agent_model_update_percept                         ( Agent* self , u32
        observation, u32 reward );
50
51
52      BitVector * Agent_encode_percept                        ( Agent* self, u32
        observation, u32 reward);
53
54      double Agent_percept_probability                        (Agent* self, u32 observation
        , u32 reward);
55      double Agent_playout                                   (Agent* self, u32 horizon);
56
57      u32 Agent_search                                       ( Agent* self);
58      u32 Agent_search_mean                                  ( Agent* self, double *mean);
59      void  Agent_reset                                       ( Agent* self );
60  #endif
```

Listing 5.7: agent.h

```
1  #include <stddef.h>
2  #include <stdarg.h>
3  #include <stdlib.h>
4  #include "../_utils/types.h"
5  #include "../_object/class.h"
6  #include "../predict/context_tree.h"
7  #include "../bit_vector.h"
8  #include "../environment/environment.r"
9  #include "../environment/environment.h"
10 #include "agent.h"
11 #include "../search/monte_node.h"
12 #include "../_utils/macros.h"
13 #include <assert.h>
14
15 Agent* Agent_init ( Agent* self, void * _env, u32 learn  ) {
16     const struct Coin_Flip * env = _env;
17     TRACE("Prepping to build agent\n", "agent");
18     self -> environment = cpy ( env );
19     self -> age = 0;
20     self -> learning_period = learn;
21     self -> last_update = action_update;
22     self -> total_reward = 0.0;
23     self -> horizon = 6;
24     u32 depth = 192;
25
26     TRACE("Building context tree for Agent", "agent");
27     self->context_tree = ctw_create(depth);
28
29     #ifdef DEBUG
30         TRACE("learning period = %d, horizon = %d, depth = %d", \
31                 learn,self->horizon,depth);
32     #endif
33     Agent_reset(self);
```

```
34      return self;
35  }
36
37  void Agent_delete ( void * _self ) {
38      int *t = malloc(1)
39      free(t)
40  }
41
42  AgentUndo* Agent_clone_into_temp(Agent* self) {
43    AgentUndo* undo = (AgentUndo *) malloc(sizeof(AgentUndo));
44    undo->age = self->age;
45    undo->total_reward = self->total_reward;
46    undo->history_size = Agent_history_size(self);
47    undo->last_update = self->last_update;
48    return undo;
49  }
50
51  u32 Agent_decode_action(Agent* self, BitVector* symbols) {
52    return bv_peek_uint32(symbols);
53  }
54
55  u32 Agent_decode_observation(Agent* self, BitVector* symbols) {
56    return bv_peek_uint32(symbols);
57  }
58
59  u32 Agent_decode_reward(Agent * self, BitVector* symbols) {
60    return bv_peek_uint32(symbols);
61  }
62
63  u32Tuple* Agent_decode_percept(Agent* self, BitVector* symbols) {
64    uint64_t i;
65    BitVector* reward_symbols = bv_create();
66    BitVector* observation_symbols = bv_create();
67
68    for (i = 0; i < 32; i++) {
69      bv_push(reward_symbols, bv_test(symbols, i));
70    }
71    for (i = 32; i < 64; i++) {
72      bv_push(observation_symbols, bv_test(symbols, i));
73    }
74
75    // Decode both
76    u32 reward = Agent_decode_reward(self, reward_symbols);
77    u32 observation = Agent_decode_observation(self, observation_symbols);
78    u32Tuple* tuple;
79    if (is_valid_reward(self->environment, reward) &&
80        is_valid_observation(self->environment, observation)) {
81      tuple = malloc(sizeof(u32Tuple));
82      tuple->first = observation;
83      tuple->second = reward;
84    } else {
85      tuple = NULL;
86    }
87
88    return tuple;
89  }
90
91  BitVector * Agent_encode_action(Agent* self, u32 action) {
```

```
92    return bv_from_uint32(action);
93  }
94
95  BitVector * Agent_encode_percept ( Agent * self, u32 observation, u32 reward) {
96    BitVector* a = bv_from_uint32(reward);
97    BitVector* b = bv_from_uint32(observation);
98    bv_append(a, b);
99    bv_free(b);
100   return a;
101 }
102
103 u32 Agent_generate_action(Agent* self) {
104   assert(self->last_update == percept_update);
105
106   BitVector* random = ctw_gen_random_symbols(self->context_tree, 32);
107   return Agent_decode_action(self, random);
108 }
109
110 u32Tuple* Agent_generate_percept(Agent* self) {
111   BitVector* random = ctw_gen_random_symbols_and_update(self->context_tree, 64);
112
113   u32Tuple *percept = Agent_decode_percept(self, random);
114   if (percept == NULL) {
115     percept = malloc(sizeof(u32Tuple));
116     percept->first = rand() % 2;
117     percept->second = rand() % 2;
118     ctw_revert(self->context_tree, 64);
119     BitVector* symbols = Agent_encode_percept(self, percept->first, percept->second)
      ;
120     ctw_update_vector(self->context_tree, symbols);
121     bv_free(symbols);
122   }
123   return percept;
124 }
125
126 u32Tuple * Agent_generate_percept_and_update(Agent*  self) {
127   u32Tuple* tuple = Agent_generate_percept(self);
128   self->total_reward += tuple->second;
129   self->last_update = percept_update;
130   return tuple;
131 }
132
133 double Agent_get_predicted_action_probability(Agent* self, u32 action) {
134   BitVector* symbols = Agent_encode_action(self, action);
135   return ctw_predict_vector(self->context_tree, symbols);
136 }
137
138 u32 Agent_history_size(Agent* self) {
139   return self->context_tree->history->size;
140 }
141
142 u32 Agent_maximum_bits_needed(Agent * self) {
143     return 32;
144 }
145
146 void Agent_model_revert(Agent * self, AgentUndo* undo) {
147   while(Agent_history_size(self) > undo->history_size)  {
148     if(self->last_update == percept_update){
```

```
149       ctw_revert(self->context_tree, 32);
150       self->last_update = action_update;
151     } else {
152       ctw_revert_history(self->context_tree, 32);
153       self->last_update = percept_update;
154     }
155   }
156
157   if (Agent_history_size(self) != undo->history_size) {
158     printf("hist size should be equal %u %u\n", Agent_history_size(self), undo->
      history_size);
159     exit(1034109);
160   }
161
162   self->age = undo->age;
163   self->total_reward = undo->total_reward;
164   self->last_update = undo->last_update;
165 }
166
167 u32 Agent_model_size ( Agent* self) {
168   return ctw_size(self->context_tree);
169 }
170
171 void Agent_model_update_action ( Agent* self, u32 action) {
172   BitVector* action_symbols = Agent_encode_action(self, action);
173   ctw_update_history(self->context_tree, action_symbols);
174   self->age++;
175   self->last_update = action_update;
176 }
177
178 void Agent_model_update_percept ( Agent * self, u32 observation, u32 reward ) {
179   BitVector* symbols = Agent_encode_percept(self, observation, reward);
180
181   if((self->learning_period > 0 ) && (self->age > self->learning_period)) {
182     printf("not learning any more\n");
183     ctw_update_history(self->context_tree, symbols);
184   } else {
185     ctw_update_vector(self->context_tree, symbols);
186   }
187
188   self->total_reward += reward;
189   self->last_update = percept_update;
190 }
191
192 double Agent_percept_probability(Agent* self, u32 observation, u32 reward) {
193   BitVector* symbols = Agent_encode_percept(self, observation, reward);
194   return ctw_predict_vector(self->context_tree, symbols);
195 }
196
197 double Agent_playout(Agent* self, u32 horizon) {
198   double total_reward = 0;
199
200   for(u32 i = 0; i < horizon; i++) {
201     u32 action = Agent_generate_random_action(self);
202     Agent_model_update_action(self, action);
203
204     u32Tuple* tuple = Agent_generate_percept_and_update(self);
205     total_reward += tuple->second;
```

```
206   }
207
208   return total_reward;
209 }
210
211 void Agent_reset ( Agent* self ) {
212   ctw_clear(self->context_tree);
213
214   self->age = 0;
215   self->total_reward = 0.0;
216   self->last_update = action_update;
217 }
218
219 u32 Agent_search_mean(Agent* self, double *mean) {
220
221   printf("start search\n");
222   AgentUndo* undo = Agent_clone_into_temp(self);
223
224   MonteNode* node = monte_create_tree(NODE_TYPE_DECISION);
225
226   printf("start sampling\n");
227   for(u32 i = 0; i < 50;i++ ) {
228     monte_sample(node, self, self->horizon);
229     Agent_model_revert(self, undo);
230   }
231
232   printf("done sampling\n");
233   u32 best_action = Agent_generate_random_action(self);
234   double best_mean = -1;
235
236   for(u32 i = 0; i < self->environment->num_actions; i++) {
237     u32 action =  self->environment->_valid_actions[i];
238     MonteNode* searchNode = dict_find(node->actions, action);
239
240     if(searchNode != NULL) {
241       double mean = searchNode->mean + ((float)rand()/(float)(RAND_MAX)) * 0.0001;
242       if(mean > best_mean) {
243   best_mean = mean;
244   best_action = action;
245       }
246     }
247   }
248
249   printf("done search\n");
250   *mean = best_mean;
251   return best_action;
252 }
253
254 u32 Agent_search(Agent* self) {
255   double mean;
256   u32 action = Agent_search_mean(self, &mean);
257   return action;
258 }
259
260 double Agent_average_reward ( Agent * self) {
261   double average = 0.0;
262   if ( self -> age > 0 )
263     average = ( self -> total_reward ) / ( self->age );
```

```
264   return average;
265 }
266
267 u32 Agent_generate_random_action ( Agent * self) {
268   int actionIndex = rand() % self->environment->num_actions;
269   return self->environment->_valid_actions[actionIndex];
270 }
271
272 u32 Agent_maximum_action ( Agent* self) {
273   return maximum_action(self->environment);
274 }
275
276 u32 Agent_maximum_reward ( Agent* self)
277 {
278   return maximum_reward(self->environment);
279 }
```

Listing 5.8: agent.c

5.4 Search

```
1  #ifndef _SEARCH_C
2  #define _SEARCH_C
3
4
5  #include <stdlib.h>
6  #include <stdbool.h>
7  #include <stdio.h>
8  #include <math.h>
9  #include <float.h>
10 #include "dict.h"
11 #include "monte_node.h"
12 #include "../agent/agent.h"
13 #include "../environment/environment.r"
14 #include "../_utils/types.h"
15 #include "../_utils/macros.h"
16
17 #define ARC4RANDOM_MAX      0x100000000
18
19 #define MONTE_UNEXPLORED_BIAS 100000000000.0f
20
21
22 MonteNode* monte_create_tree(u32 nodeType) {
23     MonteNode* root = (MonteNode* ) malloc(sizeof(MonteNode));
24
25     root->mean = 0.0;
26     root->type = nodeType;
27     root->visits = 0;
28
29     root->children = dict_new();
30     root->actions = dict_new();
31
32     return root;
33 }
34
```

```
35 u32 _monte_select_action(MonteNode* tree, Agent* agent) {
36     // returns -1 if no vaild action
37 
38     float agent_horizon = agent->horizon;
39     float agent_max_reward = Agent_maximum_reward(agent)/1;
40 
41     float explore_bias = agent_horizon * agent_max_reward;
42     float exploration_numerator = (float) (2.0f* log((double) tree->visits));
43 
44 
45     // desu??? Mondaiji-tachi ga Isekai kara Kuru Sou Desu yo?
46     u32 best_action = 0;
47     double best_priority = -FLT_MAX;
48     u32 i = 0;
49     for(i = 0; i < agent->environment->num_actions; i++) {
50         u32 action = agent->environment->_valid_actions[i];
51         MonteNode* node = dict_find(tree->actions, action);
52 
53         double priority = 0;
54 
55         if(node == NULL  || node->visits == 0)  {
56             priority = MONTE_UNEXPLORED_BIAS;
57         } else {
58           priority = node->mean - (explore_bias * sqrt(exploration_numerator / node
    ->visits));
59         }
60 
61         if(priority > (best_priority + ((float)rand()/(float)(RAND_MAX)) * 0.001)) {
62             best_action = action;
63             best_priority = priority;
64         }
65 
66     }
67     return best_action;
68 }
69 
70 
71 float monte_sample(MonteNode* tree, Agent* agent, u32 horizon) {
72     double reward = 0.0;
73 
74     if(horizon == 0) {
75         return reward;
76     } else if(tree->type == NODE_TYPE_CHANCE) {
77         u32Tuple* tuple = Agent_generate_percept_and_update(agent);
78 
79         u32 observation = tuple->first;
80         u32 random_reward = tuple->second;
81 
82         bool notInTreeYet = dict_find(tree->children, observation) == NULL;
83 
84         if(notInTreeYet) {
85             MonteNode* newChild = monte_create_tree(NODE_TYPE_DECISION);
86             dict_add(tree->children, observation, newChild);
87         }
88 
89         MonteNode* child = dict_find(tree->children, observation);
90 
91         if(child == NULL) {
```

```
92              printf("wtf?? child was not found.. abort!\n");
93              exit(1337);
94          }
95
96          reward = random_reward + monte_sample(child, agent, horizon - 1);
97      } else if (tree->visits == 0) {
98          reward = Agent_playout(agent, horizon);
99      }
100     else {
101         u32 action = _monte_select_action(tree, agent);
102         Agent_model_update_action(agent, action);
103
104         MonteNode* child = dict_find(tree->actions, action);
105         if(child == NULL) {
106             child = monte_create_tree(NODE_TYPE_CHANCE);
107           dict_add(tree->actions, action, child);
108         }
109
110         if(child == NULL) {
111             printf("wtf??? child was not found.. abort!\n");
112             exit(1338);
113         }
114
115         reward = monte_sample(child, agent, horizon);
116     }
117     tree->mean = (reward + (tree->visits * tree->mean)) / (tree->visits + 1.0);
118     tree->visits = tree->visits + 1;
119
120     return reward;
121 }
122
123 #endif
```

Listing 5.9: search.c

```
1  #ifndef _MONTE_NODE_H_
2  #define _MONTE_NODE_H_
3
4  #define NODE_TYPE_CHANCE 0
5  #define NODE_TYPE_DECISION 1
6
7  #include "dict.h"
8  #include "../agent/agent.h"
9  #include "../_utils/types.h"
10
11 typedef struct MonteNode {
12     double mean;
13     int type;
14     int visits;
15
16     dict_t children;
17     dict_t actions;
18 } MonteNode;
19
20 MonteNode* monte_create_tree(u32 nodeType);
21 u32 _monte_select_action(MonteNode* tree, Agent* agent);
22 float monte_sample(MonteNode* tree, Agent* agent, u32 horizon);
23
```

```
24
25 #endif
```

Listing 5.10: monte$_n$*ode.h*

```
1  #ifndef _DICT_H_
2  #define _DICT_H_
3
4  #include <stddef.h>
5  #include <stdlib.h>
6
7  typedef struct MonteNode MonteNode;
8
9  typedef struct dict_entry_s {
10     int key;
11     MonteNode* value;
12 } dict_entry_s;
13
14 typedef struct dict_s {
15     int len;
16     int cap;
17     dict_entry_s *entry;
18 } dict_s, *dict_t;
19
20 int dict_find_index(dict_t, const int);
21
22 MonteNode* dict_find(dict_t, const int);
23
24 void dict_add(dict_t, const int, MonteNode*);
25
26 dict_t dict_new(void);
27
28 void dict_free(dict_t);
29
30 #endif
```

Listing 5.11: dict.h

```
1  #include <stddef.h>
2  #include <stdlib.h>
3  #include "dict.h"
4  #include "monte_node.h"
5
6  int dict_find_index(dict_t dict, const int key) {
7      for (int i = 0; i < dict->len; i++) {
8          if (dict->entry[i].key == key) {
9              return i;
10         }
11     }
12     return -1;
13 }
14
15 MonteNode* dict_find(dict_t dict, const int key) {
16     int idx = dict_find_index(dict, key);
17     return idx == -1 ? NULL : dict->entry[idx].value;
18 }
19
```

```
20 void dict_add(dict_t dict, const int key, MonteNode* value) {
21     int idx = dict_find_index(dict, key);
22     if (idx != -1) {
23         dict->entry[idx].value = value;
24         return;
25     }
26     if (dict->len == dict->cap) {
27         dict->cap *= 2;
28         dict->entry = realloc(dict->entry, dict->cap * sizeof(dict_entry_s));
29     }
30     dict->entry[dict->len].key = key;
31     dict->entry[dict->len].value = value;
32     dict->len++;
33 }
34
35 dict_t dict_new(void) {
36     dict_s proto = {0, 10, malloc(10 * sizeof(dict_entry_s))};
37     dict_t d = malloc(sizeof(dict_s));
38     *d = proto;
39     return d;
40 }
41
42 void dict_free(dict_t dict) {
43
44     for (int i = 0; i < dict->len; i++) {
45         free(dict->entry[i].value);
46     }
47
48     free(dict->entry);
49     free(dict);
50 }
```

Listing 5.12: dict.c

5.5 Predict

```
1 #ifndef _CTW_TREE_
2 #define _CTW_TREE_
3
4 #include "context_tree_node.h"
5 #include "ctw_list.h"
6
7 typedef struct ContextTree {
8   uint32_t depth;
9   ContextTreeNode *root;
10  BitVector *history;
11  CTWNodeList *context;
12 } ContextTree;
13
14 ContextTree *ctw_create(uint32_t);
15
16 void ctw_free(ContextTree *);
17
18 void ctw_clear(ContextTree *);
19
```

```
20 uint64_t ctw_size(ContextTree *);
21
22 void ctw_print(ContextTree *);
23
24 void ctw_update_context(ContextTree *);
25
26 void ctw_revert(ContextTree *, uint64_t);
27
28 void ctw_update_symbol(ContextTree *, bool);
29
30 void ctw_update_vector(ContextTree *, BitVector *);
31
32 void ctw_update_history(ContextTree *, BitVector *);
33
34 double ctw_predict_symbol(ContextTree *, bool);
35
36 double ctw_predict_vector(ContextTree *, BitVector *);
37
38 BitVector *ctw_gen_random_symbols_and_update(ContextTree *, uint64_t);
39
40 BitVector *ctw_gen_random_symbols(ContextTree *, uint64_t);
41
42 void ctw_revert_history(ContextTree *, uint64_t);
43
44 void ctw_save(ContextTree *, char *);
45
46 void ctw_load(ContextTree *, char *);
47
48 #endif
```

Listing 5.13: context$_t$*ree.h*

```
1 #include <math.h>
2 #include <stdint.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <errno.h>
6 #include "../bit_vector.h"
7 #include "ctw_list.h"
8 #include "context_tree.h"
9
10 ContextTree *ctw_create(uint32_t depth) {
11   ContextTree *tree = (ContextTree *) malloc(sizeof(ContextTree));
12   tree->depth = depth;
13   tree->root = ctw_node_create();
14   tree->history = bv_create();
15   tree->context = ctw_list_create();
16   return tree;
17 }
18
19 void ctw_free(ContextTree *tree) {
20   ctw_node_free(tree->root);
21   bv_free(tree->history);
22   ctw_list_free(tree->context);
23   free(tree);
24 }
25
26 void ctw_clear(ContextTree *tree) {
```

```
27    bv_clear(tree->history);
28    ctw_node_free(tree->root);
29    tree->root = ctw_node_create();
30    ctw_list_clear(tree->context);
31  }
32
33  uint64_t ctw_size(ContextTree *tree) {
34    return ctw_node_size(tree->root);
35  }
36
37  void ctw_print(ContextTree *tree) {
38    printf("Context Tree { depth: %d, size: %llu}\n", tree->depth, ctw_size(tree));
39    printf("History: ");
40    bv_print(tree->history);
41  }
42
43  void ctw_update_context(ContextTree *tree) {
44    if (tree->history->size < tree->depth) {
45      perror("Not enough history to update context\n");
46    }
47    ctw_list_free(tree->context);
48    tree->context = ctw_list_create();
49    ctw_list_push(tree->context, tree->root);
50    ContextTreeNode *node = tree->root;
51    uint64_t update_depth = 1;
52    int64_t i;
53    for (i = tree->history->size-1; i >= 0; i--) {
54      bool symbol = bv_test(tree->history, i);
55
56      if (symbol && node->one_child != NULL) {
57        node = node->one_child;
58      } else if (!symbol && node->zero_child != NULL) {
59        node = node->zero_child;
60      } else {
61        ContextTreeNode *new_node = ctw_node_create();
62        if (symbol) {
63  node->one_child = new_node;
64        } else {
65  node->zero_child = new_node;
66        }
67        node = new_node;
68      }
69      ctw_list_push(tree->context, node);
70      update_depth += 1;
71      if (update_depth > tree->depth) {
72        break;
73      }
74    }
75  }
76
77  void ctw_revert(ContextTree *tree, uint64_t n) {
78    uint64_t i;
79    for (i = 0; i < n; i++) {
80      if (tree->history->size == 0) {
81        return;
82      }
83
84      bool symbol = bv_pop(tree->history);
```

```
85
86      if (tree->history->size >= tree->depth) {
87        ctw_update_context(tree);
88
89        int64_t j;
90        for (j = tree->depth-1; j >= 0; j--) {
91    ctw_node_revert(ctw_list_get(tree->context, j), symbol);
92        }
93      }
94    }
95  }
96
97  void ctw_update_symbol(ContextTree *tree, bool symbol) {
98    if (tree->history->size >= tree->depth) {
99      ctw_update_context(tree);
100     int64_t i;
101     for (i = tree->depth-1; i >= 0; i--) {
102       ctw_node_update(ctw_list_get(tree->context, i), symbol);
103     }
104   }
105   bv_push(tree->history, symbol);
106 }
107
108 void ctw_update_vector(ContextTree *tree, BitVector *symbols) {
109   uint64_t i;
110   for (i = 0; i < symbols->size; i++) {
111     bool symbol = bv_test(symbols, i);
112     ctw_update_symbol(tree, symbol);
113   }
114 }
115
116 void ctw_update_history(ContextTree *tree, BitVector *symbols) {
117   bv_append(tree->history, symbols);
118 }
119
120 double ctw_predict_symbol(ContextTree *tree, bool symbol) {
121   if (tree->history->size + 1 <= tree->depth) {
122     return 0.5;
123   }
124   double prob_history = tree->root->log_probability;
125   ctw_update_symbol(tree, symbol);
126   double prob_sequence = tree->root->log_probability;
127   ctw_revert(tree, 1);
128   return exp(prob_sequence - prob_history);
129 }
130
131 double ctw_predict_vector(ContextTree *tree, BitVector *symbols) {
132   if (tree->history->size + symbols->size <= tree->depth) {
133     return pow(0.5, symbols->size);
134   }
135
136   double prob_history = tree->root->log_probability;
137   ctw_update_vector(tree, symbols);
138   double prob_sequence = tree->root->log_probability;
139   ctw_revert(tree, symbols->size);
140   return exp(prob_sequence - prob_history);
141 }
142
```

```
143 BitVector *ctw_gen_random_symbols_and_update(ContextTree *tree, uint64_t n) {
144   BitVector *symbols = bv_create();
145   uint64_t i;
146   for (i = 0; i < n; i++) {
147     double p = ((double) rand()) / ((double) RAND_MAX);
148     bool symbol;
149     if (p < ctw_predict_symbol(tree, true)) {
150       symbol = true;
151     } else {
152       symbol = false;
153     }
154     bv_push(symbols, symbol);
155     ctw_update_symbol(tree, symbol);
156   }
157   return symbols;
158 }
159
160 BitVector *ctw_gen_random_symbols(ContextTree *tree, uint64_t n) {
161   BitVector *symbols = ctw_gen_random_symbols_and_update(tree, n);
162   ctw_revert(tree, n);
163   return symbols;
164 }
165
166 void ctw_revert_history(ContextTree *tree, uint64_t n) {
167
168   if (tree->history->size < n) {
169     perror("not enough history to revert\n");
170   }
171   uint64_t i;
172   for (i = 0; i < n; i++) {
173     bv_pop(tree->history);
174   }
175 }
176
177 void ctw_save(ContextTree *tree, char *file_name) {
178   uint64_t count = 0;
179
180   FILE *fp = fopen(file_name, "w");
181   fwrite(&(tree->depth), sizeof(uint32_t), 1, fp);
182   bv_save(tree->history, fp);
183
184   CTWNodeList *stack = ctw_list_create();
185   ctw_list_push(stack, tree->root);
186   while (stack->size > 0) {
187     ContextTreeNode *node = ctw_list_pop(stack);
188     fwrite(&(node->log_kt), sizeof(double), 1, fp);
189     fwrite(&(node->log_probability), sizeof(double), 1, fp);
190     fwrite(&(node->ones_in_history), sizeof(uint32_t), 1, fp);
191     fwrite(&(node->zeroes_in_history), sizeof(uint32_t), 1, fp);
192
193     bool zero_child;
194     bool one_child;
195
196     if (node->one_child != NULL) {
197       ctw_list_push(stack, node->one_child);
198       one_child = true;
199     } else {
200       one_child = false;
```

```
201     }
202
203     if (node->zero_child != NULL) {
204       ctw_list_push(stack, node->zero_child);
205       zero_child = true;
206     } else {
207       zero_child = false;
208     }
209
210     fwrite(&zero_child, sizeof(bool), 1, fp);
211     fwrite(&one_child, sizeof(bool), 1, fp);
212     count += 1;
213   }
214   fclose(fp);
215   printf("Wrote %llu nodes to %s\n", count, file_name);
216 }
217
218 void ctw_load(ContextTree *tree, char *file_name) {
219   ctw_clear(tree);
220
221   uint64_t count = 0;
222   FILE *fp = fopen(file_name, "r");
223   fread(&(tree->depth), sizeof(uint32_t), 1, fp);
224   bv_load(tree->history, fp);
225   printf("Read history\n");
226
227   CTWNodeList *stack = ctw_list_create();
228   ContextTreeNode *parent = NULL;
229   do {
230     ContextTreeNode *node = ctw_node_create();
231     if (stack->size == 0 && parent == NULL) {
232       tree->root = node;
233     }
234     if (parent != NULL) {
235       parent->zero_child = node;
236     } else if (stack->size != 0) {
237       ContextTreeNode *one_parent = ctw_list_pop(stack);
238       one_parent->one_child = node;
239     }
240
241     fread(&(node->log_kt), sizeof(double), 1, fp);
242     fread(&(node->log_probability), sizeof(double), 1, fp);
243     fread(&(node->ones_in_history), sizeof(uint32_t), 1, fp);
244     fread(&(node->zeroes_in_history), sizeof(uint32_t), 1, fp);
245
246     bool zero_child;
247     fread(&zero_child, sizeof(bool), 1, fp);
248     if (zero_child) {
249       parent = node;
250     } else {
251       parent = NULL;
252     }
253
254     bool one_child;
255     fread(&one_child, sizeof(bool), 1, fp);
256     if (one_child) {
257       ctw_list_push(stack, node);
258     }
```

```
259
260     count += 1;
261   } while (stack->size > 0 || parent != NULL);
262   fclose(fp);
263
264   printf("Read %llu nodes to %s\n", count, file_name);
265 }
```

Listing 5.14: context$_t$ree.c

```
1 #ifndef _CTW_NODE_
2 #define _CTW_NODE_
3
4 #include <stdint.h>
5 #include <stdbool.h>
6 #include "../bit_vector.h"
7
8 typedef struct ContextTreeNode {
9   double log_kt;
10  double log_probability;
11  uint32_t ones_in_history;
12  uint32_t zeroes_in_history;
13  struct ContextTreeNode *zero_child;
14  struct ContextTreeNode *one_child;
15 } ContextTreeNode;
16
17 ContextTreeNode *ctw_node_create();
18
19 void ctw_node_free(ContextTreeNode *);
20
21 bool ctw_node_is_leaf(ContextTreeNode *);
22
23 uint32_t ctw_node_visits(ContextTreeNode *);
24
25 double ctw_node_log_kt_multiplier(ContextTreeNode *, bool);
26
27 void ctw_node_update_log_probability(ContextTreeNode *);
28
29 void ctw_node_revert(ContextTreeNode *, bool);
30
31 uint32_t ctw_node_size(ContextTreeNode *);
32
33 void ctw_node_update(ContextTreeNode *, bool);
34
35 void ctw_node_print(ContextTreeNode *);
36 #endif
```

Listing 5.15: context$_t$ree$_n$ode.h

```
1 #include <stdlib.h>
2 #include <math.h>
3 #include <stdint.h>
4 #include <stdio.h>
5 #include <stdbool.h>
6 #include <errno.h>
7 #include "context_tree_node.h"
8
```

```
9  ContextTreeNode *ctw_node_create() {
10   ContextTreeNode *node = (ContextTreeNode *) malloc(sizeof(ContextTreeNode));
11   node->log_kt = 0.0;
12   node->log_probability = 0.0;
13   node->ones_in_history = 0;
14   node->zeroes_in_history = 0;
15   node->zero_child = NULL;
16   node->one_child = NULL;
17   return node;
18 }
19
20 void ctw_node_free(ContextTreeNode *node) {
21   if (node == NULL) {
22     return;
23   }
24   ctw_node_free(node->zero_child);
25   ctw_node_free(node->one_child);
26   free(node);
27 }
28
29 bool ctw_node_is_leaf(ContextTreeNode *node) {
30   return node->zero_child == NULL && node->one_child == NULL;
31 }
32
33 uint32_t ctw_node_visits(ContextTreeNode *node) {
34   return node->ones_in_history + node->zeroes_in_history;
35 }
36
37 double ctw_node_log_kt_multiplier(ContextTreeNode *node, bool symbol) {
38   uint32_t numerator;
39   if (symbol) {
40     // symbol is 1
41     numerator = node->ones_in_history;
42   } else {
43     // symbol is 0
44     numerator = node->zeroes_in_history;
45   }
46
47   uint32_t denominator = ctw_node_visits(node);
48   return log((numerator + 0.5) / (denominator + 1.0));
49 }
50
51 void ctw_node_update_log_probability(ContextTreeNode *node) {
52   if (ctw_node_is_leaf(node)) {
53     node->log_probability = node->log_kt;
54   } else {
55     double log_child_prob = 0.0;
56     if (node->zero_child != NULL) {
57       log_child_prob += node->zero_child->log_probability;
58     }
59     if (node->one_child != NULL) {
60       log_child_prob += node->one_child->log_probability;
61     }
62
63     double a, b;
64     if (node->log_kt >= log_child_prob) {
65       a = node->log_kt;
66       b = log_child_prob;
```

```
67      } else {
68        a = log_child_prob;
69        b = node->log_kt;
70      }
71      node->log_probability = log(0.5) + a + log1p(exp(b - a));
72    }
73  }
74
75  void ctw_node_revert(ContextTreeNode *node, bool symbol) {
76    // This is called in a loop from leaf to root, so we know that the
77    // node's children have already been treated
78
79    if (symbol && node->ones_in_history > 0) {
80      // symbol is 1
81      node->ones_in_history -= 1;
82    } else if (!symbol && node->zeroes_in_history > 0) {
83      // symbol is 0
84      node->zeroes_in_history -= 1;
85    }
86
87    // need to remove redundant nodes, since this has already been called on
88    // the node's children, they may have 0 visits now
89    if (symbol) {
90      if (node->one_child != NULL && ctw_node_visits(node->one_child) == 0) {
91        free(node->one_child);
92        node->one_child = NULL;
93      }
94    } else {
95      if (node->zero_child != NULL && ctw_node_visits(node->zero_child) == 0) {
96        free(node->zero_child);
97        node->zero_child = NULL;
98      }
99    }
100
101   node->log_kt -= ctw_node_log_kt_multiplier(node, symbol);
102   ctw_node_update_log_probability(node);
103 }
104
105 uint32_t ctw_node_size(ContextTreeNode *node) {
106   uint32_t zero_size = 0;
107   if (node->zero_child != NULL) {
108     zero_size = ctw_node_size(node->zero_child);
109   }
110
111   uint32_t one_size = 0;
112   if (node->one_child != NULL) {
113     one_size = ctw_node_size(node->one_child);
114   }
115
116   return 1 + zero_size + one_size;
117 }
118
119 void ctw_node_update(ContextTreeNode *node, bool symbol) {
120   node->log_kt += ctw_node_log_kt_multiplier(node, symbol);
121
122   ctw_node_update_log_probability(node);
123
124   if (symbol) {
```

```
125     node->ones_in_history += 1;
126   } else {
127     node->zeroes_in_history += 1;
128   }
129 }
130
131 void ctw_node_print(ContextTreeNode *node) {
132   if (node == NULL) {
133     printf("CTWN NULL\n");
134     return;
135   }
136   printf("----------\n");
137   printf("0: %u 1: %u\n", node->zeroes_in_history, node->ones_in_history);
138   printf("probs: %f %f\n", node->log_probability, node->log_kt);
139   ctw_node_print(node->zero_child);
140   ctw_node_print(node->one_child);
141 }
```

Listing 5.16: context$_t$ree$_n$ode.c

```
1 #ifndef _CTW_LIST_
2 #define _CTW_LIST_
3
4 #include <stdint.h>
5 #include "context_tree_node.h"
6
7 typedef struct CTWNodeList {
8   uint64_t size;
9   uint64_t capacity;
10  ContextTreeNode **nodes;
11 } CTWNodeList;
12
13 CTWNodeList *ctw_list_create();
14
15 void ctw_list_free(CTWNodeList *);
16
17 void __ctw_list_check_bounds(CTWNodeList *, uint64_t);
18
19 void __ctw_list_grow(CTWNodeList *);
20
21 ContextTreeNode *ctw_list_get(CTWNodeList *, uint64_t);
22
23 void ctw_list_set(CTWNodeList *, uint64_t, ContextTreeNode *);
24
25 void ctw_list_push(CTWNodeList *, ContextTreeNode *);
26
27 ContextTreeNode *ctw_list_pop(CTWNodeList *);
28
29 void ctw_list_clear(CTWNodeList *);
30
31 #endif
```

Listing 5.17: ctw$_l$ist.h

```
1 #include <errno.h>
2 #include <stdint.h>
3 #include <stdio.h>
```

```c
#include <stdlib.h>
#include "context_tree_node.h"
#include "ctw_list.h"

CTWNodeList *ctw_list_create() {
  CTWNodeList *ctw_list = (CTWNodeList *) malloc(sizeof(CTWNodeList));
  ctw_list->size = 0;
  ctw_list->capacity = 8;
  ContextTreeNode **nodes = (ContextTreeNode **) malloc(ctw_list->capacity * sizeof(
    ContextTreeNode *));
  if (nodes == NULL) {
    perror("Failed to allocate nodes list");
  }

  ctw_list->nodes = nodes;
  return ctw_list;
}

void ctw_list_free(CTWNodeList *ctw_list) {
  // we do not free the nodes in the list, those nodes need to be freed from
  // the ContextTree
  free(ctw_list->nodes);
  free(ctw_list);
}

void __ctw_list_check_bounds(CTWNodeList *ctw_list, uint64_t index) {
  if (index >= ctw_list->size) {
    fprintf(stderr, "CT, Index out of bounds, index: %llu size: %llu\n", index,
    ctw_list->size);
    perror("Wowowowow. Out of bounds mannn");
  }
}

void __ctw_list_grow(CTWNodeList *ctw_list) {
  uint64_t new_capacity = ctw_list->capacity * 2;
  ContextTreeNode **new_nodes = (ContextTreeNode **) malloc(new_capacity * sizeof(
    ContextTreeNode *));
  uint64_t i;
  for (i = 0; i < ctw_list->size; i++) {
    new_nodes[i] = ctw_list->nodes[i];
  }

  free(ctw_list->nodes);
  ctw_list->capacity = new_capacity;
  ctw_list->nodes = new_nodes;
}

ContextTreeNode *ctw_list_get(CTWNodeList *ctw_list, uint64_t index) {
  __ctw_list_check_bounds(ctw_list, index);
  return ctw_list->nodes[index];
}

void ctw_list_set(CTWNodeList *ctw_list, uint64_t index, ContextTreeNode *node) {
  __ctw_list_check_bounds(ctw_list, index);
  ctw_list->nodes[index] = node;
}

void ctw_list_push(CTWNodeList *ctw_list, ContextTreeNode *node) {
```

```
59   if (ctw_list->size == ctw_list->capacity) {
60     __ctw_list_grow(ctw_list);
61   }
62   ctw_list->nodes[ctw_list->size] = node;
63   ctw_list->size += 1;
64 }
65
66 ContextTreeNode *ctw_list_pop(CTWNodeList *ctw_list) {
67   if (ctw_list->size == 0) {
68     perror("The list is empty, can't pop\n");
69   }
70   ContextTreeNode *node = ctw_list->nodes[ctw_list->size-1];
71   ctw_list->size -= 1;
72   return node;
73 }
74
75 void ctw_list_clear(CTWNodeList *ctw_list) {
76   // Just sets size to 0
77   // we may want to resize the array if memory is a problem
78   ctw_list->size = 0;
79 }
```

Listing 5.18: ctw$_l$*ist.c*

5.6 Environment

```
1  #ifndef ENV_H
2      #define ENV_H
3      #include "../_utils/types.h"
4
5      extern const void * Environment;
6
7      // define any functions here
8      u32  action_bits           ( void * _self );
9      u32  observation_bits      ( void * _self );
10     u32  reward_bits           ( void * _self );
11     u32  percption_bits        ( void * _self );
12
13     u08  is_valid_action       ( void * _self, u32   action );
14     u08  is_valid_observation  ( void * _self, u32   observation );
15     u08  is_valid_reward       ( void * _self, u32   reward );
16
17     u32  maximum_action        ( void * _self );
18     u32  maximum_observation   ( void * _self );
19     u32  maximum_reward        ( void * _self );
20
21     u32  minimum_action        ( void * _self );
22     u32  minimum_observation   ( void * _self );
23     u32  minimum_reward        ( void * _self );
24
25     u32 LG2 ( u32 x );
26
27 #endif
```

Listing 5.19: environment.h

```
#ifndef ENV_R
    #define ENV_R
    #include "../_utils/types.h"
    #include <stdarg.h>
    // Class Environment:
    struct Environment
    {
        const void * class; // must be first
        u32       _action;
        u08       _is_finished;
        u32       _observation;
        va_list   _options;
        u08       _reward;
        u32       *_valid_actions;
        u32       *_valid_observations;
        u32       *_valid_rewards;
        u32       num_actions;
    };//--------------------------------------------------

    #define action(e) (((const struct Environment *)(e)) -> _action)
    #define is_finished(e) ((const struct Environment *)(e)) -> _is_finished
    #define observation(e) (((const struct Environment *)(e)) -> _observation)
    #define reward(e) (((const struct Environment *)(e)) -> _reward)
    #define valid_actions(e) (((const struct Environment *)(e)) -> _valid_actions)
    #define valid_observations(e) (((const struct Environment *)(e)) ->
    _valid_observations)
    #define valid_rewards(e) (((const struct Environment *)(e)) -> _valid_rewards)

#endif
```

Listing 5.20: environment.r

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "../_utils/types.h"
#include "../_utils/macros.h"
#include "class.h"
#include "class.r"
#include "environment.h"
#include "environment.r"

// def __init__():
static void * Environment_init ( void * _self, va_list * args )
{
    struct Environment * self = _self;

    va_copy ( self -> _options, *args );
    self -> _is_finished   = 0x00;
    self -> _reward        = 0x00;
    self -> _action        = 0x00;

    #ifdef DEBUG
        TRACE("Environment initialized\n","Environment_init\n");
    #endif

    return self;
```

```
}//------------------------------------------------------------------------

// def __delete__():
static void * Environment_delete ( void * _self )
{
    struct Environment * self = _self;

    //free ( self->_options ),            self->_options = 0;
    //free ( self->_observation )
    //self->_observation = 0;
    //free ( self->_valid_observations )//, self->_valid_observations = 0;
    //free ( self->_valid_actions )//,      self->_valid_actions = 0;
    free ( self );

    #ifdef DEBUG
        TRACE("Environment destroyed\n","Environment_delete\n");
    #endif

    return self;
}//------------------------------------------------------------------------

// def secure-copy():
static void * Environment_cpy ( const void * _self )
{
    const struct Environment * self = _self;

    #ifdef DEBUG
        TRACE("Environment copied\n","Environment_cpy\n");
    #endif

    return new ( Environment , self->_options );
}//------------------------------------------------------------------------

// def ___str___():
static void * Environment_str ( const void * _self )
{
    const struct Environment * self = _self;

    // reserve 255 Characters for print string
    char *  pstring = malloc(sizeof(char) * 0xFF);

    sprintf ( pstring, "action = %x, observation = %x, reward = %x\n",
              self->_action, self->_observation, self->_reward );
    #ifdef DEBUG
        TRACE(pstring," Environment_init");
    #endif

    return pstring;
}//------------------------------------------------------------------------


static const struct Class _Environment = {
    sizeof(struct Environment),
    Environment_init,                   // done
    Environment_delete,                 // done
    Environment_cpy,                    // done
    Environment_str,                    // done
}; //-----------------------------------------------------
```

```
84
85 const void * Environment = & _Environment;
86
87 // def action_bits():
88 u32  action_bits ( void * _self )
89 {
90     struct Environment * self = _self;
91     assert ( self->_valid_actions != NULL);
92
93     u32 max_action = 0;
94
95     foreach ( u32 const * action, self->_valid_actions )
96         max_action = *action && *action > max_action ? *action : max_action;
97
98     return LG2( max_action );
99
100 }//----------------------------------------------------------------------
101
102 // def observation_bits():
103 u32  observation_bits ( void * _self )
104 {
105     struct Environment * self = _self;
106     assert ( self->_valid_observations != NULL);
107
108     u32 max_observation = 0;
109
110     foreach ( u32 const * observation, self->_valid_observations )
111         max_observation = *observation && *observation > max_observation ? *
    observation : max_observation;
112
113     return LG2( max_observation );
114
115 }//----------------------------------------------------------------------
116
117 // def reward_bits():
118 u32  reward_bits ( void * _self )
119 {
120     struct Environment * self = _self;
121     assert ( self->_valid_rewards != NULL);
122
123     u32 max_reward = 0;
124
125     foreach ( u32 const * reward, self->_valid_rewards )
126         max_reward = *reward && *reward > max_reward ? *reward : max_reward;
127
128     return LG2( max_reward );
129
130 }//----------------------------------------------------------------------
131
132 // def perception_bits():
133 u32  percption_bits ( void * _self )
134 {
135     struct Environment * self = _self;
136     return reward_bits(self) + action_bits(self);
137
138 }//----------------------------------------------------------------------
139
140 // check if the action is valid
```

```
141 u08  is_valid_action ( void * _self, u32   action )
142 {
143
144     struct Environment * self = _self;
145
146     foreach( u32 const * a , self->_valid_actions)
147         if ( * a == action ) return TRUE;
148
149     return FALSE;
150
151 }//------------------------------------------------------------------
152
153 // find out if the observation is a valid one
154 u08  is_valid_observation ( void * _self, u32   observation )
155 {
156
157     struct Environment * self = _self;
158
159     foreach( u32 const * o , self->_valid_observations)
160         if ( * o == observation ) return TRUE;
161
162     return FALSE;
163
164 }//------------------------------------------------------------------
165
166 // check if the action is a valid action
167 u08  is_valid_reward ( void * _self, u32   reward )
168 {
169
170     struct Environment * self = _self;
171
172     foreach( u32 const * r , self->_valid_rewards)
173         if ( * r == reward ) return TRUE;
174
175     return FALSE;
176
177 }//------------------------------------------------------------------
178
179 // Get maximum action
180 u32  maximum_action ( void * _self )
181 {
182     struct Environment * self = _self;
183     u16 idx = 0;
184
185     foreach ( u32 const * x , self->_valid_actions)
186         idx++;
187
188     return self->_valid_actions[idx];
189
190 }//------------------------------------------------------------------
191
192 // Get maximum observation
193 u32  maximum_observation ( void * _self )
194 {
195     struct Environment * self = _self;
196     u16 idx = 0;
197
198     foreach ( u32 const * x , self->_valid_observations)
```

```
199         idx++;
200 
201     return self->_valid_observations[idx];
202 
203 }//---------------------------------------------------------------------------
204 
205 
206 // Get maximum reward
207 u32  maximum_reward ( void * _self )
208 {
209     struct Environment * self = _self;
210     u16 idx = 0;
211 
212     foreach ( u32 const * x , self->_valid_rewards)
213     {
214         if ( *x ) idx++;
215         else break;
216     }
217 
218     return self->_valid_rewards[idx];
219 
220 }//---------------------------------------------------------------------------
221 
222 // Get minimum action
223 u32  minimum_action ( void * _self )
224 {
225     struct Environment * self = _self;
226     return self->_valid_actions[0];
227 
228 }//---------------------------------------------------------------------------
229 
230 // Get minimum observation
231 u32  minimum_observation ( void * _self )
232 {
233     struct Environment * self = _self;
234     return self->_valid_observations[0];
235 
236 }//---------------------------------------------------------------------------
237 
238 // Get minimum reward
239 u32  minimum_reward ( void * _self )
240 {
241     struct Environment * self = _self;
242     return self->_valid_rewards[0];
243 
244 }//---------------------------------------------------------------------------
245 
246 u32 LG2 ( u32 x )
247 {
248     #define LT(n) n, n, n, n, n, n, n, n, n, n, n, n, n, n, n, n
249     const char LogTable256[256] =
250     {
251         -1, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3,
252         LT(4), LT(5), LT(5), LT(6), LT(6), LT(6), LT(6),
253         LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7)
254     };
255 
256     register u32 ret, t, tt; // temp var2
```

```
257
258     if ((tt = x >> 16))
259         ret = (t = tt >> 8) ? 24 + LogTable256[t] : 16 + LogTable256[tt];
260     else
261         ret = (t = x >> 8) ? 8 + LogTable256[t] : LogTable256[x];
262
263     return (u32) ret;
264 }//-------------------------------------------------------------------------
```

Listing 5.21: environment.c

```
1  #ifndef COIN_FLIP_H
2      #define COIN_FLIP_H
3      #define HEADS_VAL 0x00000001
4      #define TAILS_VAL 0x00000000
5
6      #include <stddef.h>
7
8      extern const void * Coin_Flip;
9      // Usage
10     //       new ( Coin_Flip , (double) probability );
11     //
12
13
14     double __rp();
15     u32Tuple * perform_action ( void * _self, u32 action_t );
16     static void CF_print(void * _self);
17
18 #endif
```

Listing 5.22: coin$_f$*lip.h*

```
1  #ifndef COIN_FLIP_R
2      #define COIN_FLIP_R
3      #include "../_utils/types.h"
4     struct Coin_Flip {
5         struct Environment _;
6         double probability;
7     };
8      #define probability(e) (((const struct Coin_Flip*)(e)) -> probability)
9
10     typedef enum { Tails , Heads } _action_enum;
11     typedef enum { Tails , Heads } _observation_enum;
12     typedef enum { Loss , Win } _reward_enum;
13
14     static double default_probability = 7e-1;
15 #endif
```

Listing 5.23: coin$_f$*lip.r*

```
1  #include <stdlib.h>
2  #include <time.h>
3  #include <assert.h>
4  #include <stdio.h>
5  #include <stddef.h>
6  #include "../_utils/types.h"
7  #include "../_utils/macros.h"
```

```
8  #include "class.h"
9  #include "class.r"
10 #include "environment.h"
11 #include "environment.r"
12 #include "coin_flip.h"
13 #include "coin_flip.r"
14
15 void * CF_init ( void * _self, va_list * args )
16 {
17     struct Coin_Flip * self =
18         ((const struct Class *) Environment) -> __init__( _self , args );
19
20     self -> _ . num_actions             = 2;
21
22     self -> _ . _valid_actions          = calloc (1, 2 * sizeof ( u32 ) );
23     self -> _ . _valid_actions[0]       = 0;
24     self -> _ . _valid_actions[1]       = 1;
25
26     self -> _ . _valid_observations     = calloc (1, 2 * sizeof ( u32 ) );
27     self -> _ . _valid_observations[0]  = 0;
28     self -> _ . _valid_observations[1]  = 1;
29
30     self -> _ . _valid_rewards          = calloc (1, 2 * sizeof ( u32 ) );
31     self -> _ . _valid_rewards[0]       = 0;
32     self -> _ . _valid_rewards[1]       = 1;
33
34    double probability_t = va_arg ( * args , double );
35    if ( probability_t <= 0.0001 || probability_t >= 1.0001 ) probability_t = 0.7;
36
37     #ifdef DEBUG
38         TRACE ( "Probability = %d\n", probability_t );
39     #endif
40
41    self -> probability = probability_t;
42
43    srand(time(NULL));
44    u32 random_index = rand() % 2;
45    self->_._observation = self->_._valid_observations[random_index];
46
47    return self;
48 }
49
50 double __rp() { return (double) rand() / (double)RAND_MAX; }
51
52 u32Tuple* perform_action ( void * _self, u32 action_t )
53 {
54     struct Coin_Flip * self = _self;
55
56     #ifdef DEBUG
57         TRACE ( "Action = %d\n", action_t );
58     #endif
59
60     BLOCK_START
61         u08 is_valid = 0x00;
62         foreach ( u32 const * a , valid_actions(self) )
63             if ( * a == action_t ) is_valid = !(is_valid);
64         assert ( is_valid != 0x00 );
65     BLOCK_END
```

```
66
67      self -> _ . _action = action_t;
68
69      u32 observation_t , reward_t;
70
71      if (__rp() > probability(self) ){
72          observation_t = 1;
73      } else {
74          observation_t = 0;
75      }
76
77      reward_t = ( action_t == observation_t ) ? 1 : 0;
78
79      #ifdef DEBUG
80          TRACE ( "Observation = %d Reward = %d\n", observation_t, reward_t );
81      #endif
82
83      self -> _ . _observation    = observation_t;
84      self -> _ . _reward         = reward_t;
85
86      u32Tuple* tuple = calloc (1, sizeof(u32Tuple));
87      tuple -> first = observation_t;
88      tuple -> second = reward_t;
89
90      return tuple;
91  }
92
93  static void CF_print(void * _self)
94  {
95      struct Coin_Flip * self = _self;
96      printf ("Prediction = %x, Observation = %x, Reward = %x\n",
97              action(self),observation(self),reward(self));
98  }
99
100 void * CF_cpy ( void * _self )
101 {
102     struct Coin_Flip * self = _self;
103     return new ( Coin_Flip , probability(_self) );
104 }
105
106 static const struct Class _Coin_Flip = {
107     sizeof(struct Coin_Flip), CF_init, NULL, CF_cpy, NULL
108 };
109
110 const void * Coin_Flip = & _Coin_Flip;
```

Listing 5.24: coin$_f$*lip.c*

5.7 Utility

```
1  #ifndef MACRO_H
2      #define MACRO_H
3
4      #include "types.h"
5
```

```
6   #define BLOCK_START {
7
8   #define BLOCK_END   }
9
10  #define FALSE (0)
11
12  #define TRUE (!FALSE)
13
14  #define MIN(a, b)  (((a) < (b)) ? (a) : (b))
15
16  #define MAX(a, b)  (((a) > (b)) ? (a) : (b))
17
18  #define ABS(a)     (((a) < 0) ? -(a) : (a))
19
20  // % used for assertions. e.g assert(IMPLIES(n > 0, array != NULL));
21  #define IMPLIES(x, y) (!(x) || (y))
22
23  // % gt 1 => x > y, eq 0 => x == y , lt 0 => x < y
24  #define COMPARE(x, y) (((x) > (y)) - ((x) < (y)))
25
26  // % return true if x is greater than 1
27  #define SIGN(x) COMPARE(x, 0)
28
29  // % determine the size of an array
30  #define ARRAY_SIZE(a) (sizeof(a) / sizeof(*a))
31
32  // % swap 2 values T is the type. e.g: SWP(a,b,int)
33  #define SWAP(x, y, T) do { T tmp = (x); (x) = (y); (y) = tmp; } while(0)
34
35  // % name says it all T = type
36  #define QSORT(a, b, T) do { if ((a) > (b)) SWAP((a), (b), T); } while (0)
37
38  // % i dont actually understand this one lol
39  #define SET(d, n, v) do{ size_t i_, n_; for (n_ = (n), i_ = 0; n_ > 0; --n_, ++
    i_) (d)[i_] = (v); } while(0)
40
41
42  #define ZERO(d, n) SET(d, n, 0)
43
44  // % For Each Loop.  pretty easy to get your head around.
45  #define foreach(item, array) \
46      for(int keep=1, \
47              count=0,\
48              size=sizeof (array)/sizeof *(array); \
49          keep && count != size; \
50          keep = !keep, count++) \
51      for(item = (array)+count; keep; keep = !keep)
52
53  // % For Debugging it prints a stack trace.
54  #if defined NDEBUG
55      #define TRACE( format, ... ) ( ( void ) 0 )
56  #else
57      #define TRACE( format, ... )   printf( "%s::%s(%d)" format, __FILE__,
    __FUNCTION__,  __LINE__, __VA_ARGS__ )
58  #endif
59
60  // % Print error info and exit
61  #define ERR(source) (fprintf(stderr,"%s:%d\n",__FILE__,__LINE__),\
```

```
62                          perror(source),kill(0,SIGKILL),      \
63                          exit(EXIT_FAILURE))
64
65      // % Debug
66      #define DBG(source) (fprintf(stderr,"%s:%d\t",__FILE__,__LINE__),\
67                               perror(source),
68
69      // % Call function and exit if return value != result
70      #define CALL_AND_CHECK(function, result)\
71      do {\
72      if (function != result)\
73      {\
74      ERR(#function);\
75      }\
76      } while (0)
77
78      // % Call function and exit if error occured
79      #define CALL_AND_EXIT_ON_ERR(function) CALL_AND_CHECK(function, 0)
80  #endif
```

Listing 5.25: macros.h

```
1   #ifndef INTDEF
2       #include <stdint.h>
3       #define INTDEF
4       #ifdef uint128_t
5           typedef uint128_t       u128;
6       #endif
7       typedef uint64_t            u64;
8       typedef uint32_t            u32;
9       typedef uint16_t            u16;
10      typedef uint8_t             u08;
11
12      #ifdef int128_t
13          typedef int_128_t       s128;
14      #endif
15      typedef int64_t             s64;
16      typedef int32_t             s32;
17      typedef int16_t             s16;
18      typedef int8_t              s08;
19  #endif
20
21  #ifndef U_ENUM
22  #define U_ENUM
23      typedef enum { action_update, percept_update } update_enum;
24  #endif
25
26  #ifndef TUPLE_32
27  #define TUPLE_32
28      typedef struct {
29          u32             first;
30          u32             second;
31      } u32Tuple;
32  #endif
33
34  #ifndef UNDO
35  #define UNDO
36      typedef struct {
```

```
37          u32             age;
38          u32             total_reward;
39          u32             history_size;
40          update_enum     last_update;
41      } AgentUndo;
42  #endif
```

Listing 5.26: types.h