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 24th 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:

```

else {
        u32 action = _monte_select_action(tree, agent);
        Agent_model_update_action(agent, action);

        MonteNode* child = dict_find(tree->actions, action);
        if(child == NULL) {
            child = monte_create_tree(NODE_TYPE_CHANCE);
          dict_add(tree->actions, action, child);
        }

        if(child == NULL) {
            exit(1338);
        }

        reward = monte_sample(child, agent, horizon);

```

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:

```
    int data;
    MPI_Recv(&data, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    ctw_load(agent->context_tree, CTW_DATA_FILE);
    double mean;
    int action = Agent_search_mean(agent, &mean);

    MPI_Send(&action, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    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:

```
for (i = 1; i < P; i++) {
  double mean;
  MPI_Recv(&action, 1, MPI_INT, i, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(&mean, 1, MPI_DOUBLE, i, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  if (mean > best_mean) {
    best_action = action;
    best_mean = mean;
  }
}
```

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.

```c
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.

```c
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.

```c
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) {
  uint32_t numerator;
  if (symbol) {
    numerator = node->ones_in_history;
  } else {
    numerator = node->zeroes_in_history;
  }
  uint32_t denominator = ctw_node_visits(node);
  return log((numerator + 0.5) / (denominator + 1.0));
}

void ctw_node_update_log_probability(ContextTreeNode *node) {
  if (ctw_node_is_leaf(node)) {
    node->log_probability = node->log_kt;
  } else {
    double log_child_prob = 0.0;
    if (node->zero_child != NULL) {
      log_child_prob += node->zero_child->log_probability;
    }
    if (node->one_child != NULL) {
      log_child_prob += node->one_child->log_probability;
    }

    double a, b;
    if (node->log_kt >= log_child_prob) {
      a = node->log_kt;
      b = log_child_prob;
    } else {
      a = log_child_prob;
      b = node->log_kt;
    }
    node->log_probability = log(0.5) + a + log1p(exp(b - a));
  }
}
```

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:

```c
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':

```
#ifndef CLASS_R
#define CLASS_R

#include <stdarg.h>
#include "../_utils/types.h"
```

```

struct Class {
    size_t  size;
    void *  ( * __init__ )              ( void * self, va_list      args );
    void *  ( * __delete__ )            ( void * self );
    void *  ( * __copy__ )              ( const void * self );
    void *  ( * __str__ )               ( const void * self );
};

#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.

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

void * new ( const void * _class, ... )
{
    const struct Class * class = _class;
    void * mem = calloc( 1, class->size );

    //TODO: Replace with better error handling.
    assert ( mem );

    * ( const struct Class ** ) mem = class;

    // This handles and vars passed to the constructor.
    if ( class->__init__ )
    {
        va_list args;
        #ifdef DEBUG
            TRACE("Class Created", "__init__",class->__str__);
        #endif
        va_start( args, _class );                   // intialize '...'
      mem = class->__init__( mem, &args );        // call constructor
        va_end( args );                             // clean
    }

    return mem;
}

void delete ( void * self )
{
    const struct Class ** parent = self;

    if ( self && * parent && ( * parent )->__delete__ ) {
        self = ( * parent )->__delete__(self);
        #ifdef DEBUG
```

```
            TRACE("Class Destroyed","delete(...)",class->__str__);
        #endif
    }
    free(self);
}

void * cpy ( const void * self )
{
    const struct Class * const * parent = self;

    assert ( self && parent && ( * parent )->__copy__);

    #ifdef DEBUG
        TRACE("Class Copied","cpy(..)",class->__str__);
    #endif

    return ( * parent )->__copy__(self);
}

char * print ( const void * self )
{
    const struct Class * const * parent = self;
    char * pstring = malloc ( 255 * (sizeof(char)));
    sprintf(pstring,"%s",( * parent )->__str__(self));

    #ifdef DEBUG
        TRACE ( "Generated Print String: ", "print(...)", pstring );
    #endif

    return pstring;
}
```

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:

```
#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 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.

```
#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("%s \n Environment_init\n", pstring);
    #endif

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

// This is kind of complicated but it basically uses the parent object class to define
// the enironment object pointer which holds all of the accessor methods.  This is all
// in the hopes that anyone can take this code and call Environment e = new(environment,...)
// then class methods can be called with e.method()
//
static const struct Class _Environment = {
    sizeof(struct Environment),
    Environment_init,                    // done
    Environment_delete,                  // done
    Environment_cpy,                     // done
    Environment_str,                     // done
```

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

const void * Environment = & _Environment;

// def action_bits():
u32  action_bits ( void * _self )
{
    struct Environment * self = _self;
    assert ( self->_valid_actions != NULL);

    u32 max_action = 0;

    foreach ( u32 const * action, self->_valid_actions )
        max_action = *action && *action > max_action ? *action : max_action;

    return LG2( max_action );

}//-------------------------------------------------------------------------

// def observation_bits():
u32  observation_bits ( void * _self )
{
    struct Environment * self = _self;
    assert ( self->_valid_observations != NULL);

    u32 max_observation = 0;

    foreach ( u32 const * observation, self->_valid_observations )
        max_observation = *observation && *observation > max_observation ? *observation : max_observation;

    return LG2( max_observation );

}//-------------------------------------------------------------------------

// def reward_bits():
u32  reward_bits ( void * _self )
{
    struct Environment * self = _self;
    assert ( self->_valid_rewards != NULL);

    u32 max_reward = 0;

    foreach ( u32 const * reward, self->_valid_rewards )
        max_reward = *reward && *reward > max_reward ? *reward : max_reward;

    return LG2( max_reward );

}//-------------------------------------------------------------------------

// def perception_bits():
u32  percption_bits ( void * _self )
{
    struct Environment * self = _self;
    return reward_bits(self) + action_bits(self);

}//-------------------------------------------------------------------------

```

```
// check if the action is valid
u08  is_valid_action ( void * _self, u32   action )
{

    struct Environment * self = _self;

    foreach( u32 const * a , self->_valid_actions)
        if ( * a == action ) return TRUE;

    return FALSE;

}//----------------------------------------------------------------------------

// find out if the observation is a valid one
u08  is_valid_observation ( void * _self, u32   observation )
{

    struct Environment * self = _self;

    foreach( u32 const * o , self->_valid_observations)
        if ( * o == observation ) return TRUE;

    return FALSE;

}//----------------------------------------------------------------------------

// check if the action is a valid action
u08  is_valid_reward ( void * _self, u32   reward )
{

    struct Environment * self = _self;

    foreach( u32 const * r , self->_valid_rewards)
        if ( * r == reward ) return TRUE;

    return FALSE;

}//----------------------------------------------------------------------------

// Get maximum action
u32  maximum_action ( void * _self )
{
    struct Environment * self = _self;
    u16 idx = 0;

    foreach ( u32 const * x , self->_valid_actions)
        idx++;

    return self->_valid_actions[idx];

}//----------------------------------------------------------------------------

// Get maximum observation
u32  maximum_observation ( void * _self )
{
    struct Environment * self = _self;
    u16 idx = 0;

```

```c
    foreach ( u32 const * x , self->_valid_observations)
      idx++;

    return self->_valid_observations[idx];

}//------------------------------------------------------------------------------


// Get maximum reward
u32  maximum_reward ( void * _self )
{
    struct Environment * self = _self;
    u16 idx = 0;

    foreach ( u32 const * x , self->_valid_rewards)
    {
        if ( *x ) idx++;
        else break;
    }

    return self->_valid_rewards[idx];

}//------------------------------------------------------------------------------

// Get minimum action
u32  minimum_action ( void * _self )
{
    struct Environment * self = _self;
    return self->_valid_actions[0];

}//------------------------------------------------------------------------------

// Get minimum observation
u32  minimum_observation ( void * _self )
{
    struct Environment * self = _self;
    return self->_valid_observations[0];

}//------------------------------------------------------------------------------

// Get minimum reward
u32  minimum_reward ( void * _self )
{
    struct Environment * self = _self;
    return self->_valid_rewards[0];

}//------------------------------------------------------------------------------

u32 LG2 ( u32 x )
{
    #define LT(n) n, n, n, n, n, n, n, n, n, n, n, n, n, n, n, n
    const char LogTable256[256] =
    {
        -1, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3,
        LT(4), LT(5), LT(5), LT(6), LT(6), LT(6), LT(6),
        LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7)
    };

    register u32 ret, t, tt; // temp var2

    if ((tt = x >> 16))
        ret = (t = tt >> 8) ? 24 + LogTable256[t] : 16 + LogTable256[tt];
    else
        ret = (t = x >> 8) ? 8 + LogTable256[t] : LogTable256[x];

    return (u32) ret;
}//---------------------------------------------------------------------
```

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:

```
#ifndef COIN_FLIP_R
    #define COIN_FLIP_R
    #include "../_utils/types.h"
   struct Coin_Flip {
       struct Environment _;
       double probability;
   };
    #define probability(e) (((const struct Coin_Flip*)(e)) -> probability)

    //typedef enum { Tails , Heads } _action_enum;
    //typedef enum { Tails , Heads } _observation_enum;
    //typedef enum { Loss , Win } _reward_enum;

    static double default_probability = 7e-1;
#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;

    if (__rp() > probability(self) ){
        observation_t = 1;
        reward_t = ( action_t == 0 ) ? 0 : 1;
    } else {
        observation_t = 0;
        reward_t = ( action_t == 0 ) ? 1 : 0;
    }

    #ifdef DEBUG
        TRACE ( "Observation = %d Reward = %d\n", observation_t, reward_t );
    #endif

    self -> _ . _observation     = observation_t;
    self -> _ . _reward          = reward_t;

    u32Tuple* tuple = calloc (1, sizeof(u32Tuple));
    tuple -> first = observation_t;
    tuple -> second = reward_t;

    return tuple;
}

static void CF_print(void * _self)
{
    struct Coin_Flip * self = _self;
    printf ("Prediction = %x, Observation = %x, Reward = %x\n",
            action(self),observation(self),reward(self));
}

void * CF_cpy ( void * _self )
{
    struct Coin_Flip * self = _self;
    return new ( Coin_Flip , probability(_self) );
}

static const struct Class _Coin_Flip = {
    sizeof(struct Coin_Flip), CF_init, NULL, CF_cpy, NULL
};

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

```c
#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;
    bool verbose;
} app_options;

app_options* _make_default_options() {
    app_options* options = malloc(sizeof(app_options));
    options->agent = 0;
    options->agent_horizon = 5;
    options->ct_depth = 30;
    options->environment = 0;
    options->exploration = 0.001f;
    options->explore_decay = 1.0f;
    options->learning_period = 0;
    options->mc_simulations = 300;
    options->profile = false;
    options->terminate_age = 0;
    options->verbose = false;

    return options;
}

float _random_0_1() {
    return (float)rand()/(float)(RAND_MAX/1);
}

#ifdef USE_MPI
void mpi_main(Agent* agent, struct Environment* environment, app_options* options,
    int argc, const char* argv[]) {
  int rank, P;

  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) {
    printf("Error\n");
    return 1;
  }


  MPI_Comm_size(MPI_COMM_WORLD, &P);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
     printf("%d\n", agent->context_tree);
    printf("desu\n");
      agent_proc(P, agent, environment, options);
  } else {
    search_proc(rank, P, agent, environment, options);
  }

  MPI_Finalize();
}

void agent_proc(int P, Agent* agent, struct Environment* environment, app_options*
    options) {
  printf("MC AIXI training warming up...\n");
  srand(1337);

  bool explore =  options->exploration > 0;
  if(0.0f > options->exploration || 0.0f > options->explore_decay || options->
    explore_decay > 1.0f) {
    printf("Some exploration parameter is invalid. Application force quitting.\n");
    exit(1);
  }

  bool terminate_check = options->terminate_age > 0;
  bool isEnvironmentFinished = false;
  int cycle = 1;

  while(!isEnvironmentFinished) {
    int agent_age = agent->age;

    if(terminate_check && agent_age > options->terminate_age) {
      TRACE("Interaction looked broken; terminate age exceeded.\n", "main");
      break;
    }

    long cycle_start = time(NULL);

    u32 observation = environment->_observation;
    u32 reward = environment->_reward;

    if(options->learning_period > 0 && cycle > options->learning_period) {
      explore = false;
    }

    Agent_model_update_percept(agent, observation, reward);

    bool explored =  false;

    u32 best_action = 0;
    double best_mean = -1;
    u32 action;
    if (explore && _random_0_1() < options->exploration) {
      explored =  true;
      printf("Agent is trying action at random...\n");
      action = Agent_generate_random_action(agent);
    } else {
      ctw_save(agent->context_tree, CTW_DATA_FILE);
      int i;
      for (i = 1; i < P; i++) {
  int data = 1;
  MPI_Send(&data, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
      }
      for (i = 1; i < P; i++) {
  double mean;
  MPI_Recv(&action, 1, MPI_INT, i, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(&mean, 1, MPI_DOUBLE, i, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  if (mean > best_mean) {
    best_action = action;
    best_mean = mean;
  }
      }
    }


    perform_action(environment, action);

    Agent_model_update_action(agent, action);

```

```
    long ticks_taken = time(NULL) - cycle_start;

    if (cycle % 1 == 0) {
      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");
    }   // Just a large padded statement about what is going on in the world as we step through

    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));

    if(explore) {
      options->exploration *= options->explore_decay;
    }

    cycle++;

    isEnvironmentFinished = environment->_is_finished;
  }
}

void search_proc(int rank, int P, Agent* agent, struct Environment* environment, app_options* options) {
  for (;;) {
    int data;
    MPI_Recv(&data, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    ctw_load(agent->context_tree, CTW_DATA_FILE);
    double mean;
    int action = Agent_search_mean(agent, &mean);

    MPI_Send(&action, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Send(&mean, 1, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
  }
}
#endif
void _interaction_loop(Agent* agent, struct Environment* environment, app_options* options) {
    printf("MC AIXI training warming up...\n");
    srand(1337);

    bool explore =  options->exploration > 0;
    if(0.0f > options->exploration || 0.0f > options->explore_decay || options->explore_decay > 1.0f) {
        printf("Some exploration parameter is invalid. Application force quitting.\n");
        exit(1);
    }

    bool terminate_check = options->terminate_age > 0;
    bool isEnvironmentFinished = false;
    int cycle = 1;


    while(!isEnvironmentFinished) {
        int agent_age = agent->age;

        if(terminate_check && agent_age > options->terminate_age) {
            TRACE("Interaction looked broken; terminate age exceeded.\n", "main");
            break;
        }

        long cycle_start = time(NULL);

  u32 observation = environment->_observation;
        u32 reward = environment->_reward;

        if(options->learning_period > 0 && cycle > options->learning_period) {
            explore = false;
        }

        Agent_model_update_percept(agent, observation, reward);

        bool explored =  false;

      u32 action;
        if (explore && _random_0_1() < options->exploration) {
      explored =  true;
      printf("Agent is trying action at random...\n");
      action = Agent_generate_random_action(agent);
  } else {
      action = Agent_search(agent);
  }

        perform_action(environment, action);

        Agent_model_update_action(agent, action);

        long ticks_taken = time(NULL) - cycle_start;

  if (cycle % 1 == 0) {
    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");
    }   // Just a large padded statement about what is going on in the world as we step through
        //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");
        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));

        if(explore) {
            options->exploration *= options->explore_decay;
        }

        cycle++;

        isEnvironmentFinished = environment->_is_finished;
    }

}
```

```c

int main(int argc, const char* argv[]) {
    app_options* appOptions = _make_default_options();

    printf("Booting MC AIXI kernel...\n");

    TRACE("Creating environment...\n", "desu");

    struct Coin_Flip* environment = new (Coin_Flip, 0.9f);

    TRACE("Creating agent... please be patient\n", "desu");

    TRACE("allocate agent\n", "main");
    Agent* agent = malloc(sizeof(Agent));
    TRACE("init agent\n", "main");
    agent = Agent_init(agent, environment, appOptions->learning_period);


    if(appOptions->profile) {
        printf("Profiling is not currently supported. Ignoring.\n");
    }

#ifndef USE_MPI
    _interaction_loop(agent, (struct Environment *) environment, appOptions);
#else
    mpi_main(agent, environment, appOptions, argc, argv);
#endif
}
```

Listing 5.1: main.c

```c
#include <stdint.h>
#include <stdbool.h>
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "bit_vector.h"

BitVector *bv_create() {
  BitVector *bit_vector = (BitVector *) malloc(sizeof(BitVector));
  bit_vector->size = 0;
  bit_vector->capacity = DEFAULT_BIT_VECTOR_CAPACITY;
  bool *bits = (bool *) malloc(bit_vector->capacity * sizeof(bool));
  assert(bits != NULL);
  bit_vector->bits = bits;
  return bit_vector;
}

void bv_free(BitVector *bv) {
  free(bv->bits);
  free(bv);
}

BitVector *bv_from_char(char c) {
  BitVector *bv = bv_create();
  int64_t j;
  for (j=sizeof(char) * 8 - 1; j >= 0; j--) {
    bv_push(bv, (c >> j) % 2 == 1);
  }
  return bv;
}

BitVector *bv_from_uint32(uint32_t v) {
  BitVector *bv = bv_create();
  int64_t j;
  for (j=sizeof(uint32_t) * 8 - 1; j >= 0; j--) {
    bv_push(bv, (v >> j) % 2 == 1);
  }
  return bv;
}

BitVector *bv_from_uint64(uint64_t v) {
  BitVector *bv = bv_create();
  int64_t j;
  for (j=sizeof(uint64_t) * 8 - 1; j >= 0; j--) {
    bv_push(bv, (v >> j) % 2 == 1);
  }
  return bv;
}

BitVector *bv_from_double(double d) {
  // interpret the double bits as an int
  uint64_t v = *((uint64_t *) &d);
  return bv_from_uint64(v);
}

BitVector *bv_from_float(float f) {
  // interpret the float bits as an int
  uint32_t v = *((uint32_t *) &f);
  return bv_from_uint32(v);
}

uint32_t bv_peek_uint32(BitVector *bv) {
  uint32_t result = 0;
  int64_t i;
  for (i = sizeof(uint32_t) * 8; i > 0; i--) {
    result = result << 1;
    if (bv_test(bv, bv->size - i)) {
      result += 1;
    }
  }
  return result;
}

uint64_t bv_peek_uint64(BitVector *bv) {
  uint64_t result = 0;
  int64_t i;
  for (i = sizeof(uint64_t) * 8; i > 0; i--) {
    result = result << 1;
    if (bv_test(bv, bv->size - i)) {
      result += 1;
    }
  }
  return result;
}

```

```c
void bv_append(BitVector *a, BitVector *b) {
  uint64_t i;
  for (i = 0; i < b->size; i++) {
    bv_push(a, bv_test(b, i));
  }
}

void __bv_check_bounds(BitVector *bv, uint64_t index) {
  if (index >= bv->size) {
    printf("BV, Index out of bounds, index: %llu size: %llu\n", index, bv->size);
    assert(false);
  }
}

void __bv_grow(BitVector *bv) {
  uint64_t new_capacity = bv->capacity * BIT_VECTOR_GROW_RATE;
  bool *new_bits = (bool *) malloc(new_capacity * sizeof(bool));
  uint64_t i;
  for (i = 0; i < bv->size; i++) {
    new_bits[i] = bv->bits[i];
  }

  free(bv->bits);
  bv->capacity = new_capacity;
  bv->bits = new_bits;
}

bool bv_test(BitVector *bv, uint64_t index) {
  __bv_check_bounds(bv, index);
  return bv->bits[index];
}

void bv_set(BitVector *bv, uint64_t index, bool bit) {
  __bv_check_bounds(bv, index);
  bv->bits[index] = bit;
}

void bv_push(BitVector *bv, bool bit) {
  if (bv->size == bv->capacity) {
    __bv_grow(bv);
  }
  bv->bits[bv->size] = bit;
  bv->size += 1;
}

bool bv_peek(BitVector *bv) {
  assert(bv->size > 0);
  return bv->bits[bv->size -1];
}

bool bv_pop(BitVector *bv) {
  assert(bv->size > 0);
  bool bit = bv->bits[bv->size -1];
  bv->size -= 1;
  return bit;
}

void bv_clear(BitVector *bv) {

  bv->size = 0;
}

void bv_print(BitVector *bv) {
  printf("Size: %llu\n", bv->size);
  if (bv->size > 400) {
    return;
  }
  uint64_t i;
  for (i = 0; i < bv->size; i++) {
    if (bv_test(bv, i)) {
      printf("1");
    } else {
      printf("0");
    }
  }
  printf("\n");
}

void bv_print_ascii(BitVector *bv) {
  uint64_t i;
  for (i = 0; i < bv->size - (bv->size % 8); i += 8) {
    char c = 0;
    uint64_t j;
    for (j = 0; j < 8; j++) {
      c = c << 1;
      if (bv_test(bv, j+i)) {
  c += 1;
      }
    }
    printf("%c", c);
  }
  printf("\n");
}

void bv_save(BitVector *bv, FILE *fp) {
  fwrite(&(bv->size), sizeof(uint64_t), 1, fp);
  uint64_t i;
  for (i = 0; i < bv->size; i++) {
    bool bit = bv_test(bv, i);
    fwrite(&bit, sizeof(bool), 1, fp);
  }
}

void bv_load(BitVector *bv, FILE *fp) {
  bv_clear(bv);
  uint64_t size;
  fread(&size, sizeof(uint64_t), 1, fp);
  uint64_t i;
  for (i = 0; i < size; i++) {
    bool bit;
    fread(&bit, sizeof(bool), 1, fp);
    bv_push(bv, bit);
  }
}

BitVector *bv_slice(BitVector *bv, uint64_t start, uint64_t end) {
  BitVector *slice = bv_create();
  for (uint64_t i = start; i < end; i++) {
    bv_push(slice, bv_test(bv, i));
  }
  return slice;
}
```

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

```c
#ifndef _BIT_VECTOR_
#define _BIT_VECTOR_

#include <stdbool.h>
#include <stdio.h>
#include <stdint.h>

#define DEFAULT_BIT_VECTOR_CAPACITY 16
#define BIT_VECTOR_GROW_RATE 2

typedef struct BitVector {
  uint64_t size;
  uint64_t capacity;
  bool *bits;
} BitVector;

BitVector *bv_create();

void bv_free(BitVector *);

BitVector *bv_from_char(char);

BitVector *bv_from_uint32(uint32_t);

BitVector *bv_from_uint64(uint64_t);

void bv_append(BitVector *, BitVector *);

uint32_t bv_peek_uint32(BitVector *);

uint64_t bv_peek_uint64(BitVector *);

void __bv_check_bounds(BitVector *, uint64_t);

void __bv_grow(BitVector *);

bool bv_test(BitVector *, uint64_t);

void bv_set(BitVector *, uint64_t, bool);

void bv_push(BitVector *, bool);

bool bv_peek(BitVector *);

bool bv_pop(BitVector *);

void bv_clear(BitVector *);

void bv_print(BitVector *);

void bv_print_ascii(BitVector *);

void bv_save(BitVector *, FILE *);

void bv_load(BitVector *, FILE *);

BitVector *bv_slice(BitVector *bv, uint64_t start, uint64_t end);

#endif
```

Listing 5.3: bit$_{v}ector.h$

5.2 Object

```
#ifndef NEW_H
#define NEW_H

void * new      ( const void * class, ... );
void delete     ( void * item );
void * cpy      ( const void * self );
char * print    ( const void * self );

#endif
```

Listing 5.4: class.h

```
#ifndef CLASS_R
#define CLASS_R

#include <stdarg.h>
#include "../_utils/types.h"

struct Class {
    size_t  size;
    void *  ( * __init__ )          ( void * self, va_list      args );
    void *  ( * __delete__ )        ( void * self );
    void *  ( * __copy__ )          ( const void * self );
    void *  ( * __str__ )           ( const void * self );
};



#endif
```

Listing 5.5: class.r

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

```

```
void * new ( const void * _class, ... )
{
    const struct Class * class = _class;
    void * mem = calloc( 1, class->size );

    assert ( mem );

    * ( const struct Class ** ) mem = class;

    // This handles and vars passed to the constructor.
    if ( class->__init__ )
    {
        va_list args;
        #ifdef DEBUG
            TRACE("Class Created", "__init__",class->__str__);
        #endif
        va_start( args, _class );                 // intialize '...'
      mem = class->__init__( mem, &args );      // call constructor
        va_end( args );                           // clean
    }

    return mem;
}

void delete ( void * self )
{
    const struct Class ** parent = self;

    if ( self && * parent && ( * parent )->__delete__ ) {
        self = ( * parent )->__delete__(self);
        #ifdef DEBUG
            TRACE("Class Destroyed","delete(...)",class->__str__);
        #endif
    }
    free(self);
}

void * cpy ( const void * self )
{
    const struct Class * const * parent = self;

    assert ( self && parent && ( * parent )->__copy__);

    #ifdef DEBUG
        TRACE("Class Copied","cpy(..)",class->__str__);
    #endif

    return ( * parent )->__copy__(self);
}

char * print ( const void * self )
{
    const struct Class * const * parent = self;
    char * pstring = malloc ( 255 * (sizeof(char)));
    sprintf(pstring,"%s",( * parent )->__str__(self));

    #ifdef DEBUG
        TRACE ( "Generated Print String: ", "print(...)", pstring );
    #endif

    return pstring;
}
```

Listing 5.6: class.c

5.3 Agent

```
#ifndef AGENT_H
    #define AGENT_H
    #include <stdarg.h>
    #include "../_utils/types.h"
    #include "../bit_vector.h"
    #include "../predict/context_tree.h"

    typedef struct Agent
    {
        struct Environment * environment;
        va_list              _options;
        double               total_reward;
        update_enum          last_update;
        u32                  age;
        u32                  horizon;
        u32                  learning_period;
        ContextTree*         context_tree;
    } Agent;

    Agent* Agent_init ( Agent* self, void * _env, u32 learn  );
    AgentUndo* Agent_clone_into_temp                    (Agent* self);

    double  Agent_average_reward                        ( Agent* self);

    u32  Agent_generate_random_action                   ( Agent* self);
    u32  Agent_maximum_action                           ( Agent* self  );
    u32  Agent_maximum_reward                           ( Agent* self  );
    u32  Agent_model_size                               ( Agent* self  );

    void  Agent_model_update_action                     ( Agent* self, u32 action );
    BitVector * Agent_encode_action                     ( Agent* self , u32 action );

    // decoding
    u32 Agent_decode_action                             (Agent* self  , BitVector* symbols);
    u32 Agent_decode_observation                        (Agent* self  , BitVector* symbols);
    u32 Agent_decode_reward                             (Agent* self , BitVector* symbols);
    u32Tuple* Agent_decode_percept                      (Agent* self  , BitVector* symbols);

    // generators
    u32 Agent_generate_action                           (Agent* self);
    u32Tuple * Agent_generate_percept                          (Agent* self );
    u32Tuple * Agent_generate_percept_and_update               (Agent* self );

    u32 Agent_history_size                             (Agent* self );
    double Agent_get_predicted_action_probability (Agent* self , u32 action);
    u32 Agent_maximum_bits_needed                      (Agent* self );

    void Agent_model_revert                            (Agent* self , AgentUndo* undo);
    void Agent_model_update_percept                    ( Agent* self , u32 observation , u32 reward );


    BitVector * Agent_encode_percept                   ( Agent* self , u32 observation , u32 reward);

    double Agent_percept_probability                   (Agent* self , u32 observation , u32 reward);
    double Agent_playout                              (Agent* self , u32 horizon);

    u32 Agent_search                                  ( Agent* self);
    u32 Agent_search_mean                             ( Agent* self , double *mean);
    void  Agent_reset                                  ( Agent* self );
#endif
```

Listing 5.7: agent.h

```
#include <stddef.h>
#include <stdarg.h>
#include <stdlib.h>
#include "../_utils/types.h"
#include "../_object/class.h"
#include "../predict/context_tree.h"
#include "../bit_vector.h"
#include "../environment/environment.r"
#include "../environment/environment.h"
#include "agent.h"
#include "../search/monte_node.h"
#include "../_utils/macros.h"
#include <assert.h>

Agent* Agent_init ( Agent* self , void * _env, u32 learn  ) {
    const struct Coin_Flip * env = _env;
    TRACE("Prepping to build agent\n", "agent");
    self -> environment = cpy ( env );
    self -> age = 0;
    self -> learning_period = learn;
    self -> last_update = action_update;
    self -> total_reward = 0.0;
    self -> horizon = 6;
    u32 depth = 192;

    TRACE("Building context tree for Agent", "agent");
    self->context_tree = ctw_create(depth);

    #ifdef DEBUG
        TRACE("learning period = %d, horizon = %d, depth = %d", \
                learn,self->horizon,depth);
    #endif
    Agent_reset(self);
    return self;
}

void Agent_delete ( void * _self ) {
    int *t = malloc(1)
    free(t)
}

AgentUndo* Agent_clone_into_temp(Agent* self) {
  AgentUndo* undo = (AgentUndo *) malloc(sizeof(AgentUndo));
  undo->age = self->age;
  undo->total_reward = self->total_reward;
  undo->history_size = Agent_history_size(self);
  undo->last_update = self->last_update;
  return undo;
}

u32 Agent_decode_action(Agent* self, BitVector* symbols) {
  return bv_peek_uint32(symbols);
}

u32 Agent_decode_observation(Agent* self, BitVector* symbols) {
  return bv_peek_uint32(symbols);
}

u32 Agent_decode_reward(Agent * self, BitVector* symbols) {
  return bv_peek_uint32(symbols);
}

u32Tuple* Agent_decode_percept(Agent* self, BitVector* symbols) {
  uint64_t i;
  BitVector* reward_symbols = bv_create();
  BitVector* observation_symbols = bv_create();

  for (i = 0; i < 32; i++) {
    bv_push(reward_symbols, bv_test(symbols, i));
  }
  for (i = 32; i < 64; i++) {
    bv_push(observation_symbols, bv_test(symbols, i));
  }

  // Decode both
  u32 reward = Agent_decode_reward(self, reward_symbols);
  u32 observation = Agent_decode_observation(self, observation_symbols);
  u32Tuple* tuple;
  if (is_valid_reward(self->environment, reward) &&
      is_valid_observation(self->environment, observation)) {
    tuple = malloc(sizeof(u32Tuple));
    tuple->first = observation;
    tuple->second = reward;
  } else {
    tuple = NULL;
  }

  return tuple;
}

BitVector * Agent_encode_action(Agent* self, u32 action) {
  return bv_from_uint32(action);
}

BitVector * Agent_encode_percept ( Agent * self, u32 observation, u32 reward) {
  BitVector* a = bv_from_uint32(reward);
  BitVector* b = bv_from_uint32(observation);
  bv_append(a, b);
  bv_free(b);
  return a;
}

u32 Agent_generate_action(Agent* self) {
  assert(self->last_update == percept_update);

  BitVector* random = ctw_gen_random_symbols(self->context_tree, 32);
  return Agent_decode_action(self, random);
}

u32Tuple* Agent_generate_percept(Agent* self) {
  BitVector* random = ctw_gen_random_symbols_and_update(self->context_tree, 64);

  u32Tuple *percept = Agent_decode_percept(self, random);
  if (percept == NULL) {
    percept = malloc(sizeof(u32Tuple));
    percept->first = rand() % 2;
    percept->second = rand() % 2;
    ctw_revert(self->context_tree, 64);
    BitVector* symbols = Agent_encode_percept(self, percept->first, percept->second)
    ;
    ctw_update_vector(self->context_tree, symbols);
    bv_free(symbols);
  }
  return percept;
}

u32Tuple * Agent_generate_percept_and_update(Agent*  self) {
  u32Tuple* tuple = Agent_generate_percept(self);
  self->total_reward += tuple->second;
  self->last_update = percept_update;
  return tuple;
}

double Agent_get_predicted_action_probability(Agent* self, u32 action) {
  BitVector* symbols = Agent_encode_action(self, action);
  return ctw_predict_vector(self->context_tree, symbols);
}

u32 Agent_history_size(Agent* self) {
  return self->context_tree->history->size;
}

u32 Agent_maximum_bits_needed(Agent * self) {
    return 32;
}

void Agent_model_revert(Agent * self, AgentUndo* undo) {
  while(Agent_history_size(self) > undo->history_size)  {
    if(self->last_update == percept_update){
```

```c
      ctw_revert(self->context_tree, 32);
      self->last_update = action_update;
    } else {
      ctw_revert_history(self->context_tree, 32);
      self->last_update = percept_update;
    }
  }

  if (Agent_history_size(self) != undo->history_size) {
    printf("hist size should be equal %u %u\n", Agent_history_size(self), undo->history_size);
    exit(1034109);
  }

  self->age = undo->age;
  self->total_reward = undo->total_reward;
  self->last_update = undo->last_update;
}

u32 Agent_model_size ( Agent* self) {
  return ctw_size(self->context_tree);
}

void Agent_model_update_action ( Agent* self, u32 action) {
  BitVector* action_symbols = Agent_encode_action(self, action);
  ctw_update_history(self->context_tree, action_symbols);
  self->age++;
  self->last_update = action_update;
}

void Agent_model_update_percept ( Agent * self, u32 observation, u32 reward ) {
  BitVector* symbols = Agent_encode_percept(self, observation, reward);

  if((self->learning_period > 0 ) && (self->age > self->learning_period)) {
    printf("not learning any more\n");
    ctw_update_history(self->context_tree, symbols);
  } else {
    ctw_update_vector(self->context_tree, symbols);
  }

  self->total_reward += reward;
  self->last_update = percept_update;
}

double Agent_percept_probability(Agent* self, u32 observation, u32 reward) {
  BitVector* symbols = Agent_encode_percept(self, observation, reward);
  return ctw_predict_vector(self->context_tree, symbols);
}

double Agent_playout(Agent* self, u32 horizon) {
  double total_reward = 0;

  for(u32 i = 0; i < horizon; i++) {
    u32 action = Agent_generate_random_action(self);
    Agent_model_update_action(self, action);

    u32Tuple* tuple = Agent_generate_percept_and_update(self);
    total_reward += tuple->second;
  }

  return total_reward;
}

void Agent_reset ( Agent* self ) {
  ctw_clear(self->context_tree);

  self->age = 0;
  self->total_reward = 0.0;
  self->last_update = action_update;
}

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");
  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;
}

u32 Agent_search(Agent* self) {
  double mean;
  u32 action = Agent_search_mean(self, &mean);
  return action;
}

double Agent_average_reward ( Agent * self) {
  double average = 0.0;
  if ( self -> age > 0 )
    average = ( self -> total_reward ) / ( self->age );
  return average;
}

u32 Agent_generate_random_action ( Agent * self) {
  int actionIndex = rand() % self->environment->num_actions;
  return self->environment->_valid_actions[actionIndex];
}

u32 Agent_maximum_action ( Agent* self) {
  return maximum_action(self->environment);
}

u32 Agent_maximum_reward ( Agent* self)
{
  return maximum_reward(self->environment);
}
```

Listing 5.8: agent.c

5.4 Search

```c
#ifndef _SEARCH_C
#define _SEARCH_C


#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>
#include <math.h>
#include <float.h>
#include "dict.h"
#include "monte_node.h"
#include "../agent/agent.h"
#include "../environment/environment.r"
#include "../_utils/types.h"
#include "../_utils/macros.h"

#define ARC4RANDOM_MAX      0x100000000

#define MONTE_UNEXPLORED_BIAS 100000000000.0f


MonteNode* monte_create_tree(u32 nodeType) {
    MonteNode* root = (MonteNode* ) malloc(sizeof(MonteNode));

    root->mean = 0.0;
    root->type = nodeType;
    root->visits = 0;

    root->children = dict_new();
    root->actions = dict_new();

    return root;
}

```

```c
u32 _monte_select_action(MonteNode* tree, Agent* agent) {
    // returns -1 if no vaild action

    float agent_horizon = agent->horizon;
    float agent_max_reward = Agent_maximum_reward(agent)/1;

    float explore_bias = agent_horizon * agent_max_reward;
    float exploration_numerator = (float) (2.0f* log((double) tree->visits));


    // desu??? Mondaiji-tachi ga Isekai kara Kuru Sou Desu yo?
    u32 best_action = 0;
    double best_priority = -FLT_MAX;
    u32 i = 0;
    for(i = 0; i < agent->environment->num_actions; i++) {
        u32 action = agent->environment->_valid_actions[i];
        MonteNode* node = dict_find(tree->actions, action);

        double priority = 0;

        if(node == NULL  || node->visits == 0)  {
            priority = MONTE_UNEXPLORED_BIAS;
        } else {
          priority = node->mean - (explore_bias * sqrt(exploration_numerator / node->visits));
        }

        if(priority > (best_priority + ((float)rand()/(float)(RAND_MAX)) * 0.001)) {
            best_action = action;
            best_priority = priority;
        }

    }
    return best_action;
}


float monte_sample(MonteNode* tree, Agent* agent, u32 horizon) {
    double reward = 0.0;

    if(horizon == 0) {
        return reward;
    } 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);
        }

        MonteNode* child = dict_find(tree->children, observation);

        if(child == NULL) {
            printf("wtf?? child was not found.. abort!\n");
            exit(1337);
        }

        reward = random_reward + monte_sample(child, agent, horizon - 1);
    } else if (tree->visits == 0) {
        reward = Agent_playout(agent, horizon);
    }
    else {
        u32 action = _monte_select_action(tree, agent);
        Agent_model_update_action(agent, action);

        MonteNode* child = dict_find(tree->actions, action);
        if(child == NULL) {
            child = monte_create_tree(NODE_TYPE_CHANCE);
          dict_add(tree->actions, action, child);
        }

        if(child == NULL) {
            printf("wtf??? child was not found.. abort!\n");
            exit(1338);
        }

        reward = monte_sample(child, agent, horizon);
    }
    tree->mean = (reward + (tree->visits * tree->mean)) / (tree->visits + 1.0);
    tree->visits = tree->visits + 1;

    return reward;
}

#endif
```

Listing 5.9: search.c

```c
#ifndef _MONTE_NODE_H_
#define _MONTE_NODE_H_

#define NODE_TYPE_CHANCE 0
#define NODE_TYPE_DECISION 1

#include "dict.h"
#include "../agent/agent.h"
#include "../_utils/types.h"

typedef struct MonteNode {
    double mean ;
    int type;
    int visits;

    dict_t children;
    dict_t actions;
} MonteNode;

MonteNode* monte_create_tree(u32 nodeType);
u32 _monte_select_action(MonteNode* tree, Agent* agent);
float monte_sample(MonteNode* tree, Agent* agent, u32 horizon);

```

```

#endif
```

Listing 5.10: monte$_{n}ode.h$

```
#ifndef _DICT_H_
#define _DICT_H_

#include <stddef.h>
#include <stdlib.h>

typedef struct MonteNode MonteNode;

typedef struct dict_entry_s {
    int key;
    MonteNode* value;
} dict_entry_s;

typedef struct dict_s {
    int len;
    int cap;
    dict_entry_s *entry;
} dict_s, *dict_t;

int dict_find_index(dict_t, const int);

MonteNode* dict_find(dict_t, const int);

void dict_add(dict_t, const int, MonteNode*);

dict_t dict_new(void);

void dict_free(dict_t);

#endif
```

Listing 5.11: dict.h

```
#include <stddef.h>
#include <stdlib.h>
#include "dict.h"
#include "monte_node.h"

int dict_find_index(dict_t dict, const int key) {
    for (int i = 0; i < dict->len; i++) {
        if (dict->entry[i].key == key) {
            return i;
        }
    }
    return -1;
}

MonteNode* dict_find(dict_t dict, const int key) {
    int idx = dict_find_index(dict, key);
    return idx == -1 ? NULL : dict->entry[idx].value;
}

```

```c
void dict_add(dict_t dict, const int key, MonteNode* value) {
    int idx = dict_find_index(dict, key);
    if (idx != -1) {
        dict->entry[idx].value = value;
        return;
    }
    if (dict->len == dict->cap) {
        dict->cap *= 2;
        dict->entry = realloc(dict->entry, dict->cap * sizeof(dict_entry_s));
    }
    dict->entry[dict->len].key = key;
    dict->entry[dict->len].value = value;
    dict->len++;
}

dict_t dict_new(void) {
    dict_s proto = {0, 10, malloc(10 * sizeof(dict_entry_s))};
    dict_t d = malloc(sizeof(dict_s));
    *d = proto;
    return d;
}

void dict_free(dict_t dict) {

    for (int i = 0; i < dict->len; i++) {
        free(dict->entry[i].value);
    }

    free(dict->entry);
    free(dict);
}
```

Listing 5.12: dict.c

5.5 Predict

```c
#ifndef _CTW_TREE_
#define _CTW_TREE_

#include "context_tree_node.h"
#include "ctw_list.h"

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

ContextTree *ctw_create(uint32_t);

void ctw_free(ContextTree *);

void ctw_clear(ContextTree *);

```

```
uint64_t ctw_size(ContextTree *);

void ctw_print(ContextTree *);

void ctw_update_context(ContextTree *);

void ctw_revert(ContextTree *, uint64_t);

void ctw_update_symbol(ContextTree *, bool);

void ctw_update_vector(ContextTree *, BitVector *);

void ctw_update_history(ContextTree *, BitVector *);

double ctw_predict_symbol(ContextTree *, bool);

double ctw_predict_vector(ContextTree *, BitVector *);

BitVector *ctw_gen_random_symbols_and_update(ContextTree *, uint64_t);

BitVector *ctw_gen_random_symbols(ContextTree *, uint64_t);

void ctw_revert_history(ContextTree *, uint64_t);

void ctw_save(ContextTree *, char *);

void ctw_load(ContextTree *, char *);

#endif
```

Listing 5.13: context$_{t}ree.h$

```
#include <math.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include "../bit_vector.h"
#include "ctw_list.h"
#include "context_tree.h"

ContextTree *ctw_create(uint32_t depth) {
  ContextTree *tree = (ContextTree *) malloc(sizeof(ContextTree));
  tree->depth = depth;
  tree->root = ctw_node_create();
  tree->history = bv_create();
  tree->context = ctw_list_create();
  return tree;
}

void ctw_free(ContextTree *tree) {
  ctw_node_free(tree->root);
  bv_free(tree->history);
  ctw_list_free(tree->context);
  free(tree);
}

void ctw_clear(ContextTree *tree) {
  bv_clear(tree->history);
  ctw_node_free(tree->root);
  tree->root = ctw_node_create();
  ctw_list_clear(tree->context);
}

uint64_t ctw_size(ContextTree *tree) {
  return ctw_node_size(tree->root);
}

void ctw_print(ContextTree *tree) {
  printf("Context Tree { depth: %d, size: %llu}\n", tree->depth, ctw_size(tree));
  printf("History: ");
  bv_print(tree->history);
}

void ctw_update_context(ContextTree *tree) {
  if (tree->history->size < tree->depth) {
    perror("Not enough history to update context\n");
  }
  ctw_list_free(tree->context);
  tree->context = ctw_list_create();
  ctw_list_push(tree->context, tree->root);
  ContextTreeNode *node = tree->root;
  uint64_t update_depth = 1;
  int64_t i;
  for (i = tree->history->size-1; i >= 0; i--) {
    bool symbol = bv_test(tree->history, i);

    if (symbol && node->one_child != NULL) {
      node = node->one_child;
    } else if (!symbol && node->zero_child != NULL) {
      node = node->zero_child;
    } else {
      ContextTreeNode *new_node = ctw_node_create();
      if (symbol) {
  node->one_child = new_node;
      } else {
  node->zero_child = new_node;
      }
      node = new_node;
    }
    ctw_list_push(tree->context, node);
    update_depth += 1;
    if (update_depth > tree->depth) {
      break;
    }
  }
}

void ctw_revert(ContextTree *tree, uint64_t n) {
  uint64_t i;
  for (i = 0; i < n; i++) {
    if (tree->history->size == 0) {
      return;
    }

    bool symbol = bv_pop(tree->history);

    if (tree->history->size >= tree->depth) {
      ctw_update_context(tree);

      int64_t j;
      for (j = tree->depth-1; j >= 0; j--) {
  ctw_node_revert(ctw_list_get(tree->context, j), symbol);
      }
    }
  }
}

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_update_vector(ContextTree *tree, BitVector *symbols) {
  uint64_t i;
  for (i = 0; i < symbols->size; i++) {
    bool symbol = bv_test(symbols, i);
    ctw_update_symbol(tree, symbol);
  }
}

void ctw_update_history(ContextTree *tree, BitVector *symbols) {
  bv_append(tree->history, symbols);
}

double ctw_predict_symbol(ContextTree *tree, bool symbol) {
  if (tree->history->size + 1 <= tree->depth) {
    return 0.5;
  }
  double prob_history = tree->root->log_probability;
  ctw_update_symbol(tree, symbol);
  double prob_sequence = tree->root->log_probability;
  ctw_revert(tree, 1);
  return exp(prob_sequence - prob_history);
}

double ctw_predict_vector(ContextTree *tree, BitVector *symbols) {
  if (tree->history->size + symbols->size <= tree->depth) {
    return pow(0.5, symbols->size);
  }

  double prob_history = tree->root->log_probability;
  ctw_update_vector(tree, symbols);
  double prob_sequence = tree->root->log_probability;
  ctw_revert(tree, symbols->size);
  return exp(prob_sequence - prob_history);
}
```

```c
BitVector *ctw_gen_random_symbols_and_update(ContextTree *tree, uint64_t n) {
  BitVector *symbols = bv_create();
  uint64_t i;
  for (i = 0; i < n; i++) {
    double p = ((double) rand()) / ((double) RAND_MAX);
    bool symbol;
    if (p < ctw_predict_symbol(tree, true)) {
      symbol = true;
    } else {
      symbol = false;
    }
    bv_push(symbols, symbol);
    ctw_update_symbol(tree, symbol);
  }
  return symbols;
}

BitVector *ctw_gen_random_symbols(ContextTree *tree, uint64_t n) {
  BitVector *symbols = ctw_gen_random_symbols_and_update(tree, n);
  ctw_revert(tree, n);
  return symbols;
}

void ctw_revert_history(ContextTree *tree, uint64_t n) {

  if (tree->history->size < n) {
    perror("not enough history to revert\n");
  }
  uint64_t i;
  for (i = 0; i < n; i++) {
    bv_pop(tree->history);
  }
}

void ctw_save(ContextTree *tree, char *file_name) {
  uint64_t count = 0;

  FILE *fp = fopen(file_name, "w");
  fwrite(&(tree->depth), sizeof(uint32_t), 1, fp);
  bv_save(tree->history, fp);

  CTWNodeList *stack = ctw_list_create();
  ctw_list_push(stack, tree->root);
  while (stack->size > 0) {
    ContextTreeNode *node = ctw_list_pop(stack);
    fwrite(&(node->log_kt), sizeof(double), 1, fp);
    fwrite(&(node->log_probability), sizeof(double), 1, fp);
    fwrite(&(node->ones_in_history), sizeof(uint32_t), 1, fp);
    fwrite(&(node->zeroes_in_history), sizeof(uint32_t), 1, fp);

    bool zero_child;
    bool one_child;

    if (node->one_child != NULL) {
      ctw_list_push(stack, node->one_child);
      one_child = true;
    } else {
      one_child = false;
    }

    if (node->zero_child != NULL) {
      ctw_list_push(stack, node->zero_child);
      zero_child = true;
    } else {
      zero_child = false;
    }

    fwrite(&zero_child, sizeof(bool), 1, fp);
    fwrite(&one_child, sizeof(bool), 1, fp);
    count += 1;
  }
  fclose(fp);
  printf("Wrote %llu nodes to %s\n", count, file_name);
}

void ctw_load(ContextTree *tree, char *file_name) {
  ctw_clear(tree);

  uint64_t count = 0;
  FILE *fp = fopen(file_name, "r");
  fread(&(tree->depth), sizeof(uint32_t), 1, fp);
  bv_load(tree->history, fp);
  printf("Read history\n");

  CTWNodeList *stack = ctw_list_create();
  ContextTreeNode *parent = NULL;
  do {
    ContextTreeNode *node = ctw_node_create();
    if (stack->size == 0 && parent == NULL) {
      tree->root = node;
    }
    if (parent != NULL) {
      parent->zero_child = node;
    } else if (stack->size != 0) {
      ContextTreeNode *one_parent = ctw_list_pop(stack);
      one_parent->one_child = node;
    }

    fread(&(node->log_kt), sizeof(double), 1, fp);
    fread(&(node->log_probability), sizeof(double), 1, fp);
    fread(&(node->ones_in_history), sizeof(uint32_t), 1, fp);
    fread(&(node->zeroes_in_history), sizeof(uint32_t), 1, fp);

    bool zero_child;
    fread(&zero_child, sizeof(bool), 1, fp);
    if (zero_child) {
      parent = node;
    } else {
      parent = NULL;
    }

    bool one_child;
    fread(&one_child, sizeof(bool), 1, fp);
    if (one_child) {
      ctw_list_push(stack, node);
    }

    count += 1;
  } while (stack->size > 0 || parent != NULL);
  fclose(fp);

  printf("Read %llu nodes to %s\n", count, file_name);
}
```

Listing 5.14: context$_tree.c$

```
#ifndef _CTW_NODE_
#define _CTW_NODE_

#include <stdint.h>
#include <stdbool.h>
#include "../bit_vector.h"

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;

ContextTreeNode *ctw_node_create();

void ctw_node_free(ContextTreeNode *);

bool ctw_node_is_leaf(ContextTreeNode *);

uint32_t ctw_node_visits(ContextTreeNode *);

double ctw_node_log_kt_multiplier(ContextTreeNode *, bool);

void ctw_node_update_log_probability(ContextTreeNode *);

void ctw_node_revert(ContextTreeNode *, bool);

uint32_t ctw_node_size(ContextTreeNode *);

void ctw_node_update(ContextTreeNode *, bool);

void ctw_node_print(ContextTreeNode *);
#endif
```

Listing 5.15: context$_tree_node.h$

```
#include <stdlib.h>
#include <math.h>
#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>
#include <errno.h>
#include "context_tree_node.h"

```

```
ContextTreeNode *ctw_node_create() {
  ContextTreeNode *node = (ContextTreeNode *) malloc(sizeof(ContextTreeNode));
  node->log_kt = 0.0;
  node->log_probability = 0.0;
  node->ones_in_history = 0;
  node->zeroes_in_history = 0;
  node->zero_child = NULL;
  node->one_child = NULL;
  return node;
}

void ctw_node_free(ContextTreeNode *node) {
  if (node == NULL) {
    return;
  }
  ctw_node_free(node->zero_child);
  ctw_node_free(node->one_child);
  free(node);
}

bool ctw_node_is_leaf(ContextTreeNode *node) {
  return node->zero_child == NULL && node->one_child == NULL;
}

uint32_t ctw_node_visits(ContextTreeNode *node) {
  return node->ones_in_history + node->zeroes_in_history;
}

double ctw_node_log_kt_multiplier(ContextTreeNode *node, bool symbol) {
  uint32_t numerator;
  if (symbol) {
    // symbol is 1
    numerator = node->ones_in_history;
  } else {
    // symbol is 0
    numerator = node->zeroes_in_history;
  }

  uint32_t denominator = ctw_node_visits(node);
  return log((numerator + 0.5) / (denominator + 1.0));
}

void ctw_node_update_log_probability(ContextTreeNode *node) {
  if (ctw_node_is_leaf(node)) {
    node->log_probability = node->log_kt;
  } else {
    double log_child_prob = 0.0;
    if (node->zero_child != NULL) {
      log_child_prob += node->zero_child->log_probability;
    }
    if (node->one_child != NULL) {
      log_child_prob += node->one_child->log_probability;
    }

    double a, b;
    if (node->log_kt >= log_child_prob) {
      a = node->log_kt;
      b = log_child_prob;
```

```c
    } else {
      a = log_child_prob;
      b = node->log_kt;
    }
    node->log_probability = log(0.5) + a + log1p(exp(b - a));
  }
}

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);
}

uint32_t ctw_node_size(ContextTreeNode *node) {
  uint32_t zero_size = 0;
  if (node->zero_child != NULL) {
    zero_size = ctw_node_size(node->zero_child);
  }

  uint32_t one_size = 0;
  if (node->one_child != NULL) {
    one_size = ctw_node_size(node->one_child);
  }

  return 1 + zero_size + one_size;
}

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;
  }
}

void ctw_node_print(ContextTreeNode *node) {
  if (node == NULL) {
    printf("CTWN NULL\n");
    return;
  }
  printf("----------\n");
  printf("0: %u  1: %u\n", node->zeroes_in_history, node->ones_in_history);
  printf("probs: %f %f\n", node->log_probability, node->log_kt);
  ctw_node_print(node->zero_child);
  ctw_node_print(node->one_child);
}
```

Listing 5.16: context$_{t}ree_{n}ode.c$

```c
#ifndef _CTW_LIST_
#define _CTW_LIST_

#include <stdint.h>
#include "context_tree_node.h"

typedef struct CTWNodeList {
  uint64_t size;
  uint64_t capacity;
  ContextTreeNode **nodes;
} CTWNodeList;

CTWNodeList *ctw_list_create();

void ctw_list_free(CTWNodeList *);

void __ctw_list_check_bounds(CTWNodeList *, uint64_t);

void __ctw_list_grow(CTWNodeList *);

ContextTreeNode *ctw_list_get(CTWNodeList *, uint64_t);

void ctw_list_set(CTWNodeList *, uint64_t, ContextTreeNode *);

void ctw_list_push(CTWNodeList *, ContextTreeNode *);

ContextTreeNode *ctw_list_pop(CTWNodeList *);

void ctw_list_clear(CTWNodeList *);

#endif
```

Listing 5.17: ctw$_{l}ist.h$

```c
#include <errno.h>
#include <stdint.h>
#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) {
  if (ctw_list->size == ctw_list->capacity) {
    __ctw_list_grow(ctw_list);
  }
  ctw_list->nodes[ctw_list->size] = node;
  ctw_list->size += 1;
}

ContextTreeNode *ctw_list_pop(CTWNodeList *ctw_list) {
  if (ctw_list->size == 0) {
    perror("The list is empty, can't pop\n");
  }
  ContextTreeNode *node = ctw_list->nodes[ctw_list->size -1];
  ctw_list->size -= 1;
  return node;
}

void ctw_list_clear(CTWNodeList *ctw_list) {
  // Just sets size to 0
  // we may want to resize the array if memory is a problem
  ctw_list->size = 0;
}
```

Listing 5.18: ctw$_{l}ist.c$

5.6 Environment

```c
#ifndef ENV_H
    #define ENV_H
    #include "../_utils/types.h"

    extern const void * Environment;

    // define any functions here
    u32  action_bits          ( void * _self );
    u32  observation_bits     ( void * _self );
    u32  reward_bits          ( void * _self );
    u32  percption_bits       ( void * _self );

    u08  is_valid_action      ( void * _self, u32   action );
    u08  is_valid_observation ( void * _self, u32   observation );
    u08  is_valid_reward      ( void * _self, u32   reward );

    u32  maximum_action       ( void * _self );
    u32  maximum_observation  ( void * _self );
    u32  maximum_reward       ( void * _self );

    u32  minimum_action       ( void * _self );
    u32  minimum_observation  ( void * _self );
    u32  minimum_reward       ( void * _self );

    u32 LG2 ( u32 x );

#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
}; //-------------------------------------------------------
```

```

const void * Environment = & _Environment;

// def action_bits():
u32  action_bits ( void * _self )
{
    struct Environment * self = _self;
    assert ( self->_valid_actions != NULL);

    u32 max_action = 0;

    foreach ( u32 const * action, self->_valid_actions )
        max_action = *action && *action > max_action ? *action : max_action;

    return LG2( max_action );

}//-----------------------------------------------------------------------------

// def observation_bits():
u32  observation_bits ( void * _self )
{
    struct Environment * self = _self;
    assert ( self->_valid_observations != NULL);

    u32 max_observation = 0;

    foreach ( u32 const * observation, self->_valid_observations )
        max_observation = *observation && *observation > max_observation ? *observation : max_observation;

    return LG2( max_observation );

}//-----------------------------------------------------------------------------

// def reward_bits():
u32  reward_bits ( void * _self )
{
    struct Environment * self = _self;
    assert ( self->_valid_rewards != NULL);

    u32 max_reward = 0;

    foreach ( u32 const * reward, self->_valid_rewards )
        max_reward = *reward && *reward > max_reward ? *reward : max_reward;

    return LG2( max_reward );

}//-----------------------------------------------------------------------------

// def perception_bits():
u32  percption_bits ( void * _self )
{
    struct Environment * self = _self;
    return reward_bits(self) + action_bits(self);

}//-----------------------------------------------------------------------------

// check if the action is valid
```

```
u08  is_valid_action ( void * _self, u32   action )
{

    struct Environment * self = _self;

    foreach( u32 const * a , self->_valid_actions)
        if ( * a == action ) return TRUE;

    return FALSE;

}//-----------------------------------------------------------------------

// find out if the observation is a valid one
u08  is_valid_observation ( void * _self, u32   observation )
{

    struct Environment * self = _self;

    foreach( u32 const * o , self->_valid_observations)
        if ( * o == observation ) return TRUE;

    return FALSE;

}//-----------------------------------------------------------------------

// check if the action is a valid action
u08  is_valid_reward ( void * _self, u32   reward )
{

    struct Environment * self = _self;

    foreach( u32 const * r , self->_valid_rewards)
        if ( * r == reward ) return TRUE;

    return FALSE;

}//-----------------------------------------------------------------------

// Get maximum action
u32  maximum_action ( void * _self )
{
    struct Environment * self = _self;
    u16 idx = 0;

    foreach ( u32 const * x , self->_valid_actions)
        idx++;

    return self->_valid_actions[idx];

}//-----------------------------------------------------------------------

// Get maximum observation
u32  maximum_observation ( void * _self )
{
    struct Environment * self = _self;
    u16 idx = 0;

    foreach ( u32 const * x , self->_valid_observations)
```

```
        idx++;

    return self->_valid_observations[idx];

}//----------------------------------------------------------------------------


// Get maximum reward
u32  maximum_reward ( void * _self )
{
    struct Environment * self = _self;
    u16 idx = 0;

    foreach ( u32 const * x , self->_valid_rewards)
    {
        if ( *x ) idx++;
        else break;
    }

    return self->_valid_rewards[idx];

}//----------------------------------------------------------------------------

// Get minimum action
u32  minimum_action ( void * _self )
{
    struct Environment * self = _self;
    return self->_valid_actions[0];

}//----------------------------------------------------------------------------

// Get minimum observation
u32  minimum_observation ( void * _self )
{
    struct Environment * self = _self;
    return self->_valid_observations[0];

}//----------------------------------------------------------------------------

// Get minimum reward
u32  minimum_reward ( void * _self )
{
    struct Environment * self = _self;
    return self->_valid_rewards[0];

}//----------------------------------------------------------------------------

u32 LG2 ( u32 x )
{
    #define LT(n) n, n, n, n, n, n, n, n, n, n, n, n, n, n, n, n
    const char LogTable256[256] =
    {
        -1, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3,
        LT(4), LT(5), LT(5), LT(6), LT(6), LT(6), LT(6),
        LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7)
    };

    register u32 ret, t, tt; // temp var2

    if ((tt = x >> 16))
        ret = (t = tt >> 8) ? 24 + LogTable256[t] : 16 + LogTable256[tt];
    else
        ret = (t = x >> 8) ? 8 + LogTable256[t] : LogTable256[x];

    return (u32) ret;
}//-----------------------------------------------------------------------------
```

Listing 5.21: environment.c

```
#ifndef COIN_FLIP_H
    #define COIN_FLIP_H
    #define HEADS_VAL 0x00000001
    #define TAILS_VAL 0x00000000

    #include <stddef.h>

    extern const void * Coin_Flip;
    // Usage
    //      new ( Coin_Flip , (double) probability );
    //


    double __rp();
    u32Tuple * perform_action ( void * _self, u32 action_t );
    static void CF_print(void * _self);

#endif
```

Listing 5.22: coin$_{f}$*lip.h*

```
#ifndef COIN_FLIP_R
    #define COIN_FLIP_R
    #include "../_utils/types.h"
   struct Coin_Flip {
       struct Environment _;
       double probability;
   };
    #define probability(e) (((const struct Coin_Flip*)(e)) -> probability)

    typedef enum { Tails , Heads } _action_enum;
    typedef enum { Tails , Heads } _observation_enum;
    typedef enum { Loss , Win } _reward_enum;

    static double default_probability = 7e-1;
#endif
```

Listing 5.23: coin$_{f}$*lip.r*

```
#include <stdlib.h>
#include <time.h>
#include <assert.h>
#include <stdio.h>
#include <stddef.h>
#include "../_utils/types.h"
#include "../_utils/macros.h"
```

```
#include "class.h"
#include "class.r"
#include "environment.h"
#include "environment.r"
#include "coin_flip.h"
#include "coin_flip.r"

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];

   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;

    if (__rp() > probability(self) ){
        observation_t = 1;
    } else {
        observation_t = 0;
    }

    reward_t = ( action_t == observation_t ) ? 1 : 0;

    #ifdef DEBUG
        TRACE ( "Observation = %d Reward = %d\n", observation_t, reward_t );
    #endif

    self -> _ . _observation    = observation_t;
    self -> _ . _reward         = reward_t;

    u32Tuple* tuple = calloc (1, sizeof(u32Tuple));
    tuple -> first = observation_t;
    tuple -> second = reward_t;

    return tuple;
}

static void CF_print(void * _self)
{
    struct Coin_Flip * self = _self;
    printf ("Prediction = %x, Observation = %x, Reward = %x\n",
            action(self),observation(self),reward(self));
}

void * CF_cpy ( void * _self )
{
    struct Coin_Flip * self = _self;
    return new ( Coin_Flip , probability(_self) );
}

static const struct Class _Coin_Flip = {
    sizeof(struct Coin_Flip), CF_init, NULL, CF_cpy, NULL
};

const void * Coin_Flip = & _Coin_Flip;
```

Listing 5.24: coin$_{f}$*lip.c*

5.7 Utility

```
#ifndef MACRO_H
    #define MACRO_H

    #include "types.h"

```

```
#define BLOCK_START {

#define BLOCK_END   }

#define FALSE (0)

#define TRUE (!FALSE)

#define MIN(a, b)  (((a) < (b)) ? (a) : (b))

#define MAX(a, b)  (((a) > (b)) ? (a) : (b))

#define ABS(a)     (((a) < 0) ? -(a) : (a))

// % used for assertions. e.g assert(IMPLIES(n > 0, array != NULL));
#define IMPLIES(x, y) (!(x) || (y))

// % gt 1 => x > y, eq 0 => x == y , lt 0 => x < y
#define COMPARE(x, y) (((x) > (y)) - ((x) < (y)))

// % return true if x is greater than 1
#define SIGN(x) COMPARE(x, 0)

// % determine the size of an array
#define ARRAY_SIZE(a) (sizeof(a) / sizeof(*a))

// % swap 2 values T is the type. e.g: SWP(a,b,int)
#define SWAP(x, y, T) do { T tmp = (x); (x) = (y); (y) = tmp; } while(0)

// % name says it all T = type
#define QSORT(a, b, T) do { if ((a) > (b)) SWAP((a), (b), T); } while (0)

// % i dont actually understand this one lol
#define SET(d, n, v) do{ size_t i_, n_; for (n_ = (n), i_ = 0; n_ > 0; --n_, ++i_) (d)[i_] = (v); } while(0)


#define ZERO(d, n) SET(d, n, 0)

// % For Each Loop.  pretty easy to get your head around.
#define foreach(item, array) \
    for(int keep=1, \
           count=0,\
           size=sizeof (array)/sizeof *(array); \
        keep && count != size; \
        keep = !keep, count++) \
    for(item = (array)+count; keep; keep = !keep)

// % For Debugging it prints a stack trace.
#if defined NDEBUG
    #define TRACE( format, ... ) ( ( void ) 0 )
#else
    #define TRACE( format, ... )   printf( "%s::%s(%d)" format, __FILE__, __FUNCTION__,  __LINE__, __VA_ARGS__ )
#endif

// % Print error info and exit
#define ERR(source) (fprintf(stderr,"%s:%d\n",__FILE__,__LINE__),\
                        perror(source),kill(0,SIGKILL),     \
                        exit(EXIT_FAILURE))

    // % Debug
    #define DBG(source) (fprintf(stderr,"%s:%d\t",__FILE__,__LINE__),\
                            perror(source),

    // % Call function and exit if return value != result
    #define CALL_AND_CHECK(function, result)\
    do {\
    if (function != result)\
    {\
    ERR(#function);\
    }\
    } while (0)

    // % Call function and exit if error occured
    #define CALL_AND_EXIT_ON_ERR(function) CALL_AND_CHECK(function, 0)
#endif
```

Listing 5.25: macros.h

```
#ifndef INTDEF
    #include <stdint.h>
    #define INTDEF
    #ifdef uint128_t
        typedef uint128_t       u128;
    #endif
    typedef uint64_t            u64;
    typedef uint32_t            u32;
    typedef uint16_t            u16;
    typedef uint8_t             u08;

    #ifdef int128_t
        typedef int_128_t       s128;
    #endif
    typedef int64_t             s64;
    typedef int32_t             s32;
    typedef int16_t             s16;
    typedef int8_t              s08;
#endif

#ifndef U_ENUM
#define U_ENUM
    typedef enum { action_update, percept_update } update_enum;
#endif

#ifndef TUPLE_32
#define TUPLE_32
    typedef struct {
        u32             first;
        u32             second;
    } u32Tuple;
#endif

#ifndef UNDO
#define UNDO
    typedef struct {
        u32             age;
        u32             total_reward;
        u32             history_size;
        update_enum     last_update;
    } AgentUndo;
#endif
```

Listing 5.26: types.h