cpp-AIF

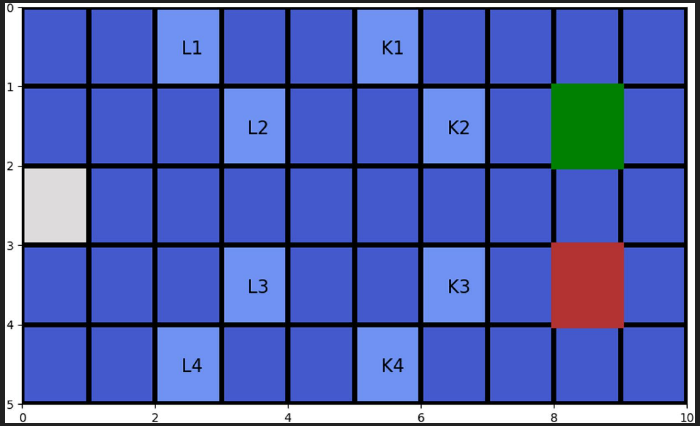
- C++ header-only library
- simulates active inference agents in discrete space and time, using partially-observed Markov Decision Processes (POMDPs) as a generative model class
- implements a multi-threads parallelization of the most computationally intensive operations

https://github.com/fgregoretti/Cpp-AIF

the need for cpp-AIF

- existing softwares implementing active inference like DEM (Matlab) and pymdp (Python) are far away from being efficient
- this is more evident when the sizes of states and outcomes increase, and the computational cost of a single operation becomes more demanding

cpp-AIF vs pymdp



extended epistemic chaining problem:
it is a 10 × 5 grid where a sequence of three cues reveals the locations of both a rewarding (indicated by the colour green) and a penalising (indicated by the colour red) outcome

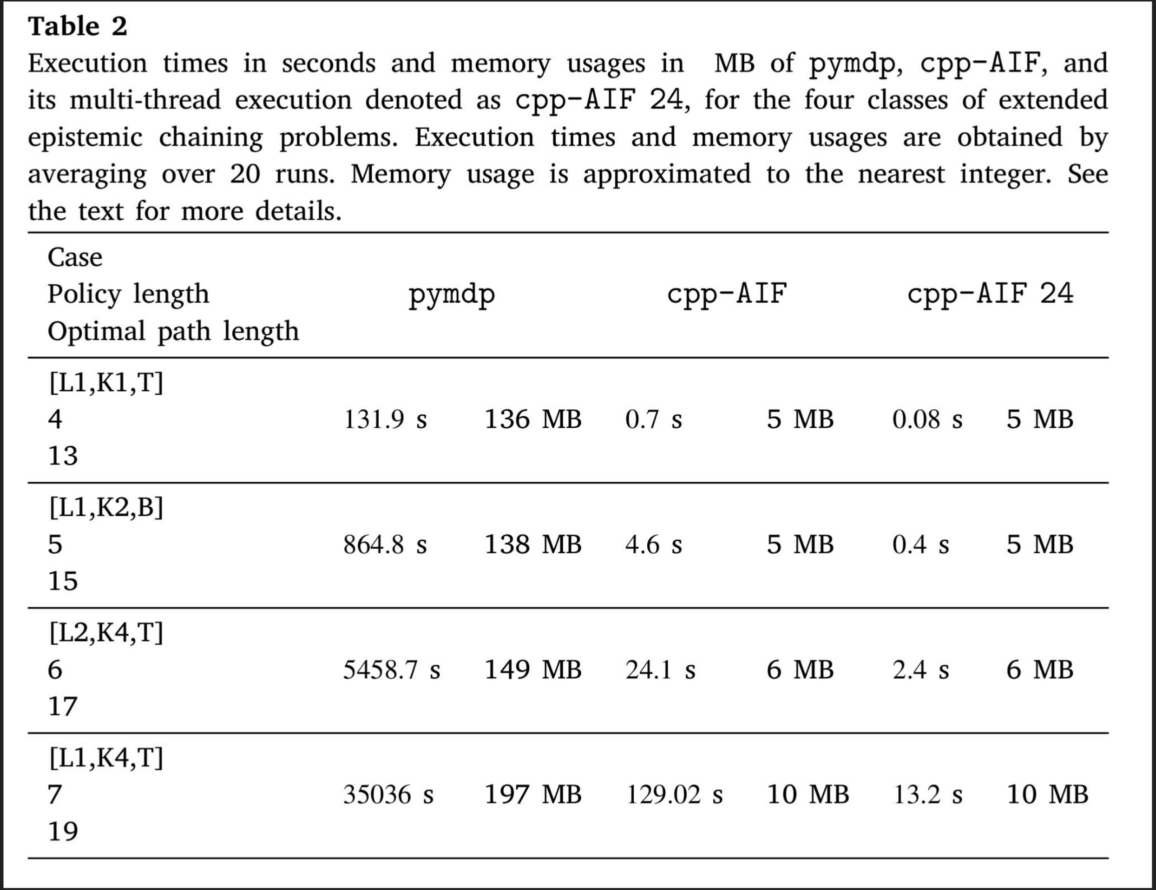


Table 2
Execution times in seconds and memory usages in MB of pymdp, cpp-AIF, and its multi-thread execution denoted as cpp-AIF 24, for the four classes of extended epistemic chaining problems. Execution times and memory usages are obtained by averaging over 20 runs. Memory usage is approximated to the nearest integer. See the text for more details.

Case Policy length Optimal path length	pymdp		cpp-AIF		cpp-AIF 24	
[L1,K1,T] 4 13	131.9 s	136 MB	0.7 s	5 MB	0.08 s	5 MB
[L1,K2,B] 5 15	864.8 s	138 MB	4.6 s	5 MB	0.4 s	5 MB
[L2,K4,T] 6 17	5458.7 s	149 MB	24.1 s	6 MB	2.4 s	6 MB
[L1,K4,T] 7 19	35036 s	197 MB	129.02 s	10 MB	13.2 s	10 MB

cpp-AIF features

The library is designed to provide:

- scalability
- extensibility
- portability
- high efficiency

cpp-AIF architecture

- `MDP` class is the main API: allows the specification of the generative model, performs inference and generates actions
- `States`, `likelihood`, `Transitions`, `Priors`, and `Beliefs` classes allow to specify the parameters of the generative model:
  - `States` hidden states and observations
  - `likelihood` the likelihood of observations (**A**)
  - `Transitions` the transition function (**B**)
  - `Priors` the ‘a priori’ probability over observations (**C**)
  - `Beliefs` the initial beliefs (**D**)

cpp-AIF data structure

generative model distributions as well as expectations of hidden states, states and observations are represented as:

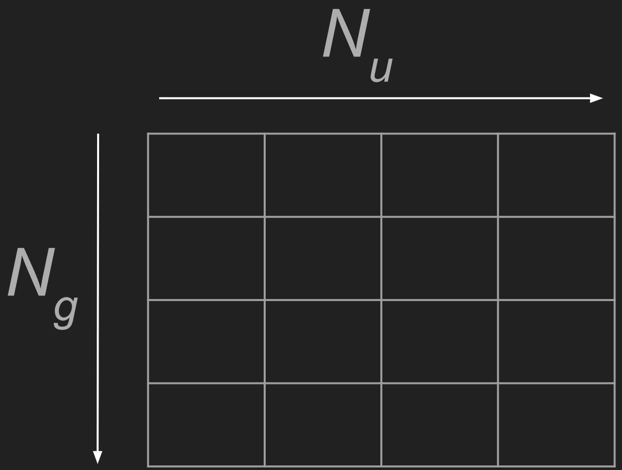
- vector of vectors (**A** and **B**)
- vector (all the others)

of objects (instances of the classes `likelihood`, `Transitions`, `Priors`, `Beliefs` and `States`)

cpp-AIF data structure

N_g the number of observation factors

N_u the number of control states

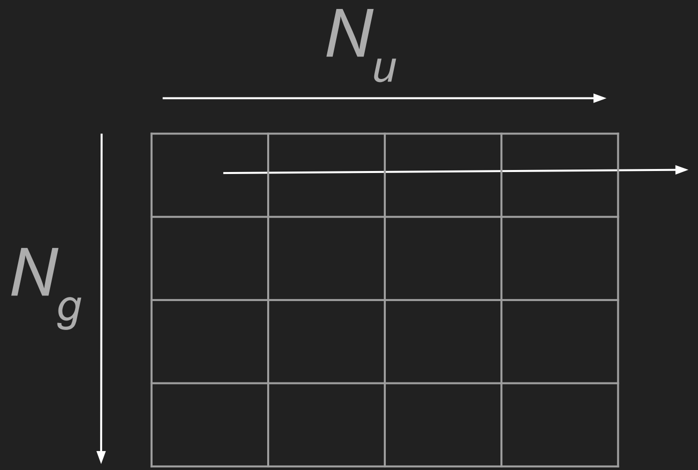


A is a vector of size N_g whose each element will contain a vector of size N_u (or **1** if the factor is uncontrollable) of (pointers to) `likelihood` objects

cpp-AIF data structure

N_g the number of observation factors

N_u the number of control states

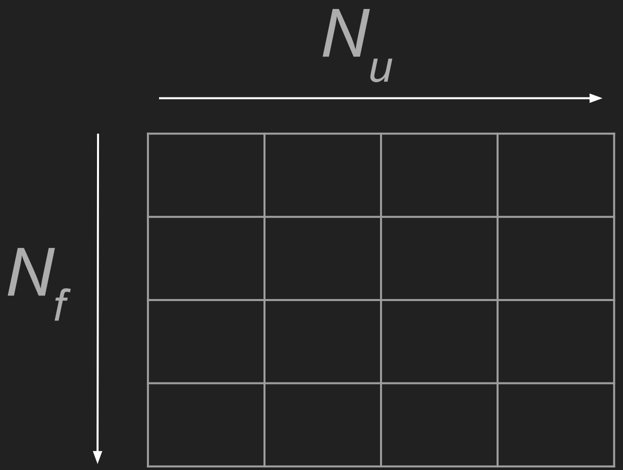


each `likelihood` object is a multidimensional array encoding conjunctive relationships between combinations of hidden states and observations

cpp-AIF data structure

N_f the number of hidden state factors

N_u the number of control states



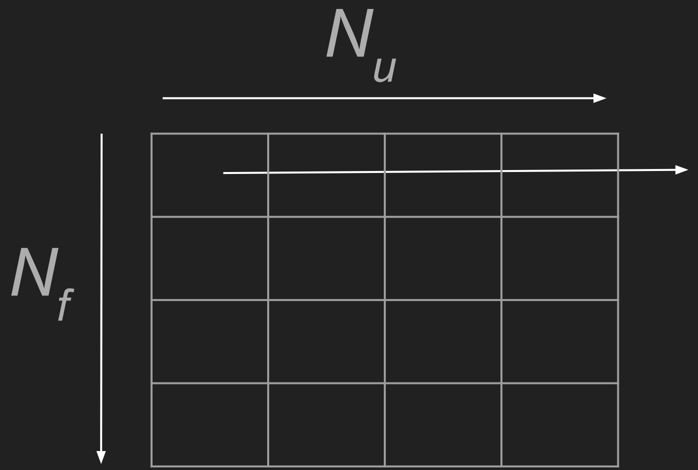
B is a vector of size N_f whose each element will contain a vector of size N_u (or **1** if the factor is uncontrollable) of (pointers to) `Transitions` objects

cpp-AIF data structure

N_f the number of hidden state factors

N_u the number of control states

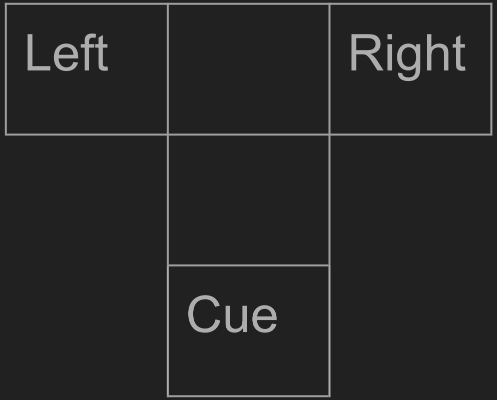
$\mathbf{N_s}$ the vector containing the number of hidden states for each factor



each `Transitions` object is a 2D array of size $N_s(f)$ x $N_s(f)$ encoding the conditional dependencies between states for a given factor *f*

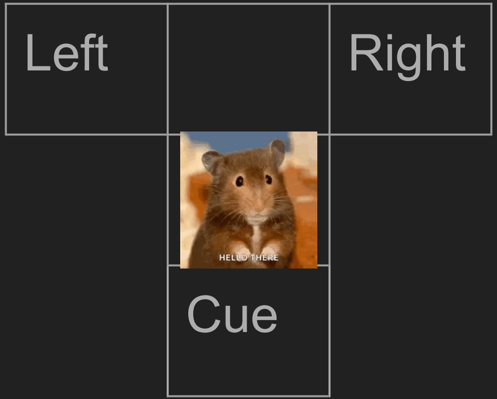
cpp-AIF use case

T-Maze: in this problem, we examine an agent traversing a T-maze with three arms, with the agent's initial position being the central location. The bottom arm of the maze includes a cue that provides information about the probable location of a reward in either of the two top arms



cpp-AIF use case

T-Maze: in this problem, we examine an agent traversing a T-maze with three arms, with the agent's initial position being the central location. The bottom arm of the maze includes a cue that provides information about the probable location of a reward in either of the two top arms



T-Maze

MDP class allows you to run active inference processes

Firstly, you need to:

- initialize state **S** arrays by setting up true initial state
- build a generative model in terms of **A**, **B**, **D**, and **C** arrays
- (optionally) build a vector of policies

T-Maze: environment

$N_f = 2$

First hidden state factor is a variable with values in (0, 1, 2, 3) corresponding to (Center, Left, Right, Bottom): encodes the agent's present location

$S^0 = 0$ if the agent is in the center location

Second hidden state factor is a variable with values in (0, 1) corresponding to (Reward on the Left, Reward on the Right): encodes the reward condition of the trial

$S^1 = 0$ if the trial has reward on the left

$\mathbf{N_s} = [4, 2]$

T-Maze: environment

set the true Initial State at time step 0

we define a vector of two (pointers to) `States` objects, one for each factor

```
int T = 3;
std::vector<States*> S;

States s0(T);
s0.Zeros();
s0.Set(0,0);
S.push_back(&s0);


States s1(T);
s1.Zeros();
s1.Set(0,0);
S.push_back(&s1);
```

T-Maze: environment

set the true Initial State at time step 0

we define a vector of two (pointers to) `States` objects, one for each factor

```
int T = 3;
std::vector<States*> S;

States s0(T);
s0.Zeros();
s0.Set(0,0);
S.push_back(&s0);


States s1(T);
s1.Zeros();
s1.Set(0,0);
S.push_back(&s1);
```

$\longrightarrow$ `States(unsigned int T_)`
T_ temporal horizon size of the integer array

T-Maze: environment

set the true Initial State at time step 0

we define a vector of two (pointers to) `States` objects, one for each factor

```
int T = 3;
std::vector<States*> S;

States s0(T);
s0.Zeros();
s0.Set(0,0);
S.push_back(&s0);


States s1(T);
s1.Zeros();
s1.Set(0,0);
S.push_back(&s1);
```

```
void Set(unsigned int val, unsigned int t);
```
Assign a value val to a specific time step t

T-Maze: environment

set the true Initial State at time step 0

we define a vector of two (pointers to) `States` objects, one for each factor

```
int T = 3;
std::vector<States*> S;

States s0(T);
s0.Zeros();
s0.Set(0,0);
S.push_back(&s0);

States s1(T);
s1.Zeros();
s1.Set(0,0);
S.push_back(&s1);
```

```
void push_back(const value_type& val);
```
Adds a new element (s0) at the end of the vector

T-Maze: outcome mapping

$N_g = 2$

First observation factor is a variable with values in (0, 1, 2, 3) corresponding to (CueCenter, CueLeft, CueRight, CueBottom): provide agent's position in the maze

Second observation factor is a variable with values in (0, 1, 2, 3) corresponding to (CueLeft, CueRight, Reward, NoReward): specify the reward levels and where they are deployed in the maze

$\mathbf{N_o} = [4, 4]$

T-Maze: outcome mapping

Each factor-specific **A** array is stored as a multidimensional array with $N_o(m)$ rows and as many lagging dimensions as there are hidden state factors.

Remembering that $\mathbf{N_o}$ = [4, 4] and $\mathbf{N_s}$ = [4, 2]

$\mathbf{A}^0$ and $\mathbf{A}^1$ are two 3-dimensional arrays with dimensions 4 x 4 x 2 ($N_o(0)$ x $N_s(0)$ x $N_s(1)$ and $N_o(1)$ x $N_s(0)$ x $N_s(1)$) and are the same for each action

T-Maze: outcome mapping

we define two likelihood objects of size 4 x 4 x 2 and we fill out them

```
likelihood<double,3> a0(4,4,2);
likelihood<double,3> a1(4,4,2);

a0.Zeros();
a0(0,0,0)=1; a0(1,1,0)=1; a0(2,2,0)=1; a0(3,3,0)=1;
a0(0,0,1)=1; a0(1,1,1)=1; a0(2,2,1)=1; a0(3,3,1)=1;

const double a = .9;
const double b = 1.-a;

const double d = 1.;
const double e = 1.-d;

a1.Zeros();
a1(0,0,0)=0.5; a1(0,3,0)=d; a1(1,0,0)=0.5; a1(1,3,0)=e;
a1(2,1,0)=a;   a1(2,2,0)=b; a1(3,1,0)=b;   a1(3,2,0)=a;
a1(0,0,1)=0.5; a1(0,3,1)=e; a1(1,0,1)=0.5; a1(1,3,1)=d;
a1(2,1,1)=b;   a1(2,2,1)=a; a1(3,1,1)=a;   a1(3,2,1)=b;
```

T-Maze: outcome mapping

we define **A** as a vector of vectors of (pointers to) likelihood objects

```
std::vector<std::vector<likelihood<double,3>*>> A;
```

we define two vectors of (pointers to) likelihood objects each containing one likelihood object (`a0` and `a1` respectively)

```
std::vector<likelihood<double,3>*> A0;
A0.push_back(&a0);
A.push_back(A0);
```

```
std::vector<likelihood<double,3>*> A1;
A1.push_back(&a1);
A.push_back(A1);
```

T-Maze: A

$N_g = 2$

a0
a1

T-Maze: transition distribution

Each factor-specific **B** array ($\mathbf{B}^f$) encodes the probability of transitions between the states of a given factor *f*

$\mathbf{B}^0$ and $\mathbf{B}^1$ are 2D arrays

T-Maze: transition distribution

The first hidden state factor (Location) is within the agent's control

If **U**=[MoveToCenter, MoveToLeft, MoveToRight, MoveToBottom] contain the four actions taking the agent directly to each of the four locations, we have to define the four transition distribution matrix $\mathbf{B^0}$ taking into account that the probability of being in a specific state is determined by the previous state and by the actions taken

$$\begin{tikzcd} & \mathbf{B^0_0} \\ \mathbf{B^0} \arrow[r] & \mathbf{B^0_1} \\ & \mathbf{B^0_2} \\ & \mathbf{B^0_3} \end{tikzcd}$$

T-Maze: transition distribution

```
std::vector<std::vector<double>> B0_0 {
               { 1, 0, 0, 1 },
               { 0, 1, 0, 0 },
               { 0, 0, 1, 0 },
               { 0, 0, 0, 0 }
          };

 std::vector<std::vector<double>> B0_1 {
               { 0, 0, 0, 0 },
               { 1, 1, 0, 1 },
               { 0, 0, 1, 0 },
               { 0, 0, 0, 0 },
          };

 std::vector<std::vector<double>> B0_2 {
               { 0, 0, 0, 0 },
               { 0, 1, 0, 0 },
               { 1, 0, 1, 1 },
               { 0, 0, 0, 0 },
          };

 std::vector<std::vector<double>> B0_3 {
               { 0, 0, 0, 0 },
               { 0, 1, 0, 0 },
               { 0, 0, 1, 0 },
               { 1, 0, 0, 1 },
          };
```

entry (*i*, *j*) in a particular matrix indicates the probability of transitioning to state *i* given that the system was in state *j* taken the action *u*

T-Maze: transition distribution

we define four Transitions objects (we pass the constructor the 2D arrays previously created as parameter)

```
Transitions<double> b0(B0_0);
Transitions<double> b1(B0_1);
Transitions<double> b2(B0_2);
Transitions<double> b3(B0_3);
```

T-Maze: transition distribution

The transition array for the second hidden state factor (reward condition) is a "trivial" identity matrix

we encodes that the reward condition remains unchanged over time, as it is mapped from its current value to the same value in the next time step

we define the identity matrix

```
std::vector<std::vector<double>> eye {
                    { 1., 0. },
                    { 0., 1. }
              };
```

we define one Transitions object

```
Transitions<double> bb(eye);
```

T-Maze: transition distribution

we define **B** as a vector of vectors of (pointers to) Transitions objects

```
std::vector<std::vector<Transitions<double>*>> B;
```

we define two vectors of (pointers to) Transitions objects

```
std::vector<Transitions<double>*> B0;
B0.push_back(&b0);
B0.push_back(&b1);
B0.push_back(&b2);
B0.push_back(&b3);
B.push_back(B0);

std::vector<Transitions<double>*> B1;
B1.push_back(&bb);
B.push_back(B1);
```

T-Maze: B

$N_f=2$

B0_0	B0_1	B0_2	B0_3
bb			

T-Maze: agent's initial beliefs

we encode the agent's initial beliefs regarding its starting location and reward condition: we define two arrays, each corresponding to a specific hidden state factor

```
std::vector<double> D0 = {1., 0., 0., 0.}; /* location states */
std::vector<double> D1 = {1./2, 1./2}; /* cue left, cue right */
```

we define two Beliefs objects

```
Beliefs<double> d0(D0);
Beliefs<double> d1(D1);
```

we define **D** as a vector of (pointers to) Beliefs objects

```
std::vector<Beliefs<double>*> D;
D.push_back(&d0);
D.push_back(&d1);
```

T-Maze: priors

we ensure that the agent is motivated to choose the arm that maximizes the probability of receiving a reward by setting up **priors:** we define two arrays, one for each observation factor

```
std::vector<double> C0 = {1., 1., 1., 1.};
softmax<double>(C0);
```

```
std::vector<double> C1 = {0., 0., 2, -2};
softmax<double>(C1);
```

we define two Priors objects

```
Priors<double> c0(C0);
Priors<double> c1(C1);
```

we define **C** as a vector of (pointers to) Priors objects

```
std::vector<Priors<double>*> C;
C.push_back(&c0);
C.push_back(&c1);
```

T-Maze: policies

we can either create an empty vector of policies and in this case the MDP constructor will generate the policies, or we can build our own vector of policies

```
std::vector<std::vector<int>> V;
```

T-Maze: MDP

we have abstracted many of the computations necessary for active inference into the MDP class

stores the generative model parameters, the agent's current observations and actions, and execute action/perception through functions like `infer_states` and `infer_policies`

to create an instance of the `MDP`, simply call the `MDP` constructor with a list of arguments

```
int seed = 0;
MDP<double,3> mdp(D,S,B,A,C,V,T,64,4,1./4,1,4,seed);
```

T-Maze: active inference

To run the active inference process we can use the basic active inference procedure implemented as MDP class method

```
mdp.active_inference();
```

Users can define a customized `active_inference()`

T-Maze

MDP public members

```
std::vector<std::vector<int>> _st
```

   _st[*i*][*j*] sampled state for factor *j* at time step *i*

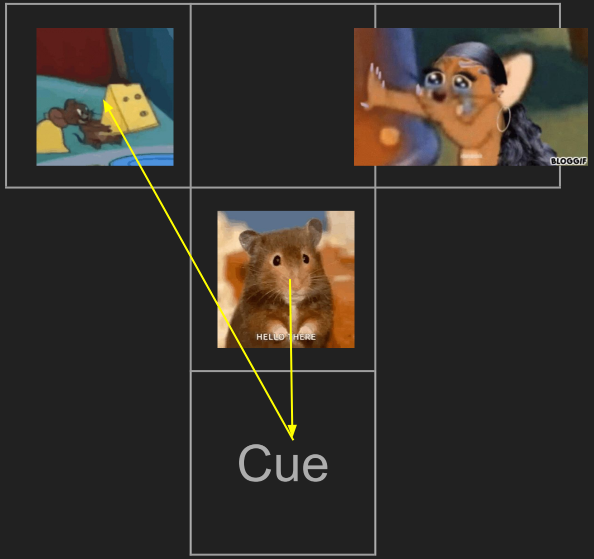
```
std::vector<std::vector<int>> _ot
```

   _ot[*i*][*j*] observed state for factor *j* at time step *i*

```
for (int i = 0; i < T; i++)
  std::cout << "T=" << i+1
            << " Location: [" << mdp._st[i][0] << "] "
            << "Observation: [" << mdp._ot[i][1] << "]"
            << " Action: [" << mdp.getU(i) << "]"
            << std::endl
```

T-Maze

```
T=1 Location: [Center] Observation: [Cue Left] Action: [Move to Bottom]
T=2 Location: [Bottom] Observation: [Cue Left] Action: [Move to Left]
T=3 Location: [Left] Observation: [Reward!] Action: [Move to Bottom]
```


BLOGGIF
HELLO THERE
Cue