CSM: An Improved Algorithm for Instance-based Reinforcement Learning

Feng Liu
Nanjing University
Nanjing, China
fengliu@nju.edu.cn

Haomin Qiu
Nanjing University
Nanjing, China
aquafits@outlook.com

Peng Huang
Nanjing University
Nanjing, China
paulwongpang@foxmail.com

Chongjun Wang
Nanjing University
Nanjing, China
chjwang@nju.edu.cn

Abstract

Instance-based approaches are effective ways to solve reinforcement learning problems. Utile Suffix Memory (USM) algorithm has shown decent results for distinguishing different states from instance chains and generating Q-value of actions of each state, but involving exponentially expanded state space and a number of redundant states. In this paper we propose a new state space compressed algorithm, called Compressed Suffix Memory (CSM) algorithm. CSM algorithm obtains heuristic information of environments by blind exploration, e.g. the path length of most solutions and goal frequencies, to improve efficiency and resist overfitting. We use Boltzmann sampling to balance between exploration and exploitation. Our experiments show that both the efficiency and the effect have been improved a lot by CSM algorithm compared with USM algorithm.

1 Introduction

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal in a provided environment [9]. In many reinforcement learning scenarios such as robotic exploration [8] and autonomous driving [2], the agent is only able to gain partial and noisy observations form environment, so POMDP (partially observable Markov decision process) model is wildly adopted. According to the observations, reward and the historical information, POMDPs provide a rich mathematical approach to solving sequential problems by calculating the Q-value of actions.

As the agent does not directly observe the underlying state, the generation of the state space is the key to POMDP-based reinforcement learning algorithms. Many instance based methods have been put forward, including Nearest Sequence Memory (NSM) algorithm [5] and Utile Suffix Memory (USM) algorithms [4]. USM algorithm presents the state space by tree-nodes in a suffix tree building from the instance chains, and is proved effective maximizing the Q-value of actions. However, the state space of USM algorithm exponentially expands during iteration and comprises many redundant states, which reduces the efficiency. Furthermore, because the ϵ-greedy policy of USM treats the second best and the worst the same, it may lead to overfitting.

In this paper, we propose a new algorithm, called Compressed Suffix Memory (CSM) algorithm, which optimizes the generation of a utile tree and decision process. First, the heuristic information is obtained by the agent during a blind exploration of the environment, e.g., l as the length of the longest non-repetitive observation sequence and p the probability of goal during this exploration. Second, the maximum depth of the suffix tree is limited to doubled l and the minimum instances required to trigger state splitting is deduced from p. Finally, after initializing the agent with a random policy, boltzmann sampling approach will be applied. Experiment has shown that both the efficiency and the effect have been greatly improved by CSM compared with USM.

The paper outline follows. We will briefly review the basics of reinforcement learning, POMDP problems and Utile Suffix Memory (USM) algorithm, in Section 2. Next we propose Compressed Suffix Memory (CSM) algorithm, which solves the weaknesses of USM by new means of utile tree generation, new exploration method and self correction mechanism, in Section 3. We measure the performance of CSM and see improvements comparing with USM, in Section 4. We close in Section 5, with a brief summary and possible means of improvement of CSM.

2 Background

2.1 Reinforcement Learning in Partially Observable Environment

Reinforcement learning is about how an agent learning to map states to actions and produce a maximized reward in a provided environment. In general, a reinforcement learning agent interacts with the environment over time. At each time step t, it determines its state s_t from a state space S, and chooses a best action a_t from an action space A

The 1st International Conference on Distributed Artificial Intelligence (DAI 2019), Beijing, China.

according to the policy π. The agent gets a instant reward r_t according to the reward function $R(s_t, a_t)$ and transfer to the next state s_{t+1} according to the transition probability $T(s_{t+1}|s_t, a_t)$. The reward is normally discounted with factor $\gamma \in (0, 1]$, and the accumulated reward at t_n is defined as $R_{t_n} = \sum_{t=0}^{t_n} \gamma^t r_t$. When a problem satisfies the Markov property, the problem can be formulated as a Markov decision process (MDP), which is defined by the 5-tuple (S, A, T, R, γ).

However, in most cases, an agent cannot directly observe the states of the MDP model underlying the provided environment, but can only deduce a state by an observation. It is necessary to bring in partially observable Markov decision process (POMDP). It defines Ω as a set of observations and O as a set of conditional observation probabilities-mapping current state s_t and previous action a_{t-1} to the probability of current observation o_t. In that way, a POMDP can be defined by the 7-tuple $(S, A, T, R, \Omega, O, \gamma)$.

When the model is given, i.e., all elements of the 7-tuple are known, there are plenty of algorithms to calculate a best policy [7]. However, in model-free methods, an agent can learn with trail-and-error from experience directly and a policy can generated before grasping all information of a model [3]. An important way to implement model-free methods is to make an agent comprise some sort of internal memory [1, 4, 6] (see Figure 1). For example, the states of a model can be expressed by some nearest observations and internal states of an agent. After learning the state space, the agent can solve it by model-free methods, such as HQ-learning [10].

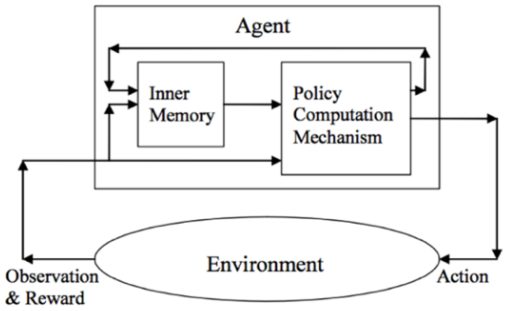

Figure 1: The agent model that has internal memory

The internal memory of an agent can comprise instances that record what it has encountered at each time step. Nearest Sequence Memory (NSM) and Utile Suffix Memory (USM) algorithms are based on the instances.

2.2 Utile Suffix Memory Algorithm

The interaction between the agent implementing USM and the environment is described by A, O, R, which are finite set of actions, finite set of observations and a reward function. Like other instance based algorithms, USM records each of its raw instances [4]. At each time step t, the agent executes action $a_t \in A$ to get a new instance, gets observation $o_{t+1} \in O$, and get a instant reward r_{t+1} according to R, which is determined by the environment. That new instance is formulated as

$$T_{t+1} = (T_t, a_t, o_{t+1}, r_{t+1}). \quad (1)$$

T_{t+1} is T_t's successor and, similarly, T_{t-1} is T_t's predecessor. Normally, in an environment with goals for agent to reach, chains of instances-from initial instance to instance reached goal-are built during the learning phase.

In order to find hidden states from those instances chains, USM creates a suffix tree, whose leaves present the state space and store clustered instances. Each node of the tree can thus be uniquely identified by the string of labels on the path from node to the root, and the string is called the node's suffix. An instance is always deposited into the nodes whose suffix matches its observation and action context, or suffix. That is, for an instance T_i, if its suffix $[\ldots, o_{t-3}, a_{t-3}, o_{t-2}, a_{t-2}, o_{t-1}, a_{t-1}]$ matches the suffix of some node, it would be put into that node (see Figure 2). The set of instances that a node contains is written as $I(s)$. The suffix tree leaf which instance T belongs to is written as $L(T)$. It is inevitable that the action layer and the observation layer appears alternately when tree grows (see Figure 2).

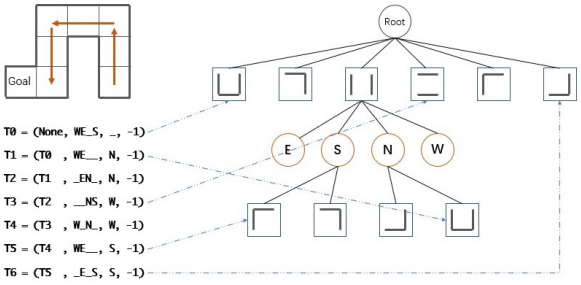

Figure 2: The agent navigates itself in a maze (probably not the first time), generates a sequence of instances, and builds a suffix tree. The action layer and the observation layer appears alternately.

Besides the general nodes, there is another type of node called "fringe" node [4]. The fringe nodes are actually the deepest layers of the suffix tree, however, we treated them the inner instance buckets of the leaves previously mentioned. Fringe nodes also contain instances according to the same suffix criterion used by non-fringe nodes. That is, if the suffix of a leaf is $[a_{t-2}, o_{t-1}, a_{t-1}]$, an 1-layer deeper fringe nodes of it will stores the instance that matches suffix $[o_{t-2}, a_{t-2}, o_{t-1}, a_{t-1}]$. The agent decides whether it should promote fringe nodes to leaves by Kolmogorov-Smirnov test, which determine the instances in the fringes and that of their parent are drawn the same distribution.

The steps of USM is as below:

1. The agent begins with a suffix tree that comprises a root node and an layer of observation nodes as leaves, i.e., the agent only acts according to its observation.

2. The agent chooses an action according to ϵ-greedy policy, executes it and generates an instance (see Equation 1).
3. The agent insert the instance into the suffix tree. The instance is classified to leaves and fringes with the same suffix with that instance.
4. The agent triggers K-S test every n additions of instance. If the Q-value of fringes and their parent are from different distributions, the fringes will be promoted to leaves. The instance will always be added to fringes and leaves that have the same suffix with the instance. Q-value table will be updated after every addition of instance. Time step increases and algorithm jumps to step 2.

There are two obvious deficiencies of the USM algorithm. Firstly, in the USM algorithm, the number of states may exponentially increase as the number of steps grows, but many states are redundant, which correspond to the same state in the real environment. Thus, it will reduce the efficiency of the algorithm. Secondly, since the initial Q value of all leaves is initialized with 0, the ϵ-greedy policy may lead to overfitting. Because the ϵ-greedy policy of USM treats the second best and the worst the same, it drastically decrease the probability to explore other actions which may get higher reward. For example, we found that in the first few steps, the agent may get negative default return after choosing actions at some states, which decreases the probability of exploring those actions when at those states again.

3 Compressed Suffix Memory Algorithm

In order to overcome the weaknesses of USM. We reconstructed the work flow of agent in maze environments. Firstly, we introduces the blind exploration. Exploration and exploitation is a key issue in reinforcement learning. Agent could first make a blind policy exploration of the environment, obtaining some heuristic experience, and apply it to following reinforcement learning. We found that the longest non-repetitive observations of the maze and the frequency of goals can decently presents the local complexity and the scale of the environment, respectively. The heuristic experience can be used to improve the algorithm:

1. The depth of the suffix tree could be constrained. We tried to locate the redundant states of USM and found a potential cause. The suffix tree may grow so deep that a leaf's suffix is longer than most paths from start to goal. By deducing the length of most paths from start to goal as l and limiting the suffix tree's depth to $2l$ can reduce the generation of less useful leaves. So we set l as the sum of longest horizontal edge and vertical edge found by the blind search, since in most situations, paths generated by the best policy are shorter than l.
2. Instances in tree node could be denser. Instead of doing K-S test every n instances added, which may lead to overfitting at some leaves, agent only triggers K-S test when leaves and their fringe nodes are holding enough instances. The minimum number b of instances to do K-S test is deduced from blind exploration, since b should vary with environment scale. In practice, we find the frequency of goals in blind exploration well describes the environment scale.
3. The ϵ-greedy approach can be improved. The main problem of ϵ-greedy approach is that it treats the second best and the worst the same, hence decreasing the probability to explore the second best action. We use boltzmann sampling to balance between exploration and exploitation, which assigns actions with similar Q value similar probabilities to encourage agent to explore or exploit them all, instead of only the best. The probability to choose an action a_i at leaf s with temperature t is

$$p(s, a_i) = e(s, a_i) / \sum_a e(s, a), \tag{2}$$

which e is

$$e(s, a) = \exp((Q(s, a) - \max Q(s))/t) \tag{3}$$

The details and the pseudo code (see Algorithm 1) form of the CSM algorithm is as below.

1. The agent begins with a suffix tree that comprises a root node and an layer of observation nodes as leaves, i.e., the agent only acts according to its observation.
2. The agent randomly explores the environment for a number of times. The agent deduces the the effective path length l to goal and minimum number b of instances to do K-S test in that exploration. The maximum depth of the suffix tree is set to $2l$.
3. The agent executes action and interacts with the environment at time step t. It records its learning history as an instance T_t (see Equation 1). For all s, no matter s is leaf or fringe, if the suffix of s matches the suffix of T_t. Their instance set are updated as:

$$I(s) \leftarrow I(s) \cup T_t. \tag{4}$$

Let $L(T)$ be the leaf which instance T belongs to. $L(T_t)$ is cached to agent memory.
4. For each instance added, the agent does one Bellman iteration with the leaves of of the states:

$$Q(s, a) \leftarrow R(s, a) + \gamma Pr(s'|s, a)U(s'). \tag{5}$$

Let $I(s, a)$ be the subset of $I(s)$ that contains all the instances that executed action a. $U(s')$ is the utility of the state s', calculated as $U(s) = \max_{a \in A} Q(s, a)$. $R(s, a)$ and $Pr(s'|s, a)$ are the estimated immediate reward and the transition probability, respectively, that drawn from the instance chains:

$$R(s, a) = \frac{\sum_{T_i \in I(s,a)} r_i}{|I(s, a)|}, \tag{6}$$

$$Pr(s'|s, a) = \frac{|\forall T_i \in I(s, a)\ s.t.\ L(T_{i+1} = s')|}{|I(s, a)|}. \tag{7}$$

5. Agent perform Kolmogorov-Smirnov when $L(T_t)$ is not the same as $L(T_{t-1})$ and $L(T_t)$ holds enough instances (more than b). The expected discounted reward of instance T_i is written as $H(T_i)$, and is defined as:

$$H(T_i) = r_i + \gamma U(L(T_{i+1})) \tag{8}$$

Agent calculates every $Q(T_i)$ for instances of leaf and those of immediate fringe nodes, then determines whether they are from the same distribution. If they are not, those fringe nodes would be promoted as leaves.

6. Agent does Boltzmann sampling to choose the next action, i.e., with probability $p(s, a_i)$ the agent chooses a_i as a_{t+1} (see Equation 2). In practice, we encourage agent to do random exploration for n_r times, and then perform Boltzmann sampling method with descendant temperature. Time step increase to $t+1$ and algorithm jumps to step 3.

4 Experiment

4.1 Experiment Setup

We run algorithms with three well-known benchmarks: Tiger-Grid, Hallway and McCallum as well as a large Prim Maze generated by Prim algorithm randomly (see Figure 3). The characteristics of POMDP benchmarks are described below (see Table 1), where $|S|$ denotes the number of states, $|A|$ denotes the number of actions and $|\Omega|$ denotes the number of observations.

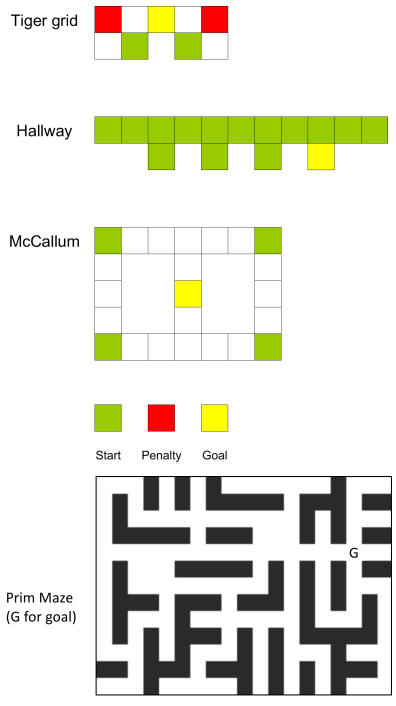

Figure 3: Benchmark mazes

Algorithm 1 Compressed Suffix Algorithm

Input: iteration steps n, radom steps in iteration n_r
Output: average discounted return $E(r)$

Initialize path length l, and minimum number b of instance required to do K-S test by blind exploration
Initialize temperature t as positive infinity
Initialize a suffix tree with depth limitation $2l$
Initialize agent with random start position
for $i = 0$ to n **do**
  if $i > n_r$ **then**
    t decreases from 0.5 to nearly 0 as i increases
  end if
  Choose action a by boltzmann sampling and execute it (see Equation 2)
  while not moved after executed a **do**
    Choose action a by boltzmann sampling with higher t and execute it
  end while
  Generate instance T according to the execution of a just now
  Insert T into the suffix tree, and find the leaf node s the instance belongs
  $I(s) = I(s) \cup T_t$
  $Q(s, a) = R(s, a) + \gamma Pr(s'|s, a)U(s')$
  $R(s, a) = \sum_{T_i \in I(s,a)} r_i / |I(s, a)|$
  $Pr(s'|s, a) = |\forall T_i \in I(s, a)\ s.t.\ L(T_{i+1} = s')| / |I(s, a)|$
  if $|I(s)| > b$ **then**
    do K-S test between $H(T)$s of s and its fringe nodes, get p_value p
    if $p < 0.1$ **then**
      promote fringe nodes to leaves
    end if
  end if
  if agent at goal **then**
    Initialize agent with random start position
  end if
end for
Use boltzmann sampling at low temperature on $Q(s, a)$ to produce average discounted reward $E(r)$
return $E(r)$

In Tiger-Grid, the agents are initially assigned to one of two definite starting positions. The probability of each starting position is 0.5. The goal of the model is to make the agent reach the target position as soon as possible. Once the agent reaches the target position, it is reset to two starting positions. The action of the agent includes: moving forward, turning left, turning right and turning backwards. Tiger-Grid's maze world contains two punitive positions, and the agent will receive a negative return after entering these locations.

Hallway problems are similar to Tiger-Grid issues but has some differences. When the agent arrives at the target, Hallway will reset the agent to a random state in the labyrinth. The Hallway problem does not include a penalty position, and add landmarks to the four places for observation.

Table 1: Benchmarks

problem	$\|S\|$	$\|A\|$	$\|\Omega\|$
Tiger-Grid	9	4	7
Hallway	15	4	4
McCallum	23	4	9
Prim Maze	167	4	16

Table 2: Experiment Results

	Max ADR		Time to Max ADR/sec	
	USM	CSM	USM	CSM
Tiger Grid	3.81	8.70	10.75	9.45
Hallway	8.43	10.13	13.52	6.67
McCallum	2.82	5.90	34.39	14.15
Prim Maze	7.13	8.30	49.30	18.24

In McCallum, the agent's task is to navigate to the goal using action for moving north, south, east and west. After reaching the goal, the agent executes any action and begins a new trail in a randomly chosen corner of the world. The agent receives reward 5.0 upon reaching the goal, and reward -0.1 otherwise.

In the experiment, we evaluate the following criteria:

1. Average discounted reward (ADR), $E(r)$

$$\frac{\sum_{i=0}^{n_{trails}} \sum_{j=0}^{n_{steps}} \gamma^j r^j}{n_{trails}}, \tag{9}$$

   ADR is widely considered to be a good evaluation of the quality of a value function.

2. Iteration duration, which measures the speed of making decision in each iteration.

Our experiment uses a discount factor $\gamma = 0.9$, an exploration probability $\epsilon = 0.1$ for USM. The boundary value of the Kolmogorov-Smirnov test is $p = 0.1$. We iterate the USM, CSM and random algorithm 2048 times, and set check points per 48 iteration. In each check point we test our model 256 times within 96 steps, i.e., get 256 ADRs. We will show the changes in average ADR along with runtime in different benchmarks.

4.2 Experiment Results

The experiment results of CSM, USM and random algorithm are shown in Table 2 and Figure 4 to 7. It should be noted that due to the randomness of the ADR measurement method, the results of ADR may fluctuate. Therefore, the Max ADR in Table 2 represents the ADR when the algorithm results start to stabilize, rather than the maximum value of the experimental ADR results.

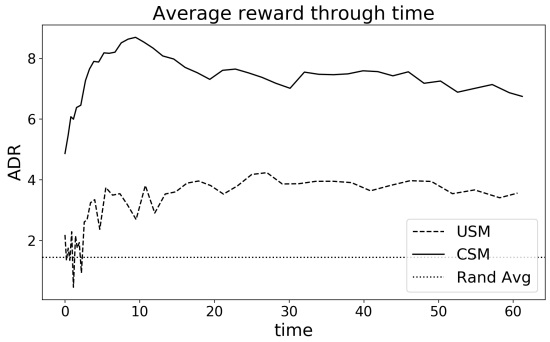

Figure 4: ADR by time(sec) for Tiger-Grid

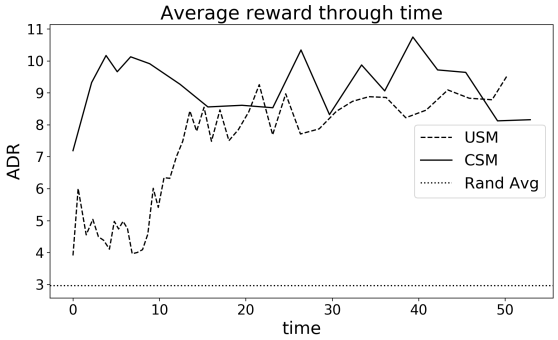

Figure 5: ADR by time(sed) for Hallway

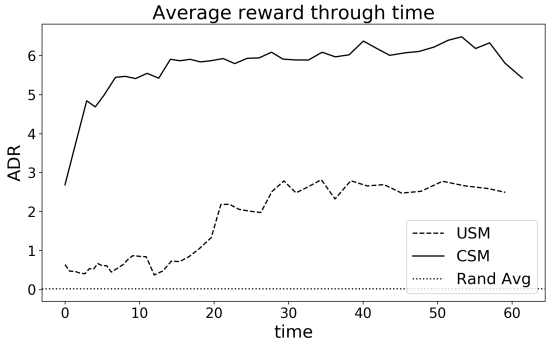

Figure 6: ADR by time(sec) for McCallum

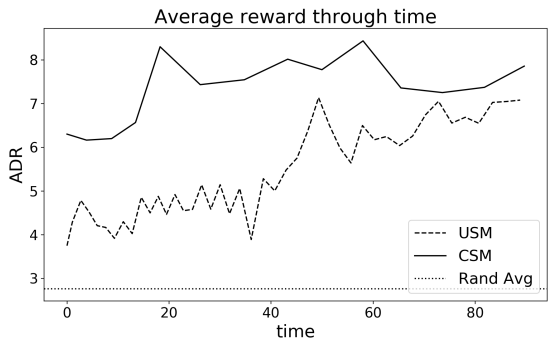

Figure 7: ADR by time(sec) for Prim Maze

4.3 Experiment Results Analysis

On the Tiger-Grid problem, the CSM algorithm quickly reached a very high ADR in a short time after the start of the experiment, and as the experiment progressed, the ADR of the CSM was basically stable at around 8. In contrast, the experimental results of the USM algorithm have large fluctuations in the early stage of the experiment, and the experimental results in the latter part of the experiment are basically stable at 4, which is only half of the CSM.

On the Hallway problem, the experimental ADR results of both algorithms showed large fluctuations, and they had very similar stable maximum ADR. However, from the time point of view, the CSM algorithm reaches the stable maximum ADR running time is only half of the USM, that is, the CSM has twice the processing speed of the USM on the Hallway problem.

On the McCallum Maze problem, similar to the performance in the Tiger Grid problem, the CSM algorithm quickly reached a very high ADR in the short time after the start of the experiment, and as the experiment progressed, the ADR of the CSM was basically stable at 6. In contrast, the experimental results of the USM algorithm did not increase significantly in the early stage of the experiment, and only showed a growth trend in the later stage of the experiment. Despite this, USM's final ADR is only about 3, far less than CSM. This shows that in the case of McCallum Maze problem, CSM can explore the whole problem and find the optimal strategy more quickly and better than USM.

In large dataset problems, such as the Prim Maze problem in the experiment, CSM can also perform faster and better than USM. From the experimental results, the CSM achieved a stable maximum ADR 8 in about 20 seconds, while the USM had an ADR of less than 5 in the same time. Even in the mid-term of the experiment, USM has experienced rapid growth, but its time cost is almost three times that of CSM, and ADR is still lower than CSM.

Overall, It can be seen that the CSM algorithm is superior to the USM algorithm.in terms of speed and return.

5 Conclusion

In this paper, the USM algorithm is described in detail, and the defects of USM algorithm are proposed. Based on the original algorithm, some suggestions for improvement are proposed. Then the CSM (Compressed Suffix Memory) algorithm is proposed. At the end of this paper, the content of the maze experiment is introduced to test the CSM algorithm, and compared with the results of the USM algorithm, it shows that the CSM algorithm does have some improvement effects.

Submissions to DAI should be made at the following site:

`http://www.adai.ai/call-for-papers.html`

The paper length is limited to 6 pages, with 1 additional page containing only bibliographic references. Authors may use as many pages of appendices (after the bibliography) as they wish, but reviewers are not required to read these. Any paper exceeding this length (except for the appendices) will automatically be rejected.

5.1 Style

Papers to be submitted to DAI 2019 must be prepared by LaTeX using the style file `dai_2019.sty`. As the review process is double blind, please use the style package as

```
\usepackage{dai_2019}
```

which automatically anonymizes the submission.

The style file also provide an optional option `final`, which creates a a camera-ready for your paper

```
\usepackage[final]{dai_2019}
```

The paper size, font sizes, margins, spaces between lines and headers or around figures/tables, etc, preset by the style file must *not* be modified. Any modification would lead to rejection without further notification.

6 Headings: first level

Example for first-level headings.

6.1 Headings: second level

Example for second-level headings.

6.1.1 Headings: third level

Example for third-level headings.

Headings: forth level

Example for forth-level headings.

Paragraphs Example for paragraphs.

7 Citations, figures, tables, references

7.1 Citations

The `natbib` package will be used by default. The citation format could be author/year or numeric, as long as it is internally consistent. As the submission is double blind, please refer to your own work in the third person.

7.2 Figures

Example for figures (see Figure 8).

Figure 8: Example figure caption.

7.3 Tables

Example for tables (see Table 3).

Table 3: Example table caption.

Alpha		
Beta	Gamma	Delta
x^2	$f(y)$	$z \sim \mathcal{N}(0, 1)$

7.4 References

It is acceptable to reduce the font size to `\small` when listing the references.

Acknowledgments

Do not include acknowledgments in the submission, only in the final paper.

References

[1] Douglas Aberdeen et al. Policy-gradient algorithms for partially observable markov decision processes. 2003.

[2] Haoyu Bai, Shaojun Cai, Nan Ye, David Hsu, and Wee Sun Lee. Intention-aware online pomdp planning for autonomous driving in a crowd. In *2015 ieee international conference on robotics and automation (icra)*, pages 454–460. IEEE, 2015.

[3] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[4] R Andrew McCallum. Instance-based utile distinctions for reinforcement learning with hidden state. In *Machine Learning Proceedings 1995*, pages 387–395. Elsevier, 1995.

[5] R McCallum. Reinforcement learning with selective perception and hidden state. 1997.

[6] Nicolas Meuleau, Leonid Peshkin, Kee-Eung Kim, and Leslie Pack Kaelbling. Learning finite-state controllers for partially observable environments. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 427–436. Morgan Kaufmann Publishers Inc., 1999.

[7] Guy Shani, Joelle Pineau, and Robert Kaplow. A survey of point-based pomdp solvers. *Autonomous Agents and Multi-Agent Systems*, 27(1):1–51, 2013.

[8] Trey Smith and Reid Simmons. *Probabilistic planning for robotic exploration.* PhD thesis, Carnegie Mellon University, The Robotics Institute, 2007.

[9] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[10] Marco Wiering and Jürgen Schmidhuber. Hq-learning. *Adaptive Behavior*, 6(2):219–246, 1997.

A Example appendix section

A.1 Example appendix subsection

B Another example appendix section

As long as you need contents in appendices.

fsdflsdjflsdf jhjkhkhkkkkkk dfdfdsfdf