A Reinforcement Learning Approach to Constrained Resource Allocation Problems

C. Vic Hu
Department of Electrical & Computer Engineering
University of Texas at Austin

***Abstract*—Resource allocation has been extensively studied in various fields such as wireless communication, intelligent traffic routing, industrial management, parallel architectures and distributed systems. Although there have been many efficient and powerful algorithms proposed in each domains, very few of them is able to address more generalized resource allocation problems on a higher level.**

This paper proposes a generalized framework and makes three main contributions to solving constrained resource allocation problem using a modified reinforcement learning algorithm. First, we designed a general architecture, Constrained Resource Allocation Framework (CRAF), for the ease of generalized problem mapping and learning. Second, we defined four properties and three measurements to both quantitatively and qualitatively analyze how well an algorithm performs in CRAF. Finally, we developed three benchmark experiments to demonstrate how a reinforcement algorithm can successfully solve very different resource allocating problems with CRAF.

I. Introduction

Resource allocation is an old and widely-solved problem in many fields, including electrical engineering, computer science, economics, management science and many more. It typically involves a fixed number of resource units to be distributed to a set of tasks over a period of time, such as landing aircraft scheduling, wireless communication routing, social security welfare and shared resources in parallel computer architectures. The problems of resource allocating are ubiquitous, but they all essentially boil down to one simple notion–based on a set of criteria, how many resource units should be allocated to which task first, and for how long.

To name a few remarkable research examples to solve this type of problems, Dresner and Stone proposed s reservation system for autonomous intersection management [1] to allocate right of road for crossing traffic, Perkins and Royer presented a novel routing algorithm for ad-hoc on-demand mobile nodes management [2], and Foster et al. came up with a reservation and allocation architecture for heterogeneous resource management in computer network. While they all addressed an elegant solution to a specific domain, most of the techniques cannot be easily transferred to similar decision problems in other domains. In this paper, we focus on addressing exactly this issue and forming a general resource allocation framework that is suitable to be solved by a reinforcement learning method.

The first contribution of this paper is to form the Constrained Resource Allocation Framework (CRAF), in which we designed a generalized architecture to capture most of the resource allocation problems. In addition to the conventional first-come, first-served basis, we introduced the notion of constraints and queue propagation to reflect a more realistic setting and to relax more complicated systems into a single-frontier problem.

Secondly, we defined a collection of analysis criteria and evaluation methods to both qualitatively and quantitatively study how a reinforcement learning algorithm perform on our framework. Lastly, we proposed three benchmark problems to empirically demonstrate how CRAF applies to different resource allocation problems consistently and effectively.

II. Background

Unlike the Markov Decision Processes (MDPs) that most of the reinforcement learning algorithms are designed to solve, the notion of state representation, transition functions and reward functions in resource allocation problems such as aircraft landing scheduling can be very complex and challenging to define. Furthermore, the state space representing the entire global snapshot could be too large to be useful for effective learning.

Instead of trying to model a resource allocation problem as MDP, we found it more intuitive to formalize it as a multi-armed bandit problem, which has been extensively researched in the field of reinforcement learning [6]. Assuming that we

can reasonably classify each incoming task candidate to one of the K prototypes, from which we use the approximated reward functions to determine which candidate receives the resource unit in each episode. Before we formalize our definitions and notations of the framework, let us go through some of the existing K-armed algorithms and see how they can be useful to the resource allocation problem.

A. The K-armed Bandit Problem

The problem is formalized by a fixed number of slot gambling machines, each defined by a random variable $X_{i,n}$, for $1 \leq i \leq K$ and $n \geq 1$. A sequential N plays of machine i give rewards of $X_{i,1}, X_{i,2}, \ldots, X_{i,N}$, which are all independent of each other and identically distributed. Based on the sequence of playing history and obtained rewards, one can form an allocation algorithm to pick the next machine. To evaluate the performance of such algorithms, one criterion, the $regret$, is defined as

$$regret = \mu^* \, n - \mu_j \sum_{j=1}^{K} E[T_j(n)]$$

where μ_i is the true mean of the generative distribution of X_i, and $\mu^* \equiv \max_{1 \leq i \leq K} \mu_i$, $T_j(n)$ is the number of times machine j has been played during the n plays. Therefore, $regret$ is essentially the expected loss function (opportunity cost) of the played allocation strategy. In reality we don't know the true μ_i of any slot machines, and thus we need a more empirical method to estimate the payoffs. However, $regret$ gives us an intuition about one of the fundamental problem the reinforcement learning is trying to solve–the dilemma between exploration and exploitation.

B. Balancing Exploration and Exploitation

As introduced in the previous section, $regret$ is a theoretical index to tell us how close we are from the optimal playing strategy in a K-armed bandit game. To play the optimal action of the game, the learning agent needs to build up a knowledge base of how well each action will pay off. The behavior of trying unknown or low confident actions is called *exploration*. At certain point, the agent may decide it is confident enough to simply play the best action based on the learned knowledge so far, and that playing a suboptimal *exploration* policy is not worthwhile. This idea of greedily choosing whatever is currently the best is known as *exploitation*. It is crucial that a learning agent need to balance both *exploration* and *exploitation* to find the optimal policy in almost any reinforcement learning settings.

To quantitatively describe how 'good' an action is, we can define the action-value function prior to time step t to be

$$Q_t(a) = \frac{\sum_{i=1}^{N_a} r_i}{N_a}$$

where N_a is the number of times action a was chosen prior to t. In other words, $Q_t(a)$ is the sampled arithmetic average of the rewards received so far, which approaches to the true action-value $Q^*(a)$ as $N_a \to \infty$. Alternatively, instead of keeping track of the entire history of rewards received r_i, $i \in \{1, \ldots, N_a\}$ we can simply increment the current action-value toward the new one with a step-size parameter α:

$$Q_{k+1} = Q_k + \alpha \left[r_{k+1} - Q_k \right]$$

When we choose $\alpha = \frac{1}{k+1}$, the action-value function falls back to:

$$Q_{k+1} = \frac{1}{k+1} \sum_{i=1}^{k+1} r_i$$

Which is exactly the original sampled average form. To guarantee a general step-size α such that $\lim_{k \to \infty} Q_k = Q^*$, it follows that the steps must be large enough to encounter biased initial conditions and randomness, while it cannot be too large for the action-value function to converge eventually. Based on these intuitions, the following conditions must hold according to the stochastic approximation theory in [8]:

$$\sum_{k=1}^{\infty} \alpha_k(a) = \infty \text{ and } \sum_{k=1}^{\infty} \alpha_k^2(a) < \infty$$

One straightforward policy is to always exploit the greedy action $a^* = \arg\max_a Q_t(a)$to maximize the immediate reward at time t . To balance *exploitation* with *exploration*, here are a few alternative methods we will consider:

ϵ-greedy: The simplest way is to explore once in a while and to exploit greedy actions the rest of the time. In other words, we can formalize a policy function as the probability of playing action a at time t:

$$\pi_t(a) = \begin{cases} 1 - \epsilon, & \text{for } a = a^* = \arg\max_a Q_t(a) \\ \epsilon, & \text{choose any action randomly} \end{cases}$$

Although ϵ-greedy method does guarantee $N_a \to \infty$ as $t \to \infty$ and thus $Q_t(a) \to Q^*(a)$ in theory, it is definitely

not the most efficient learning strategy in practice, especially the evaluative feedback is non-stationary , which is true in most decision learning processes. However, it is a popular and effective way to balance exploration and exploitation when augmented with more sophisticated algorithms.

Softmax: When exploring suboptimal actions, it makes more sense to choose risky action-values with lower probability, which intuitively blends in a little bit exploitation within the exploration itself. For instance, one may define the policy to be a probabilistic distribution in proportion to the action-values:

$$\pi_t(a) = \frac{e^{Q_t(a)/\tau}}{\sum_{\hat{a}\in A} e^{Q_t(\hat{a})/\tau}},\ \tau > 0$$

Notice that as the tuning parameter $\tau \to 0$, this softmax method is equivalent to the greedy policy, while higher τ leads to a more uniform action selection.

Reinforcement Comparison: This method is based on a simple idea of "preferring the good while avoiding the bad." Trivial and obvious as it sounds, we need to formalize our definitions of 'preference' and 'good/bad' to make this concept useful on a multi-armed bandit problem. First, we define the notion of the *reference reward* at time t, $\overline{r}_t$:

$$\overline{r}_{t+1} = \overline{r}_t + \alpha\left[r_t - \overline{r}_t\right]$$

which is almost identical to the action-value function Q_t we defined earlier, except that it is the incremental average of all the received rewards, regardless of the actions taken. Now we have the *reference reward* to tell us what's the reward an average action should receive, we can establish our preference to an action based on its reward compared to the average:

$$p_{t+1}(a_t) = p_t(a_t) + \beta\left[r_t - \overline{r}_t\right],\ \beta > 0$$

which basically increases or decreases how much we prefer an action a according to its reward, parameterized by another step-size parameter β. Finally, we can determine our policy just like the softmax method:

$$\pi_t(a) = \frac{e^{p_t(a)}}{\sum_{\hat{a}\in A} e^{p_t(\hat{a})}}$$

Pursuit: The *pursuit* method maintains both the action-value estimates $Q_t(a)$ for choosing the greedy action a_t^* and the policy $\pi_t(a)$, which is updated by incrementing toward one and zero according to the following rule:

$$\pi_{t+1}(a) = \begin{cases} \pi_t(a) + \beta\left[1 - \pi_t(a)\right], & \text{for } a = a_{t+1}^* \\ \pi_t(a) + \beta\left[0 - \pi_t(a)\right], & \text{for } a \neq a_{t+1}^* \end{cases}$$

More Advanced Algorithms: More recently, Auer et al. proposed a new set of more sophisticated algorithms to solve the multi-armed bandit problem in finite time [6]. Here we list a few that we consider to implement on our framework. The details and analysis of these algorithms are beyond the scope of this paper, but we encourage the reader to refer to their original work [6]

UCB1: deterministic policy
Initialization: play each action once
For each iteration: play action a^*

$$a_t^* = \arg\max_{a\in A} Q_t(a) + \sqrt{\frac{2\ln N}{N_a}}$$

where $Q_t(a)$ is computed as the sampled average reward, N_a is the number of a played so far, and N is the total rounds played ($N = \sum_{a\in A} N_a$)

UCB2: deterministic policy
Initialization: Set $r_a = 0\ \forall a \in A$ and play each action once
For each iteration:

1) Choose

$$a^* = \arg\max_{a\in A} Q_t(a) + \sqrt{\frac{(1+\alpha)\ln(eN/\tau(r_a))}{2\tau(r_a)}}$$

$$\tau(x) = \lceil (1+\alpha)^x \rceil,\ 0 < \alpha < 1$$

2) Play action a exactly $\tau(r_a + 1) - \tau(r_a)$ times
3) $r_a \leftarrow r_a + 1$
4) slowly decrease α

ϵ_n-greedy: randomized policy (ϵ needs to go to 0)
Initialization: Define $\epsilon_n \in (0, 1]$, $n = 1, 2, \ldots$

$$\epsilon_n = min\{1, \frac{cK}{d^2 n}\},\ c > 0 \text{ and } 0 < d < 1$$

For $n = 1, 2, \cdots$:

1) Let $a_n = \arg\max_{a\in A} Q_n(a)$

2) Play with action selected according to policy:

$$\pi_n(a) = \begin{cases} 1-\epsilon_n, & \text{for } a_n \\ \epsilon_n, & random \end{cases}$$

UCB1-NORMAL: deterministic policy

For $n = 1, 2, \cdots$:

1) If $n_a < \lceil 8 \log n \rceil \ \forall a \in A$, play a
2) Otherwise, play

$$a_n^* = \arg\max_{a \in A} Q_n(a) + \sqrt{16 \cdot \frac{q_a - n_a Q_n^2(a)}{n_a - 1} \cdot \frac{\ln(n-1)}{n_a}}$$

where q_a is the sum of squared rewards from action a so far.

3) Update $Q_n(a)$ and q_a with the obtained reward

C. Prototype Classification

1) Distance Functions: The idea of a particle is essentially a sampled instance of the topic distribution over a time sequence, represented by a vector $\mathbf{w}_i \in \mathbb{R}^{|V|},\ i \in \{1, \cdots, N\}$, where $|V|$ is the total vocabulary size of all the topic words appeared. Since one of our intermediate objectives is to formalize clusters between these particles, we need to first define how we will measure the similarity or distance between any pair of particles $\mathbf{w}_i,\ \mathbf{w}_j$.

Minkowski:

$$d = \sqrt[p]{\sum_{k=1}^{|V|} |w_{ik} - w_{jk}|^p}$$

Note that when $p = 1$, the Minkowski reduces to the city block distance, while $p = 2$ gives the Euclidean distance and $p = \infty$ yields the Chebychev distance.

Cosine:

$$d = 1 - \frac{\mathbf{w}_i \, \mathbf{w}_j^T}{|\mathbf{w}_i|_2 |\mathbf{w}_j|_2}$$

Correlation:

$$d = 1 - \frac{(\mathbf{w}_i - \overline{\mathbf{w}}_i)(\mathbf{w}_j - \overline{\mathbf{w}}_j)^T}{|(\mathbf{w}_i - \overline{\mathbf{w}}_i)|_2 |(\mathbf{w}_j - \overline{\mathbf{w}}_j)|_2}$$

where

$$\overline{\mathbf{w}}_i = \frac{1}{|V|} \sum_{k=1}^{|V|} w_{ik},\ \overline{\mathbf{w}}_j = \frac{1}{|V|} \sum_{k=1}^{|V|} w_{jk}$$

Jaccard:

$$d = \frac{\#\left[(w_{ik} \neq w_{jk}) \cap ((w_{ik} \neq 0) \cup (w_{jk} \neq 0))\right]}{\#\left[(w_{ik} \neq 0) \cup (w_{jk} \neq 0)\right]}$$

2) Clustering Algorithms: LDA is essentially pulling out the 'principal components' from the text documents, reducing a large archive of data into just K representative topics $\beta_{1:K}$. Therefore, our assumption is that by grouping similar particles induced from past topics, we can observe the clustering patterns and make reasonable predictions on how the future topics will be like.

Now we have a collection of well-defined distance functions, we can start looking at clustering methods to group our particles together accordingly, and here are a set of common clustering algorithms we will consider:

K-Means: As the name itself suggests, the K-Means clustering algorithm consists of K cluster centroids and moves these means iteratively towards the center of its closest neighbors, until they no longer change. Although K-Means has been proved to be guaranteed for convergence [**?**], its clustering performance is often correlated to how the seeds are initialized at the beginning, and the optimal choice of K is often not apparent (in our case, it is the same as the number of topics.)

```
1. Initialize the means by picking
   K samples at random
2. Iterate
2.a. Assign each instance to
     its nearest mean
2.b. Move the current means to
 the center of all the
 associated points
```

Hierarchical Agglomerative Clustering (HAC): This algorithm starts with treating every instances as individual cluster, and iteratively joins pairs of similar clusters repeatedly until there is only one. If we take the merging history and form a hierarchical binary tree, it will look like the dendrogram in Fig. 1.

Although we have defined the distance functions in the previous section, we have yet formalized the similarity functions between clusters. In our method, we will focus on the

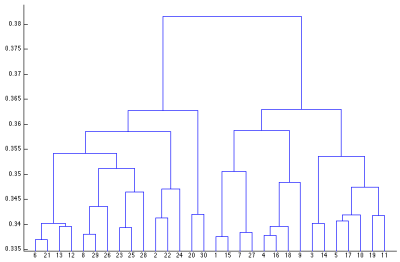

Fig. 1: A sample dendrogram by HAC

following four similarity functions:

- **Single-linkage:** Also known as the nearest neighbor, computes the distance between the two closest elements from two clusters
- **Complete-linkage:** The conjugate of **single-linkage**, also known as the farthest neighbor, computes the distance between the two farthest elements (maximum distance) from two clusters
- **UPGMA:** Unweighted Pair Group Method with Averaging calculates the distance between two clusters, $C_i \& C_j$, by averaging all distances between any pair of objects from the two clusters.

$$dist(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{w_i \in C_i} \sum_{w_j \in C_j} dist(w_i, w_j)$$

Now, let's call this newly-formed cluster C_{ij} and compare its distance with another cluster C_k:

$$dist(C_{ij}, C_k) = \frac{|C_i| dist(c_i, c_k) + |C_j| dist(c_j, c_k)}{|C_i| + |C_j|}$$

where c_i, c_j, c_k are the centroids for clusters C_i, C_j, C_k

- **WPGMA** Weighted Pair Group Method with Averaging is similar to UPGMA except that the cluster distance is now calculated as:

$$dist(C_{ij}, C_k) = \frac{dist(c_i, c_k) + dist(c_j, c_k)}{2}$$

The behavior of HAC is often dominated by the chosen similarity function. While each variant has a different set of clustering patterns it is good at capturing, all of them have certain vulnerabilities.

III. The Constrained Resource Allocation Framework

Formally, the Constrained Resource Allocation Framework (CRAF) consists of a sequence of task instances, and a centralized learning agent who distributes resource units to the task instances and receives rewards according to the decision they make. The task instances can come in one or more channels $\mathbf{x}_{c,i}$, $1 \leq c \leq C$, $1 \leq i \leq capacity(c)$, where i denotes the i-th instance and c denotes the c-th channel. To transform a resource allocation problem into a learnable framework, the formalization process under CRAF can be broke down into three phases:

1) Phase One
   - **Constraint**: The instance $x_{c,i}$ needs a total of $num(x_{c,i})$ resource units before the deadline $exp(x_{c,i})$
   - **Objective**: A high-level criteria that the learning agent should achieve, which adds bias against the reward functions
2) Phase Two
   - **Priority Score**: $ps(x_{c,i})$, a function of both the **Constraint** and **Objective**
   - **Queue Propagation**: $qp(x_{c,i}) = \gamma qp(x_{c,i+1}) + \tau\, ct(x_{c,i})$ A linear combination of the current constraint and the same function from the next instance in queue
   - **Feature Selection**: Use the relevant features to augment the instance $x_{c,i}$ into a feature vector in a higher dimensional space. Domain knowledge may help reduce the dimensionality.
3) Phase Three
   - **Prototype Classification**: Given a manually-chosen K, the current instance $x_{c,i}$ is classified into one of the K clustering prototypes with respect to the two parameters obtained in the previous phase $ps(x_{c,i})$, $qp(x_{c,i})$
   - **Reward Function**: The learning agent receives a reward of $r_p + r_o$ if it fulfills the constraint of $x_{c,i}$, or $r_f - r_o$ otherwise

Now we formed the K prototypes, we can have the resource allocating agent learn to choose from the channel frontiers just like in the multi-armed bandit problem. For each round, the agent faces a total of C channels, while each channel has a number of instance tasks waiting to be processed. The agent is given the prototype number, k, of the first instance (frontier) of each channel, and it has to choose which frontier to give its resource for that round based on k. Unlike the traditional K-armed bandit problem, there are only a subset of K presented at a time if $C < K$. Furthermore, Each channel

frontier doesn't necessarily have distinct k, which the agent break the tie randomly after deciding which k to choose from the available subset. The pseudo code of this algorithm can be found in Table. I.

Step	Instruction
1	For each time step n
2	$K_n = \{K(x_{1,c}),\, c = 1, \ldots, C\}$ //Observe the prototypes of all the available channel frontiers as the resource candidates
3	$k_* = bandit(K_n)$ //Choose the optimal candidate instance to give the resource by solving a k-armed bandit problem
4	Ready for the next round

TABLE I: Pseudo code for the frontier choosing algorithm in CRAF.

To examine how well an algorithm solves a general resource allocation problem, we proposed the following properties:

- **Safety**: One resource cannot be shared by more than one task instance at a time.
- **Fairness**: Task instances with higher priority scores should always be served first.
- **Progression**: Every task instance should get the resource in bounded and finite time.

The algorithm we proposed satisfies the *safety* property since the resource is controlled by a centralized allocating agent, and thus no more than one task instance may share the same resource at any time. However, the *fairness* and *progression* properties of CRAF depend on which K-armed bandit algorithm is chosen.

To quantitatively analyze a resource allocating algorithm, we introduced the following measurement criteria:

- **Latency**: The average time each task instance has to wait in line (channel).
- **Throughput**: The average number of task instances processed in a period of time.
- **Rewards**: The average reward accumulated in a period of time.

IV. Experimental Results

A. Candy Distribution

In this section, we propose an example testing bed to demonstrate how exactly CRAF works in a straightforward problem. Suppose it's the desert break in a kindergarden and there are C lines of children (channel), but only one teacher distributing the candies. There are constantly children coming inside the classroom and waiting in a random line for the candy. Each child has different patience and desire for the candy, which impact how happy they will be when receiving the candy. The teacher is trying to maximize the total amount of happiness in the classroom.

In this problem, we define the i-th child in line c as a task instance, $x_{i,c}$, which has two constraints–one for their desire, $c_1(x_{i,c})$, and the other for their patience, $c_2(x_{i,c})$. In this case, the objective is to maximize children's happiness, which will be embedded in the reward function. That's all for *Phase One*. We've defined our constraints and objective, and now let's pass these variables to the next phase.

In *Phase Two*, we need to map the constraints and objective functions we defined in the previous phase into a set of normalized features for clustering. Here, we introduced two more features–$c_3(x_{i,c}) = \sum_{n=i}^{|C_c|} \#x_{i,c}$, the number of children behind $x_{i,c}$, and $c_4(x_{i,c}) = \sum_{c \in C} \#x_{i,c}$, the number of frontiers competing at the front of every lines. Now we have these two new features in addition to the two we introduced in *Phase One*, we normalize these four parameters and pass them to *Phase Three* for clustering.

Finally in *Phase Three*, we apply one of the clustering algorithms and distance functions we mentioned earlier to group these parameterized instances into K clusters. Furthermore, we define the reward function of choosing channel m to be $R(m) = pr(m) + \sum_{j \neq m} nr(j)$, which consists of a positive reward received from the chosen channel and the negative rewards given by the rest of missed channels. To simplify our model, we define $pr(j) = c_1(x_{i,j}) + c_2(x_{i,j})$ and $nr(j) = -0.1\, c_1(x_{i,j}) \times c_2(x_{i,j})$. Note that we intentionally make up a convoluted reward function that is not linear, and leave c_3, c_4 completely independent to the reward to examine the learning agent's ability to discover this nontrivial pattern.

In Fig. 2, 3, and 4, we compared how clustering algorithms with different variations impact the outcome of the candy distribution example. For the K-Means algorithm, we experimented with the Minkowski-2 (Euclidean), Minkowski-1 (city block), cosine, and correlation distance functions. As shown in Fig. 2, all flavors of the K-Means yield a fairly consistent distribution, with a majority in one big clusters and many smaller ones. For the Hierarchical Agglomerative Clustering (HAC) algorithms comparisons in Fig. 3, however, we found that the correlation distance function gave us only one cluster, while the others consistently resulted in two comparable ones.

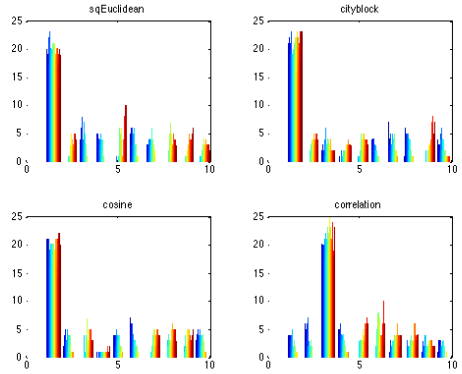

Fig. 2: A histogram to compare the effects of using different distance functions in K-Means. The column stands for $K = 10$ topics, while each color represents the number of corresponding instances in each channel.

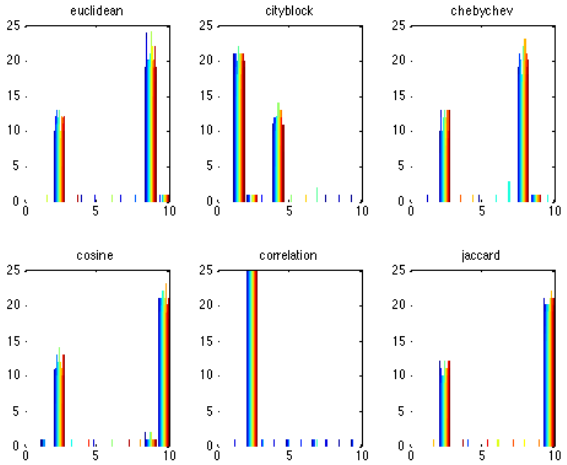

Fig. 3: A histogram to compare the effects of using different distance functions in HAC. The column stands for $K = 10$ topics, while each color represents the number of corresponding instances in each channel.

For the linkage function comparisons in Fig. 4, single linkage method appears to be cleaner and more deterministic about the two-cluster pattern, compared to the other linkage methods.

Now we have the clustering results, we can leave the rest to our k-armed bandit algorithms. In Fig. 5, we compared the accumulated rewards obtained from the candy distribution example by the four basic algorithms we introduced earlier–ϵ-greedy, softmax, reinforcement comparison, and pursuit. We can see that the pursuit method generally has a poorer performance than the other three.

In Fig. 10, we demonstrated the estimated action-value, $Q_t(a)$, where the action is equivalent to the chosen k. Softmax

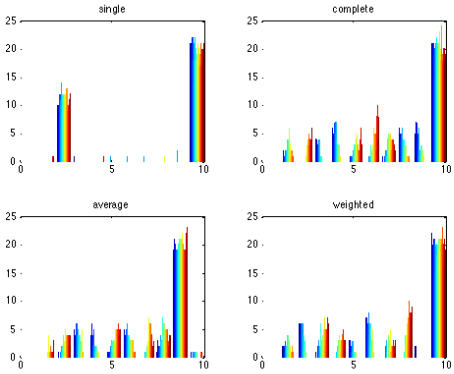

Fig. 4: A histogram to compare the effects of using different linkage functions in HAC. The column stands for $K = 10$ topics, while each color represents the number of corresponding instances in each channel.

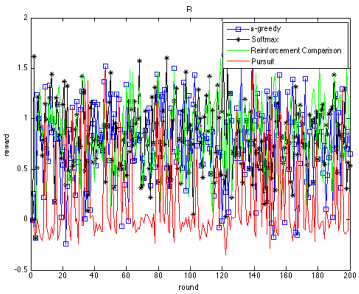

Fig. 5: The rewards obtained by the ϵ-greedy, Softmax, Reinforcement Comparison, and Pursuit methods on the candy distribution problem

(top right) has the most smooth estimation, which also shows a lack of exploration.

Now we look at the learning curve obtained with the UCB algorithm family–the UCB1, UCB2, ϵ_n-greedy, and UCB1-Normal algorithms. In Fig. 7, we can observe almost identical learning curves for UCB1 and UCB1-Normal. Note that the learning curve for UCB2 is always zero, which is most likely resulted from an implementation error. The estimated action-value functions are fairly similar among the UCB1, ϵ_n-greedy, and UCB1-Normal methods as shown in Fig. 8.

B. Randomized Test

Besides the candy distribution testbed, we tested the K-armed bandit algorithms on a separate benchmark to generalize our results. In this testbed, we created C channels of instances with randomly-generated prototype numbers and the corresponding rewards. In this case, the positive reward function, $pr = k + \mathbf{N}(0, 0.01)$, is defined to be proportional to the

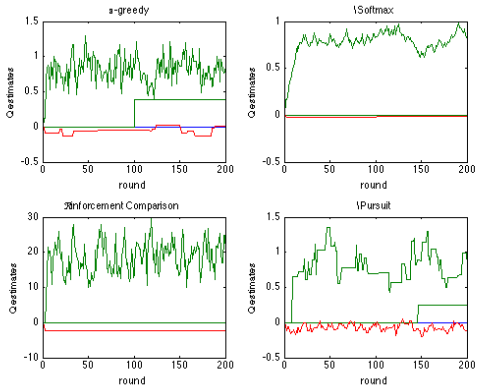

Fig. 6: The estimated action-value functions obtained by the ϵ-greedy, Softmax, Reinforcement Comparison, and Pursuit methods on the candy distribution problem

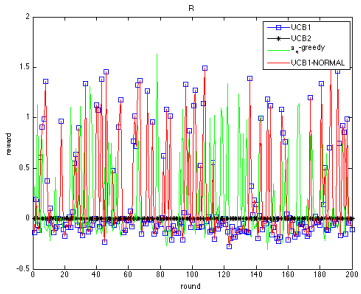

Fig. 7: The rewards obtained by the UCB1, UCB2, ϵ_n-greedy, and UCB1-NORMAL methods on the candy distribution problem

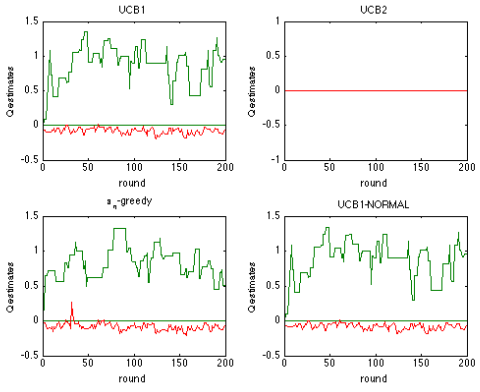

Fig. 8: The estimated action-value functions obtained by the UCB1, UCB2, ϵ_n-greedy, and UCB1-NORMAL methods on the candy distribution problem

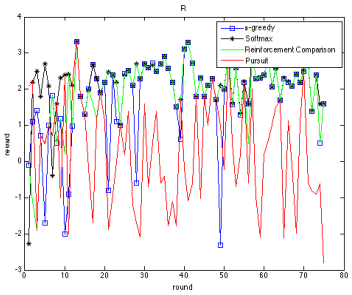

Fig. 9: The rewards obtained by the ϵ-greedy, Softmax, Reinforcement Comparison, and Pursuit methods on the randomized testbed

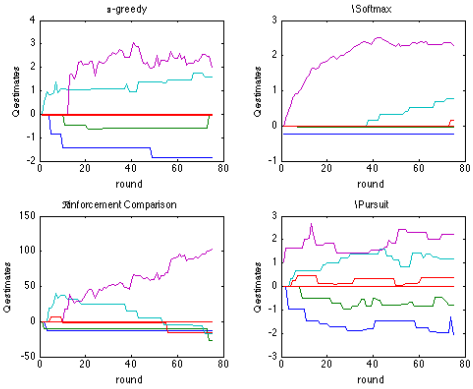

Fig. 10: The estimated action-value functions obtained by the ϵ-greedy, Softmax, Reinforcement Comparison, and Pursuit methods on the randomized testbed

prototype index k with a small gaussian noise. The negative reward function, $nr = -\frac{pr}{C}$, is a scaled negative of pr.

In Fig. 9, we experimented the four basic algorithms again and found that the pursuit method still underperforms the others. Moreover, we can now see a clear distinction that softmax has the highest accumulated rewards overall, followed by the reinforcement comparison method, and then the ϵ-greedy.

The performance in Fig. 9 can be reasonably explained when we look at the estimated action-value functions in Fig. 10. Softmax has a clean distinction between the action-values but again demonstrated a sign of lack of exploration. On the other hand, we can see a fair amount of exploration for both the ϵ-greedy and pursuit algorithms, which can be demonstrated from their more accurate value estimations. Nonetheless, they don't seem to be able to catch up the optimal strategy quickly enough to compete with softmax.

With the UCB algorithms in Fig. 11, the differences between the UCB1, UCB-Normal, and ϵ_n-greedy methods become

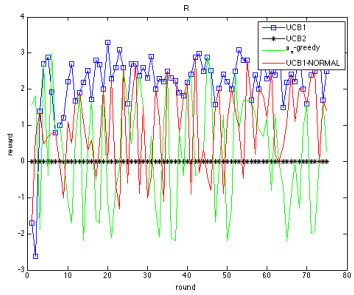

Fig. 11: The rewards obtained by the UCB1, UCB2, ϵ_n-greedy, and UCB1-NORMAL methods on the randomized testbed

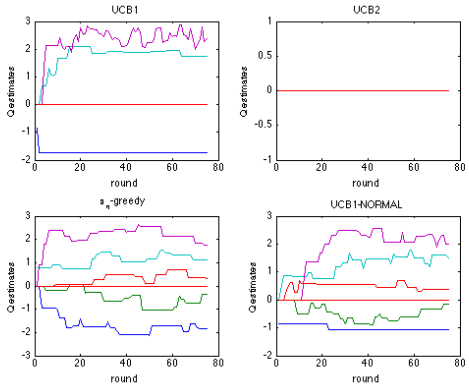

Fig. 12: The estimated action-value functions obtained by the UCB1, UCB2, ϵ_n-greedy, and UCB1-NORMAL methods on the randomized testbed

more clear as well; we can see that UCB1 quickly picked up the optimal strategy and stayed there, while UCB1-Normal and ϵ_n-greedy seems to spend more time exploring other options. We can see that UCB1-Normal does have an uprising learning curve during its exploration, which demonstrates its ability to effectively balance exploitation and exploration.

In Fig. 12, we can see that UCB1 shows a monotone pattern on the high action-values, which is similar to the softmax in Fig. 10. The ϵ_n-greedy and UCB1-Normal have a relatively better estimation due to their ability to balance exploitation with exploration.

Overall, we observed a similar correlation between the learning curve of a k-armed bandit algorithm and its ability to correctly estimate true action-value functions in both the candy distribution and random testbeds.

V. Conclusion

In this paper, we proposed a generalized resource allocation framework to be solved as a classic K-armed bandit problem in reinforcement learning. We demonstrated two simple benchmark domains and compared the results learned by a collection of K-armed bandit algorithms. Our experimental results reinforced the relations of exploitation and exploration tradeoff, and fortified a new concept to solve resource allocation problem in the reinforcement learning fashion.

VI. Future Work

VII. Acknowledgements

We really appreciate Professor Stone's knowledgable guidance and tremendous patience with this project. This work will not be possible without him.

References

[1] Dresner, K. and Stone, P. A Multiagent Approach to Autonomous Intersection Management. *Journal of Artificial Intelligence Research*, 31:591-656, March 2008.

[2] Perkins, C. E., and Royer, E. M. (1999, February). Ad-hoc on-demand distance vector routing. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA'99. Second IEEE Workshop on (pp. 90-100).* IEEE.

[3] Foster, I., Kesselman, C., Lee, C., Lindell, B., Nahrstedt, K., and Roy, A. (1999). A distributed resource management architecture that supports advance reservations and co-allocation. In *Quality of Service, 1999. IWQoS'99. 1999 Seventh International Workshop on (pp. 27-36).* IEEE.

[4] Baruah, S. K., Cohen, N. K., Plaxton, C. G., and Varvel, D. A. (1996). Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6), 600-625.

[5] Thomas, D. (1990). Intra-household resource allocation: An inferential approach. *Journal of human resources*, 635-664.

[6] Auer, P., Cesa-Bianchi, N. and Fischer, P. Finite-time Analysis of the Multiarmed Bandit Problem. In *Proc. of 15th International Conference on Machine Learning, pages 100-108*. Morgan Kaufmann, 1998.

[7] Diuk, C., Li, L. and Leffler, B. R. The Adaptive k-Meteorologists Problem and Its Application to Structure Learning and Feature Selection in Reinforcement Learning. In *Proceedings of the 26th International Conference of Machine Learning*. Montreal, Canada, 2009.

[8] Bertsekas, D. P., and Tsitsiklis, J. N. (1996). Neuro-Dynamic Programming (Optimization and Neural Computation Series, 3). *Athena Scientific*, 7, 15-23.