Fast Reinforcement Learning Algorithms for Resource Allocation in Data Centers

Yuang Jiang*, Murali Kodialam†, T. V. Lakshman†, Sarit Mukherjee†, Leandros Tassiulas*
*Yale University, New Haven, CT, USA. Email: {yuang.jiang, leandros.tassiulas}@yale.edu
†Nokia Bell Labs, Holmdel, NJ, USA. Email: {murali.kodialam, tv.lakshman, sarit.mukherjee}@nokia-bell-labs.com

***Abstract*—Dynamic resource allocation to satisfy varying, concurrent and unpredictable demands from multiple applications is a key need in cloud systems. A fundamental challenge is the need to find the right balance between over-allocation, which satisfies each application's varying needs without requiring frequent allocation changes, and system efficiency which requires that the allocation exactly matches the application needs. However, allocating resources close to current needs will result in frequent allocation changes. This can be detrimental to applications since there may be fixed costs (state replication, policy reconfiguration, etc.) that need to be incurred by applications for each allocation change. In this paper, we develop an MDP-based dynamic allocation scheme that uses reinforcement learning to satisfy unpredictable application demands. It minimizes the overall resource allocation needed to satisfy varying application demands while meeting application constraints on the rate of allocation changes. We prove convergence bounds and use real-world traces to study the performance.**

***Index Terms*—resource allocation, reinforcement learning, data center**

I. Introduction

The economic model of data centers is the savings that result by multiplexing resources among different users. Customers can flexibly purchase additional resources when needed, and trim these down when the need has past, while data center service providers can direct resources when and where customers might require. In such a manner, providers can maintain a manageable load across data-centers even while customers encounter large fluctuations in demand. This vision of dynamic resource allocation (and deallocation) comes with its challenges, one of which is how best to manage resource allocation. Resources are allocated to (or deallocated from) a customer's application by scaling the application horizontally or vertically [1].

In horizontal scaling new Virtual Machines (VMs) are added to an already-existing VM cluster executing the application. For example, an application's VMs may sit behind a load balancer that enables the application to grow or shrink dynamically. Each time an allocation or deallocation of VMs happens, the load balancer is reconfigured to allow the new resources to be used, or old resources to be decommissioned. The cost of reconfiguration of the VM cluster can vary based on the application. In vertical scaling, virtualization is used to scale up (or down) the resource allocation within a host for a live VM executing the application. For example, a VM can be resized to have more (or less) CPU based on the current workload. Such scaling has the cost of re-provisioning a host and some downtime for the application. More details on cost associated with resource reconfiguration can be found in [1]–[3].

ISBN 978-3-903176-28-7

It is clear from the above discussion that any resource scaling mechanism can be costly from the perspectives of both cloud provider and customer. For the provider the allocation process can involve initializing dormant hardware and all the supporting mechanisms involved, as well as the short-term impact on customer QoS while this process is taking place. For the customer, the allocation process can involve complex software flows as part of the customer application integrating the new VM or storage into the running application, as well as the time it takes to do so. These challenges are especially significant in a distributed Cloud, comprising of multiple medium-scale data centers, such as those considered by providers for support of network function virtualization (NFV). In these systems, capacity at a given location might be highly utilized and thus improved multiplexing might be very important. As a result, both provider and customer would prefer it if resource allocation would be continuous, fast and with low energy and management overhead. There has been some work [4]–[6] where researchers predict/model the applications workload for future allocation of resources that is as close to the application's real demand as possible. However, these comes with large management overhead per-application, and might not be a scalable solution for the provider.

In this paper, we propose a reinforcement learning based automatic resource scaling method that minimizes the amount of over-provisioning while ensuring the rate of allocation/de-allocation of resources is upper bounded by some user specified threshold. We show how to exploit the problem structure to speed up the execution time of the algorithm. We study the performance of the algorithm on synthetic as well as real-world traces.

II. Problem Formulation

For the ease of exposition, in this section we first consider the single resource allocation problem across multiple time periods. This single resource can be thought of as compute power, disk space, etc. The techniques can be extended directly to the multi-resource allocation problem. Assuming time t is discretized, we denote by $d(t)$ the amount of requested resource from the user at time t and $u(t)$, the decision variable which specifies the amount of resource allocated by the data

center provider. For all t, the provider must ensure $u(t) \geq d(t)$. It is also easy to extend the algorithms developed in this paper to the case where resource shortage is permitted with some shortage costs. If the data center provider attempts to allocate resource to exactly match the demand in every time period, there are two potential drawbacks:

1) There is significant overhead on the data center service provider to monitor a huge number of users constantly and scale up or down resources as needed.
2) Applications can suffer performance degradation when resources are being augmented or deallocated [5].

Therefore, depending on the application, the data service provider ideally would like to reduce the number of reconfigurations (i.e. changes of resource allocation). An easy way to achieve this will be to set the resource allocation at a relatively high value. This will, however, result in wastage of resources that can potentially be used by other customers. Thus, a balance has to be struck between avoiding resource wastage and lowering the number of reconfigurations in resource allocation. For $t \geq 1$, we define

$$\delta(t) = \begin{cases} 1 & \text{if} \quad u(t) \neq u(t-1) \\ 0 & \text{Otherwise} \end{cases}$$

to be the indicator of resource reconfiguration at time t. Ideally, we would like the average reconfiguration rate to be less than an upper bound R. The rate R is specific to the application and will be larger for applications that do not have a significant overhead when changes are made. The average reconfiguration rate constraint can be written as $\frac{1}{T}\sum_{t=1}^{T}\delta(t) \leq R$ considering some finite time interval T, and we have the following formulation:

$$\begin{aligned} Q = \min & \sum_{t=1}^{T} [u(t) - d(t)] \\ u(t) \quad &\geq \quad d(t) \quad 1 \leq t \leq T \\ \sum_{t=1}^{T} \delta(t) \quad &\leq \quad R\,T \end{aligned}$$

The objective is to minimize the amount of over-provisioning subject to the constraints that the allocation amount is greater than the demand and the rate of reconfiguration is bounded above by R.

III. Offline Solution

Before considering the online decision problem, we first solve the offline, finite horizon problem with T time steps. In this offline resource allocation problem, the input are the demand vector $d(1), d(2), \ldots, d(T)$ and upper bound on the number of resource reconfigurations $K = RT$. The output are allocation vector $u(1), u(2), \ldots, u(T)$ that minimizes total resource cost while not having more than K resource reconfigurations.

We use a dynamic programming based approach to solve this problem. Let us define $m_a^b = \max_{a \leq t \leq b} d(t)$ and we denote by $\phi_k(a)$ the minimum objective function value required to cover the demands $d(1) \ldots d(T)$ between time periods a and T using at most k change points. Note that

$$\phi_1(i) = \min_{i \leq j \leq T} (j - i + 1)\, m_i^j + (T - j) m_{j+1}^T.$$

Since $\phi_1(i)$ denotes the optimum allocation for the problem starting from time period i when one jump is permitted. Assume that the jump is after time period $j \geq i$. In this case for the $(j - i + 1)$ intervals before the jump, the function value is set to the maximum value of the demand curve in this interval which is denoted by m_i^j. For the $(T - j)$ intervals after the jump the solution value is set to m_{j+1}^T. The value of $\phi_1(i)$ is to pick the jump point j that minimizes the total allocation. We compute the $\phi_1(i)$ for all values of $1 \leq i \leq T$. This computation takes $O(T^2)$ time. Similarly, for $k = 2, \ldots, K$, we replace the second part with $\phi_{k-1}(j + 1)$:

$$\phi_k(i) = \min_{i \leq j \leq T} (j - i + 1)\, m_i^j + \phi_{k-1}(j + 1).$$

This algorithm is outlined in Algorithm 1. The running time of the algorithm is $O(KT^2)$, and the solution to the offline resource allocation problem serves as a lower bound for the online allocation problem that we deal with next.

Algorithm 1: Offline Resource Allocation

```
for i = 1, 2, ... T do
    m_i^i = d_i
    for j = i + 1, i + 2, ..., T do
        m_i^j = max{m_i^{j-1}, d_j}
    end
end
for i = 1, 2, ... T do
    φ_1(i) = ∞
    for j = i, i + 1, ..., T do
        v = (j − i + 1) m_i^j + (T − j) m_{j+1}^T
        if φ_1(i) > v then
            φ_1(j) = v
            u_1(i) = j
        end
    end
end
for k = 2, 3, ..., K do
    for i = 1, 2, ... T do
        φ_k(i) = ∞
        for j = i, i + 1, ..., T do
            v = (j − i + 1) m_i^j + φ_{k−1}(j + 1)
            if φ_k(i) > v then
                φ_k(j) = v
                u_k(i) = j
            end
        end
    end
end
```

IV. Online Resource Allocation

In the online resource allocation model, demands are revealed at the end of previous time interval. The resource allocation for the current time interval is decided without knowledge of future demands. One promising approach for addressing these types of problems is a reinforcement learning (RL) framework. In reinforcement learning, decisions are made in an online manner while the decision maker learns the outcome of decisions made earlier in previous visits to similar states. However, the constraint for average number of reconfigurations cannot be handled directly in the RL framework. We use a Lagrangian relaxation based approach to indirectly handle the average reconfiguration rate constraint.

A. Enforcing the Reconfiguration Rate Constraint

We relax the average reconfiguration rate constraint by using a Lagrange multiplier of $\lambda \geq 0$ and taking it into the objective function. We can now rewrite the formulation as:

$$\underline{Q}(\lambda) = min \sum_{t=1}^{T} [u(t) - d(t)] + \lambda \left[\sum_{t=1}^{T} \delta(t) - R\,T \right]$$

$$u(t) \quad \geq \quad d(t) \quad 1 \leq t \leq T$$

The Lagrangian relaxation is amenable for solution using reinforcement learning. Moreover, by choosing the Lagrange multiplier λ carefully, we can ensure that the solution obtained is optimal to the original problem. The basis for this is the following result.

Theorem IV.1. *Let $\underline{Q}(\lambda)$ be defined as above. Then $\underline{Q}(\lambda) \leq Q$ for all $\lambda \geq 0$. Moreover, if we find a $\hat{\lambda}$ such that the optimal solution $\hat{u}(t)$ and the corresponding $\hat{\delta}(t)$ to $\underline{Q}(\hat{\lambda})$ satisfies*

$$\hat{\lambda} \left[\sum_{t=1}^{T} \delta(t) - R\,T \right] = 0,$$

then $\hat{u}(t)$ is optimal to the original problem.

Proof. For any $\lambda \geq 0$, it is obvious to see $Q \geq \underline{Q}(\lambda)$. Since $\hat{u}$ is feasible, we know $Q \leq \sum_{t=1}^{T} [\hat{u}(t) - d(t)]$. With $\lambda = \hat{\lambda}$, we have

$$\begin{aligned} Q \geq \underline{Q}(\hat{\lambda}) &= \sum_{t=1}^{T} [\hat{u}(t) - d(t)] + \hat{\lambda} \left[\sum_{t=1}^{T} \delta(t) - R\,T \right] \\ &= \sum_{t=1}^{T} [\hat{u}(t) - d(t)] \geq Q \end{aligned}$$

The second equality holds due to the assumption in the theorem. Combining the results above, we get

$$Q = \underline{Q}(\hat{\lambda}).$$

□

Removing the constant terms from the objective function, for a given λ we effectively have the following objective function:

$$\min \sum_{t} [u(t) - d(t)] + \lambda \sum_{t} \delta(t).$$

We can view λ as the cost of making a change in the allocation. In reality, the time horizon T increases by one in each new time slot. Hence, we will attempt to enforce the average rate constraint asymptotically:

$$\lim_{T \to \infty} \frac{\sum_{t=1}^{T} \delta(t)}{T} \leq R.$$

This is done by checking the average reconfiguration rate for every time window W. If it is less than R we decrease λ and if it is greater than R, we increase λ. More details can be found in section VI.

B. Formulation of Reinforcement Learning

After leaving out the constant terms in the objective function, the online optimization problem that we have to solve is the the following:

$$min \sum_{t=1}^{T} [u(t) - d(t)] + \lambda \sum_{t=1}^{T} \delta(t)$$

$$u(t) \quad \geq \quad d(t) \quad 1 \leq t \leq T$$

We generalize single resource to N types of resource (e.g. $N = 3$ if we consider CPU, disk space and memory resources), and use the following notations for the underlying Markov decision process model:

1) $\mathcal{S}$ represents the *state space*. It is a 2-tuple, $(\boldsymbol{u}, \boldsymbol{d})$, comprising of the current resources allocation and demand in vector form. The i-th element of such vectors represents the value for the i-th type of resources. We assume that resource demand/allocation vectors are selected from the set
$$S = \{0, ..., s^1_{\max}\} \times \ldots \times \{0, ..., s^N_{\max}\}$$
where $s^i_{\max}$ denotes the discretized maximum resource allocation of the i-th resource. Thus, the state space is
$$\mathcal{S} = \{(\boldsymbol{u}, \boldsymbol{d}) | \boldsymbol{u}, \boldsymbol{d} \in S\}.$$
2) $\mathcal{A}$ represents the *action space*. The action $\boldsymbol{a}(t)$ at time t is just deciding the allocation of resources at next time $t+1$, i.e. $\boldsymbol{u}(t+1) = \boldsymbol{a}(t)$. Hence,
$$\mathcal{A} = S.$$
In the following text, we also use $\mathcal{A}(\boldsymbol{u}, \boldsymbol{d})$ to denote the eligible actions at state $(\boldsymbol{u}, \boldsymbol{d})$, i.e. $\mathcal{A}(\boldsymbol{u}, \boldsymbol{d}) = \{\boldsymbol{a} | \boldsymbol{a} \succeq \boldsymbol{d}\}$. We use symbol $\succeq$ for element-wise comparison, i.e. $\boldsymbol{x} \succeq \boldsymbol{y}$ means $x_i \geq y_i$, $\forall i$, and $\boldsymbol{x} \not\succeq \boldsymbol{y}$ means $\exists i$ s.t. $x_i < y_i$.
3) In Markov decision processes, the transition from one time period to another depends on the current state and the current action. We assume that the demand process is independent of the the actions taken by the algorithm, and it is Markovian where
$$\Pr[\boldsymbol{d}(t) = \boldsymbol{x} | \boldsymbol{d}(t-j) = \boldsymbol{x}_j, 1 \leq j \leq t-1] =$$
$$\Pr[\boldsymbol{d}(t) = \boldsymbol{x} | \boldsymbol{d}(t-1) = \boldsymbol{x}_1] = P(\boldsymbol{x}_1, \boldsymbol{x}).$$

$P(\boldsymbol{x}_1, \boldsymbol{x})$ is the probability that a (vector) demand of $\boldsymbol{x}_1$ is followed by a demand of $\boldsymbol{x}$.

4) There are two *costs* in the objective function. First is the cost of providing the resource, we denote by c_i the unit cost associated with the type-i resource, so the total cost will be $c(\boldsymbol{u}(t)) = \sum_i c_i u_i(t)$. Second is the change cost λ, the cost resource reconfiguration. This cost is independent of the amount by which the resource is changed.

At each time step t, the provider does the following in sequence:

1) Pay for the current resource allocation $c\big(\boldsymbol{u}(t)\big)$.
2) Observe new demand $\boldsymbol{d}(t+1)$.
3) Decide the next resource allocation $\boldsymbol{u}(t+1)$.
4) If $\boldsymbol{u}(t+1) \neq \boldsymbol{u}(t)$, pay the change cost λ.

Since the objective function of the online optimization problem does not use discounted costs, we use an average-cost reinforcement learning algorithm. We can either use a model-free approach like Q-learning or a model-based approach where the transition matrix P is explicitly estimated. We show in Figure 2 in section VI that the convergence of model-free approaches like Q-learning is extremely slow even when we use discounted learning. Therefore, an average-cost, model-based reinforcement learning approach is adopted.

V. Model-Based Average-Cost Optimization

We can now solve the online optimization problem using model-based average-cost reinforcement learning. A naive implementation of the Model-Based Average-Cost Reinforcement Learning (MARCO) is shown in Algorithm 2. MARCO is based on (relative) value iteration [7]. At each time step, a sweep over the state space is performed, in which an average cost J and the transition matrix are updated using the observed information at this step. In the standard algorithm, we select a reference state $\boldsymbol{s}_0$. Two sets of value functions: $V(\cdot)$ and the relative value function w.r.t. $\boldsymbol{s}_0$, $V_{\text{rel}}(\cdot)$ are maintained. In this algorithm, there is no discount factor γ (or $\gamma = 1$), and after each round of updates, a normalization step is performed. One key element in a model-based approach is the estimation of the transition matrix. Assume that we observe n_{ij} transitions from demand i to demand j. The estimate of the transition probability is

$$P(i,j) = \frac{n_{ij}}{\sum_j n_{ij}}.$$

Alternatively, we can use a Bayesian approach with a Dirichlet prior. The resource allocation procedure is a value iteration based learning algorithm that uses the estimated traffic matrix. The algorithm is shown in Algorithm 2. The running time of the algorithm is $O(|S|^4)$ at each iteration. In the next section, we use the structure of our problem to improve the standard algorithm.

A. VIRU: Value Iteration With Rapid Updates

The standard algorithm as outlined in Algorithm 2 does not take into account the special structure of the online optimization problem: If the current amount of resource $\boldsymbol{u}(t) \succeq \boldsymbol{d}(t+1)$ then the resource allocation algorithm can either leave the resource unchanged, or change the amount of resource to satisfy future demand requests; If $\boldsymbol{u}(t) \not\succeq \boldsymbol{d}(t+1)$, the amount of resource has to be increased and the allocation algorithm has to decide how much resource to allocate. We now exploit the structure of the optimization problem to develop a Value Iteration scheme with Rapid Updates (VIRU). VIRU exploits the cost structure as well as the fact that the transition probability relies only on the demand. Let $\boldsymbol{d}'$ be the next demand, and $I_{\boldsymbol{a}\neq\boldsymbol{u}}$ be the indicator for a change on resource provision. Using a relative value iteration scheme, we have

$$\begin{aligned} V(\boldsymbol{u},\boldsymbol{d}) &= \min_{\boldsymbol{a}\succeq\boldsymbol{d}} \sum_{\boldsymbol{d}'\in S} P(\boldsymbol{d},\boldsymbol{d}')\Big[c(\boldsymbol{u}) + I_{\boldsymbol{a}\neq\boldsymbol{u}}\cdot\lambda + V_{\text{rel}}(\boldsymbol{a},\boldsymbol{d}')\Big] \\ &= \min_{\boldsymbol{a}\succeq\boldsymbol{d}} \Big[(c(\boldsymbol{u}) + I_{\boldsymbol{a}\neq\boldsymbol{u}}\cdot\lambda) + \sum_{\boldsymbol{d}'\in S} P(\boldsymbol{d},\boldsymbol{d}')V_{\text{rel}}(\boldsymbol{a},\boldsymbol{d}')\Big] \end{aligned}$$

Using the fact that the term $\sum_{\boldsymbol{d}'\in\mathcal{S}} P(\boldsymbol{d},\boldsymbol{d}')V_{\text{rel}}(\boldsymbol{a},\boldsymbol{d}')$ is independent of $\boldsymbol{u}$, we obtain a function

$$F(\boldsymbol{a}) = \sum_{\boldsymbol{d}'\in S} P(\boldsymbol{d},\boldsymbol{d}')V_{\text{rel}}(\boldsymbol{a},\boldsymbol{d}')$$

for $\boldsymbol{a} \succeq \boldsymbol{d}$. We denote the minimum value of this function by $f_m = \min_{\boldsymbol{a}\succeq\boldsymbol{d}} F(\boldsymbol{a})$ and its minimizer by a^*. Then the value function can be re-written as

$$V(\boldsymbol{u},\boldsymbol{d}) = \min_{\boldsymbol{a}\succeq\boldsymbol{d}} \Big[(c(\boldsymbol{u}) + I_{\boldsymbol{a}\neq\boldsymbol{u}}\cdot\lambda) + F(\boldsymbol{a})\Big].$$

At time t, there are two possible situations: at least one type of the current allocation is less than the newly observed demand $\big(\boldsymbol{u}(t) \not\succeq \boldsymbol{d}(t+1)\big)$, or the contrary $\big(\boldsymbol{u}(t) \succeq \boldsymbol{d}(t+1)\big)$. In the first case, the data center operator has the only choice to increase its allocation to match or exceed the new demand.

Algorithm 2: MARCO: Model-Based Average-Cost Algorithm

```
Initialize estimated transition matrix P, value function
  V, relative value function V_rel, and average cost J
  arbitrarily.
Select reference state s_0.
for t = 1, 2, ... do
    foreach (u, d) do
        V(u, d) ←
          min_{a∈A(u,d)} { Σ_{d'∈S} P(d, d') [(c(u) +
          I_{a≠u} · λ) + V_rel(a, d')] }
    J ← V(s_0)
    foreach (u, d) do
        V_rel(u, d) ← V(u, d) − J
    Observe next demand d' and update transition
      probabilities
```

In this case, $I_{\boldsymbol{a}\neq\boldsymbol{u}} = 1$. Since $\boldsymbol{a}^*$ is also the minimizer of $V(\boldsymbol{u},\boldsymbol{d})$, the best action is to provide resource of $\boldsymbol{a}^*$.

$$\begin{aligned} V(\boldsymbol{u},\boldsymbol{d}) &= \min_{\boldsymbol{a}\succeq\boldsymbol{d}} \Big[(c(\boldsymbol{u}) + 1\cdot\lambda) + F(\boldsymbol{a})\Big] \\ &= c(\boldsymbol{u}) + \lambda + f_m \end{aligned}$$

In the second case where the new demand is already satisfied by current allocation, the operator has two choices: change allocation or keep the same allocation. The value function of keeping the allocation is

$$\begin{aligned} V(\boldsymbol{u},\boldsymbol{d}) =& c(\boldsymbol{u}) + \sum_{\boldsymbol{d}'} P(\boldsymbol{d},\boldsymbol{d}') V_{\text{rel}}(\boldsymbol{u},\boldsymbol{d}') \\ =& c(\boldsymbol{u}) + F(\boldsymbol{u}) \end{aligned}$$

without a change cost. If we choose to update the allocation, we must change to $\boldsymbol{a}^*$, and the value function is

$$V(\boldsymbol{u},\boldsymbol{d}) = c(\boldsymbol{u}) + \lambda + f_m$$

We take the minimum over the above 2 quantities. Thus, we update the value functions as follows:

$$V(\boldsymbol{u},\boldsymbol{d}) = \begin{cases} c(\boldsymbol{u}) + \lambda + f_m & \text{for } \boldsymbol{u} \not\succeq \boldsymbol{d} \\ c(\boldsymbol{u}) + \min\{\lambda + f_m, F(\boldsymbol{u})\} & \text{for } \boldsymbol{u} \succeq \boldsymbol{d} \end{cases}$$

Based on this equation, we propose a fast resource allocation algorithm *VIRU* in Algorithm 3. The complexity analysis of Algorithm 3 is presented in Theorem V.1 and its convergence analysis is presented in Theorem V.2.

Algorithm 3: VIRU: Value Iteration with Rapid Updates

```
Initialize estimated transition matrix P, value function
  V, relative value function V_rel, and average cost J
  arbitrarily.
Initialize reference state s_0.
for t = 1, 2, ... do
    foreach d ∈ S do
        for a ⪰ d do
            F_a ← Σ_{d'∈S} P(d, d') V_rel(a, d')
        f_m ← min_{a⪰d} F_a
        foreach u ⋡ d do
            V(u, d) ← c(u) + λ + F_a
        foreach u ⪰ d do
            V(u, d) ← c(u) + min{λ + f_m, F_a}
    J ← V(s_0)
    foreach (u, d) do
        V_rel(u, d) ← V(u, d) − J
    Observe next demand d' and update transition
      probabilities
```

Theorem V.1. *The standard value iteration algorithm MARCO takes $O(|S|^4)$ computations per iteration, and VIRU reduces the complexity to $O(|S|^3)$ computations per iteration.*

Proof. In each iteration of MARCO (Algorithm 2), there are $|S|^2$ states to update. For each state $(\boldsymbol{u},\boldsymbol{d})$, there are $|\mathcal{A}(\boldsymbol{u},\boldsymbol{d})| = O(|S|)$ summations over the set S. Therefore, the complexity is $O(|S|^4)$.

In each iteration of VIRU (Algorithm 3), there are $|S|$ loops for each demand $\boldsymbol{d}$. For each $\boldsymbol{d}$, the computation with highest complexity is the vector $F_{\boldsymbol{a}}$, which takes $|\mathcal{A}(\boldsymbol{u},\boldsymbol{d})| \cdot |S| = O(|S|^2)$ computations. Thus, the complexity of VIRU is $O(|S|^3)$. □

Theorem V.2. *Let $J_n(P_t)$ denote the average cost after n iterations using the estimated traffic matrix P_t, and $J^*(P)$ denote the optimal cost. Then,*

$$|J^*(P) - J_n(P_t)| \leq C_1 \cdot r^n + C_2 \cdot \|P - P_t\|_\infty$$

where C_1, C_2 and $r < 1$ are constants.

Proof. See section IX for an outline of proof. □

Discussion. It's worth mentioning that, though a single, universal change cost λ is used throughout this paper, we can directly extend it to multiple change costs for different types of scaling. We claim without proof that if the number of different change costs are finite, Theorem V.2 still holds. Also notice that the estimation of transition matrix and the value iteration can be performed asynchronously. For example, at a certain time, value iteration may be run n times while transition matrix is updated $t \neq n$ times.

The "curse of dimentionality" is a problem of table-based RL algorithms as the number of states grow exponentially in its dimension (in our case, N). To reduce the computational overhead, we can run VIRU asynchronously in parallel while preserving the convergence results, provided that every state gets updated infinitely often in a infinite time horizon [8]. Moreover, the size of state space can be far smaller than an exponential function of N in practice since the use of different resources are usually highly correlated. Most cloud provides, e.g. Amazon [9], support limited number of resource combinations rather than an arbitrary granularity (Amazon EC2 on-demand service provides less than 200 instances in total).

VI. Performance Evaluation

The evaluation is performed on the public Google cluster data [10]. This data represents cell information, including machine attributes, machine events, job and task events, resource usage information, etc. on a cluster of about 12.5k machines for 29 days. To simplify our analysis, we ignored the hidden scheduling information and analyze the "task_events" data set which records all time stamps of events happened of one specific user. We consider a "SUBMITTED" or "UPDATE_RUNNING" event as the beginning of a task's life cycle and a "FINISH" or "LOST" event as a task's end of life cycle. The requested resource usage is recorded during the task's life cycle. The original usages are rounded to discrete numbers and normalized with a maximum of 20. Time is rounded to seconds. All experiments are performed

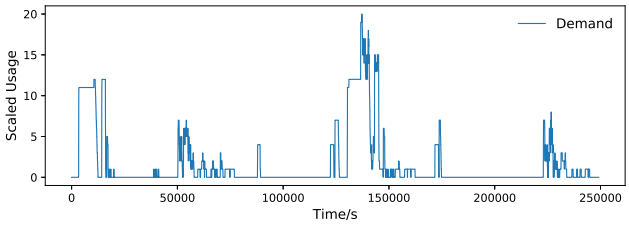

Fig. 1. Demand curve.

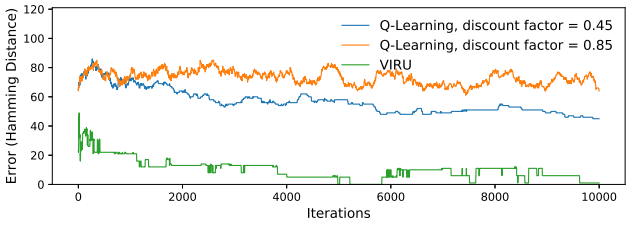

Fig. 2. Convergence speed of different algorithms.

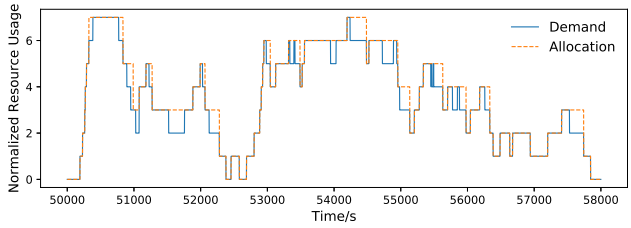

Fig. 3. Demand and allocation curves from time 50000 to 58000.

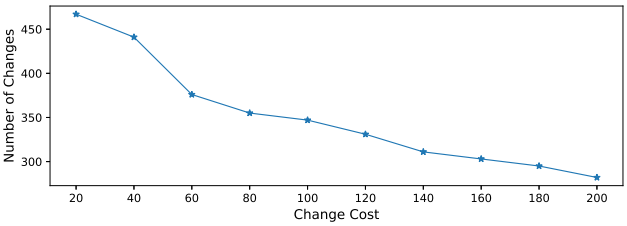

Fig. 4. Number of reconfigurations vs. change cost.

on a physical machine with 2.5 GHz Intel Core i7 processor and 16GB memory. The following evaluations will be based on the data of requested memory from one user, from time stamp 1503960208646 to time stamp 1759531162512 (in μs) as shown in Figure 1.

A. Comparing the Speed of Convergence

To compare the convergence rate of different algorithms, we generated a random 11×11 transition matrix. We implemented Q-learning with $\gamma = 0.85$, $\alpha = 0.5$, as in [11], $\gamma = 0.45$, $\alpha = 0.8$ as in [12], and VIRU for 10000 steps. Actions taken in Q-learning methods are according to an ϵ-greedy policy with $\epsilon = 0.1$. The performance measure is the Hamming distance to the policy derived from average-cost value iteration. As presented in Figure 2, Q-learning with $\gamma = 0.45$ has a trend to converge, while Q-learning with $\gamma = 0.85$ shows a fluctuation with a slower convergence. VIRU shows a significantly faster convergence compared with the Q-learning methods. Note that, discounted Q-learning with an overly small discount factor (e.g. $\gamma = 0.45$) may not have the same convergence as an average-cost algorithm.

B. Performance of VIRU on Trace Data

We run the VIRU algorithm with change cost = 100, and show the demand and allocation curves in Figure 3 in an active period between $t = 50000$ and $t = 58000$ (in seconds). We observe that the allocation curve does not follow every jump on the demand curve, nor does it stay at a high value and fail to react to the demand change. A balance was struck between the frequency of reconfigurations and the resource wastage.

Next, we study the effect of change cost λ on the number of reconfigurations in the whole time interval. Applying different change cost to the demand curve, there is a reduction of number of reconfigurations as change cost increases (Figure 4). As change cost goes from 20 to 200, the number of reconfigurations decreases from 462 to 283.

C. Online Solution vs. Offline Solution

We take the trace data from $t = 50000$ to $t = 58000$ (in seconds), and compare the performance of online and offline algorithms in Figure 5. The online solution curve connects the data points of change costs from 500 to 20 with an interval of 20. The first online data point annotated with 500 means that the algorithm using change cost = 500 has a total cost of 40576, and it reconfigures the allocation 20 times in this time interval. Similar rules applied to the following points with change cost $480, \ldots, 20$. While the offline curve is strictly decreasing, the online curve fluctuates since the algorithm may not be converged completely before the analyzed time interval. We also find that with change cost = 20, the online and offline curve arrive at the same point, because it follows every jump (83 times in total, see Table I) that the demand curve has. The offline solution is obtained from Algorithm 1. It serves as an absolute lower bound of online solution for it is computed using the knowledge of future demands. Moreover, this lower bound is not always achievable since the offline algorithm seeks a local solution between time 50000 and 58000, ignoring data which lies out of the time period, while VIRU seeks a global solution considering all data points. For example, the $(0, 56000)$ point on the offline curve is not achievable, since forcing no reconfiguration will keep the online solution on the max allocation of 20 for all time, and thus the total cost will be $8000 \times 20 = 160000$ instead of $8000 \times \max_t\{d(t)\} = 56000$. That being said, the worst performance of online solution in the figure still achieves 76.8% utilization of resource at change cost = 500.

D. Changing λ to Achieve R

VIRU (Algorithm 3) can be directly applied when an estimate of change cost is available. However, in some situations, it could be too expensive to estimate such a cost caused by state replication, policy reconfigurations etc., or the cost itself could vary over time that an accurate estimate is difficult to

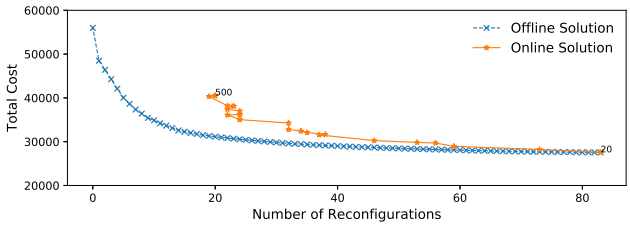

Fig. 5. Online algorithm vs. offline algorithm (online algorithm annotated with change cost).

obtain. In the case where we do not have an estimate of λ but we do have a target reconfiguration rate R, the way we control our algorithm over time is to tune the change cost λ according to the target rate and current rate. In this experiment, we set the target reconfiguration rate R to 5 reconfigurations per 600 seconds, and we measure the actual number of reconfigurations in a window of 600s. The change cost is updated by the following simple, heuristic function:

$$\lambda(t+1) = \max\{140, \lambda(t) + \beta \cdot (R - R_t)\}$$

where β is set to -1, and $R_t = \frac{1}{600}\sum_{\tau=t-599}^{t}\delta(\tau)$. We call this variation of algorithm VIRU with varying λ (vVIRU). Again, we take the data from time 50000 to 58000 and show the number of reconfigurations in the last 600s in Figure 6, 7, comparing two curves with $\lambda = 0$ and varying λ. We find that using the simple updating rule, the reconfiguration rate is pulled towards the target rate $R = 5$, and the number of reconfigurations exceeding 5 per 600s decreased from 4817 times to 1023 times. Furthermore, vVIRU still achieves 82.6% resource utilization (see Table I). It is clear that even with a naive control approach, the reconfiguration rate can be tuned as desired. One is free to tune the algorithm parameters, or to devise a more sophisticated algorithm accordingly to achieve better performance.

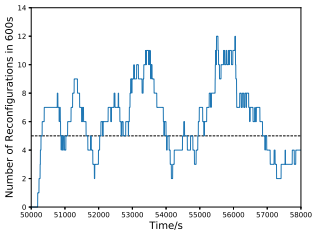

Fig. 6. Number of reconfigurations in 600s with $\lambda = 0$.

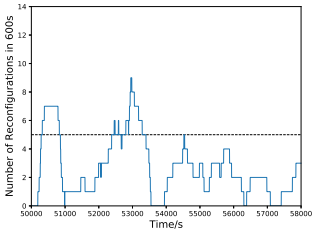

Fig. 7. Number of reconfigurations in 600s with varying λ.

E. Performance Comparison

This section presents the comparison of different autoscaling policies, namely, (i) A threshold-based policy available in Amazon EC2 [13], (ii) A target-tracking based policy available in Amazon EC2 [13] (iii) A Q-learning based policy adapted from [12], and (iv) vVIRU algorithm from section VI-D (since we do not model change cost explicitly). The threshold-based and target-tracking algorithms are implemented according to Amazon's descriptions. We adopt the formulation

TABLE I
PERFORMANCE OF 4 AUTOSCALING POLICIES.

Policy	#Reconf.	Total Cost	Utilization (%)
Demand	83	27569	100.0
vVIRU	37	33372	82.6
Threshold-based	44	54210	50.1
Target-tracking	83	54234	50.8
Q-learning	201	28402	97.1

and algorithm parameters from [12] but we also modify the parameters that are specific to the data traces. To make the algorithms directly comparable, we add an additional rule that the allocation must match or exceed the demand. The key performance measurements are listed in Table I, and the demand and allocation curves are depicted in Figure 8, 9, 10, 11.

We observe that the threshold-based and target-tracking policies behave conservatively, keeping the resource allocation at a high level. The target-tacking policy achieves marginally better utilization rate than the threshold-based policy but it incurs more reconfigurations. vVIRU demonstrates its superiority in terms of both number of reconfigurations and utilization rate compared with threshold-based and target-tracking policies. The Q-learning based algorithm does not aim to reduce the number of reconfigurations. It only penalizes adding more resources, and hence the allocation curve tries to match the demands most of the time. The spikes in Figure 11 are due to the slow convergence of Q-learning. It is clear that even though Q-learning based policy reduces the resource wastage, it reconfigures the resource allocation significantly more often than our algorithm, and in real cloud operations, an ϵ-greedy based policy could be too expensive to be applied due to the unaffordable cost of exploration in the training phase.

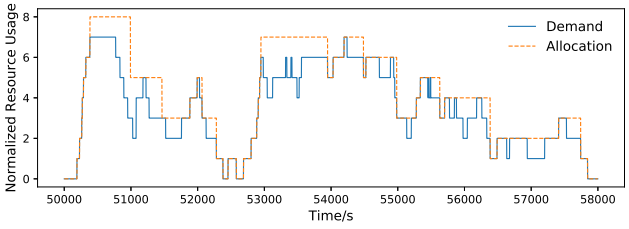

Fig. 8. VIRU

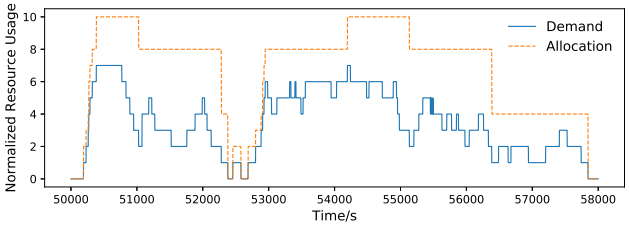

Fig. 9. Threshold-based

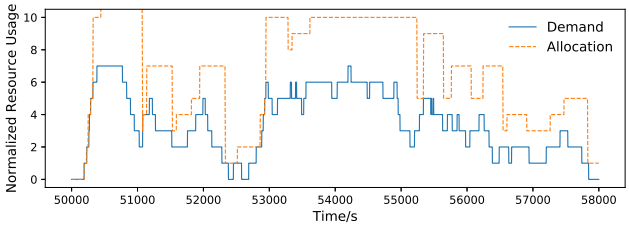

Fig. 10. Target-tracking

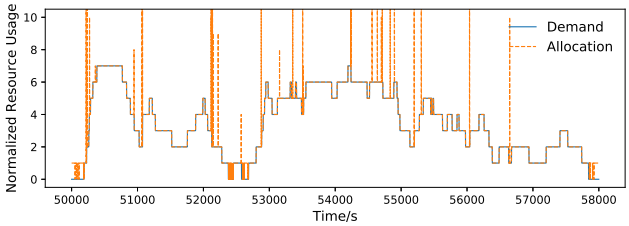

Fig. 11. Q-learning

VII. Related Work

The application of reinforcement learning to autoscaling in data centers has been extensively studied. Different combination of metrics such as the demand at current time, demand at previous time, response time etc. are used to represent the system state. Tesauro et al. [14] proposed a horizontal scaling algorithm in which both SARSA and queuing model were employed in a hybrid way. A neural network is implemented to maintain the state values. To speed up the convergence of RL, Dutreilh et al. [12] gave an initial approximation of the Q-function for faster convergence. There is also research [11], [15]–[17] for vertical scaling . In [15], simulated experiences is used for value function estimation in order to reduce the training time, while in [16] from the same authors, a distributed learning mechanism was used to facilitates the VM resource provisioning. Parallelism is utilized in Q-learning [11] to accelerate the learning, where each agent can learn the value of unvisited state from its neighbors. All these papers use a discounted cost function. However, we abandoned the discount factor and adopted an average-cost method instead since the autoscaling problem can be naturally cast as an average-cost problem and it is not clear what discount factor should be used. A discount factor is expedient for speeding up convergence rather than desired. A model-based approach is adopted in [15] to address scalability and adaptability issues. As in [12] we use a model-based RL because such approach is more data efficient and converges much faster compared with model-free methods. Other techniques for autoscaling include threshold based techniques [18], time series analysis [19], [20] and queuing theory [21], [22] based techniques.

VIII. Conclusion

In this work, we present a model-based, average-cost reinforcement learning algorithm and its extension for resource autoscaling in clouds. Our goal is to restrict the reconfiguration frequency while still achieving good resource utilization (VIRU) even when the change cost is hard to model (vVIRU). Theoretically, we exploit the structure of the optimization problem and improve the computational cost of a standard value iteration algorithm by a factor of $|S|$. Moreover, we prove the convergence bound of the proposed VIRU algorithm. Experimentally, we show the convergence rate of VIRU and compare vVIRU with a threshold-based policy, a target-tracking policy and a Q-learning policy that are either used by real-world vendors or popular in literature. Results demonstrate our algorithm's superiority over other methods in terms of both resource utilization and reconfiguration frequency.

References

[1] S. Dutta, S. Gera, A. Verma, and B. Viswanathan, "SmartScale: Automatic Application Scaling in Enterprise Clouds," in *IEEE Fifth International Conference on Cloud Computing*, 2012.

[2] S. Chaisiri, B.-S. Lee, and D. Niyato, "Optimization of Resource Provisioning Cost in Cloud Computing," *IEEE Transactions on Services Computing*, 2012.

[3] A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. R. Kompella, " ElastiCon: An Elastic Distributed SDN Controller ," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2014.

[4] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai, "Optimal Autoscaling in a IaaS Cloud," in *International Conference on Autonomic Computing*, 2012.

[5] N. Roy, A. Dubey, and A. Gokhale, "Efficient Autoscaling in the Cloud using Predictive Models for Workload Forecasting," in *IEEE 4th International Conference on Cloud Computing*, 2011.

[6] C. Bunch, V. Arora, N. Chohan, C. Krintz, S. Hegde, and A. Srivastava, "A Pluggable Autoscaling Service for Open Cloud PaaS Systems," in *IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, 2012.

[7] D. J. White, "Dynamic programming, markov chains, and the method of successive approximations," *Journal of Mathematical Analysis and Applications*, vol. 6, no. 3, pp. 373–376, 1963.

[8] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and distributed computation: numerical methods*. Prentice hall Englewood Cliffs, NJ, 1989, vol. 23.

[9] "Amazon ec2 instance types - amazon web services." [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[10] "Google cluster data," Dec 2017. [Online]. Available: https://github.com/google/cluster-data

[11] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.

[12] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow," in *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, 2011, pp. 67–74.

[13] "Amazon ec2 autoscaling." [Online]. Available: https://aws.amazon.com/ec2/autoscaling/

[14] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A hybrid reinforcement learning approach to autonomic resource allocation," in *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on*. IEEE, 2006, pp. 65–73.

[15] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, "Vconf: a reinforcement learning approach to virtual machines auto-configuration," in *Proceedings of the 6th international conference on Autonomic computing*. ACM, 2009, pp. 137–146.

[16] J. Rao, X. Bu, C.-Z. Xu, and K. Wang, "A distributed self-learning approach for elastic provisioning of virtualized cloud resources," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*. IEEE, 2011, pp. 45–54.

[17] P. Tang, F. Li, W. Zhou, W. Hu, and L. Yang, "Efficient auto-scaling approach in the telco cloud using self-learning algorithm," in *Global Communications Conference (GLOBECOM), 2015 IEEE*. IEEE, 2015, pp. 1–6.

[18] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 2012, pp. 644–651.
[19] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-aware server provisioning and load dispatching for connection-intensive internet services." in *NSDI*, vol. 8, 2008, pp. 337–350.
[20] A. Chandra, W. Gong, and P. Shenoy, "Dynamic resource allocation for shared data centers using online measurements," in *International Workshop on Quality of Service*. Springer, 2003, pp. 381–398.
[21] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth, "Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control," in *Proceedings of the 3rd workshop on Scientific Cloud Computing*. ACM, 2012, pp. 31–40.
[22] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 3, no. 1, p. 1, 2008.
[23] P. J. Schweitzer and A. Federgruen, "Geometric convergence of value-iteration in multichain markov decision problems," *Advances in Applied Probability*, vol. 11, no. 1, pp. 188–217, 1979.
[24] G. E. Cho and C. D. Meyer, "Comparison of perturbation bounds for the stationary distribution of a markov chain," *Linear Algebra and its Applications*, vol. 335, no. 1-3, pp. 137–150, 2001.

IX. Appendix: Proof of Convergence (Outline)

In this section, we present an outline of the proof of Theorem V.2. We use simplifying assumptions listed below:

1) The Markov chain we consider has only one ergodic class, with possible transient states.
2) $\lim_{t\to\infty} \Pr\left[P_t(i,j) \neq 0 \Leftrightarrow P(i,j) \neq 0\right] = 1$. This assumption ensures that eventually the estimated Markov chain has an identical connectivity structure as the true Markov chain.
3) The policies throughout this proof are always deterministic. Ties can be broken arbitrarily.

In an average-cost value iteration, the optimal average cost is defined as

$$\begin{aligned} J^* &= \min_{\pi} \sum_{\boldsymbol{u},\boldsymbol{d}\in S} \mu_\pi(\boldsymbol{u},\boldsymbol{d}) c_\pi(\boldsymbol{u},\boldsymbol{d}) \\ &= \min_{\pi} \boldsymbol{\mu}_\pi \cdot \boldsymbol{c}_\pi \end{aligned}$$

where $\mu_\pi(\boldsymbol{u},\boldsymbol{d})$, $c_\pi(\boldsymbol{u},\boldsymbol{d})$ denote the stationary distribution and the cost of state $(\boldsymbol{u},\boldsymbol{d})$ under policy π, respectively. The convergence indicator is the distance between two average costs

$$D = |J^*(P) - J_n(P_t)|$$

where $J_n(P_t)$ denotes the *estimated* average cost after n sweeps of value iteration, using the *estimated* transition matrix P_t with t updates, and $J^*(P)$ denotes the *optimal* average cost $J^*(P)$ of *true* transition matrix P. We break D into 2 parts:

$$\begin{aligned} D &= |J^*(P) - J_n(P_t)| \\ &\leq D_1 + D_2 \end{aligned}$$

where $D_1 \triangleq |J_n(P_t) - J^*(P_t)|$ and $D_2 \triangleq |J^*(P_t) - J^*(P)|$.

Lemma IX.1 and IX.2 give bounds for the two distances mentioned above.

Lemma IX.1. *Distance D_1 is bounded by*

$$|J_n(P_t) - J^*(P_t)| \leq C_1 \cdot r^n$$

where C_1 and $r < 1$, are two constants depending on the MDP.

Proof. Please refer to [23] for the details of proof and the values of C_1 and r. □

Lemma IX.2. *Distance D_2 is bounded by*

$$|J^*(P_t) - J^*(P)| \leq C_2 \cdot \|P_t - P\|_\infty$$

where $\|\cdot\|_\infty$ denotes the infinity matrix norm, and C_2 is a constant depending on the MDP.

Proof.

$$\begin{aligned} |J^*(P_t) - J^*(P)| &= |\min_{\pi} \boldsymbol{\mu}'_\pi \cdot \boldsymbol{c}_\pi - \min_{\pi} \boldsymbol{\mu}_\pi \cdot \boldsymbol{c}_\pi| \\ &\leq \max_{\pi} |\boldsymbol{\mu}'_\pi \cdot \boldsymbol{c}_\pi - \boldsymbol{\mu}_\pi \cdot \boldsymbol{c}_\pi| \\ &\leq \max_{\pi} \|\boldsymbol{\mu}'_\pi - \boldsymbol{\mu}_\pi\|_2 \cdot \|\boldsymbol{c}_\pi\|_2 \\ &\leq |S| \cdot c_{\max} \cdot \max_{\pi} \|\boldsymbol{\mu}'_\pi - \boldsymbol{\mu}_\pi\|_\infty \end{aligned}$$

The last inequality follows from the relationship between 2-norm and ∞-norm. $c_{\max}$ denotes the maximum possible immediate cost; $\boldsymbol{\mu}'_\pi$ denotes the stationary distribution with policy π and transition matrix P_t.

Now the problem boils down to the perturbation of stationary distribution of Markov chain. Supposing for simplicity that time t is large enough so the condition $P_t(i,j) \neq 0 \Leftrightarrow P(i,j) \neq 0$ in assumption 2 is already satisfied, we only need to consider the recurrent states of this MDP. Using conclusions from [24], and the assumption that P does not depend on allocation u, we have:

$$\|\boldsymbol{\mu}'_\pi - \boldsymbol{\mu}_\pi\|_\infty \leq \kappa_\pi \|P_t - P\|_\infty$$

where

$$\kappa_\pi = \frac{1}{2} \max_j \left\{ \frac{\max_{i\neq j} m_{ij}}{m_{jj}} \right\}.$$

m_{ij} denotes the mean first passage time from recurrent state i to recurrent state $j \neq i$, and $m_{jj} = 1/\mu_\pi(j)$ for a recurrent state j.

$$\begin{aligned} |J^*(P_t) - J^*(P)| &\leq |S| r_{\max} \max_{\pi} \|\boldsymbol{\mu}'_\pi - \boldsymbol{\mu}_\pi\|_\infty \\ &\leq |S| r_{\max} \max_{\pi} \kappa_\pi \|P_t - P\|_\infty \end{aligned}$$

Since π is from a finite deterministic policy set, we can define $\kappa_{\max} \triangleq \max_\pi \kappa_\pi < \infty$. Letting $C_2 = |S| r_{\max} \kappa_{\max}$, we arrive at the following conclusion:

$$|J^*(P_t) - J^*(P)| \leq C_2 \cdot \|P_t - P\|_\infty$$

□

Using conclusions from lemma IX.1 and IX.2, we complete the proof of Theorem V.2

Proof.

$$\begin{aligned} |J^*(P) - J_n(P_t)| &\leq |J_n(P_t) - J^*(P_t)| + |J^*(P_t) - J^*(P)| \\ &\leq C_1 \cdot r^n + C_2 \cdot \|P - P_t\|_\infty \end{aligned}$$

□