Taking Out the GaRLbage

Mark Sabini – `msabini` & Clare Chen – `cchen9`

October 30, 2017

Introduction

Modern programming languages afford users countless features, ranging from first-class functions to operator overloading. Although these conveniences can make code both cleaner and easier to write, it is often easy to overlook the ultimate headache-reliever: garbage collection (GC). As a form of automatic memory management, GC reduces the cognitive load on the programmer, helping to prevent issues such as memory leaks and double frees.

Despite its benefits, GC requires both sufficient memory and an efficient implementation to perform at a level comparable to manual memory management. That being said, researchers Matthew Hertz and Emery Berger demonstrated that provided enough memory, GC could perform on-par with explicit memory management [2]. As a result, we turn our attention to the crafting and design of an effective garbage collector.

When implementing a garbage collector, it is possible to utilize a fixed set of rules for all programs in order to decide whether to garbage-collect. However, the optimal set of rules R for a program P, could differ from the optimal set of rules R' for a different program P'. Thus, it can be beneficial to construct a more adaptive garbage collector that dynamically decides which rules to apply. In their paper "To Collect or Not To Collect? Machine Learning for Memory Management", Andreasson et al. detail a GC approach that uses reinforcement learning (RL) to train an agent to decide whether to carry out GC [1]. The resulting agent was found to learn heuristics similar to those used in modern JVMs, showing that RL can provide valuable insight into the construction of adaptive garbage collectors.

Problem Formulation

We aim to use RL to automate the extraction of optimal policies for GC across various applications, so that engineers can incorporate these policies into existing GC protocols. We base our approach off of [1], and model GC as a Markov Decision Process (MDP):

- **States**: Let $m_{1,t}$ be the available memory at the current timestep t, and let $m_{2,t} = m_{1,t} - m_{1,t-1}$ be the change in memory from the last timestep. We can model the agent's state at timestep t as $s_t = (m_{1,t}, m_{2,t})$. Other possible features for the MDP state include (but are not limited to) the current heap fragmentation factor and memory allocation rate.

- **Actions**: At any given timestep t, the agent can either choose to do nothing ($a_t = 0$) or garbage-collect ($a_t = 1$). If time permits, other possible actions to investigate include growing or shrinking the heap ($a_t = 2, 3$, respectively).
- **Rewards**: Actions such as GC and increasing the heap size give a reward $-r_g$ ($r_g > 0$), as they represent time-intensive operations. To discourage running out of memory, we assign a reward of $-r_m$ ($r_m \gg r_g$), as the application will be stalled until GC is finished.

Techniques & Algorithms

We plan to use OpenAI Gym as a framework for hosting our algorithms in order to decouple the approach and application as much as possible. We will implement our own environment conforming to the specification above. Using OpenAI Gym allows us to not only easily train our RL agent, but also quickly simulate various memory allocation patterns. In turn, we will be able to examine learned policies, and qualitatively infer heuristics for different use cases.

We will use a tile-coding method (similar to [1]) to discretize the continuous state space and the Q-learning algorithm to extract an optimal policy. The Q-learning algorithm applies incremental updates to the Bellman equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s,a) \right) \tag{1}$$

Evaluation

To evaluate the policies learned by our RL agent, we will compute the average reward-per-episode [3], which acts as a proxy for successful memory management. Using this metric, we will compare the performance of our RL agent to that of the following baselines:

1. **Threshold GC**: $\pi((m_{1,t}, m_{2,t})) = \mathbf{1}[m_{1,t} < \alpha]$ (for some threshold α). This policy will garbage-collect when the amount of available memory drops below α.
2. **Horizon GC**: $\pi((m_{1,t}, m_{2,t})) = \mathbf{1}\left[\frac{m_{1,t}}{m_{2,t}} < \beta\right]$ (for some threshold β). This policy will garbage-collect when the expected number of timesteps before running out of memory drops below β.

References

[1] Andreasson, Eva, Frank Hoffmann, and Olof Lindholm. "To Collect or Not to Collect? Machine Learning for Memory Management." *Java Virtual Machine Research and Technology Symposium.* 2002.

[2] Hertz, Matthew, and Emery D. Berger. "Quantifying the performance of garbage collection vs. explicit memory management." *ACM SIGPLAN Notices.* Vol. 40. No. 10. ACM, 2005.

[3] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).