Reinforcement Learning in Garbage Collection

Andrii Kuch, Kenneth B. Kent, Gerhard W. Dueck
University of New Brunswick, IBM Canada
Faculty of Computer Science
andrii.kuch@unb.ca, ken@unb.ca, gdueck@unb.ca

Background

- The balanced collector amortizes the cost of global garbage collection across many collection pauses, reducing the effect of whole heap collection times.
- Each pause should attempt to perform a self contained collection, returning free memory back to the application for immediate reuse.
- The balanced collector dynamically selects heap areas to collect in order to maximize the return-on-investment of time and effort. The set of regions selected for garbage collection is called a collection set.
- By manipulating the collection set we can manipulate performance of balanced collector.

Problem Formulation

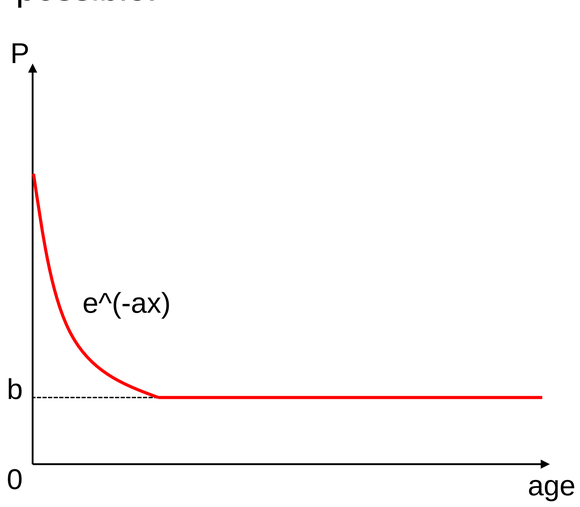
- The balanced collector leverages the observation that recently allocated objects are likely to quickly become garbage by adding regions with younger objects to the collection set more frequently.
- As various applications are allocating memory differently, optimal rules to pick a collection set can be determined case-by-case.
- Questions the project aims to answer:
  - “Objects die young” — is “young” the same for all applications?
  - Can these allocation differences be noticed and leaned?
  - Can we use it to improve performance of garbage collection?

Why reinforcement learning?

- Relevant properties of reinforcement learning:
  - A model of the environment is known, but an analytical solution is not available — it is easier to find collector parameters by trial and error, than to analyze application’s code.
  - Interaction with environment happens in discrete time steps — collections are periodic, one collection is one step.
  - The only way to collect information about the environment is by interacting with it — the only way to know if collection is effective is to actually perform it.

Intended design

- Model is probabilistic — heap regions are selected by chance, depending on age.
- Probability is described by function *P(age)*. The function has few parameters. Changing these parameters reshapes function and thus influences distribution of regions in final collection set.
- Parameters are discretized to create a checker-like world that algorithm can travel and explore.
- The process of learning is formulated as **Markov decision process** (MDP) where sets of parameters are states and moving between them are the actions the learning algorithm can take.
- Learning algorithm tries different parameters to reshape the *P(age)* function. On each collection, algorithm has a chance a try one set of parameters.
- Ratio *retrieved_memory / allocated_memory* is used as feedback, i.e. reinforcement to give the algorithm an idea if last try was good or bad.
- Example: Algorithm is checking different sets of parameters one by one and gathering feedback. Type of *P(age)* function is one of many possible.



Variations of parameter **a**

Variations of parameter **b**

0.45	0.63	0.42	0.32	0.18
0.53	0.43	0.37	0.30	0.19
0.49	0.39	0.29	0.26	0.25
0.40	0.33	0.26	0.23	0.18
0.31	0.28	0.16	0.19	0.15

- After some time of exploration, algorithm will be able to generalize obtained knowledge into a policy — understanding of what set of parameters is more promising.

Flexibility and variations

- Some element of the model can be modified to fine tune it:
  - By changing how continuous parameters are discretized, we can change the size of the world to be explored, thus we can balance between accuracy and exploration time.
  - General type of the probability function has be set in advance. Different types can have different performance.