Choosing a Set of Partitions to Collect in a Connectivity-Based Garbage Collector

Martin Hirzel Harold N. Gabow Amer Diwan
{hirzel,hal,diwan}@cs.colorado.edu
Department of Computer Science
University of Colorado, Boulder 80309-0430

Technical Report CU-CS-958-03
August 8, 2003

Abstract

Connectivity-Based Garbage Collection is a new family of garbage collection algorithms that are based on potential object connectivity properties. Objects are placed into partitions based on a static connectivity analysis. When the program needs memory, the collector can choose a set of partitions to do a garbage collection on. This choice should maximize the expected benefit in reclaimed memory, while minimizing the cost in expended work. We formalize this problem and present the *flow-based* chooser, an algorithm that uses network flow to find an optimal solution. We compare it to the *greedy* chooser, a simpler algorithm that may not find an optimal solution.

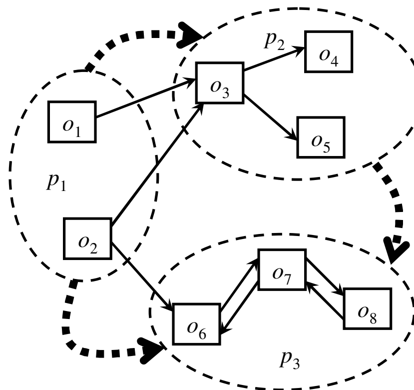

Figure 1: Example partitioning. Solid boxes are objects, solid arrows are pointers, dashed ovals are partitions, and dashed arrows are partition edges.

1 Problem statement

A connectivity-based garbage collector (Cbgc) divides the set O of heap objects into a set P of partitions based on a conservative estimate of their connectivity [6]. A *partitioning* (m, P, E) of the objects consists of a *partition map* $m : O \to P$ and a *partition dag* (P, E) (a dag is a directed acyclic graph). The partition map m associates each object $o \in O$ with its partition $m(o) \in P$. The edges E of the partition dag represent the *may-point-to* relations. In other words, if a pointer may exist between two heap objects, then either the objects must be in the same partition, or there must exist an edge between their partitions in the partition dag. Figure 1 gives an example partitioning.

When the Cbgc needs to free up some memory, it scavenges a subset $C \subseteq P$ of the partitions. The goal is to choose C such that (i) the objects in C can be collected independently from the rest of the heap, and (ii) the benefit/cost ratio for collecting C is as high as possible.

For the independence property (i), we make use of the connectivity information that the partition dag gives us: we pick a set $C \subseteq P$ of partitions that is closed under the predecessor relation ($q \in C \wedge (p, q) \in E \Rightarrow p \in C$). When an object in C is not reachable from the roots via any objects in C, it is not reachable from the roots via any objects in O, and is therefore garbage; its memory can be reclaimed.

For the benefit/cost ratio property (ii), we start out by estimating the local cost and benefit of collecting an individual partition. This estimate is represented by a pair of functions $\mathit{dead}, \mathit{live} : P \to \mathbb{N}$ mapping partitions to nonnegative integers. The number of dead objects $\mathit{dead}(p)$ in a partition p is the benefit of collecting it, since their memory can be reclaimed for reuse. The number of live objects $\mathit{live}(p)$ in a partition p is the cost of collecting it, since they need

to be traversed before unreachable objects can be reclaimed as garbage. In this write-up, we assume the problem of coming up with the two functions *dead* and *live* has already been solved.

Figure 2 shows an example of a partition dag with cost/benefit estimates. For instance, partition p_1 may contain objects pointing to objects in p_3, and we estimate that p_2 contains four dead objects and three live objects $(4:3)$.

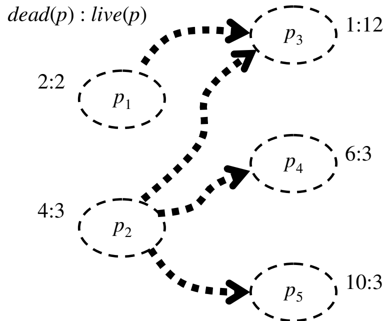

Figure 2: Example partition dag annotated with the estimated *dead* and *live* functions.

We define the quality of a set C of partitions as $quality(C) = \sum_{p \in C} dead(p) / \sum_{p \in C} live(p)$. The problem that we are interested in in this technical report is the following:

> Given a partition dag (P, E) and a pair of functions $dead, live : P \to \mathbb{N}$, find a closed subset $C \subseteq P$ of partitions that maximizes $quality(C)$.

The solution to this problem may not be uniquely defined. The naive algorithm of computing the quality of all closed sets of partitions has complexity $O(2^P)$.

Table 1 shows all closed subsets of the partition dag in Figure 2 and their quality. The best quality set is $\{p_2, p_5\}$.

2 The flow-based chooser

The *flow-based* chooser finds an optimal solution to the problem stated in 1. It does so by reducing it to a max-weight closed set problem, for which the literature has solutions using network flow algorithms. Section 2.1 describes the reduction, Section 2.2 reviews some basics on network flow, and Section 2.3 reviews the solution to the max-weight closed set problem from the literature.

Table 1: Qualities of closed subsets of the partition dag in Figure 2.

Closed set $C \subseteq P$					Quality
p_1	p_2	p_3			$7/17 = 0.41$
p_1	p_2	p_3	p_4		$13/20 = 0.65$
p_1	p_2	p_3		p_5	$17/20 = 0.85$
p_1	p_2	p_3	p_4	p_5	$23/23 = 1.00$
	(empty set)				$0/0 = \text{undefined}$
p_1					$2/2 = 1.00$
	p_2				$4/3 = 1.33$
	p_2		p_4		$10/6 = 1.67$
	p_2		p_4	p_5	$16/9 = 1.78$
	p_2			p_5	$14/6 = 2.33$

2.1 Reduction to max-weight closed set problem

In Section 2.1.1 we prove two lemmas and make some observations that allow us to reduce the Cbgc partition selection problem to the max-weight closed set problem. In Section 2.1.2, we formulate the algorithm that does that. Section 2.1.3 describes how our algorithm uses geometry so it needs to solve only a logarithmic number of instances of the max-weight closed set problem.

2.1.1 Preparation

One difficulty of the problem we are trying to solve is that the properties of individual partitions do not simply add up ($\frac{d_1}{l_1} + \frac{d_2}{l_2} \neq \frac{d_1+d_2}{l_1+l_2}$). Therefore, our first goal is to reduce the problem to one where they *do* add up. To this end, we define a family $w : \mathbb{R} \times P \to \mathbb{R}$ of weight functions on partitions. For each real number $x \in \mathbb{R}$, the weight function $w_x : P \to \mathbb{R}$ is given by $w_x(C) = dead(p) - x \cdot live(p)$. Table 2 shows the weights w_x for the example in Figure 2 and $x = \frac{21}{11}$.

Table 2: The weight function $w_{\left(\frac{21}{11}\right)}$ for the partitions in Figure 2.

Partition p	p_1	p_2	p_3	p_4	p_5
$\frac{dead(p)}{live(p)}$	$\frac{2}{2}$	$\frac{4}{3}$	$\frac{1}{12}$	$\frac{6}{3}$	$\frac{10}{3}$
$w_{\left(\frac{21}{11}\right)}(p)$	$-\frac{20}{11}$	$-\frac{19}{11}$	$-\frac{241}{11}$	$\frac{3}{11}$	$\frac{47}{11}$

Lemma 1 gives a feeling for how the weight-functions are related to our problem.

Lemma 1 *For all $x \in \mathbb{R}$ and all non-empty closed sets of partitions $C \subseteq P$, we have*

(i) $0 \leq \sum_{p \in C} w_x(p) \Leftrightarrow x \leq \mathit{quality}(C)$
(ii) $0 \geq \sum_{p \in C} w_x(p) \Leftrightarrow x \geq \mathit{quality}(C)$
(ii) $0 = \sum_{p \in C} w_x(p) \Leftrightarrow x = \mathit{quality}(C)$

Proof. Part (i) can be seen by symbol-pushing:

$$
\begin{aligned}
& 0 \leq \sum_{p \in C} w_x(p) && \Leftrightarrow \\
& 0 \leq \Big(\sum_{p \in C} \mathit{dead}(p) - x \cdot \mathit{live}(p) \Big) && \Leftrightarrow \\
& 0 \leq \Big(\sum_{p \in C} \mathit{dead}(p) \Big) - x \cdot \Big(\sum_{p \in C} \mathit{live}(p) \Big) && \Leftrightarrow \\
& x \leq \Big(\sum_{p \in C} \mathit{dead}(p) \Big) / \Big(\sum_{p \in C} \mathit{live}(p) \Big) && \Leftrightarrow \\
& x \leq \mathit{quality}(C)
\end{aligned}
$$

In the above, we used the property that the domain of the functions *live* and *dead* is non-negative. We required that C is non-empty, since otherwise $\mathit{quality}(C) = 0/0$ is undefined. The proof for the other two parts of the lemma is analogous. □

Using Lemma 1, we can show how the solution of maximizing a straight sum correlates with the solution of our problem at hand. Let K be the set of non-empty closed subsets of P.

Lemma 2 *For all $x \in \mathbb{R}$, we have*

$$
\begin{aligned}
&(i) \quad 0 \leq \max_{C \in K} \Big\{ \sum_{p \in C} w_x(p) \Big\} \Rightarrow x \leq \max_{C \in K} \{ \mathit{quality}(C) \} \\
&(ii) \quad 0 \geq \max_{C \in K} \Big\{ \sum_{p \in C} w_x(p) \Big\} \Rightarrow x \geq \max_{C \in K} \{ \mathit{quality}(C) \} \\
&(iii) \; 0 = \max_{C \in K} \Big\{ \sum_{p \in C} w_x(p) \Big\} \Rightarrow x = \max_{C \in K} \{ \mathit{quality}(C) \}
\end{aligned}
$$

Proof. For part (i), let C_1 be a witness for the premise, in other words, let $C_1 \in K$ satisfy $0 \leq \sum_{p \in C_1} w_x(p)$. Then by Lemma 1, we have $x \leq \mathit{quality}(C_1)$. Since we also have $\mathit{quality}(C_1) \leq \max_{C \in K}\{\mathit{quality}(C)\}$, we get the conclusion $x \leq \max_{C \in K}\{\mathit{quality}(C)\}$.
Part (ii) can be seen with a proof by contradiction. Assume that $0 \geq \max_{C \in K}\{\sum_{p \in C} w_x(p)\}$, but that $x < \max_{C \in K}\{\mathit{quality}(C)\}$. Then there exists a closed set of partitions $C_2 \in K$ for which $x < \mathit{quality}(C_2)$. By Lemma 1 that means that $\sum_{p \in C_2} w_x(p) > 0$. But then we would have $\sum_{p \in C_2} w_x(p) > \max_{C \in K}\{\sum_{p \in C} w_x(p)\}$, contradicting the maximality.
Part (iii) follows from the conjunction of parts (i) and (ii). □

In Section 2.3, we will see how to find a solution (along with a witness) to the max-weight closed set problem

$$\max_{C \in K \cup \{\emptyset\}} \Big\{ \sum_{p \in C} w_x(p) \Big\}.$$

Here, we sketch how to use that know-how to find a solution (along with a witness) to our main problem

$$\max_{C \in K} \{ \mathit{quality}(C) \}.$$

The basic idea is to try different values for x and to use Lemma 2 to search for an x_M that satisfies

$$x_M = \max_{C \in K} \{ \mathit{quality}(C) \}.$$

We first observe some properties of the search space.

1. For every partition q that has no predecessors ($\neg \exists p \,.\, (p, q) \in E$), the singleton set of partitions $\{q\}$ is closed ($\{q\} \in K$). Hence, $x_{\min} = \max\{\mathit{quality}(\{q\}) \mid \neg \exists p \,.\, (p,q) \in E\}$ is a lower bound on x_M. For example, in Figure 2, $x_{\min} = \max\{\mathit{quality}(\{p\}) \mid p \in \{p_1, p_2\}\} = \mathit{quality}(\{p_2\}) = \frac{4}{3}$.
2. Either $\sum_{p \in P} \mathit{dead}(p) = 0$, or the solution has positive quality, and hence contains at least one partition. We will ignore the case $\sum_{p \in P} \mathit{dead}(p) = 0$, since in that case, it would not make sense to attempt scavenging at all. Let $x_{\max} = \max\{\mathit{quality}(\{p\})\}$ be the best quality of any singleton set of partitions (including singleton sets of partitions with predecessor, which are not closed). Since the quality of a set of partitions is at most as high as the quality of its best member, $x_{\max}$ is an upper bound on x_M. For example, in Figure 2, $x_{\max} = \max\{\mathit{quality}(\{p\})\} = \mathit{quality}(\{p_5\}) = \frac{10}{3}$.
3. Let $D = \sum_{p \in P} \mathit{dead}(p)$ and $L = \sum_{p \in P} \mathit{live}(p))$. We know that x_M must be of the form d/l where $0 < d \leq D$ and $0 < l \leq L$. Hence, there are $O(DL)$ valid values for x_M. For example, in Figure 2, $D = 23$ and $L = 23$, so x_M must be of the form d/l with $0 < d \leq 23$ and $0 < L \leq 23$. Note that in general $D \neq L$.

2.1.2 Flow-based chooser algorithm

We can now formulate the algorithm, see Figure 3. If D is the total number of dead objects, L the total number of live objects, P the number of nodes and E the number of edges in the partition dag, then the total complexity of the algorithm is

$$O\Big(\log(DL) \cdot \Big(\min\{D, L\} + PE \log\Big(\frac{P^2}{E}\Big) \Big) \Big)$$

The algorithm works as follows. Lines 1 and 2 initialize $[\mathit{low}, \mathit{high}]$ to the range in which the solution of the partition selection problem must reside.

	action	complexity
1	$\mathit{low} \leftarrow x_{\min} = \max\{\mathit{quality}(\{q\}) \mid \neg\exists p \,.\, (p,q) \in E\}$;	$P+$
2	$\mathit{high} \leftarrow x_{\max} = \max\{\mathit{quality}(\{p\})\}$;	$P+$
3	**do**{	(
4	choose $x = \frac{d}{l}$ such that $\mathit{low} \leq x \leq \mathit{high} \wedge 0 < d \leq D \wedge 0 < l \leq L$ and such that x halves the search space;	$\min\{D, L\}+$
5	find y_x, C_x such that $y_x = \max\limits_{C \in K \cup \{\emptyset\}} \left\{ \sum\limits_{p \in C} w_x(p) \right\} = \sum\limits_{p \in C_x} w_x(p)$;	$PE \log(P^2/E)+$
6	**if**($C_x = \{\} \vee 0 > y_x$){	max{
7	$\mathit{high} \leftarrow \max\{\frac{d}{l} \mid \frac{d}{l} < x \wedge 0 < d \leq D \wedge 0 < l \leq L\}$;	$\min\{D, L\}$
8	}**else if**($0 < y_x$){	,
9	$\mathit{low} \leftarrow \min\{\frac{d}{l} \mid \frac{d}{l} > x \wedge 0 < d \leq D \wedge 0 < l \leq L\}$;	$\min\{D, L\}$
10	}	}
11	}**while**($C_x = \{\} \vee 0 \neq y_x$);	$) \cdot \log(DL)+$
12	**return** C_x;	1

Figure 3: Algorithm for *flow-based* chooser.

The do-while-loop in lines 3 to 11 repeatedly solves max-weight closed set problems for values of x in the search space, and uses the solution y_x to either narrow the range $[\mathit{low}, \mathit{high}]$ or to determine that it has found the solution to the CBGC partition selection problem. Line 4 chooses an x that is a possible solution between *low* and *high* such that the search space is halved (see Section 2.1.3). Line 5 solves the max-weight closed set problem for the weight function $w = w_x$ (see Section 2.3), finding the maximum weight y_x and a witness set of partitions C_x that has the maximum weight. If C_x is empty or $0 > y_x$, then x was too large and line 7 sets *high* to a possible solution value $\frac{d}{l}$ just below x. If $0 < y_x$, then x was too small and line 9 sets *low* to a possible solution value $\frac{d}{l}$ just above x. When the algorithm has found a non-empty set C_x that maximizes $\sum_{p \in C_x} w_x(p)$, then according to Lemmas 1 and 2 the set C_x is also a solution to the CBGC partition selection problem, and the algorithm terminates.

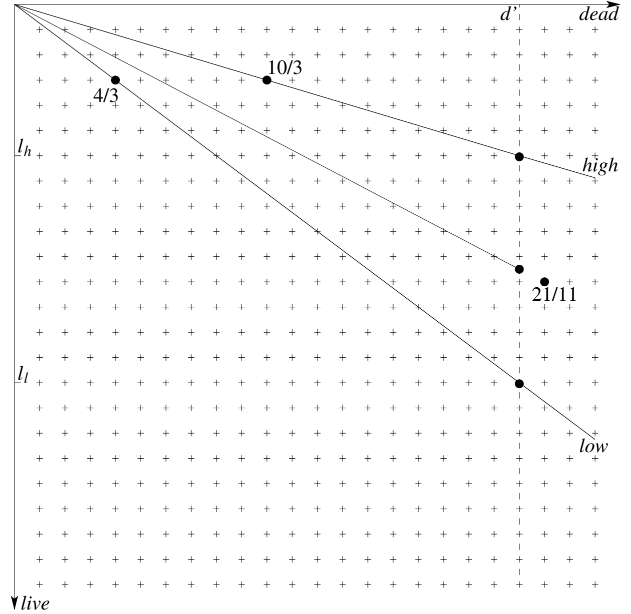

Figure 4: Solution space.

2.1.3 Search space halving

The search space for solutions to the CBGC partition selection problem is $\{d/l \mid 0 < d \leq D \wedge 0 < l \leq L\}$. Figure 4 visualizes the search space for $D = L = 23$ as a two-dimensional grid of numbers.

Each number in the search space corresponds to a ray originating in $(0, 0)$. Figure 4 shows the rays corresponding to $\mathit{low} = \frac{4}{3}$ and $\mathit{high} = \frac{10}{3}$. The solution x_M must be one of the fractions in the area between the rays.

Without loss of generality we assume that $\frac{D}{L} \leq \mathit{low}$, in other words, both rays *low* and *high* are above the diagonal of the rectangular solution grid. We pick $d' = \mathrm{lcm}(\mathit{low.dead}, \mathit{high.dead})$; in Figure 4, we have $d' = \mathrm{lcm}(4, 10) = 20$. The rays *low* and *high* intersect the line $\mathit{dead} = d'$ at $l_l = \mathit{low.live} \cdot d'/\mathit{low.dead}$ and $l_h = \mathit{high.live} \cdot d'/\mathit{high.dead}$.

We are looking for a point that halves the area of the triangle $((0,0),(d',l_l),(d',l_h))$. Let A_l be the area of the triangle $((0,0),(d',l_l),(d',0))$, which is $\frac{1}{2}(d' \cdot l_l)$, and let A_h be the area of the triangle $((0,0),(d',l_h),(d',0))$, which is $\frac{1}{2}(d' \cdot l_h)$. We are looking for a point $x' = (d', l_x)$ such that $A_x = \frac{1}{2}(d' \cdot l_x) = \frac{1}{2}(A_l + A_h)$. It is easy to see that $l_x = \frac{1}{2}(l_l + l_h)$.

Line 4 of the algorithm in Figure 3 finds an $x' =$

d'/l_x as described above, and then rounds it to the closest $x = d/l$ that corresponds to a point in the grid, i.e. that satisfies $0 < d \leq D$ and $0 < l \leq L$. For example, for $low = \frac{4}{3}$ and $high = \frac{10}{3}$, we have $x' = \frac{10.5}{20}$, and the closest legal x is $\frac{11}{21}$.

2.2 Flow networks

Before we look at max-weight closed sets in Section 2.3, let us review some folklore on flow networks, as described e.g. in [3] chapter 27.

A *flow network* consists of a set V of vertices, with two special vertices $s, t \in V$, the source s and the sink t, and a capacity function $c : V \times V \to \mathbb{R}^+$. It can be represented as a graph with edges $E = \{(p, q) \in V \times V \mid c(p, q) > 0\}$.

A *flow* is a function $f : V \times V \to \mathbb{R}$ on pairs of vertices in a flow network that satisfies the three flow properties

- Capacity constraint: for all $p, q \in V$, we require $f(p, q) \leq c(p, q)$.
- Skew symmetry: for all $p, q \in V$, we require $f(p, q) = -f(q, p)$.
- Flow conservation: for all $p \in V \setminus \{s, t\}$, we require $\sum_{q \in V} f(p, q) = 0$.

The value $\mathit{value}(f)$ of a flow is the total flow coming out of the source, in other words $\mathit{value}(f) = \sum_{q \in V} f(s, q)$. A max-flow is a flow of maximum value for its network. A flow defines a residual capacity function $c_f(p, q) = c(p, q) - f(p, q)$ on pairs of vertices.

A cut is a partition (S, T) of the vertices of a flow network into a source side $S \subset V$ with $s \in S$ and a sink side $T = V \setminus S$ with $t \in T$. The capacity $c(S, T)$ of a cut is the total capacity of the edges that cross the partition, in other words, $c(S, T) = \sum_{p \in S, q \in T} c(p, q)$. A min-cut is a cut of minimum capacity for its network.

There is a duality between max-flows and min-cuts.

Theorem 1 (Max-flow min-cut.)
$\max_{f\ \mathit{flow}} \{\mathit{value}(f)\} = \min_{(S,T)\ \mathit{cut}} \{c(S, T)\}$

Proof. See [3] page 593. This theorem is also constructive: given a max-flow, you can find a min-cut by choosing all vertices reachable from s via edges with positive residual capacities as S. □

The max-flow problem is to find a max-flow along with a witness. The fairly straight-forward lift-to-front algorithm from [3] page 621 solves it in $O(V^3)$. The Goldberg-Tarjan algorithm is an extension of the lift-to-front algorithm that solves the problem even faster, namely in $O(VE \log(V^2/E))$ [4].

2.3 Max-weight closed sets

Max-weight closed set problems and their solution using network flow are described on pages 719-721 of [1]. We modify the problem slightly by solving for sets that are closed under the predecessor relation, instead of closed under the successor relation as in [1].

A max-weight closed set problem consists of a partial order $(P, <)$ and a weight function $w : P \to \mathbb{R}$ on elements of P. The goal is to find a closed set $C \subseteq P$ with maximum weight $\sum_{p \in C} w(p)$. Here, a set C is closed if for all $q \in C$, we have $p < q \Rightarrow p \in C$. Note that this definition differs from [1].

For example, the partial order in Figure 2 together with the weight function in Table 2 defines a max-weight closed set problem.

The first step is to construct a flow network from the max-weight closed set problem. We partition the elements of P into $P^+ = \{p \in P \mid w(p) \geq 0\}$ and $P^- = \{p \in P \mid w(p) < 0\}$. For the vertices V of the flow problem, we choose two additional vertices s, t as source and sink and set $V = P^+ \cup P^- \cup \{s, t\}$. The capacity function $c : V \times V \to \mathbb{R}^+ \cup \infty$ is defined as

- $c(s, p) = w(p)$ for each $p \in P^+$
- $c(p, t) = -w(p)$ for each $p \in P^-$
- $c(q, p) = \infty$ whenever $p < q$ (this differs from [1], since the sets we are interested in are closed under the predecessor relation)

For example, the weights from Table 2 partition $P = \{p_1, p_2, p_3, p_4, p_5\}$ into $P^+ = \{p_4, p_5\}$ and $P^- = \{p_1, p_2, p_3\}$, yielding the flow network in Figure 5.

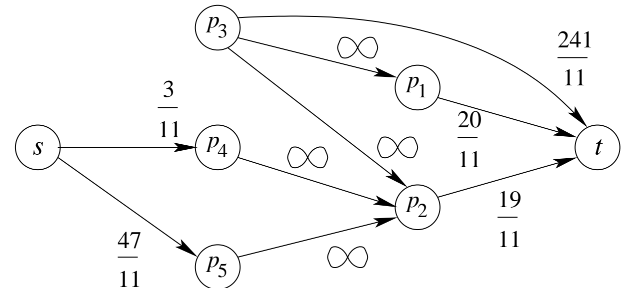

Figure 5: Example flow network.

The second step is to find a min-cut (S, T) in the flow network. The set $S \setminus \{s\}$ is a max-weight closed set. To see why this is true, we need a few lemmas.

Lemma 3 *A min-cut in a network constructed by the first step above has finite capacity.*

Proof. The cut $(\{s\}, V \setminus \{s\})$ has finite capacity, since there are no vertices $p \in V$ with $c(s,p) = \infty$. The capacity of a min-cut is less than or equal to the capacity of $(\{s\}, V \setminus \{s\})$. Hence, the capacity of a min-cut is finite. □

Lemma 4 *A subset $C \subseteq P$ of the partial order is closed if and only if $\{s\} \cup C$ gives a cut with finite capacity.*

Proof.

$$\begin{aligned} C \text{ is closed} &\Leftrightarrow (q \in C \wedge p < q) \Rightarrow p \in C \\ &\Leftrightarrow q \notin C \vee \neg(p < q) \vee p \in C \\ &\Leftrightarrow (q \in C \wedge p \notin C) \Rightarrow \neg(p < q) \\ &\Leftrightarrow (q \in C \wedge p \notin C) \Rightarrow c(q,p) < \infty \\ &\Leftrightarrow c(\{s\} \cup C, \{t\} \cup (P \setminus C)) < \infty \end{aligned}$$

□

Lemma 5 *If $c(S,T)$ is finite, then*

$$c(S,T) = \sum_{p \in P^+} w(p) - \sum_{p \in (S \setminus \{s\})} w(p).$$

Proof. Being finite, the cut (S,T) can only cross edges involving s or t. More precisely, the capacity is the total capacity of edges going from s to T, which is $\sum_{p \in T} c(s,p)$, plus the total capacity of edges going from S to t, which is $\sum_{p \in S} c(p,t)$. We can now easily see the lemma:

$$\begin{aligned} &c(S,T) \\ &= \sum_{p \in T} c(s,p) + \sum_{p \in S} c(p,t) \\ &= \sum_{p \in (P^+ \setminus (P^+ \cap S))} c(s,p) + \sum_{p \in (P^- \cap S)} c(p,t) \\ &= \sum_{p \in P^+} c(s,p) - \sum_{p \in (P^+ \cap S)} c(s,p) + \sum_{p \in (P^- \cap S)} c(p,t) \\ &= \sum_{p \in P^+} w(p) - \sum_{p \in (P^+ \cap S)} w(p) + \sum_{p \in (P^- \cap S)} (-w(p)) \\ &= \sum_{p \in P^+} w(p) - \sum_{p \in (S \setminus \{s\})} w(p) \end{aligned}$$

□

Now we can show that the reduction from a max-weight closed set problem to a min-cut problem worked.

Theorem 2 *If (S,T) is a min-cut of the flow network, then $S \setminus \{s\}$ is a max-weight closed set of the partial order $(P, <)$.*

Proof. A min-cut minimizes $c(S,T)$, and according to Lemma 5 that is equivalent to minimizing $\sum_{p \in P^+} w(p) - \sum_{p \in (S \setminus \{s\})} w(p)$. Since $\sum_{p \in P^+} w(p)$ is constant, that means that the min-cut maximizes $\sum_{p \in (S \setminus \{s\})} w(p)$. Lemma 4 states that we have indeed maximized over all closed sets. □

When there are multiple min-cuts (S,T), we choose the one with the largest S using a depth-first search starting at t.

Figure 6 shows a max-flow in the flow network from Figure 5. A min cut is $S = \{s, p_2, p_4, p_5\}, T = \{p_1, p_3, t\}$, since the residual capacities $c_f(p_2,p_3) = c_f(p_2,t) = 0$. (Note that $c_f(p_3,p_2) \neq 0$, but that is the wrong direction). The max-weight closed set is therefore $\{p_2, p_4, p_5\}$ with a weight of $-\frac{19}{11} + \frac{3}{11} + \frac{47}{11} = \frac{31}{11}$.

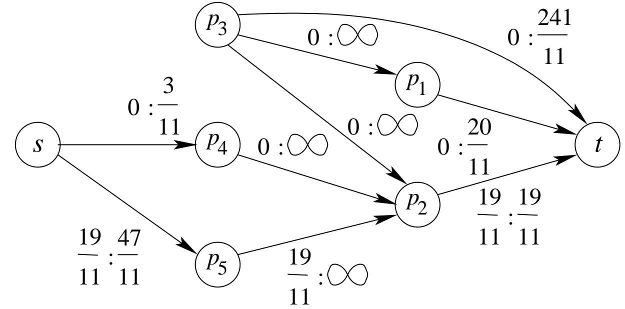

Figure 6: Max-flow in network from Figure 5.

3 The greedy chooser

The *flow-based* chooser from Section 2 finds an optimal solution to the CBGC partition selection problem from Section 1. We also developed the *greedy* chooser, which is much simpler, but may not find an optimal solution.

3.1 Greedy chooser algorithm

The *greedy* chooser works as follows [6]:

1. Initialize $C \leftarrow \emptyset$.

2. For each partition q, let $A(q) = \{p \in P \setminus C \mid p \rightarrow^* q\}$ be the set that contains all ancestors of q that have not yet been chosen. Since the ancestor relation is reflexive, q is an ancestor of itself.

3. Find the partition p with the highest *quality*$(A(p))$.

4. If $dead(C)$ is not yet enough to satisfy the current allocation request, or if $quality(C) < quality(C \cup A(q))$,
   (a) then $C \leftarrow C \cup A(p)$, and go back to Step 2,
   (b) else return the choice C.

3.2 The greedy chooser is not optimal

Figure 7 shows an example of a partition dag annotated with the estimated *dead* and *live* functions. For this example, the *greedy* chooser will not find the optimal solution.

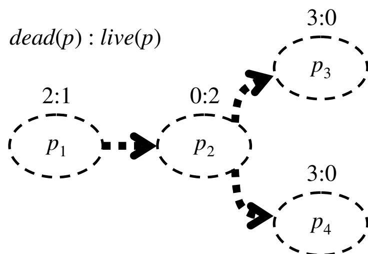

Figure 7: Example where the *greedy* chooser is not optimal.

Table 3 shows all closed subsets of the partition dag in Figure 7 and their quality. The *greedy* chooser would start by choosing $C_1 = \{p_1\}$, which has a quality of $2/1 = 2.00$. Then, it would consider adding another partition with its ancestor set. But all of the three possibilities $C_2 = \{p_1, p_2\}$, $C_3 = \{p_1, p_2, p_3\}$, or $C_4 = \{p_1, p_2, p_4\}$ have worse qualities than what has already been chosen. The only better choice, which the *flow-based* chooser would find, is $C_{\text{opt}} = \{p_1, p_2, p_3, p_4\}$ with a quality of 2.67, but the *greedy* chooser does not consider it, since it involves adding more than the ancestor set of a single partition.

Table 3: Qualities of closed subsets of the partition dag in Figure 7.

Closed set $C \subseteq P$	Quality
p_1 p_2	$2/3 = 0.67$
(empty set)	$0/0 =$ undefined
p_1 p_2 p_3	$5/3 = 1.67$
p_1 p_2 p_4	$5/3 = 1.67$
p_1	$2/1 = 2.00$
p_1 p_2 p_3 p_4	$8/3 = 2.67$

4 Results

We generated traces of Java benchmarks executing on Jikes RVM (formerly called Jalapeño [2]). These traces list all object allocations and pointer writes, as well as the precise times when objects become unreachable [5]. Using these traces, we simulate various garbage collectors in our GC simulator gcSim. Among other things, gcSim includes simple implementations of the flow-based and greedy choosers in Java. The traces and the source code for gcSim are available at `http://www.cs.colorado.edu/~hirzel/gcSim`. For details on our methodology see [6].

Table 4 shows our results. Here is how to read it:

Programs: There is one row for each of our benchmarks. They are sorted by their total allocation in bytes, which gives an indication of their size. The program *power* allocated the fewest bytes (70MB), the program *javac* allocated the most bytes (553MB).

GC work per time: This metric indicates the cost in time of garbage collection. The unit is (bytes copied / bytes allocated).

Maximum footprint: This metric indicates the cost in space of garbage collection. The unit is (maximum footprint / heap size in bytes).

Average and maximum work per GC: These metrics indicate the pause times for garbage collections. The unit is (bytes copied at current GC / heap size in bytes).

Average time choose: This metric is the average wall clock time for running the Java implementations of the choosers in gcSim, on a 1.4GHz Pentium 4 with 1GB of RAM. The unit is seconds.

AOF and AOG: These are abbreviations for the CBGC configurations. AOF uses the flow-based chooser, and AOG uses the greedy chooser. Both use the allocsite-dynamic partitioning and oracle estimator described in [6].

The metrics "GC work per time", "maximum footprint", and "average and maximum work per GC" evaluate how the quality of the chooser affects overall GC performance. If the chooser makes good choices, the numbers are lower; if it makes bad choices, they are higher. We see that in almost all cases, the flow-based and the greedy chooser make choices of identical quality. The only exceptions occur for the programs *ipsixql*, *xalan*, and *pseudojbb*. We did not investigate the cause for these variations. For *ipsixql*, the greedy chooser turns out to improve overall GC performance. At first glance, this seems to contradict

Table 4: Experimental results.

Program	GC Work per Time		Maximum Footprint		Work per GC				Average Time Choose	
					Average		Maximum			
	AOF	AOG	AOF	AOG	AOF	AOG	AOF	AOG	AOF	AOG
power	0.000	0.000	0.500	0.500	0.000	0.000	0.000	0.000	0.11s	0.13s
deltablue	0.000	0.000	0.500	0.500	0.000	0.000	0.000	0.000	1.33s	0.10s
bh	0.000	0.000	0.500	0.500	0.000	0.000	0.000	0.000	0.19s	0.11s
health	0.000	0.000	0.500	0.500	0.000	0.000	0.000	0.000	0.62s	0.11s
db	0.000	0.000	0.500	0.500	0.000	0.000	0.000	0.000	0.34s	0.10s
compress	0.000	0.000	0.498	0.498	0.000	0.000	0.000	0.000	3.97s	0.04s
mtrt	0.057	0.057	0.530	0.530	0.000	0.000	0.037	0.037	2.14s	0.12s
ipsixql	0.151	0.137	0.532	0.518	0.000	0.000	0.054	0.049	4.81s	0.13s
jess	0.236	0.236	0.538	0.538	0.001	0.001	0.046	0.046	6.69s	0.18s
jack	0.092	0.092	0.537	0.537	0.000	0.000	0.047	0.047	3.03s	0.15s
xalan	0.333	0.328	0.536	0.561	0.001	0.001	0.045	0.130	12.62s	0.32s
pseudojbb	0.239	0.240	0.550	0.549	0.000	0.000	0.081	0.081	4.25s	0.18s
javac	0.234	0.234	0.564	0.564	0.000	0.000	0.076	0.076	5.02s	0.24s

the optimality of the flow-based chooser; but it can happen if the choosers make different choices of similar quality that lead to different fragmentation effects down the line.

The wall-clock times have to be taken with a grain of salt, since we put little effort into optimizing the choosers. The flow-based chooser implemented in gc-Sim uses the cubic lift-to-front flow algorithm described in [3], but in the literature, there are algorithms with better complexity (e.g. [4]). The greedy chooser is not very efficient either, it builds up auxiliary data structure for every choice that could be cached in a better implementation.

Table 4 shows that the wall-clock time of the flow-based chooser is quite high. In fact, given that a GC should take less than around 0.3 seconds to be imperceptible by the user, we can probably not afford it, even if we can optimize it by an order of magnitude. The greedy chooser performs much better, and with a little more optimization, its choice times will be acceptable.

5 Conclusions

We looked at two choosers for CBGC: flow-based and greedy. In theory, the flow-based chooser is optimal, while the greedy chooser can make sub-optimal choices. In practice, the quality of the two choosers is usually almost the same. From a prototype implementation, the flow-based chooser appears to be too slow, whereas the greedy chooser is probably usable. Nevertheless, the flow-based chooser was important for our experiments; at the very least, it helped demonstrate that the greedy chooser makes good choices in practice.

References

[1] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT press, 1990.

[4] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, pages 921–940, 1988.

[5] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error-free garbage collection traces: How to cheat and not get caught. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2002.

[6] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.