A Brouwerian Model of the Run-Time Memory*,+

WUU YANG
Department of Computer Science
National Chiao-Tung University
Hsinchu, 300 Taiwan
E-mail: wuuyang@cs.nctu.edu.tw

The run-time memory of a program may be described with a directed graph in which nodes represent chunks of memory and edges represent references. We define a closed cluster induced by a node n, denoted as CC(n), as the largest set of nodes that are reachable from n but are unreachable from nodes outside the closed cluster. Based on closed clusters, there is a Brouwerian structure under the run-time memory. We present the Brouwerian model and discuss its properties, transformations, and applications. We also propose a two-counter algorithm for calculating CC(n). The two-counter algorithm is never slower than a traditional one-counter algorithm. Our study of the Brouwerian structure is motivated by work on garbage collection.

Keywords: Brouwerian algebra, closed cluster, cyclic structure, depth-first search, graph theory, garbage collection, reference count, run-time memory, strongly connected component

1. INTRODUCTION

The run-time memory of a program consists of global variables, the runtime stack, and the dynamic allocation heap, with pointers pointing to one another. Pointers can be created and destroyed freely and hence arbitrary linked structures can be built up. Even with this much freedom, we can still identify a Brouwerian structure under the run-time memory. The Brouwerian structure provides an insight into the dynamic allocation heap. The insight helps designing memory-management algorithms.

The run-time memory of a program may be described with a directed graph in which nodes represent chunks of memory and edges represent references. The Brouwerian model is based on closed clusters in a directed graph. We define the closed cluster induced by a node n, denoted as CC(n), as the largest set of nodes that are reachable from n but are unreachable from nodes outside the closed cluster. The run-time memory is structured as various overlapping closed clusters. These closed clusters, together with appropriate operations, form a Brouwerian algebra [21, 24, 25], which is a distributive lattice with a pseudo-difference operation, x ⊕ y, characterized by the assertion that z ≥ x ⊕ y if and only if z v y ≥ x. Every memory operation (creating and destroying a reference) transforms one Brouwerian structure to another. The Brouwerian model allows to study the dynamic behavior of the memory system.

There are three applications of the Brouwerian model: (1) it helps in determining whether a garbage-collection algorithm will suffer from the memory-leak problem; (2) it can guide the memory-allocation algorithms; and (3) it can guide the compaction opera-
A garbage collector is called complete if it eventually reclaims all garbage nodes [22].

Our study of the Brouwerian model is motivated by work on garbage collection. Reference counting is a major technique for garbage collection. A problem with reference counting is that cyclic garbage is difficult to reclaim. In addition to examining nodes’ counters, (a part of) the virtual memory still needs to be scanned separately in order to identify cyclic garbage.

There are two issues related to scanning the virtual memory: when a scan operation should be triggered and which portion of the virtual memory should be scanned. The two issues will affect the overall efficiency of the computer system.

In algorithms based on reference counting [5, 7, 14, 19], a garbage-collection decision has to be made whenever a reference $x \rightarrow y$ is about to be destroyed. At this time, the node $y$ may become dead even if $y$’s reference count is not zero [18, 20, 22]. This is because $y$ may belong to a piece of cyclic garbage. Some aggressive collection algorithms will put $y$ on the list of potential garbage regardless of $y$’s reference count. Later a trace procedure starting from $y$ will be initiated. Other, less aggressive, algorithms will put $y$ on the list of potential garbage only if $y$’s reference count falls below a threshold. The former approach may waste time on tracing live nodes too eagerly and the latter may delay collecting cyclic garbage due to its laziness and may create memory leaks.

The problem with the above two approaches (and with reference counting in general) is that it is difficult to decide whether $y$ is dead when the reference $x \rightarrow y$ is destroyed. We propose a new collection algorithm in which each node maintains two, rather than one, reference counters, called $gcount$ and $hcount$, respectively. $gcount$ is the number of references from the global variables and from the run-time stack. $hcount$ is the number of references from the heap. Our algorithm will put a heap node $y$ on the list of potential garbage if and only if $y$’s $gcount$ becomes 0. The more precise prediction made by our algorithm results in the more efficient garbage collectors.

Distinguishing global pointers and heap pointers does not incur extra runtime overhead. Since a counter must be incremented or decremented when a reference is created or destroyed, all we need is a more careful compiler that can distinguish global and heap pointers.

Our two-counter algorithm is locally complete in that it can reclaim all the garbage that can be identified if a garbage collector is limited to examine only the nodes that are reachable from certain given nodes. Our algorithm will not cause a long interrupt to the normal computer operation since it examines a very limited portion of a program’s run-time memory. Many such partial-scan algorithms focus on cyclic garbage, which often make use of some cycle-detection techniques [8, 14, 17, 20]. In contrast, our algorithm looks for closed clusters.

The rest of this paper is organized as follows: Section 2 is a survey of related work. Section 3 gives a bird’s eye view of the memory during run time and several properties of closed clusters. The Brouwerian model is described in Section 4. An algorithm to calculate the closed clusters and an example are shown in Section 5. Section 6 gives a conclusion.

2. RELATED WORK

Brouwerian models based on graphs have been studied by Reps [24, 25], in which

---

1 A garbage collector is called complete if it eventually reclaims all garbage nodes [22].
the graph represents (data and control) dependence among the program elements. His model attempts to identify the transitive closure of (data and control) dependence on the dependence graphs, called a slice of the program. Slices are used in the study of changes to programs. The closed cluster in a directed graph is also a kind of transitive closure on the memory graphs (to be defined later). This transitive-closure property is essential in the transition from one Brouwerian structure to the next.

Garbage collection is an important topic in programming systems and compilers [10]. It has been studied intensively in the past. Garbage-collection algorithms can be classified into two broad categories: (1) some algorithms mark all live nodes and consider the rest as dead (i.e., garbage) and (2) others attempt to identify dead nodes directly. Traditional mark-sweep-compact collectors [10] belong to the first category. A naive implementation suffers from the long delays to normal computer operations because the entire virtual memory must be examined. Given today’s larger and larger virtual memory, the interrupts become less and less tolerable.

Algorithms in the second category make use of other information, mostly reference counts of various kinds, to identify dead nodes directly [7, 10, 14]. Usually, there is a counter in every node in the heap which keeps the number of references that point to that node. A node dies when its counter falls to zero.

Ben-Ari [4] proposed an on-the-fly garbage collection algorithm, which makes use of two colors. The algorithm comes with a concise but rigorous correctness proof. A variation of this algorithm is an incremental collection algorithm which may collect a few nodes, instead of a whole cycle, very rapidly. Incremental garbage collection is very useful for systems that cannot tolerate long interrupts to normal operation [9]. Jonker [16] proposed a finer-grained algorithm that makes use of only a few colors and has less overhead for the mutators. Many garbage collection algorithms mark nodes in various ways during pointer tracing [14, 15]. Some garbage collection algorithms [12, 13, 23] are able to operate in a multiprocessor or distributed environment. Ungar proposed a garbage-collection algorithm based on generations [27]. Two-space algorithm [6] divides the heap into two spaces, only one of which is active at any instant. Useful data is copied from one space to the other during garbage collection. These algorithms do not make use of two counters and do not explicitly characterize the structure of the run-time memory. Verification of garbage collection algorithms and implementations [11] is also an interesting research topic. An algebraic structure of the run-time memory reported in this paper should help in such verification.

Lins [18] extends Martinez et al.’s work [20] by searching for cyclic garbage lazily. Redundant local searches are eliminated. Bacon et al. further incorporate concurrent search for multiprocessor systems [1, 2]. It would be interesting to investigate a way to parallelize our two-counter algorithm. Garbage collection is also an important issue in real-time systems [3].

3. A BIRD’S EYE VIEW OF THE RUN-TIME MEMORY

In this paper, we assume that the computer run-time memory is partitioned into two areas: the global area, which contains global variables and the run-time stack, and the dynamic allocation heap. The global area and the heap consist of various nodes. All
nodes that are reachable from the global area are considered live nodes. The rest are dead (i.e., garbage). All dead nodes are on the heap. A reference stored in the global area is called a global pointer. All other references are called heap pointers.

For the sake of simplicity for presenting the Brouwerian structure, we assume that no reference points to a node in the global area. In a conventional programming language, as long as reference parameters and the & (address-of) operator are disallowed, there is no way to figure out the address of a variable in the global area. Then there will be no references pointing to variables in the global area. For example, a program written in the C language satisfies this assumption as long as the & (address-of) operator is not applied to variables in the global area. (There is no reference parameter in C.)

3.1 Closed Clusters

We may use a directed graph to represent the memory of a running program. Each node in the memory is represented by a node in the graph. If a node \( n \) contains a reference (i.e., a pointer) that points to node \( m \), this reference is represented as an edge \( n \rightarrow m \) in the graph.

Fig. 1. Overview of the run-time memory.

Fig. 1 shows a snapshot of a program’s run-time memory at a certain instant. There are three nodes \( A, B, \) and \( C \) in the global area. The \( D \) nodes are floating garbage that have not been collected yet. The nodes that are reachable from \( A \) are classified into the following categories:

1. The \( P \) nodes are reachable only from \( A \) (and other \( P \) nodes) but not from any other nodes in the global area and the heap. When the reference in \( A \) is destroyed, all \( P \) nodes become dead.
2. The \( Q \) nodes are reachable from \( A \) and some \( D \) node in the heap. The \( Q \) nodes are unreachable from any other nodes in the global area. If the \( D \) node has already been reclaimed, the \( Q \) nodes are really \( P \) nodes. However, since most dead nodes are not re-

\(^2\) For the purpose of garbage collection, we may ignore references pointing to nodes in the global area.
claimed immediately when they become garbage, the \( Q \) nodes may remain uncollected for some time.

3. The \( R \) nodes are reachable only from \( A \) and \( B \) but not from any other nodes in the global area. After references in both \( A \) and \( B \) are destroyed, all \( R \) nodes will become dead.

4. The \( S \) nodes are reachable only from \( A \) and \( C \) but not from any other nodes in the global area.

5. The \( T \) nodes are reachable from \( A, B, \) and \( C \).

In the Brouwerian model, we assume all \textit{dead} nodes are reclaimed immediately when they die. Therefore, there are no \( D \) nor \( Q \) nodes.

Our two-counter algorithm performs at most three (possibly partial) depth-first traversals, starting from node \( A \), in order to identify all the \( P \) nodes. The \( P \) nodes constitute the closed cluster induced by node \( A \), which is defined as follows.

**Definition:** The \textit{closed cluster induced by a node} \( n \), denoted as \( \text{CC}(n) \), in a directed graph is the largest set of nodes that are reachable from \( n \) but are unreachable from any node outside the closed cluster.

We adopt the convention that a node can reach itself. The above definition of closed clusters is for a single node. We may extend the definition to a set of nodes, as follows:

**Definition:** Let \( x \) be a set of nodes. The \textit{closed cluster induced by} \( x \), denoted as \( \text{CC}(x) \), in a directed graph is the largest set of nodes that are reachable from at least one node in \( x \) but are not reachable from any node outside the closed cluster. By convention \( \text{CC}(\emptyset) = \emptyset \).

Note that \( \text{CC}(x) \cup \text{CC}(y) \subseteq \text{CC}(x \cup y) \). In general, if there are \( n \) references \( p_1, p_2, \ldots, p_n \) in the global area, the heap will be structured as \( 2^n \) non-disjoint clusters (assuming all \textit{dead} nodes have already been reclaimed), each corresponding to a subset of \( \{p_1, p_2, \ldots, p_n\} \). A node belongs to a cluster corresponding to the subset \( \{q_1, q_2, \ldots, q_n\} \) if and only if it is reachable from at least one of \( q_1, q_2, \ldots, q_n \) but is not reachable from any node in \( \{p_1, p_2, \ldots, p_n\} \setminus \{q_1, q_2, \ldots, q_n\} \).

### 3.2 Properties of Closed Clusters

A closed cluster is similar to a strongly connected component. They share many properties. In what follows, we use the notation \( a \rightarrow^* b \) to denote that node \( a \) can reach node \( b \) through zero or more edges and the notation \( a \nrightarrow^* b \) to denote that node \( a \) cannot reach node \( b \). By convention, \( a \rightarrow^* a \), for every node \( a \). The following assertions can be deduced from the definition of a closed cluster.

**Lemma 1:** A closed cluster has no incoming edge from outside the closed cluster.

**Lemma 2:** Let \( a \) and \( b \) be two nodes. Assume \( a \rightarrow^* b \). If \( b \rightarrow^* a \), then \( \text{CC}(b) = \text{CC}(a) \). If \( b \nrightarrow^* a \), then \( \text{CC}(b) = \emptyset \).
Corollary 3: For every node \( a \), either \( a \in CC(a) \) or \( CC(a) = \emptyset \).

Corollary 4: Consider a cycle \( a_1 \to a_2 \to a_3 \to \ldots \to a_k \to a_1 \). Then \( CC(a_i) = CC(a_j) \), for \( i, j = 1, 2, \ldots, k \).

Corollary 5: Let nodes \( a_1, a_2, a_3, \ldots, a_k \) form a strongly connected component. Then \( CC(a_i) = CC(a_j) \), for \( i, j = 1, 2, \ldots, k \).

A directed graph may contain several disjoint maximal strongly connected components. Garbage-collection algorithms that focus on identifying strongly connected components may need to handle more than one maximal strongly connected component in a run. In contrast, a directed graph is divided into non-disjoint closed clusters. Each cluster contains one or more disjoint maximal strongly connected components that are connected together. Each run of garbage collection reclaims exactly one closed cluster. It is more reasonable to design garbage-collection algorithms based on closed clusters.

Lemma 6: Assume \( a \to^* c \) and \( b \to^* c \) but either \( a \not\to^* b \) or \( b \not\to^* a \). Then \( CC(c) = \emptyset \).

Proof: If \( a \to^* b \), then \( c \to^* b \). Because \( b \to^* c \), \( CC(c) = \emptyset \) by Lemma 2. A similar argument applies if \( b \to^* a \). \( \square \)

Lemma 7: Let \( a \in CC(b) \). Then (1) \( b \to^* a \); (2) either \( CC(a) = CC(b) \) (if \( a \to^* b \)) or \( CC(a) = \emptyset \) (if \( a \not\to^* b \)).

Lemma 8: Let \( a \) and \( b \) be two nodes. If \( b \) has only one incoming edge \( a \to b \) then \( b \) is in every closed cluster that includes \( a \).

Lemma 9: Assume \( a \in CC(b) \). Then \( CC(\{a, b\}) = CC(b) \).

Corollary 10: Assume \( a \in CC(b) \) and \( b \in CC(a) \). Then \( CC(a) = CC(b) \).

Fig. 2. (a) Several closed clusters. Each ellipse represents a closed cluster; (b) The collapsed graph; The big dots in (b) denote elementary closed clusters.

Since a closed cluster is also a set of nodes, one can think of calculating the closed cluster of this set. The following lemma shows that the closed cluster is a kind of transitive closure.

Lemma 11: Let \( x \) be a set of nodes. Then \( CC(CC(x)) = CC(x) \).

The closed cluster induced by a single node \( CC(a) \) is called an elementary closed
cluster. Different elementary closed clusters are either disjoint or identical. No node in an elementary closed cluster is reachable from any node in another elementary closed cluster unless the two clusters are identical. Fig. 2 (a) shows three elementary closed clusters (CC(a), CC(b), and CC(c)) and two non-elementary closed clusters CC({a, b}) and CC({a, b, c}).

A non-elementary closed cluster is induced by more than one node. There are two cases for a non-elementary closed cluster CC({a, b}):

- CC({a, b}) = CC(a) ∪ CC(b). In this case, there is no node that is reachable from both a and b.
- CC({a, b}) ≠ CC(a) ∪ CC(b). In this case, there are nodes that are reachable from both a and b. (Note that it is always true that CC({a, b}) ⊆ CC(a) ∪ CC(b).)

We may collapse an (elementary or non-elementary) closed cluster into a single node. The resulting collapsed graph is a directed acyclic graph, in which the supernode (that is, the node that represents a collapsed closed cluster) has no incoming edges. Fig. 2 (b) is obtained from Fig. 2 (a) by collapsing the three elementary closed clusters CC(a), CC(b), and CC(c). The three big dots denote supernodes, which represent the three elementary closed clusters, respectively. In the collapsed graph (e.g., Fig. 2 (b)), the elementary closed cluster induced by a supernode contains exactly the supernode itself.

4. A BROUWERIAN MODEL OF THE RUN-TIME MEMORY

We define a memory graph \(G = (V, E)\), where \(V = V_g \cup V_h\), as a directed graph satisfying the following requirements. \(V_g\) and \(V_h\) are disjoint sets of nodes. \(V_g\) is the set of nodes in the global area. \(V_h\) is the set of nodes in the dynamic allocation heap. An edge \(a \rightarrow b\) denotes a reference stored in node a that points to node b. Every node in \(V_h\) contains exactly one reference while a node in \(V_g\) contains zero or more references. Every node in \(V_h\) is reachable from at least one node in \(V_g\) (this implies that there is no garbage in the heap). All edges only point to nodes in \(V_h\).

Let \(G = (V_g \cup V_h, E)\) be a memory graph. Let \(x \subseteq V_g\). The closed cluster induced by x in G is defined as follows: \(CCG(G, x) =_{def} CC(x)\) in the memory graph G. If x contains exactly one node, say n, then \(CCG(G, \{n\}) =_{def} CC(n)\) in graph G. Note that \(CCG(G, \phi) =_{def} CC(\phi) = \phi\). Fig. 3 shows several closed clusters for the memory graph in Fig. 1. Fig. 4 shows that the run-time memory is organized as various closed clusters.

Given a memory graph G, for every set of nodes \(x \subseteq V_g\), we use \(\bar{x}\) to denote \(CCG(G, x)\). Define a set \(2^{V_g} =_{def} \{x | x \subseteq 2^{V_g}\}\) (that is, \(x \subseteq V_g\)) and four operations \(\leq, \lor, \land, \oslash\) on \(2^{V_g}\) as follows:

1. \(\bar{x} \leq \bar{y}\) if and only if \(\bar{x} \subseteq \bar{y}\).
   
   Note that the operator \(\leq\) is reflexive, transitive, and anti-symmetric (according to Lemma 13 below) and, thus, is a partial order on the set \(2^{V_g}\).

2. \(\bar{x} \lor \bar{y} =_{def} \bar{x \cup y}\).
   
   Intuitively, \(\bar{x} \lor \bar{y}\) means the closed cluster with respect to \(x \cup y\).

3. \(\bar{x} \land \bar{y} =_{def} \bar{x \cap y}\).
4. \( \bar{x} \odot \bar{y} \equiv_{def} x - y \).

We may verify the following claims:

**Lemma 12:** For all \( x \subseteq V_g, x \subseteq \bar{x} \). Furthermore, for all \( x \subseteq V_g \) and \( d \in V_g - x, d \notin \bar{x} \).

**Proof:** Note that we assume that there are no references that point to global variables and the run-time stack.

**Lemma 13:** Assume \( x, y \subseteq V_g \), \( x \subseteq y \) if and only if \( \bar{x} \subseteq \bar{y} \).

**Lemma 14 (Closure property):** Assume \( x \subseteq V_g \). Assume \( d \in \bar{x} \). For very node \( p \in V_g \), if there is a path \( p \rightarrow^* d \), then \( p \in x \). Actually, all nodes on the path \( p \rightarrow^* d \) belong to \( \bar{x} \).

**Lemma 15:** Assume \( x, y \subseteq V_g \). \( \bar{x} \triangleq \bar{y} \) if and only if \( x \subseteq y \).

**Lemma 16:** Assume \( x, y \subseteq V_g \). \( \bar{x} \cup \bar{y} \) is the least upper bound of \( \bar{x} \) and \( \bar{y} \) based on the partial ordering \( \leq \).

**Proof:** Note that \( x \subseteq x \cup y \) and \( y \subseteq x \cup y \). According to Lemma 13, \( \bar{x} \subseteq x \cup y \) and \( \bar{y} \subseteq x \cup y \). Therefore, \( x \cup y \) is an upper bound of \( \bar{x} \) and \( \bar{y} \). Furthermore, \( x \cup y \) is the smallest set that contains all elements of both \( x \) and \( y \). \( x \cup y \) is the smallest element of \( 2^{|V_g|} \) that contains all elements of \( x \) and \( y \) and hence contains all elements of \( \bar{x} \) and \( \bar{y} \). Therefore, \( x \cup y \) is the least upper bound of \( \bar{x} \) and \( \bar{y} \).

**Lemma 17:** Assume \( x, y \subseteq V_g \). \( \bar{x} \wedge \bar{y} \) is the least upper bound of \( \bar{x} \) and \( \bar{y} \) based on the partial ordering \( \leq \).
Fig. 4. The run-time memory is organized as various closed clusters. We show only a few closed clusters in this figure. Each area enclosed by a half ellipse is a closed cluster.

Finally, we need to prove the condition of a Brouwerian algebra:

**Lemma 18:** Assume \( x, y, z \subseteq V_g \). \( \bar{z} \lor \bar{y} \geq \bar{x} \) if and only if \( \bar{z} \geq \bar{x} \lor \bar{y} \).

**Proof:**
\[
\begin{align*}
\bar{z} \geq \bar{x} \lor \bar{y} & \quad \text{(according to the definition of } \lor) \\
\text{iff } \bar{z} \geq x - y & \quad \text{(by Lemma 15)} \\
\text{iff } z \supseteq (x - y) \lor y & \\
\text{iff } z \lor y \supseteq x \lor y & \\
\text{iff } z \lor y \supseteq x & \quad \text{(because } x \lor y \supseteq x) \\
\text{iff } z \lor y \geq x & \quad \text{(by Lemma 15)} \\
\text{iff } \bar{z} \lor \bar{y} \geq \bar{x} & \quad \square
\end{align*}
\]

**Lemma 19:** Assume \( x, y, z \subseteq V_g \). \( \bar{x} \lor \bar{y} = \min \{ \bar{z} | \bar{z} \lor \bar{y} \geq \bar{x} \} \).

The structure \( B = (2^{V_g}, \lor, \land, \perp) \) is a Brouwerian algebra. Most properties of \( \land \) and \( \lor \) are inherited from that of set intersection \( \cap \) and set union \( \cup \).

The \( \lor \) and \( \land \) operations are monotonic with respect to the partial ordering:

**Lemma 20:** Assume \( x, y, z, w \subseteq V_g \). \( \bar{x} \leq \bar{z} \) and \( \bar{y} \leq \bar{w} \) implies that \( \bar{x} \lor \bar{y} \leq \bar{z} \lor \bar{w} \) and \( \bar{x} \land \bar{y} \leq \bar{z} \land \bar{w} \).

The top element is \( \bar{V}_G = V_G \cup V_h = \{ \bar{x} | x \subseteq V_g \} \). The bottom element is \( \bar{\emptyset} = \emptyset = \land \{ \bar{x} | x \subseteq V_g \} \).

**Lemma 21 (Commutativity):** Assume \( x, y \subseteq V_g \). \( \bar{x} \lor \bar{y} = \bar{y} \lor \bar{x} \). \( \bar{x} \land \bar{y} = \bar{y} \land \bar{x} \).

**Lemma 22 (Associativity):** Assume \( x, y, z \subseteq V_g \). \( \bar{x} \lor (\bar{y} \lor \bar{z}) = (\bar{x} \lor \bar{y}) \lor \bar{z} \). \( \bar{x} \land (\bar{y} \land \bar{z}) = (\bar{x} \land \bar{y}) \land \bar{z} \).
Lemma 23 (Absorption): Assume \( x, y \subseteq V_g \). \( \overline{x \lor (x \land y)} = \overline{x \lor (x \land y)} = \overline{x} \).

Lemma 24 (Idempotence): Assume \( x \subseteq V_g \). \( \overline{x} \lor \overline{x} = \overline{x} \). \( \overline{x} \land \overline{x} = \overline{x} \).

Lemma 25 (Identity): Assume \( x \subseteq V_g \). \( \overline{x} \lor \phi = \overline{x} \). \( \overline{x} \land \overline{V_g} = \overline{x} \).

Lemma 26 (Distributivity): Assume \( x, y, z \subseteq V_g \). \( \overline{x} \land (y \lor z) = (\overline{x} \land y) \lor (\overline{x} \land z) \).

Though \( \overline{x \lor y} \subseteq \overline{x} \lor \overline{y} \), \( \overline{x \lor y} \neq \overline{x} \lor \overline{y} \) in general. For example, in Fig. 1, the \( R \) nodes are in \( \{A, B\} \), but are not in \( \{A\} \) nor in \( \{B\} \). On the other hand, it is less obvious that \( x \cap y \cap y \).

Lemma 27: Assume \( x, y \subseteq V_g \). \( x \cap y = \overline{x} \cap \overline{y} \). That is, \( \overline{x} \cap \overline{y} = \overline{x \cap y} \).

Proof: Because \( x \cap y \subseteq x \), \( x \cap y \subseteq x \). Similarly, since \( x \cap y \subseteq y \), \( x \cap y \subseteq y \). Therefore, \( x \cap y \subseteq x \cap y \).

Note \( x \cap y \subseteq x \cap y \) and \( x \cap y \subseteq x \cap y \). Assume \( x \cap y \subseteq x \cap y \). There is a node \( d \in V_h \), \( d \in \overline{x \cap y} \) but \( d \notin x \cap y \).

Because \( d \in x \cap y \), \( d \in x \) and \( d \notin x \). Because \( d \in x \), for every node \( p \in V_g \), if there is a path \( p \rightarrow d \), \( p \in x \) and the whole path \( p \rightarrow d \) is completely in \( x \) according to Lemma 14.

Similarly, because \( d \in y \), for every node \( p \in V_g \), if there is a path \( p \rightarrow d \), \( p \in y \) and the whole path \( p \rightarrow d \) is completely in \( y \).

Therefore, \( p \in x \cap y \), for every node \( p \in V_g \) which can reach \( d \) (through a path \( p \rightarrow d \)). We may conclude that \( d \in x \cap y \). This contradicts a previous assumption. Therefore, \( x \cap y \subseteq x \cap y \).

Lemma 28: Assume \( x, y, z \subseteq V_g \). \( (x \cap y) \circ z = \overline{x \cap y} \circ z = \overline{x \cap y} \circ z = (\overline{x \cap y} \circ z) = (\overline{x} \cap (\overline{y} \circ z)). \)

Proof: \((\overline{x \cap y} \circ z) = \overline{x \cap y} \circ z = \overline{x \cap y} \circ z = \overline{x \cap y} \circ z = \overline{x} \cap (\overline{y} \circ z) \).

Corollary 29: Assume \( x, y, z \subseteq V_g \). \( (x \cap y) \circ z = (x \cap y) \circ z \).

Corollary 30: Assume \( x, y \subseteq V_g \). \( (\overline{V_g} \circ y) \circ z = \overline{V_g} \circ (\overline{y} \circ z) \).

4.1 Iterated Garbage Collection

Let \( x \subseteq V_g \). When all the references in \( x \) are destroyed, the heap nodes in \( x \) become dead. An obvious garbage-collection algorithm is to reclaim all the heap nodes in such a closed cluster \( x \) immediately after all the references in \( x \) are destroyed. In this garbage-collection algorithm, a run of garbage collection will not leave garbage floating in the heap. Rather, floating garbage is created due to delayed garbage collection.
Lemma 31: Assume every heap node in \( G \) is reachable from a node in \( V_g \). Let \( x \subseteq V_g \). Let \( G - \bar{x} \) is still a memory graph. In particular, every heap node of \( G - \bar{x} \) is still reachable from a node in the global area of \( G - \bar{x} \), i.e., \( V_g - x \).

Let \( G \) be a memory graph. \( G - \bar{x} \) is the memory graph after a run of garbage collection starting from nodes in \( x \). Lemma 31 implies that a general mark-sweep garbage collector is not needed if every garbage-collection run will collect an entire closed cluster. This is because no garbage will be left floating around in the heap after a garbage-collection run.

**Proof:** We use proof by contradiction. Suppose heap node \( m \) is not reachable from any node in the global area of \( G - \bar{x} \). Since \( G \) is a memory graph, there must be a node \( n \) in the global area of \( G \) that can reach \( m \). Let \( n_1, n_2, n_3, \ldots, n_k \) be all the nodes in \( V_g \) that can reach \( m \) in \( G \). Then \( m \in \{n_1, n_2, n_3, \ldots, n_k\} \).

If \( \{n_1, n_2, n_3, \ldots, n_k\} \subseteq x \) and \( m \in \bar{x} \), by Lemma 13. That is, \( m \notin G - \bar{x} \). This contradicts a previous assumption about \( m \).

On the other hand, if \( \{n_1, n_2, n_3, \ldots, n_k\} \subseteq x \) and \( n_j \notin x \). Then all nodes on the path \( n_j \to m \) are in \( \bar{x} \). Thus, all nodes on the path \( n_j \to m \) are in \( G - \bar{x} \). \( m \) could not be a floating garbage node. This contradicts a previous assumption about \( m \).

Lemma 32: Assume \( x, y \subseteq V_g \). \( G - \bar{x} \) implies \( CCG(G - \bar{x}, y - x) \).

**Proof:** Let heap node \( m \in (\bar{x} \cup y) - \bar{x} \). Then \( m \) is reachable from a node \( q \) in \( y \) (for otherwise, \( m \in \bar{x} \) and hence \( m \in (\bar{x} \cup y) - \bar{x} \)). By an inductive argument, all nodes on the path from \( q \) to \( m \) are in \( (\bar{x} \cup y) - \bar{x} \).

Furthermore, because \( m \in (\bar{x} \cup y) \), \( m \) is not reachable from any node in \( V_g - (x \cup y) \). Therefore, \( m \in CCG(G - \bar{x}, y - x) \).

This property means that garbage collection based on \( x \cup y \) is the same as garbage collection based on \( x \) alone and then garbage collection based on \( y - x \) in the remaining memory graph.

Because \( x \cup y \subseteq \bar{x} \cup \bar{y} \) but \( x \cup y \neq \bar{x} \cup \bar{y} \) in general, concurrent garbage collections based on \( x \) and \( y \), respectively, must include additional mark-sweep passes (or other similar procedures) otherwise nodes in \( \bar{x} \cup \bar{y} - (\bar{x} \cup \bar{y}) \) will become floating garbage. We can use this criterion to check various concurrent garbage-collection algorithms.

Note that the powerset \( \{\{x \subseteq V_g\}, \cup, \cap\} \) is a Boolean algebra. The Brouwerian algebra \( \{\{x \subseteq V_g\}, \cup, \cap, V_g, \cap\} \) may be viewed as a distorted projection of the Boolean algebra \( \{\{x \subseteq V_g\}, \cup, \cap\} \).

4.2 Incrementally Updating the Brouwerian Structure

During run time, a program performs various operations one by one. The underlying Brouwerian structure is modified by the memory operations. There are two memory operations:

- create a reference (i.e., create an edge in the memory graph)
- destroy a reference (i.e., destroy an edge in the memory graph)

The graph-theoretical difference of a memory graph \( G \) and a set of nodes \( y \), denoted by \( G - y \), is the subgraph of \( G \) obtained from \( G \) by removing all nodes in \( y \) and all edges incident on the removed nodes.
A usual pointer assignment such as

\[ \text{aptr} := \text{bptr}; \]

is considered as two memory operations: first, the original reference in the `aptr` variable is destroyed. Then the reference in the `bptr` variable is duplicated and stored in the `aptr` variable.

After the memory operations modify the memory graph, we may recompute the underlying Brouwerian structure. Alternatively, we may incrementally change the Brouwerian structure after each memory operation.

For the operation `create a reference`, there are two cases. See Fig. 5.

**Case 1:** Add an edge \( p \rightarrow q \), assuming \( q \) has no other incoming edges. In this case, the Brouwerian structure is modified as follows:

\[ \text{Add all nodes of } CC(q) \text{ to every closed cluster that includes } p. \]

This case corresponds to allocating a new node \( q \) and hence \( CC(q) = \{ q \} \).

**Case 2:** Add an edge \( p \rightarrow q \), assuming \( q \) already has incoming edges \( s_1 \rightarrow q, s_2 \rightarrow q, \ldots, s_k \rightarrow q, \) where \( k > 0 \). In this case, the Brouwerian structure is modified as follows:

Temporarily delete all of \( q \)'s incoming edges; calculate \( CC(q) \); put all of \( q \)'s incoming edges back; delete all nodes of \( CC(q) \) from every closed cluster that includes all of \( \{ s_1, s_2, \ldots, s_k \} \) but does not include \( p \).

For the operation `destroy a reference`, there are also two cases. See Fig. 6.

**Case 1:** Destroy an edge \( p \rightarrow q \), assuming \( q \) has no other incoming edges. In this case, the Brouwerian structure is modified as follows:

Delete all nodes of \( CC(q) \) from every closed cluster that includes \( p \).

**Case 2:** Destroy an edge \( p \rightarrow q \), assuming \( q \) already has incoming edges \( s_1 \rightarrow q, s_2 \rightarrow q, \ldots, s_k \rightarrow q, \) where \( k > 0 \). In this case, the Brouwerian structure is modified as follows:
Temporarily delete all of $q$’s incoming edges; calculate $CC(q)$; put all of $q$’s incoming edges back; add all nodes of $CC(q)$ to every closed cluster that includes all of $\{s_1, s_2, \ldots, s_k\}$ but does not include $p$.

Fig. 6. Destroy a reference $p \rightarrow q$.

4.3 Applications of the Brouwerian Model

An edge $a \rightarrow b$ in the memory graph denotes a reference stored in node $a$ that points to node $b$. When an edge $a \rightarrow b$ is destroyed, all and only the (heap) nodes in the closed cluster $CC(b)$ become dead. From this observation, we may develop several applications of the Brouwerian model.

Since all the (heap) nodes in the same cluster become dead at the same time, a garbage-collection algorithm should collect garbage one cluster at a time, rather than one node or one strongly connected component at a time, for otherwise dead nodes may be left floating around pending for additional collection measures. The Brouwerian model helps in determining whether a garbage-collection algorithm will suffer from the memory-leak problem.

Since all the (heap) nodes in the same cluster live and die at the same time, it would be advantageous to allocate nodes belonging to the same cluster in the same memory “region”. The dynamic allocation heap can be partitioned into regions and nodes belonging to the same cluster had better be allocated in the same region. Of course, a powerful static and/or dynamic analysis of the closed clusters has to be developed. The Brouwerian model helps in guiding memory allocation.

In a mark-sweep-compact algorithm, the (heap) nodes surviving a run of garbage collection may be moved during the compaction phase. Since all the (heap) nodes in the same cluster live and die at the same time, heap nodes in the same closed cluster should be moved to the same region during compaction. The Brouwerian model helps in guiding the compaction operation.

5. THE \textsc{CalculateCC} Algorithm

This section presents the \textsc{CalculateCC} algorithm for computing the closed clusters. We need a few terms.

\textbf{Definition}: The \textit{span} of a node $a$, denoted as $\text{span}(a)$, where $a$ could be on the heap or in the global area, consists of all the nodes that are reachable from $a$ in the memory graph.
Fig. 7. The structure of a closed cluster. The bold arrows form a depth-first-search tree. The notation 0/1 in node $b$ means $gcount(b) = 0$ and $hcount(b) = 1$.

In Fig. 7, $\text{span}(a) = \{a, b, c, d, e, f, g, h, i, j\}$. Note that $\text{CC}(a) \subseteq \text{span}(a)$. Since spans are defined as a transitive closure, the following lemma is obvious.

**Lemma 33:** If node $b \in \text{span}(a)$ then $\text{span}(b) \subseteq \text{span}(a)$.

In a span, there may be nodes that have an incoming edge from outside the span. If a node $i$ in a span $\text{span}(a)$ has an incoming edges $p \rightarrow i$, where $p$ is a global node not in $\text{span}(a)$, then $i$ is called a g-port in $\text{span}(a)$. Similarly, if a node $d$ in $\text{span}(a)$ has an incoming edges $r \rightarrow d$, where $r$ is a heap node not in $\text{span}(a)$, then $d$ is called an h-port in $\text{span}(a)$. The span of a g-port is called a g-subspan. Similarly, the span of an h-port is called an h-subspan.

A closed cluster is a combination of a span and subspans, as shown in Fig. 7.

**Example:** Consider $\text{span}(a)$ in Fig. 7, which is the blue region (containing $a, b, c, d, e, f, g, h, i,$ and $j$). In $\text{span}(a)$, node $i$ is a g-port and node $d$ is an h-port. $\text{span}(i)$, which is the red region (containing $f, g, i,$ and $j$), is a g-subspan and $\text{span}(d)$, the green region (containing $d, e, f,$ and $g$), is an h-subspan. Note that $\text{span}(i) \subseteq \text{span}(a)$ and $\text{span}(d) \subseteq \text{span}(a)$. That is, all the g-subspans and h-subspans in the span are completely included in the span.

A closed cluster is a span excluding nodes that are reachable from nodes outside the span. Because all nodes in the g-subspans and h-subspans are reachable from nodes outside the span, they do not belong to the closed cluster. Conversely, all nodes reachable from nodes outside the span must belong to some g-subpan or h-subspan. Therefore, we may conclude that the closed cluster of a node $a$ is $\text{span}(a)$ excluding all nodes in the g-subspans and h-subspans in $\text{span}(a)$. For example, in Fig. 7,

$$\text{CC}(a) = \text{span}(a) - \text{span}(i) - \text{span}(d) = \{a, b, c, h\}.$$ 

Based on the above example, it is straightforward to develop an algorithm for computing a closed cluster.

We will use Fig. 7 to explain our algorithm. To compute $\text{CC}(a)$, We perform a first depth-first traversal, starting from node $a$, in the memory graph. During the first traver-
sal, we can identify nodes in \(\text{span}(a)\) and all the g-ports, i.e., node \(i\), in \(\text{span}(a)\). In the first depth-first traversal, we also do a depth-first traversal, starting from each g-port. All nodes in \(\text{span}(a)\) are marked dead while all nodes in the g-subspans are revived, i.e., marked as live again. Thus, at the end of the first traversal, we identified \(\text{span}(a) - \text{spa}(i)\).

Then we perform a second depth-first traversal, starting from node \(a\). During the second depth-first traversal, we can identify all the h-ports, i.e., node \(d\). In the second depth-first traversal, we also do a depth-first traversal, starting from each h-port. All nodes in the h-subspans are revived. In summary, at the end of the second traversal, we identified \((\text{span}(a) - \text{spa}(i)) - \text{span}(d)\).

Finally, we perform a third depth-first traversal, starting from node \(a\). During the third traversal, we collect all nodes that are still marked dead. \(CC(a)\) contains exactly these dead nodes.

The \(\text{CalculateCC}\) algorithm, shown in Figs. 8 and 9, computes the closed cluster induced by a node.

Each node \(n\) maintains two reference counters: \(gcount\) and \(hcount\). \(gcount(n)\) contains the number of references from the global area to node \(n\). \(hcount(n)\) contains the number of references from the heap to \(n\). Since our algorithm will perform depth-first traversals in the heap, each node (in the heap) may be visited more than once. Each node \(n\) maintains a counter \(\text{visitcount}(n)\), which records the number of times \(n\) is visited during the depth-first traversal.

Every node also contains a status variable, which could be dead or live. Every node contains a boolean visited variable, which indicates whether the node has been visited yet.

To calculate the closed cluster induced by node \(m\), we invoke \(\text{CalculateCC}(m)\). If \(m\) is in the global area but has no outgoing edge, then \(CC(m) = \{m\}\). If \(m\) is in the global area and has a (single) outgoing edge \(m \rightarrow n\), we will delete that edge (and decrement \(gcount(n)\) by 1) and calculate \(CC(n)\) (with the \(\text{CConHeap}\) function) instead. Finally, \(CC(m) = \{m\} \cup CC(n)\).

1. function \(\text{CalculateCC}(m : \text{node})\) returns a set of nodes
2. if \(m\) is in the global area and has no outgoing edge then return\(\{m\}\);
3. if \(m\) is in the global area and has an outgoing edge \(m \rightarrow n\) then
4.   delete the edge \(m \rightarrow n\);
5.   \(gcount(n) := gcount(n) - 1\);
6.   \(\text{temp} := \text{CConHeap}(n)\);
7.   re-insert the edge \(m \rightarrow n\);
8.   \(gcount(n) := gcount(n) + 1\);
9.   return\(\{m\} \cup \text{temp}\);
10. end
11. return\(\text{CConHeap}(m)\);
12. end \(\text{CalculateCC}\)

1. function \(\text{CConHeap}(m : \text{node})\) returns a set of nodes
2. /* \(m\) is a heap node. */
3. \(\text{dfsdead}(m)\); /* \text{dfsdead}(m)\) is the 1st traversal, starting from \(m\). */
4. \(\text{hcount}(m) := \text{hcount}(m) + 1\);
5. \(\text{search}(m)\); /* \text{search}(m)\) is the 2nd traversal */
6. \(\text{hcount}(m) := \text{hcount}(m) - 1\);}
The $CConHeap(m)$ function calculates the closed cluster for a heap node $m$. First we add an imaginary edge $\ldots \rightarrow m$ for the sake of uniformity in the search procedure. Hence, $hcount(m)$ is incremented by 1. $hcount(m)$ will be decremented by 1 at the end of the $CConHeap$ function.

In an invocation $CConHeap(m)$ (where $m$ must be in the heap), the node $m$ will be called the root. Note that a complete depth-first traversal, starting from the root, of a span will visit each edge in the span exactly once. The traversal will highlight a depth-first tree (dfs-tree) in the span.

A node $q$ in the current span is dead unless it satisfies one of the following three conditions:

1. $gcount(q) > 0$. This means $q$ is reachable from a node in the global area. That is, $q$ is a g-port.
2. $hcount(q) > visitcount(q)$. This means $q$ is reachable from a heap node outside the current span. That is, $q$ is an h-port.
3. $q$ is reachable from a g-port or an h-port in the current span.

```plaintext
1. procedure dfslive(m : node) /* revive a g-subspan */
2. if visited(m) = yes and status(m) = live then return;
3. visited(m) := yes;
4. status(m) := live;
5. for each outgoing edge of m (say $m \rightarrow z$) do dfslive(z);
6. end dfslive
```

```plaintext
1. procedure search(m : node) /* 2nd traversal */
2. if status(m) = live then return;
```

Fig. 8. The CalculateCC algorithm.
A BROUWERIAN MODEL OF THE RUN-TIME MEMORY

Fig. 9. The CalculateCC algorithm.

We use the dfsdead procedure to mark nodes in the current span as dead. The dfsdead procedure identifies the g-ports. The search procedure identifies the h-ports. The dfslive and revive procedures mark nodes in g-subspans and h-subspans as live.

The CConHeap(m) procedure calls dfsdead(m) to perform the first depth-first traversal, starting from node m, to identify span(m) and all g-subspans. It then calls search(m) to identify all h-subspans, and finally calls collect(m) to collect the dead nodes.

The dfsdead(m) procedure performs a (partial) depth-first traversal recursively, starting from node m. If m has already been visited, visitcount(m) is incremented by 1 and the dfsdead(m) procedure returns immediately.

On the other hand, if m has not been visited, gcourt(m) is checked. If gcourt(m) is 0, the node m is marked as (potentially) dead and dfsdead continues to m’s children recursively. If gcourt(m) > 0, m is a g-port and all nodes in g-subspan(m) are marked as live by the dfslive procedure.

Each of the dfsdead and dfslive procedures performs one partial depth-first traversal of the span. The former marks the visited nodes as (potentially) dead while the latter (definitely) live. Note that nodes in the g-subspans may be visited by both depth-first traversals.

The search procedure looks for h-ports that are marked as dead by comparing hcount and visitcount. hcount(m) is the number of m’s incoming pointers from the heap. visitcount(m) is the number of times m is visited during the dfsdead traversal. If
$hcount(m) > visitcount(m)$, the node $m$ is referenced by some nodes outside the current span. In this case, $m$ (and, hence, all nodes in $h$-$subspan(m)$) must be considered as live. The $revive$ procedure is invoked to mark $m$ and all nodes in $h$-$subspan(m)$ as live.

On the other hand, in case $hcount(m) = visitcount(m)$, all the heap pointers that point to node $m$ are from the dead nodes in the current span. Therefore, $m$ remains (potentially) dead. However, $m$ cannot be freed immediately in the $search$ procedure because $m$ might be reachable from a node that is revived by the $revive$ procedure later. Only when all calls to the $revive$ procedure finish can we be sure that the dead nodes are in the closed cluster. Finally, the $collect$ procedure accumulates these dead nodes.

The $search$ and $revive$ procedures together will visit the part of the current span that is not visited by the $dfs$-$live$ procedure. Therefore, the $dfs$-$live$, $search$, and $revive$ procedures together will perform at most one complete depth-first traversal of the current span. The $dfs$-$dead$ procedure will perform a partial depth-first traversal. The $collect$ procedure will perform another partial depth-first traversal of the current span. At most three depth-first traversals of the current span are needed in an invocation of $CConHeap$. The cycle collectors discussed in [8, 18, 20, 22, 26] also run up to three traversals in the current span.

Each node needs 5 fields: $gcount$, $hcount$, $visited$ (a boolean variable), $status$ (also a boolean variable), and $visitcount$. However, only $gcount$ and $hcount$ are permanent. The other three fields are used only during the execution of the $CConHeap$ procedure. Thus, the three fields can be declared as local array variables in $CConHeap$, with one entry for each node in the span under examination. In this implementation, each node carries only two fields: $gcount$ and $hcount$.

**Example:** Fig. 7 shows a snapshot of a computer’s memory. The numbers under the node name are the node’s $gcount$ and $hcount$, respectively. For instance, $gcount(b) = 0$ and $hcount(b) = 1$. $CalculateCC(q)$ will invoke $CConHeap(a)$, which, in turn, invokes $dfs$-$dead(a)$. $dfs$-$dead(a)$ will traverse the span, marking nodes $a$, $b$, $c$, $d$, $e$, $f$, $g$, $h$ (we assume that these nodes are visited in this order) as dead. When node $i$ is visited, $dfs$-$live(f)$ and $dfs$-$live(j)$ will be invoked since $gcount(i) > 0$. $dfs$-$live(f)$ and $dfs$-$live(j)$ will mark nodes $i$, $j$, $f$, $g$ as live. Note that nodes $f$ and $g$ are visited by both $dfs$-$dead(a)$ and $dfs$-$live(f)$. $dfs$-$dead$ and $dfs$-$live$ together will perform a complete depth-first traversal plus some portion of the $g$-subspans, which, in this example, contains nodes $f$ and $g$. The overlapping portion also depends on the order nodes are visited during the $dfs$-$dead(a)$ call.

The $search(a)$ procedure will visit nodes $a$, $b$, $c$, $d$, $h$, but not node $e$. Because $hcount(d) > visitcount(d)$, $revive(d)$ will be invoked. $revive(d)$ will revive nodes $d$ and $e$. The $collect(a)$ call will reclaim nodes $a$, $b$, $c$, $h$, which constitutes $CC(a)$. Finally, $CC(q) = \{q, a, b, c, h\}$.

### 5.1 Correctness and Efficiency of the CalculateCC Algorithm

A garbage collection algorithm must be sound and complete. Soundness means only dead nodes are reclaimed. Completeness means all dead nodes are reclaimed eventually. Completeness implies no memory leak. Though it is trivial to make any garbage-collection algorithm complete (simply by appending an extra mark-sweep pass, which is very
expensive), designing a complete collection algorithm without the extra pass is challenging.

The Brouwerian model makes it easy to verify that the $\text{CalculateCC}$ algorithm is sound and complete since our algorithm reclaim an entire cluster in every invocation.

We wish to show that the $\text{CalculateCC}(n)$ algorithm identifies all and only nodes in $CC(n)$. Let $m \in CC(n)$. Then $m$ is reachable from $n$ but is not reachable from any node outside $CC(n)$ according to the definition of closed clusters. Furthermore, all nodes on every path from $n$ to $m$ are not reachable from any node outside $CC(n)$ and their $gcounts$ are all 0. Therefore, $m$ will be marked as dead during the $\text{dfsdead}(n)$ traversal (which is a depth-first traversal of the span). Since $m$ is not reachable from any node outside $CC(n)$, $m$ will not be visited during the $\text{dfslive}$ traversal. Hence, $m$ will not be marked as live during the $\text{dfslive}$ traversal.

Furthermore, for all nodes $q$ on every path from $n$ to $m$, $gcount(q) = 0$ and $visitcount(q) = hcount(q) > 0$. Therefore, $q$ will not be visited during the $\text{revive}$ traversal and $q$’s status remains dead. Therefore, $q$ (and in particular $m$) will be collected during the $\text{collect}$ traversal.

On the other hand, if node $m$ is collected during the $\text{collect}$ traversal, $m$ must have been marked dead by the $\text{dfsdead}$ procedure and must have not been revived by either the $\text{dfslive}$ procedure or the $\text{revive}$ procedure. Therefore, $m$ is unreachable from any node with $gcount > 0$ or with $hcount > visitcount$. Therefore, $m$ must be in the closed cluster indeed.

The worst-case running time occurs when $\text{dfslive}$ and $\text{revive}$ are not invoked in the current run. Under this situation, the closed cluster is the entire current span and every edge in the current span is visited three times. This is similar to the cycle collectors discussed in [18, 20, 22], which run up to three traversals in the current span.

The above $\text{CalculateCC}$ algorithm computes the closed cluster induced by a single node. To compute the closed cluster induced by a set of global nodes, say $CC(\{a_1, a_2, a_3, \ldots, a_k\})$, we add a new node $b$ and $k$ new edges $b \rightarrow a_1, b \rightarrow a_2, b \rightarrow a_3, \ldots, b \rightarrow a_k$. The new node $b$ will be considered as a global variable and $a_1, a_2, a_3, \ldots, a_k$ will be considered as nodes on the heap. Their $gcounts$ and $hcounts$ are adjusted accordingly. We compute $CC(b)$ on this augmented graph. Then $CC(\{a_1, a_2, a_3, \ldots, a_k\}) = CC(b) = \{b\}$.

5.2 An Even Faster Variation

From the example in Fig. 7, we can see that the current span is partitioned into three disjoint sets:

- **the definitely live nodes**: the nodes that are marked as live by the $\text{dfslive}$ procedure—these nodes are reachable from variables in the global area.
- **the potentially live nodes**: the nodes that are marked as (potentially) dead but are revived by the $\text{revive}$ procedure—these nodes are reachable from nodes in the heap that are outside the current span.
- **the definitely dead nodes**: the nodes that are accumulated by the $\text{collect}$ procedure—these nodes constitute the closed cluster induced by the root of the current span.

In order to calculate a closed cluster, it is unnecessary to distinguish the definitely live nodes and the potentially live nodes. We can erase line 12 of the $\text{dfsdead}$ procedure and delete the $\text{dfslive}$ procedure. The rest of the algorithm remains unchanged. Nodes

\[ gcount(b) = hcount(b) = 0; gcount(a_i) = 1, hcount(a_i) = 0, \text{ for all } i; \text{ other nodes that have incoming edges from } \text{a}_i \text{'s need to adjust their gcounts and hcounts.} \]
that are visited by the \textit{dfs\textsc{live}} procedure will be visited by the \textit{revive} procedure instead or will not be visited at all. The result is a faster, simpler algorithm because fewer edges are examined. For the example in Fig. 7, node \( j \) will not be visited because a live node, node \( i \) in this example, appears on every path from the root \( a \) to node \( j \). Nodes \( f \) and \( g \) will be classified as potentially live nodes rather than definitely live nodes in this simplified algorithm.\(^6\)

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig10.png}
\caption{Apply the one-counter and two-counter algorithms to the same memory graph.}
\end{figure}

5.3 Comparison with the One-Counter Algorithm

Traditional one-counter algorithms operate with one counter per node. In that case, all \textit{gcounts} are 0 and there are no \( g \)-ports nor \( g \)-subspans. A closed cluster \( CC(a) \) is obtained by deleting all \( h \)-subspans from \( \text{span}(a) \).

We may create a one-counter algorithm by modifying the above two-counter algorithm by assuming \textit{gcount} is always 0. We will erase lines 6, 9, 10, 11, 12, and 13 of the \textit{dfs\textsc{dead}} procedure and the whole \textit{dfs\textsc{live}} procedure in Figs. 8 and 9.

Our two-counter algorithm examines no more edges than this one-counter algorithm. Consider the example in Fig. 10. Fig. 10 shows a memory graph with 14 nodes and 18 edges in \( \text{span}(a) \). \( CC(a) = \{ a, b, c, i \} \). Nodes \( p \) and \( q \) are in the global area and all other nodes are on the heap. The number on an edge denotes the number of times the edge is examined in the two algorithms. Fig. 10 (a) shows the result of the one-counter algorithm (the notation 2/1 in node \( j \) means \( \text{count}(j) = 2 \) and \( \text{visitcount}(j) = 1 \)). There are 47 examinations of the edges. 13 edges are examined 3 times. Fig. 10 (b) shows the result of the two-counter algorithm (the notation 0/2/1 in node \( j \) means \( \text{gcount}(j) = 0 \), \( \text{hcount}(j) = 2 \) and \( \text{visitcount}(j) = 1 \)). There are 41 examinations of the edges. 9 edges are examined 3 times.

In addition, the traditional one-counter algorithm may waste more time tracing more live nodes than the two-counter algorithm due to its im precision—tracing starts from every node whose counter is decremented. In contrast, in our two-counter algorithm, tracing only starts from nodes whose \textit{gcount} becomes zero. The two-counter algorithm performs fewer tracing operations than the one-counter algorithm.

6. CONCLUSION

We are motivated by a study of garbage collection and establish a Brouwerian

\(^{6}\) A moment’s reflection will reveal that the \textit{status} variable in each node may also be re moved. We can use the \textit{visited} variables in the nodes alone in the algorithm. For the sake of easier understanding, we keep the \textit{status} variable in the algorithm.
model of the run-time memory of a program, which in turn helps us to design, validate, and evaluate memory-management algorithms. We discussed the properties of the Brouwerian model and the transition from one Brouwerian model to the next by a memory operation. We also presented a two-counter algorithm for computing the closed clusters, which is the building blocks of the Brouwerian model. The two-counter algorithm is faster than a traditional one-counter algorithm in that it may examine fewer edges. The extra cost of the two-counter algorithm is one more counter per node than the traditional one-counter algorithm.

There are three applications of the Brouwerian model: (1) it helps in determining whether a garbage-collection algorithm will suffer from the memory-leak problem; (2) it can guide the memory-allocation algorithm; and (3) it can guide the compaction operation in a mark-sweep-compact algorithm.

REFERENCES


**Wuu Yang** (楊武) received his B.S. degree in Information Engineering from National Taiwan University in 1982 and the M.S. and Ph.D. degrees in Computer Science from the University of Wisconsin at Madison in 1987 and 1990, respectively. He joined the Computer Science Department in the National Chiao-Tung University since August 1992, where he is a Professor currently. Dr. Yang’s current research interests include systems software for embedded systems and IoT, programming languages and compilers, and attribute grammars.