A fast algorithm to compute heap memory bounds of Java Card applets

Tuan-Hung Pham, Anh-Hoang Truong,
Ninh-Thuan Truong
College of Technology, Vietnam National University
144 Xuan Thuy, Hanoi, Vietnam

Wei-Ngan Chin
School of Computing
National University of Singapore
3 Science Drive 2, Singapore 117543

Abstract

We present an approach to find upper bounds of heap space for Java Card applets. Our method first transforms an input bytecode stream into a control flow graph (CFG), and then collapses cycles of the CFG to produce a directed acyclic graph (DAG). Based on the DAG, we propose a linear-time algorithm to solve the problem of finding the single-source largest path in it. We also have implemented a prototype tool, tested it on several sample applications, and then compared the bounds found by our tool with the actual heap bounds of the programs. The experiment shows that our tool returns good estimation of heap bounds, runs fast, and has a small memory footprint.

1 Introduction

Java Card is a technology that allows us to develop software applications that can run on smart cards and devices with very limited memory and processing power. Many of these devices allow new applications to be deployed on them, even after they have been issued to end-users. Since the new applications can come from third parties, they can, unintentionally or intentionally, cause memory overflows, which may result in loss of data in smart cards and even destruction of the cards. It is, therefore, necessary for a smart card to have a program to check the maximum heap memory of new applications that are going to be installed on the card. The checker itself should also run fast and have a small memory footprint.

To adapt to the smart card environment, Java Card virtual machine (JCVM) [11, 12] is developed as a compact version of Java virtual machine (JVM) [18]. The instruction set of JCVM is only a subset of JVM's. In comparison with JVM, JCVM does not support several complex features such as threads, cloning, and finalization. As a result, JCVM does not often provide garbage collection, a resource-consuming technique for managing memory used by objects automatically. Without garbage collection, allocated objects in the Java Card platform can still consume memory even when they are no longer used or referenced. When a smart card contains many disused objects, the card can easily run out of memory. Consequently, it raises many problems of managing memory cost. One of the problems that we focus on is how to determine the maximum heap cost used by an applet without the presence of any garbage collectors.

The problem of finding an upper bound of memory usage of a program is not new, and various approaches have been developed to address it. Some approaches [16, 8, 5, 9, 10, 17] analyze the program's source code to calculate an upper bound of heap memory. Others [3, 7] propose the ideas of managing memory resources by analyzing the memory affected for each bytecode instruction in compiled code.

Extending our previous work [13], we explore a slightly different approach with a better algorithm, which is suitable to run on cards. We assume that the size of allocated arrays and the numbers of iterations of loops are input parameters or constants. When input parameters are used to bound loop executions, we assume that the used parameters are not changed inside the loops. However, if they are altered, our approach can detect the changes and indicate whether the obtained results are good or not in these cases. The main steps of our approach proceed as follows:

- First, we construct a control flow graph (CFG) of the bytecode stream of an input method in a Java Card applet. The nodes of the CFG are the bytecode instructions. For edges, if one instruction follows another in execution order, we create a directed edge from the former to the latter.
- Then, we transform the CFG into a weighted DAG. The transformation collapses cycles and assigns each edge of the CFG a corresponding non-negative weight.
- Finally, we compute the heap bound by DAG-largest-path algorithm, a linear-time algorithm to solve the single-source largest path problem in DAG.

In implementation, we combine these steps to reduce our algorithmic complexity. We obtain a tool that takes the

bytecode stream of a method of Java Card applets as input and returns a heap bound function of method parameters as output. We tested the implemented tool on several programs including applets provided in Java Card Development Kit 2.2.2 and some programs that use complex data structures such as trees, linked lists, stacks, and queues. These programs can also contain loops and method invocations. The experimental results have shown that our algorithm can not only find the lowest upper bound of heap cost in many cases, but also run fast and have a small memory footprint.

The rest of this paper is organized as follows. Section 2 briefly introduces Java Card bytecode. Section 3 presents our approach to find upper bounds of heap space for Java Card applets. In Section 4, we show our experimental results on sample applications. Related work is discussed in Section 5. Section 6 concludes.

2 Java Card bytecode

Java Card bytecodes are the form of instructions that JCVM understands and executes. Each bytecode instruction consists of one opcode and zero or more operands. The opcode represents the action JCVM needs to perform, while the operands act as arguments of the action.

JCVM is designed as a stack-based machine; thus, stacks are the center of all computations. In other words, the operand stack holds all temporary results of operations and the remains (local variables and method parameters) are stored in an array of local variables (so-called local variable table).

Among all Java Card bytecode instructions, there are three instructions that can increase heap cost. They are `new`, `newarray` and `anewarray` for creating a new instance of an object, an array of a primitive type and an array of object references, respectively. In addition, method invocation instructions (`invokeinterface`, `invokespecial`, `invokestatic`, `invokevirtual`) increase heap cost by the heap units that the invoked interface or the invoked method uses itself. The other instructions do not change heap space when they are executed.

Let $h(i)$ be the maximum heap cost allocated for instruction i. The formula to calculate $h(i)$ is presented in Figure 1, where:

- $M_{c,m,a_1,..,a_n}$ denotes the method m whose parameters are $a_1, .., a_n$ in class c, and $H(M_{c,m,a_1,..,a_n})$ denotes its heap bound function of method parameters.
- $\mathcal{M}_i$ denotes the set of methods that instruction i can invoke. Depending on the type of instruction i, we have:
  - $|\mathcal{M}_i| = 1$ if instruction i is a normal method invocation instruction (`invokeinterface`, `invokespecial`, `invokestatic`). In this case, only a specific method is called.
  - $|\mathcal{M}_i| \geq 1$ in case of dynamic dispatch where instruction i is `invokevirtual`. Depending on the runtime types of objects, different methods can be invoked in this case.
  - $|\mathcal{M}_i| = 0$ otherwise.

Figure 2 shows a method named `sample_method`. Depending on the input parameter x, the method allocates heap memory for creating an instance of `AClass` or an array of integer numbers or an array of `AClass`. JCVM compiles this source code to the bytecode stream in Figure 3. As we have seen, every instruction begins with an offset (0, 1, 3,..., 72, and 75) followed by the mnemonic of an opcode and operand values (if any). The offset of an instruction is the location where the instruction appears in the bytecode stream.

3 Our approach

First we present three steps of our approach: transforming the bytecode stream of a method into a CFG, transforming the CFG into a weighted DAG, and finding the single-source largest path in the weighted DAG. Then we present an effective algorithm for finding heap bounds of Java Card applets.

3.1 Transforming the bytecode stream of a method into a CFG

To represent all paths that may be traversed during execution of a method, we transform its bytecode stream into a CFG, a common technique in compilers [1] as well as in static analysis tools such as JULIA [14]. When constructing a CFG, many approaches [2, 3] usually store in each node a sequence of instructions that always execute sequentially, without any jumps or branching instructions. However, if we consider each bytecode instruction as a distinct node, it is easier to store and traverse the CFG. In addition, it makes our analyzer simpler, since we can omit the need in summarizing each block of sequential code that is required otherwise. Therefore, each node in our CFG corresponds to one instruction of the bytecode stream. We name each node its corresponding instruction's offset. If two instructions execute sequentially or one instruction performs a jump to another, we connect them by a directed edge. Our CFG contains an entry node e (corresponding to the entry instruction of the bytecode stream), internal nodes, and one or more exit nodes (representing `return` instructions).

Consider the bytecode stream of `sample_method` in Figure 3. Its corresponding CFG is shown in Figure 4. The entry node is 0 and the exit node is 75. Each directed edge in the CFG is formed by either two sequential instructions represented by a directed straight line, or a branch (jump) de-

$$h(i) = \begin{cases} Size(\text{the newly allocated instance}) & \text{if } i \in \{\texttt{new}, \texttt{newarray}, \texttt{anewarray}\} \\ H(M_{c_i,m_i,a_{i1},..,a_{in}}) \text{ with } M_{c_i,m_i,a_{i1},..,a_{in}} \in \mathcal{M}_i, |\mathcal{M}_i| = 1 & \\ & \text{if } i \in \{\texttt{invokeinterface}, \texttt{invokespecial}, \texttt{invokestatic}\} \\ max\{H(M_{c_i,m_i,a_{i1},..,a_{in}}) \mid M_{c_i,m_i,a_{i1},..,a_{in}} \in \mathcal{M}_i\}, |\mathcal{M}_i| \geq 1 & \\ & \text{if } i = \texttt{invokevirtual} \\ 0 & \text{otherwise} \end{cases}$$

Figure 1. Formula to calculate $h(i)$

```
public void sample_method(int x, int n,
                                  int m) {
  AClass aClass;
  int[] intArray;
  AClass[] aClassArray;
  for (int i = 0; i < x; i++) {
    switch (x % 3) {
      case 0:
        aClass = new AClass();
        break;
      case 1:
        intArray = new int[n];
        break;
      case 2:
        aClassArray = new AClass[m];
        break;
      default:
        break;
    }
  }
}
```

Figure 2. Sample method

noted by a curve arc. Three special nodes 40, 53, 61 are instructions that allocate memory for an instance of `AClass`, an array `int[n]` and an array `AClass[m]`, respectively. The edge from node 72 to node 3 creates a cycle.

For cycles, it is crucial to know how many times the cycles will execute. To do that, we analyze patterns of some common loops to identify the number of loops of these cycles, in this case the number of loops is `x`. We currently identify patterns of for loops whose number of iterations of loops are method parameters or constants, as we assumed in Section 1. It is not difficult to incorporate other kinds of loops once we know the patterns of the loop variable.

Furthermore, we also can detect if the loop counter variable (`i`) is changed inside the loop or not so that the memory bound is attached with an extra attribute, which indicates whether or not the found bound is good. When the loop variable is not altered inside a loop, the quality of our bound is good.

Note that in any valid bytecode streams, every node must be reachable from the entry node in the CFG of a method; otherwise, JCVM's verifier will reject the applet. Since we need to traverse all nodes in the CFG in our algorithm, the reachability property is required.

We denote our CFG by $G = (V, E)$ where each node $v \in V$ represents an instruction of the bytecode stream. We redefine $h(i)$ in Figure 1 as the maximum heap cost that the instruction corresponding to node i allocates (in this case, note that i denotes a node, not an instruction as in Figure 1). Since a CFG represents all paths that may be traversed during the execution of a program, the problem of computing heap bounds of Java Card applets can be restated as follows:

Given the CFG of a method in a Java Card applet, find a path p that starts at the entry node e and ends at one of the exit nodes such that the total weight of all nodes on p, $\sum_{v \in p} h(v)$, is maximum.

Note that the path p may traverse via a node more than once and each time p traverses a node, namely v, $h(v)$ is added to the total weight.

3.2 Transforming a CFG into a weighted DAG

Now we transform a CFG and its h function into a DAG, denoted by $G_\dagger = (V_\dagger, E_\dagger)$ and a weight function ω over the edges $E_\dagger$. Let u, v with indices range over nodes of CFG and DAG. For two nodes u, v that have a direct edge from u to v, we denote the edge by (u, v) and the weight of the edge by $\omega(u, v)$.

First, we assign each edge of G a non-negative weight by the following steps:

- Set $\omega(u, v) = h(v)$ for each $(u, v) \in E$.
- Create a source node s, add it to V, and connect it to the entry node e by a directed edge (s, e) with $\omega(s, e) = h(e)$.

Lemma 3.1 *The total weight $\omega(p)$ of a path $p = \{s, e, v_1, ..., v_k\}$ is the sum of the weight of its constituent edges.*

```
public void sample_method(int, int, int);
  Code:
    0: iconst_0
    1: istore 7
    3: iload 7
    5: iload_1
    6: if_icmpge 75
    9: iload_1
   10: iconst_3
   11: irem
   12: tableswitch{ //0 to 2
             0: 40;
        1: 52;
        2: 60;
        default: 69 }
   40: new #1; //class AClass
   43: dup
   44: invokespecial #2;
               //Method AClass."<init>":()V
   47: astore 4
   49: goto 69
   52: iload_2
   53: newarray int
   55: astore 5
   57: goto 69
   60: iload_3
   61: anewarray #1; //class AClass
   64: astore 6
   66: goto 69
   69: iinc 7, 1
   72: goto 3
   75: return
```

Figure 3. Bytecode stream

Proof After assigning each edge a non-negative weight, we obtain:

$$\omega(p) = h(e) + \sum_{i=1}^{k} h(v_i)$$

$$= \omega(s,e) + \omega(e,v_1) + \sum_{i=2}^{k} \omega(v_{i-1}, v_i) \quad \blacksquare$$

To generate a DAG from G, we need to collapse all cycles in G. A loop $\mathcal{L} = \{u_0, u_1, ..., u_q, u_0, u_1, u_2, v\}$ in G is formed by a sequence of nodes $u_0, u_1, ..., u_q, u_0, u_1, u_2, v$ and bounded by x ($x \geq 0$), which is a constant or a method parameter. In the loop, $u_0, u_1, ..., u_q, u_0$ form the loop body; u_1, u_2 check whether the loop condition is met; and v is the node that the control flow goes to when the loop exits. An example is the loop formed by the sequence of nodes 3, 5, ..., 72, 3, 5, 6, 75 presented in Figure 4.

We propose in Algorithm 1 a method to collapse cycles in G. Figure 5 is a graphical representation of the algorithm.

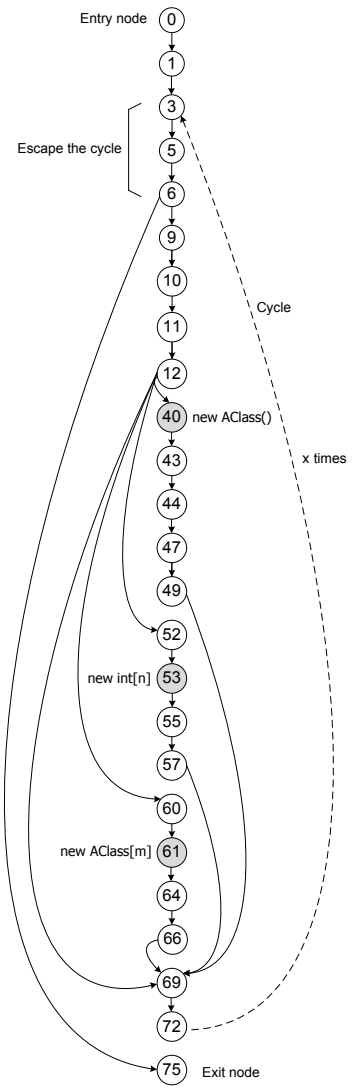

Figure 4. Control flow graph

Theorem 3.2 (Correctness of Algorithm 1) *Algorithm 1 does not change the total weight of every path that contains the collapsed cycle. It also does not change the reachability from the source node s to each node $v \in V$.*

Proof Let p be a path in G and suppose that p contains a cycle whose number of iterations is a constant or a method parameter, denoted by x ($x \geq 0$). We have the sequence of nodes of the cycle:

$$\underbrace{u_0, u_1, u_2, ..., u_q, ...}_{loop\ x\ times}, u_0, u_1, u_2, v$$

Consider a sub-path p_s from the first u_q to v. By Lemma 3.1, the weight $\omega(p_s)$ of p_s is

Algorithm 1 Collapsing a cycle in $G = (V, E)$

CollapsingCycle $(G, \mathcal{L})$:

1: $E \leftarrow E \cup (u_q, v)$
2: $\omega(u_q, v) \leftarrow \sum_{i=1}^{q} \omega(u_{i-1}, u_i) \times (x-1)$
3: $E \leftarrow E \setminus \{(u_q, u_0), (u_2, v)\}$

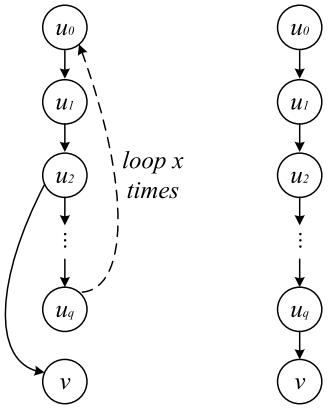

Figure 5. Collapse a cycle

$$\omega(p_s) = x \times \omega(u_q, u_0) + \sum_{i=1}^{q} \omega(u_{i-1}, u_i) \times (x-1) + \omega(u_0, u_1) + \omega(u_1, u_2) + \omega(u_2, v)$$

However, in the loop structure of bytecode stream, the weights of (u_q, u_0), (u_0, u_1), (u_1, u_2), and (u_2, v) are zeroes since they only represent jumps (from u_q to u_0 and from u_2 to v) or prepare values for checking the loop condition (the edge (u_0, u_1) and (u_1, u_2)). Thus, we obtain:

$$\omega(p_s) = \sum_{i=1}^{q} \omega(u_{i-1}, u_i) \times (x-1)$$

Since (u_q, u_0) and (u_2, v) are only used in the loop, we can safely remove them and create a new edge (u_q, v) with $\omega(u_q, v) = \omega(p_s) = \sum_{i=1}^{q} \omega(u_{i-1}, u_i) \times (x-1)$ without changing the total weight of the cycle. Thus, it does not change the total weight of every path that contains the cycle.

Although we need to remove (u_2, v) and (u_q, u_0) to collapse the cycle, the reachability from the source node s to node v is still preserved since a new edge (u_q, v) is created. Consequently, Algorithm 1 preserves the reachability from the source node s to each node in V. ∎

After collapsing all cycles in G, we produce a DAG $G_\dagger = (V_\dagger, E_\dagger)$ from G. Every edge $(u, v) \in E_\dagger$ has a non-negative weight $\omega(u, v)$ representing the maximum heap cost consumed when we traverse from node u to node v. The heap memory bound that we need to compute is now the weight of the largest path from the source node s to one of the exit nodes in $G_\dagger$. In the next subsection, we will introduce an algorithm to find the single-source largest path.

3.3 Finding the single-source largest path in the weighted DAG

This subsection shows how to solve the problem of finding the single-source largest path in a DAG generated from a CFG. We are given a DAG $G_\dagger = (V_\dagger, E_\dagger)$ in which each edge $(u, v) \in E_\dagger$ has a non-negative weight $\omega(u, v) \geq 0$. The weight of path $p_\dagger = (s, v_1, ..., v_k)$ from the source node s to v_k is the sum of the weights of its constituent edges:

$$\omega(p_\dagger) = \omega(s, v_1) + \sum_{i=2}^{k} \omega(v_{i-1}, v_i)$$

As mentioned in Subsection 3.1, every node in G must be reachable from its entry node. Accordingly, by Theorem 3.2, there is always a path from the source node s to node v in $G_\dagger$ obtained from G. We define the weight $\delta(v)$ of the largest path from the source node s to node v in $G_\dagger$ by

$$\delta(v) = max\{\omega(p_\dagger) : s \overset{p_\dagger}{\rightsquigarrow} v\}$$

Based on the DAG-shortest-paths algorithm in [6], we propose an algorithm presented in Algorithm 2 to find the largest path from the source node s to each node v in $G_\dagger$. For each node $v \in V_\dagger$, the algorithm maintains an attribute $F[v]$ - a lower bound on the weight of the largest path from the source node s to node v.

Algorithm 2 Single-source largest path in $G_\dagger = (V_\dagger, E_\dagger)$

DAG-largest-path $(G_\dagger, \omega, s)$:

1: perform a topological sort of $G_\dagger$
2: $F[s] \leftarrow 0$
3: **for** each node $v \in V_\dagger \setminus \{s\}$ **do**
4: $\quad F[v] \leftarrow -\infty$
5: **end for**
6: **for** each node $v \in V_\dagger$ taken in topological order **do**
7: $\quad F[v] = max\{F[u] + \omega(u, v) \mid (u, v) \in E_\dagger\}$
8: **end for**

In line 1, we perform a topological order $\mathcal{T} = \{v_1, v_2, ..., v_{|V_\dagger|}\}$ of $G_\dagger$ such that for each edge $(v_i, v_j) \in E_\dagger$, node v_i appears before node v_j in $\mathcal{T}$. Since Cormen et al. [6] presented a well-known topological sorting algorithm, we do not restate it for brevity.

Theorem 3.3 (Correctness of Algorithm 2) *DAG-largest-path algorithm runs on a DAG $G_\dagger = (V_\dagger, E_\dagger)$ with non-negative weight function ω and source s, terminates with $F[v] = \delta(v)$ for all $v \in V_\dagger$.*

Proof Since $F[v]$ is a lower bound of $\delta(v)$, $F[v] \leq \delta(v)$ for each node $v \in V_\dagger$. We need to prove that $F[v] = \delta(v)$ for all $v \in V_\dagger$. Suppose for the purpose of contradiction that v_q is the first node in $\mathcal{T}$ for which $F[v_q] < \delta(v_q)$. Since the largest path from s to itself is $\delta(s) = F[s] = 0$, we must have $v_q \neq s$. Therefore, the largest path from s to v_q contains at least two nodes. Let $v_p \in V_\dagger$ be the predecessor of v_q in the largest path. We have $(v_p, v_q) \in E_\dagger$ and

$$\begin{aligned} F[v_q] &< \delta(v_q) \\ &= \delta(v_p) + \omega(v_p, v_q) \end{aligned}$$

Since $(v_p, v_q) \in E_\dagger$, v_p appears before v_q in the topological order $\mathcal{T}$. Based on our supposition that v_q is the first node in $\mathcal{T}$ that $F[v_q] < \delta(v_q)$, we infer that $F[v_p] = \delta(v_p)$. Hence, we have

$$\begin{aligned} F[v_q] &< \delta(v_q) \\ &= \delta(v_p) + \omega(v_p, v_q) \\ &= F[v_p] + \omega(v_p, v_q) \end{aligned}$$

However, from line 7 in Algorithm 2, we obtain

$$\begin{aligned} F[v_q] &= max\{F[u] + \omega(u, v_q) \mid (u, v_q) \in E_\dagger\} \\ &\geq F[v_p] + \omega(v_p, v_q) \end{aligned}$$

Consequently, we derive the contradiction that $F[v_p] + \omega(v_p, v_q) \leq F[v_q] < F[v_p] + \omega(v_p, v_q)$. This contradiction completes the proof. We conclude that $F[v] = \delta(v)$ for all $v \in V_\dagger$. ∎

As stated by Cormen et al. [6], the topological sort in line 1 of Algorithm 2 can be done in $O(|V_\dagger| + |E_\dagger|)$ time. It takes $O(|V_\dagger|)$ time for the initialization from line 2 to line 5, and $O(|V_\dagger| + |E_\dagger|)$ time for computing the attribute F from line 6 to line 8. Therefore, the total running time of Algorithm 2 is $O(|V_\dagger| + |E_\dagger|)$, which is linear in the sum of the number of nodes and the number of edges in $G_\dagger$.

3.4 An effective algorithm for finding heap bounds of Java Card applets

We have shown that it is possible to compute heap bounds of Java Card applets. However, our approach needs to construct two extra data structures (CFG and DAG) and a transformation between them, which may be space-consuming phases. Since smart cards own very limited resources, we need to optimize our approach's steps to propose an effective algorithm that has a small memory footprint and runs fast enough to be suitable for smart cards. Consequently, we propose an effective algorithm for computing heap bounds of Java Card applets in Algorithm 3, which combines the three steps.

Algorithm 3 Heap analysis algorithm for Java Card applets

Input: The bytecode stream of method $M_{c,m,a_1,..,a_n}$
Output: The heap bound function $H(M_{c,m,a_1,..,a_n})$
Variables:
e: the entry node of the CFG
S: a stack holds expanded nodes for exploration
L: a list of nodes to which the chosen node connects
For each node u:
$F[u]$: the weight of the largest path from e to u
$pushed[u]$: Determine whether node u has been in S
$indegree[u]$: the indegree of node u
$goto\_jump[u]$: `goto` jump destinations of node i
$next[u]$, $if\_jump[u]$: the next node and the `if` jump target of node u

HeapBound():
1: push e into the empty stack S; $pushed[e] \leftarrow$ **true**
2: $F[e] \leftarrow h(e)$
3: $pushed[k] \leftarrow$ **false** for each node k where $k \neq e$
4: **repeat**
5: $i \leftarrow$ pop the node at the top of S; $L \leftarrow \emptyset$
6: **if** $\exists if\_jump[u]$ **then**
7: append $if\_jump[u]$ to L
8: **end if**
9: **if** $\exists next[u]$ **then**
10: append $next[u]$ to L
11: **end if**
12: **if** $goto\_jump[u] \neq \emptyset$ **then**
13: append all elements of $goto\_jump[u]$ to L
14: **end if**
15: **for all** $v \in L$ from left to right **do**
16: UpdateHeapBound(u, v)
17: $indegree[v] \leftarrow indegree[v] - 1$
18: **if** $indegree[v] = 0$ **and** $pushed[v] =$ **false then**
19: push v into S; $pushed[v] \leftarrow$ **true**
20: **end if**
21: **end for**
22: **until** $S = \emptyset$
23: **return** $max\{F[r] \mid r$ in the set of exit nodes$\}$

UpdateHeapBound(u, v):
1: **if** $pushed[v] =$ **true and** $u > v$ **then**
2: $F[v] \leftarrow F[v] + the\ number\ of\ iterations \times (F[u] - F[v])$ // A cycle is detected
3: $temp \leftarrow v$ // Move forward two nodes
4: **repeat**
5: $temp \leftarrow next[temp]$
6: $F[temp] \leftarrow F[v]$
7: **until** $\exists if\_jump[temp]$
8: $F[if\_jump[temp]] \leftarrow F[v]$
9: **else**
10: $F[v] \leftarrow max\{F[v], F[u] + h(v)\}$
11: **end if**

Our algorithm directly works with CFG to find the largest path from the entry node e to one of the exit nodes of CFG. We still use $F[v]$ as a lower bound on the weight of the largest path ending at node v. In the algorithm, we collapse a cycle by updating $F[]$ and moving forward two nodes to reach a node of `if` jump.

Regarding the data structures used in the algorithm, the most space-consuming one is for storing the CFG. We need three arrays to do this, including $next[]$, $if\_jump[]$ and $goto\_jump[]$ to represent the next (sequential) node, the `if` jump target and the set of `goto` jump destinations of each node, respectively.

The main idea of our algorithm is that we perform depth-first search (DFS) to traverse the CFG. Each time DFS is performed, a CFG's node that has zero in indegree is chosen. Since the chosen node does not have any ingoing edges, we use it to update the heap bounds of nodes to which it connects. When the heap bound of a node is updated, we reduce the indegree of the node by one. Each cycle is collapsed by moving forward two nodes to reach an `if` jump, where control flow leaves the cycle.

Since DFS is a recursive algorithm for traversing the CFG, it may result in stack overflow in Java Card environment. Therefore, we propose a non-recursive implementation of DFS in Algorithm 3 by using a stack S to hold expanded nodes for exploration.

According to Cormen et al. [6], the running time of DFS is proportional to the number of nodes $|V|$ plus the number of directed edges $|E|$ in the CFG. During CFG traversal, it takes $O(1)$ time to update the heap bound of a node. Hence, the total time complexity of our algorithm is $O(|V| + |E|)$. Our space complexity is also $O(|V| + |E|)$ for cost of storing all directed edges of the CFG and several other one-dimensional arrays used for this algorithm.

To illustrate how the algorithm works, we briefly run the algorithm with the CFG presented in Figure 4. The algorithm starts at node 0 whose heap bound is initially set to zero. There is no change of heap bound when going to nodes 1, 3, 5 and 6. After reaching node 6, a choice between the next node 9 and the conditional branch 75 appears. According to the order of CFG traversal in Algorithm 3, we visit node 9 first. When we reach node 12, there is a set of `goto` jump destinations $\{40, 52, 60, 69\}$. At node 40, the program executes an instruction that allocates memory for `AClass` object. The heap bound of node 40 is thus changed from 0 to $Size$(`AClass`). This heap bound value is preserved through nodes 43, 44, 47 and 49. Similarly, the paths from node 12 to nodes 52, 53, 55, 57 and to nodes 60, 61, 64, 66 set the heap bounds of node 57 and node 66 to $Size$(`int[n]`) and $Size$(`AClass[m]`), respectively.

Considering node 69, its heap bound is the maximum heap memory bounds of its ingoing nodes and equal to $max\{0,\ Size$(`AClass`$),\ Size$(`int[n]`$),\ Size$(`AClass[m]`$)\}$ = $max\{Size$(`int[n]`$),\ Size$(`AClass[m]`$)\}$. A cycle with `x`-time loops is detected when we reach node 72, which has a `goto` instruction jumping to node 3. It then escapes the cycle by moving forward two nodes to find an `if` jump, which locates at node 6. From node 6, an `if` jump is performed to go to the exit node 75 and update its heap bound to `x` $\times\ max\{Size$(`int[n]`$),\ Size$(`AClass[m]`$)\}$. This heap bound function of method parameters is also the maximum heap used by `sample_method` in Figure 2.

Table 1. Experimental results on Sun's applets

Sample	Size (byte)	NNodes	NEdges	Time (ms)
NullApp	916	21	18	3
Wallet	3277	64	68	7
RMIDemo	3067	111	107	11
ServiceDemo	2755	114	111	9
JavaLoyalty	2312	125	128	12
ChannelsDemo	7227	191	190	18
Transit	7765	269	276	21
Biometry	5419	271	259	19
Photocard	5383	330	340	19
Utilitydemo	10580	405	414	24
SecureRMIDemo	7411	417	424	24
JavaPurse	15489	465	479	26
JavaPurseCrypto	16713	524	538	31
SigMsgRec	4606	654	656	35

4 Experimental results

We have implemented a prototype tool that computes the heap bounds of Java Card applets using our proposed algorithm. The input of this tool is an applet and the output is the maximum heap space that its main method uses. This tool is tested on sample applets provided in Java Card Development Kit 2.2.2, and the obtained results are shown in Table 1. The first column is the name of tested samples. Each sample consists of several class files, and the total size of these classes is presented in the **Size**. The columns **NNodes** and **NEdges** contain the numbers of nodes and the numbers of directed edges of CFGs, respectively. The last column is the execution time (in milliseconds).

Since JCVM does not provide API to measure the heap memory that applets use at runtime, we do not have their actual heap bounds to compare with our results. However, by doing manual calculations, we found that the heap bound functions our tool returned are accurate.

To validate our tool by comparing its results to

Table 2. Experimental results on our samples

Sample	Size (byte)	NNodes	NEdges	HB (byte)	AHB (byte)	Time (ms)
MethodCalls	1423	41	44	504	504	4
Loops_Choices	1795	69	76	720	720	7
Tree	1551	57	63	8040	8040	5
LinkedList	1254	35	38	216	216	3
Stack	923	32	35	552	552	4
Queue	1162	37	42	432	432	3
Inaccuracy	1426	43	47	629	592	5

the actual heap bounds, we wrote several sample programs, which can be compiled by both JCVM and JVM. We first compiled them by JCVM, used our tool to find their heap bounds, and then used two methods `Runtime.getRuntime().totalMemory()` and `Runtime.getRuntime().freeMemory()` of JVM to measure the actual heap bounds they need[1]. Our samples use complex data structures like trees, linked lists, stacks and queues as well as control flow statements like choices, loops and method calls. We obtained the experimental results shown in Table 2.

In Table 2, the column **HB** represents the heap bounds that our algorithm returns. The actual heap bounds our samples allocate are shown in the column **AHB**. As we have seen, the values in the **HB** and the **AHB** are the same for most of our input samples. It means our algorithm can infer the least upper bound of heap cost used by Java Card applets in many cases. In case of the sample Inaccuracy that contains a change of method parameters in its loop body, our tool returns a non-optimal result.

We observe that the values of **NNodes** are nearly equal to ones of **NEdges** in both Table 1 and Table 2. The reason is that the number of jumps in a bytecode stream is very small in comparison with the total number of instructions. In other words, the number of nodes $|V|$ and the number of directed edges $|E|$ are almost the same. Therefore, the actual time and space complexity of our algorithm are $O(|V|)$, which is linear in the number of instructions.

For all samples in Table 1 and Table 2, our tool requires less than 3KB of space for the main memory. However, it is still preliminary and currently does not support complex features such as exceptions and subroutines. We will work on the features in the future. The experiments were carried out on an Intel P4 2.4 GHz with 1GB of RAM.

[1]See *Determining Memory Usage in Java* by Dr. Heinz M. Kabutz, available at `http://www.javaspecialists.co.za/archive/Issue029.html`

5 Related work

Many different approaches for heap space analysis have been proposed. Unnikrishnan et al. [16] propose a model analysis for inferring maximum size of live heap space of programs written in garbage-collected languages. Hofmann and Jost [9] point out a method to count the memory used for object allocation and deallocation in Java-like languages by extending the ideas of the static prediction in their previous work [8]. Chin et al. [5] present a type system to calculate memory usage of programs written in object-oriented languages. Since these techniques are performed at source-level, it is difficult to apply them to Java Card environment where source code is not available. Truong and Bezem [15] also give a type system for an abstract language. The type system can find a sharp upper bound of the number of instances but their type inference algorithm is polynomial.

Giambiagi and Schneider [7] analyze memory consumption by an algorithm that finds all potential loops and (mutually) recursive methods of Java Card applets. This algorithm improves the one presented in their certified analyzer [4] in terms of reducing memory footprint and dealing with several special features such as exceptions, subroutines, and virtual method invocations. As mentioned in their paper [7], this algorithm works directly with bytecode streams of programs without transforming them into an additional data structure such as CFG. However, the space complexity of the algorithm is higher than ours. Let N, N_U, N_C, and B be the number of instructions, the number of unconditional jumps, the number of conditional ones, and the number of bits for representing a pc-number (offset), respectively. As estimated by Giambiagi and Schneider, the complexity needed to compute $Loop$ of their algorithm is bounded by $N \times ((N_U + 2N_C + 1) \times B + 1)$, which is, unlike ours, not linear in the number of instructions. Also, they provide no experiments to support their ideas.

Albert et al. [3] extend their previous work [2], which focuses on computing cost analysis of Java programs, to propose an approach for heap space analysis of Java bytecode. In the paper [3], they base on heap space cost relations generated at compile-time to infer heap bound of

an input Java bytecode program. Their experiments, which were done on an Intel P4 Xeon 2GHz with 4GB of RAM, show that it takes hundred milliseconds for analyzing input programs whose sizes are a few kilobytes. The algorithm is not linear in complexity and take long time to run for the abstract-interpretation based size analysis. They take full bytecode instruction set but did not show how to validate their experimental results.

6 Conclusion and Acknowledgement

We have presented an approach to compute heap bounds of Java Card applets in linear time and low space complexity. We then have implemented a prototype tool that takes a set of Java Card classes together with a method name and prints out a heap bound function of method parameters that the method allocates on heap memory. This tool has been tested on various programs, and the obtained results show that our approach can find the least upper bounds of heap memory in many cases. The experiments also show that our tool not only runs fast but also has a small memory footprint.

Memory overflows of Java Card applets may result from either running out of heap space to allocate for new objects, or stack overflows. Therefore, to ensure that an applet is completely safe to run on cards, we plan to compute stack memory bounds and combine them with the heap usage analysis presented here to determine whether a Java Card applet can cause memory overflow exceptions.

This work was supported by the College of Technology, Vietnam National University, Hanoi, under the project QC.08.16. We also thank Van-Hung Dang, Viet-Ha Nguyen and anonymous reviewers for their comments on the earlier version of the paper.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In R. D. Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.

[3] E. Albert, S. Genaim, and M. Gomez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proc. of the 6th International Symposium on Memory Management*, pages 105–116, New York, NY, USA, 2007. ACM.

[4] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In *Proc. of 13th International Symposium on Formal Methods (FM'05)*, LNCS. Springer, 2005.

[5] W.-N. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory Usage Verification for OO Programs. In C. Hankin and I. Siveroni, editors, *SAS*, volume 3672 of *LNCS*, pages 70–86. Springer, 2005.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[7] P. Giambiagi and G. Schneider. Memory consumption analysis of Java smart cards. In *Proc. of XXXI Latin American Informatics Conference (CLEI 2005)*, page 12, Cali, Colombia, October 2005.

[8] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL '03: Proc. of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 185–197, New York, NY, USA, 2003. ACM.

[9] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis (for an Object-Oriented Language). In P. Sestoft, editor, *Proc. of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems*, volume 3924 of *LNCS*, pages 22–37. Springer, 2006.

[10] J. Hughes and L. Pareto. Recursion and Dynamic Datastructures in Bounded Space: Towards Embedded ML Programming. In *Proc. of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*, pages 70–81, 1999.

[11] S. Microsystems. Virtual Machine Specification, Java Card Platform, v3.0, Classic Edition. March 2008.

[12] S. Microsystems. Virtual Machine Specification, Java Card platform, v3.0, Connected Edition. March 2008.

[13] T.-H. Pham, A.-H. Truong, and N.-T. Truong. Computing Heap Space Cost of Java Card Applets. In A. Demaille, T. Cao, and B. Ho, editors, *Contributions to the 2008 IEEE International Conference on Research, Innovation and Vision for the Future in Computing & Communication Technologies*, pages 190–196, University of Science – Vietnam National University, Ho Chi Minh City, July 2008.

[14] F. Spoto. JULIA: A Generic Static Analyser for the Java Bytecode. In *Proc. of the 7th Workshop on Formal Techniques for Java-like Programs, FTfJP'2005*, Glasgow, Scotland, July 2005.

[15] H. Truong and M. Bezem. Finding Resource Bounds in the Presence of Explicit Deallocation. In D. V. Hung and M. Wirsing, editors, *Proc. of ICTAC 2005*, volume 3722 of *LNCS*, pages 227–241. Springer, 2005.

[16] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized Live Heap Bound Analysis. In *VMCAI 2003: Proc. of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 70–85, London, UK, 2003. Springer.

[17] P. B. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In P. Trinder, G. Michaelson, and R. Peña, editors, *15th International Workshop, IFL 2003, Edinburgh, UK, September 8–11, 2003.*, volume 3145 of *LNCS*, pages 86–101. Springer, 2004.

[18] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., New York, NY, USA, 2000.