Comparing An Object Oriented Runtime Complexity Metric To Depth First Search Complexity with Mobile Agents in a Mobile Autonomous Environment

J. McKinney Young¹, L. Etzkorn²

¹,²Department of Computer Science, University of Alabama in Huntsville, Huntsville, Alabama, USA
300 Technology Hall
University of Alabama at Huntsville
Huntsville, AL 35899
¹julienmckinneyyoung@mac.com, ²EtzkorL@uah.edu

Abstract - Software complexity metrics provide a way to describe and predict the resources needed to maintain and update code. Complexity metrics derived statically from source code describe the complexity of the software's code. Complexity metrics derived at run-time describe the complexity of the software's behavior. In this paper, we use an object oriented runtime complexity metric that describes the group complexity of mobile agents working atop mobile autonomous platforms. The agent/platform pairs use the Depth First Search algorithm to walk a graph in order to search for colored balls. We extend the experiment by studying the mobile agents/mobile platform's behavior complexity in three scenarios of differing difficulty. We then compare the run-time derived complexity metrics with the algorithms theoretical worst-case complexity estimation.

Keywords: Software Complexity, pathfinding, cellular automata

1. Introduction

Software complexity metrics traditionally have been computed statically using the software’s source code. This provided a descriptive means of evaluating the entire collection of code. Recently, the capture of dynamic complexity metrics of code during execution has allowed study of software behavior at runtime [3,4,5,6,7,8,9]. This view into the actual behavior of the executing code allows the software professional to better understand and predict the software’s actions. Runtime complexity metrics are especially useful with object oriented code because dynamic behavior such as inheritance and polymorphism can be hard to predict from a static source code-only review [9].

In this paper, we use an object oriented runtime complexity metric that measures the group complexity of mobile agents as they work atop mobile autonomous platforms to complete a basic walk and search task in a simulated environment. The compiled complexity data highlights the surprising ways in which actual complexity metrics gathered during execution in real world scenarios differs from theoretical static code complexity estimations.

2. Related Work

Software professionals have long noticed the relationship between a software applications code complexity and the resources required to adequately test and maintain it. The higher the complexity, the higher the level of resources required. T.J. McCabe addressed the issue in 1976 with his article, “A Complexity Measure.” In this article, McCabe presents a way to measure code complexity statically by counting all possible paths of execution that could potentially be exercised. McCabe specifies that complexity depends not on the size of the program but only on the decision structure of a program. [10]

Another method of measuring code behavior was proposed by Chidamer and Kemerer in 1994 in [1]. They proposed that coupling between objects (CBO) was a useful object-oriented metric. Coupling is defined as the manner and degree of relationship between software modules - when methods in one class use methods or instance variables defined in another. They found that the higher the amount of coupling in the code, the higher the amount of complexity is present [1].

In 2005, Mitchell and Power extended Chidamer and Kemerer’s metric, CBO in [11]. They applied it at run-time to examine coupling behavior of the actual running code. They believed that the static CBO measure did not provide an exact look at what really happened when the code actually executed. “…CBO cannot capture all the dimensions of object-oriented coupling because features of object-oriented programming such as poly-morphism, dynamic binding and inheritance render CBO imprecise in evaluating the run-time behavior of an application.” [11] Their studies showed that objects from the same class can behave differently at runtime “from the point of view of coupling” than could be described from a static analysis of the source code [11].

Five years later, Mathur and Keen proposed a different metric to study run-time complexity. Just as Mitchell and Power extended Chidamer and Kemerer’s CBO to the run-
time environment, Mathur and Keen extended McCabe’s idea of cyclomatic complexity to the run-time boundary. They introduced a metric that is based on the number of decision points evaluated by the running code [9]. The authors make the distinction between “potential” complexity - the code at compile time - and “actual” complexity - the code that actually executed. They calculated the runtime complexity of objects by counting the number of decision points that were accessed at runtime. Selection structures like if...then, if.else...then, case statements and do...while, for...while structures added to the count as executed per object. Each decision point was counted once - iterative calls were not counted. The authors, like McCabe, write that the decision points alter the control flow of the program and can affect the complexity of the running code. The authors then compared the count-generated metric of complexity against the complexity ratings given by a panel of experienced programmers who analyzed the subject source code and static complexity measures derived from the source code. They, like Chidamber and Kemerer, found that the runtime metric measured a “different aspect of complexity” than did static complexity metrics [9].

Desouky, after Mathur and Keen, examined runtime complexity expressed by decision points. Desouky extended Mathur and Keen’s decision-count metric to include iterative decision calls [5]. The author studied the metric using an open source application Rhino 1.7R4. The results were compared to bug reports because software quality is inversely related to the number of bugs found [5]. A code module that has a very low number of bugs found is considered to be of a higher quality and contain lower complexity than a module of that is found to contain a large number of bugs. The study found that the derived complexity values correlated with the number of bugs found in the code modules under test [5].

3. Background

In [6] and [7], Keen takes McCabe’s idea of complexity as a function of decision structure and combines it with the idea that run-time metrics could give a more accurate view of complexity as represented by the running code. This represents a shift in focus away from static code complexity to dynamic behavior complexity. Keen introduced the metric KeenintRM to describe the amount of complexity in a program as it executed. It is a descriptive metric that allows the software professional to compare different approaches and implementations for accomplishing the same task. The better approach is the approach that accomplishes the task with the least amount of complexity [6].

KeenintRM = 1/complexity for some particular task [6].

The metric is very similar to a runtime version of McCabe’s cyclomatic complexity [6]. It is the sum of decision points encountered by the running code plus one is added at every method invocation even if no decision structures are encountered. This represents the idea that some level of complexity is represented by the activity of processing the method even without the greater work of processing a control structure.

In [6], Keen compares the behavior of two programming approaches by implementing the approaches as agents - static-vs-mobile - running on simple mobile systems fitted with infrared sensors. The mobile systems had a task to perform - to move about a grid in search of three colored balls in a particular order. The “agents” directed the mobile systems and processed the “visual” (infrared) data that the mobile systems perceived.

In the Three-Mobile System Scenario, which compared the activity of three mobile systems with static agents and three mobile systems with mobile agents, starting at the same start points searching for the same balls in the same location, the data suggested that the mobile agent approach showed lower complexity than the static agent approach [6]. Keen’s work also demonstrated that the mobile agent approach showed greater resiliency in the face of obstacles than the static agent approach. When one mobile system got stuck in a corner and couldn’t turn enough to remove the wall/obstacle from its view, the mobile agent resident on the “stuck” mobile system was able to jump to another mobile system (that had already completed its agent’s task) and continue searching for it’s colored ball. In the static agent approach, both the “stuck” mobile system and its resident static agent where unable to complete their task.

We implemented Keen’s experiment in a simulation. The simulation was validated against the original data. The mobile platforms use the DFS pathfinding algorithm to walk the graph. In this paper, we discuss our findings when we used the simulation to compare the run-time complexity of the mobile agents/mobile platform teams walking a graph with DFS’s Order of Complexity. The comparison provides additional data to describe the aspects of complexity that the KeenintRM metric captures.

4. Case Study

4.1. Simulation Description

The simulation is implemented in the form of a cellular automaton (CA) using Microsoft Access. The researcher may observe the progress of the running code (written in Visual Basic) through the graphical user interface (GUI). The persistent datafiles in which the rules, the geography, and the intermediate and final results are stored are used to derive the results of each run.

The formal definition of a CA is expressed by four things: the array dimensionality d, the set of states (the number of cells) S, the neighborhood vector (the definition of what is a neighborhood) N, and the local rule (how local neighbor states are evaluated) f [2].

CAi = (d, S, N, f).

At time quanta zero, an initial start state is assigned to each cell. The collection of all cells’ states is called the
configuration of the cellular automaton at that time. The passing of each time quanta causes the states of all the cells to change (according to the cellular automaton’s rule) and a new configuration describes the cellular automaton at that instant in time.

The CA expresses a 2-dimensional array(d) representing geographic direction (x,y). The simulation models mobile systems that are capable of moving only “wheels on the ground.” There is no vertical component (z) to their movement. The cells (S) of the automaton comprise a 7x7 grid which has a total of 49 possible location states. The neighborhood vector (N) is made up of 4 possible cells at the North, East, South, West sides of the cell under analysis. The array of local rules are as follows:

Rules of CA behavior
1. There is a central clock.
2. Time only flows in one direction - forward.

Rules of Mobile System Behavior
1. Mobile Systems can only do one activity at each time tick:
   a. Start
   b. GoToSleep
   c. FindObject
   d. Walk
   e. Turn
   f. Lose Agent
   g. Receive Agent
2. Mobile Systems can only turn 90 degrees in one time tick.
3. A mobile system is in “Fallback” mode and the already-visited node is the just-prior node.
4. Only one mobile system at a time may occupy a node.
5. A mobile system will go to “Fallback” state on the 4th time tick after it has turned a complete circle (4 turns in 4 consecutive time ticks) and has not been able to move out of its current node.
6. A mobile system will “go to sleep” on the 4th time tick after it has turned a complete circle in the “Fallback” mode (4 turns in 4 consecutive time ticks) and has not been able to move out of its current node to its “origin” node (the node it occupied right before it moved to its current node.)

7. Each Mobile System starts the simulation with one assigned Agent.
8. Mobile Systems possess the attribute of “handedness” (left or right) - when a mobile system turns, it will turn in the direction as specified by its “handedness” setting.
9. Mobile Systems “find” a ball by stepping into a cell that contains a ball.
10. Mobile Systems walk a self-selected path using the DFS algorithm.

Rules of Agent Behavior
1. Agents can only do one activity at each time tick:
   a. DoNothing
   b. GoToSleep
   c. FindObject
   d. Jump
2. When a mobile system dies so also does its agent.

3. An agent is assigned a particular colored ball (red, blue, green) to find.
4. It can only “find” that one ball.
5. The agent “checks” if the ball is in the cell during the same time tick that the mobile system has moved into a new cell.
6. After the agent finds its ball, it goes to sleep.
7. A different agent who is still active (has not found it’s ball) can jump to a mobile system with a “sleeping” agent.
8. A different agent who is still active (has not found it’s ball) can jump to a mobile system without an agent.
9. A “jumping” agent may only leave one mobile system and arrive at another mobile system in one time tick. (The mobile system will “wake up” in the next time tick.)

4.2. Depth First Search (DFS) Description

DFS is an algorithm for traversing a graph. It was described in the 19th century by Charles Pierre Trémaux, a French mathematician [12]. In the algorithm, the search begins at some arbitrary node of the graph and “walks” or explores down as far as possible along every branch before backtracking. When an obstacle is encountered or the branch ends, one “falls back” to the next higher node and selects another node/branch down which to explore. In other words, the algorithm visits children nodes before it visits sibling nodes [12]. It’s order of complexity is defined as $O(|V|+|E|)$, where $V$ is the number of vertices (or nodes) in the graph, and $E$ is the number of edges. As an actor “walks” a graph using the DFS algorithm, the DFS order of complexity represents the fact that the actor must make decisions at each node about where to go next.

4.3. Scenario Design Description

The mobile agent/mobile platform simulation is modeled after a real-world case study using mobile agents running atop real robots who walked a graph taped on a floor in a real building. The real-world scenario forced the agent/robot teams to interact with an uncertain environment that included each other, temperature and changing light conditions based on the position of the sun. The advantages of this approach are that the agent/platform pairs represent actual entities who must move “wheels on the ground” in linear time, our hypothesis was that the agent/platform pairs would...
experience more complexity than what a theoretical estimation would describe.

For that purpose, we compare the values of KeenintRM collected for the group of agent/platform pairs while traversing a graph with the theoretical values proposed by the DFS algorithms order of complexity for the same traversed graph.

The three agent/platform pairs walk autonomously the graph searching for colored balls. The presence of three pairs interjects uncertainty with regards to path openness. Even though the mobile agents do check if the mobile platform has “found” the appropriate ball, that decision structure is not counted for this experiment. In the three mobile system/mobile platform teams, the start states are described below.

**Scenario 1 - Length: 120 time quanta**
B1: red, at 2,2  
B2: green, at 4,5  
B3: blue, 1,6  
R1, agent1 - read ball, 7,1, left-handed  
R2, agent2 - green ball, 7,2, right-handed  
R3, agent3 - blue ball, 1,7, right-handed

**Scenario 2 - Length: 120 time quanta**
B1: red, at 2,4  
B2: green, at 5,3  
B3: blue, 6,6  
R1, agent1 - read ball, 7,1, left-handed  
R2, agent2 - green ball, 7,2, right-handed  
R3, agent3 - blue ball, 1,7, right-handed

**Scenario 3 - Length: 120 time quanta**
B1: red, at 1,4  
B2: green, at 1,6  
B3: blue, 7,3  
R1, agent1 - read ball, 7,1, left-handed  
R2, agent2 - green ball, 7,2, right-handed  
R3, agent3 - blue ball, 1,7, right-handed

As Table 1 describes, the placement of the balls and the start location of the mobile platforms in Scenario 1 makes this scenario the most arduous for three agent/platform pairs. They visit 81 nodes in the 120 sec experiment. (And then revisit 9 of those nodes in “fallback” mode.) Comparing Scenario 2 data with Scenario 1 data, one sees a 24% increase in the number of nodes visited from Scenario 2’s 65 nodes to Scenario 1’s 81 nodes but there is a 54.5% increase in the KeenintRM value. The difference between Scenario 2’s activity and Scenario 3’s activity is more dramatic. The number of nodes visited more than doubles from Scenario 2 to Scenario 3 but the KeenintRM value increases by a factor of 7. In comparison, the Order of Complexity given by the DFS algorithm shows only an arithmetic increase between the three scenarios related to the number of nodes and edges visited.
5. Conclusion and Future Work

Implementing the DFS algorithm as a path finding strategy in a simulation where the moving objects are specifically designed to mimic real, autonomous, mobile platforms highlighted several critical differences between how the algorithm models movement and time and how real objects experience movement and time as described by KeenintRM. The Order of Complexity given by the DFS algorithm shows only an arithmetic increase between the three scenarios related to the number of nodes and edges visited while the KeenintRM values show a greater than arithmetic growth between the scenarios. In Scenario 1, the agent/platform pairs visit a total of 90 nodes in the 120 second experiment (81 nodes in normal mode and 9 in “fallback” mode.) Comparing Scenario 2 activity with Scenario 1 activity, there is a 24% increase from Scenario 2’s 65 nodes to Scenario 1’s 81 nodes visited in normal mode. However, there is a 54.5% increase in the KeenintRM value from Scenario 2 to Scenario 1. Comparing Scenario 3 with Scenario 1 shows that the agent/platform pairs visited almost 2 1/2 times more nodes in “normal” mode in Scenario 1 then they did in Scenario 3. KeenintRM, however, shows an almost 12-fold increase between Scenario 3 and Scenario 1.

What is the difference between what the DFS order of complexity represents and what the KeenintRM captures? First and foremost, in the DFS algorithm, an actor will “fallback” to its prior node instantaneously and effortlessly. Revisiting a “fallback” node does not add any complexity to the DFS algorithm’s estimation. As the algorithm is written, an actor who is blocked from moving forward to the next unvisited node will drop out of the “next step” subroutine and return to the calling routine with all position data either updated or still available. DFS implemented recursively with a real mobile platform has many more actions to complete before it is back at its prior location. First, the real object does not fly. It must move wheels on the ground. And it must be able to “see” where it's going. That means that platform may be a more effective rule for the mobile platform to move into the now open, unvisited, adjacent node and skip the remainder of the “fallback” exercise. As DFS does not include interim steps between the decision to fallback and the actual fallback arrival at the calling routine, there is no mechanism to describe interim steps and how best to handle them in terms of the traversal exercise.

The complexity metric KeenintRM captures more decision making with regards to WHY a node may be ineligible to visit than the theoretical DFS algorithm does. In theoretical DFS, each node is checked once to see if it is open to be visited. In implemented DFS, however, an additional nested case statement is added to the algorithm to handle node evaluation when the platform is in “fallback” mode rather than “normal” mode. The additional condition checks represent decisions made by the platforms executing code and add to the complexity value. The case study shows that intermittently accessing different levels of nested control structures will affect the complexity metric in a way that is not represented by theoretical DFS’s time complexity metric of $|V| + |E|$. Theoretical DFS time complexity value should increase at an arithmetic rate with the addition of nodes and edges to the walked path. The KeenintRM metric shows a greater than arithmetic increase in complexity with the addition of nodes and edges to the path as shown between Scenario 2 and Scenario 1.

Additional study could further discriminate between “normal” mode complexity values and “fallback” mode complexity values. An interesting question would be, is the greater than arithmetic growth in KeenintRM due solely to “fallback” mode behavior? Or does the metric represent additional actual traversal exercise complexity that is not captured in the theoretical DFS estimation?

Another logical difference between theoretical DFS and implemented DFS is that when the platform turns to “fallback”, it still has the capability to evaluate other open nodes. What if the platform must turn 180 degrees in order to “fallback” to its prior node? That would take it two time ticks and two turns. Once the platform turns 90 degrees and is facing an adjacent node (that was evaluated as occupied several time ticks ago - and thus ineligible for moving to), what if that adjacent node (which is not the fallback node) is now open? In real life, if an actor were traversing a graph, it may be more effective for the mobile platform to move into the now open, unvisited, adjacent node and skip the direction. It must move wheels on the ground. And it must keep in mind that the platform has direction. It must then turn in order to evaluate possible “next steps” from the current-on-the-way-to-the-“fallback” node. At each time interval during the “fallback” exercise, the platform makes decisions on how to move (turn or walk) and what is the correct node to walk to (it can’t walk to just any open node - it’s in fallback mode, so it can walk only to its preceding node.) Once, the platform does determine that it is facing the proper fallback node, it then must decide if the node is empty. These determinations are represented by decision structures in the running code that are NOT part of the DFS algorithm. As the data in Table 1 shows, each node visited (or revisited), regardless of mode, adds to the complexity of the system.
6. References


