An Analysis of Black Hole Search Algorithms in Asynchronous Rings

Sean Floyd[1] and Jonathan Guillotte-Blouin[2]

[1] sfloy029@uottawa.ca, (6778524)
[2] jguil098@uottawa.ca, (7900293)

Abstract. This report analyzes five black hole search algorithms in the ring, using the asynchronous white-board model. Our experiments show that the actual time and move complexities are in line with the theoretical values presented by their authors. No major issues were encountered in the undertaking of this project.

1 Introduction

1.1 The Problem

Black hole search (*BHS*) is a multi-agent graph exploration problem in a dangerous graph. Agents are tasked to discover the location of a black hole in a graph. A black hole is a node in the graph that destroys any incoming agent without any evidence of it doing so.
In order to solve the *BHS* problem, at least one agent must survive the search process and know the location of the black hole in the graph.

The algorithms presented in section 1.2 solve the *BHS* problem in a specific topology: an asynchronous ring. Additionally, they all contribute to the body of work dealing with the asynchronous white-board model, in which each node in the graph provides a bounded shared-memory area dedicated to the communication between agents. An agent can only read or write, in mutual exclusion, from/on the white-board of the node it is located at. Finally, each agent also has access to a bounded memory area of its own, knows the topology of the graph, knowledge of n, has local orientation and a sense of direction.

In order to compare various *BHS* algorithms, the following measures are used to evaluate their cost: the sum of the number of moves performed by each agent, the number of agents used, and the quantity of time units (in ideal-time) taken to discover the location of the black hole and return to an agreed location.

1.2 Algorithms

We will not present the algorithms listed below in detail, as they have been presented in class. However, we have noted their complexities.

N.B.: n represents the number of nodes in a ring.

Dobrev et al. present, among others, the following two algorithms, in [1]:

- *OptTime*: $n-1$ co-located agents are tasked to discover the location of the black hole in $(2n-4)$ time units,
- *Divide*: two co-located agents are tasked to discover the location of the black hole using $2n * lg(n) + O(n)$ moves.

Balamohan et al. present three algorithms in [2] that they compare to *OptTime* in [1] while still using $O(n^2)$ moves

- *OptAvgTime*: optimizes average-time complexity by doubling the number of agents required to $2(n-1)$,
- *Group*: slightly enhances the average-time complexity while still using $(n-1)$ agents,
- *OptTeamSize*: uses the optimal (2) number of agents while still running in linear-time in average and worst-cases.

1.3 Outline

In section 2 we will go over our experiments and their results. Then, in section 3, we will discuss the main issues we have encountered. Finally, we will conclude our report in section 4

2 Implementation and evaluation

2.1 Implementation

The *Go* programming language developed by Google was chosen for the implementation of our selected algorithms, since it is intrinsically designed for concurrent programs. The concurrent element provided by Go is crucial for emulating these asynchronous algorithms as the distributed agents execute in parallel in the algorithms.

We created a small package called "bhs" in which we defined the properties and methods of a Ring network (ring.go), of an agent (agent.go), of a Node (node.go) and one file for each of our algorithms. Our main program (main.go) launches our simulation and prints the statistics. Finally, we have a test file (main_test.go) in which we defined test functions, as well as benchmarking functions.

There are three possible things to run here, assuming Go 1.5+ is installed and that you are at the root of the project in the command-line:

1. Run the algorithms and print evaluation measure statistics: `go run main.go` To see the flags available, add the flag `-help` after.
2. Benchmark the algorithms: `go test -bench=.`
3. Test the algorithms: `go test` or `go test -v` for more details

2.2 Evaluation

The benchmarks are very straightforward: they allow us to see how many nanoseconds each algorithm takes to complete on average. Each algorithm is run in a benchmarking function which repeatedly runs an algorithm until the time it takes to execute converges on a value. This value is printed with the number of runs it took to converge on it. In the benchmarking function, the black hole is always located at the last node in the ring.

We also wrote a ruby script to run the benchmark multiple times for different ring sizes, as we could not do it within the go test file. The ruby script asks the user for a starting ring size, the increment and the maximum ring size to reach. The script then uses a regular expression to edit the go test file inplace to replace the previous ring size with a new ring size, then executes the benchmark. It then uses another regular expression to scan the printed results from the benchmark to obtain the values for each algorithm. After running a benchmark for each increment between the starting and ending ring size, the results are written to a comma-separated-value *(.csv)* file. We then used Google Sheets to create some charts.

We also wanted to obtain the actual costs for the time and move cost measures. In order to get these back after running an algorithm, each agent needs to return the number of times it moved from one node to another. To compute the move cost for an algorithm, we simply sum the number returned by each agent.

To compute the cost for time, we decided to simply return the maximum value returned by an agent, giving us the ideal time. However, each algorithm has variations, therefore we sometime need to get the maximum returned value of pairs of agents, or in the case of the tie breaker group, we need to add $2i$ to the value returned because it is not launched at the same time as other agents.

The main function is comprised of two nested loops: the first loop runs once per algorithm, and the second loop contains $(n-1)$ iterations, to run the algorithm for every possible location of the black hole. At the end of each iteration of the first loop, we print the *minimum*, *average*, and *maximum* values of the time and move cost measures returned when searching for a black hole at each of the $(n-1)$ possible locations. At the same time, we make sure that the ID of the black hole found by the agent(s) is indeed the right one.

2.3 Results

Figure 1 shows the results obtained from running the benchmarks. We noticed that the average time it took to run each algorithm was far different from what we expected from the theoretical average-case or worst-case time complexities and did not provide any means to verify the move complexity. For example, *OptTeamSize*, which is supposed to be the slowest of all algorithms in terms of

time complexity is the fastest of the five by a large margin. Additionally we can see from figure 2 that Divide and OptTeamSize appear to run in linear time, whereas the other algorithms seem to run in polynomial time. This could be because our simulation can't emulate perfectly the environment that the algorithms are supposed to run in; in an ideal world, all agents are supposed to run in parallel independently from the other agents, but that is not possible on a single machine with very few cores as we do. A limited number of agents can truly run in parallel on a computer, so having only two agents like in *OptTeamSize* is much different than having a linear amount of agents; these 2 agents (or threads in the case of the simulation), even though their underlying algorithm is technically slower, will run much faster because they won't have to compete with as many threads for resources, wait for context switches and other OS-level realities. It was therefore necessary to ignore these results and rather analyze the number of moves to measure the time complexity.

Indeed, looking at table 1, which shows the computed statistics, we can see that each algorithm returns a value that fits within the margins of its average and worst-case in both the time and move complexities. That is, except for *OptTeamSize*, for which it is nearly impossible to reach the worst-case complexities because they can only be achieved by forcing delays on every link that the "big" agent traverses, which we did not have time to do.

It should be noted that the *minimum* and *maximum* values shown do not represent the best and worst-case, only that those were the extreme values found over the executions to find the black hole at every possible position.

Finally, all experiments were run on a laptop running macOS, with 16GB of RAM and a 2.2GHz Intel Core i7-4770HQ.

3 Discussion of issues

For the most part, we did not encounter any major issues. However, we dealt with our fair share of bugs regarding concurrency control.

Algorithms *Divide* and *OptTeamSize* were the source of many bugs mainly due to their use of white-boards, compared to the other three algorithms. Managing the logic of mutual-exclusion and timing the locking/unlocking can be cumbersome and complex at times to wrap our heads around.

Another issue we had was with the benchmarking, as it did not seem to produce the results we expected, or even produce a logical ranking of the algorithms by running time. This problem led us to pursue our research by counting the number of moves of the agents, as discussed previously.

4 Conclusion

While the results are in line with what we expected, further work can be done to better control the link delays so that they are not as dependent on a given go-routine's priority or the computer's CPU load. By adding random timeouts, we could also emulate better the asynchronous nature of the problem and analyze some test cases that never occurred, for example the "big" agent (discussed previously). There are however limits that cannot be overcome with such a simulation: running thousands or hundreds of thousands of agents simultaneously can not be emulated perfectly unless the means to recreate this environment are directly proportional to the objectives envisioned. More experiments should be undertaken to verify that algorithms with lower time complexities have smaller execution times, on a system that would not be held back by hardware limitations.

References

1. S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro, "Mobile search for a black hole in an anonymous ring," *Algorithmica*, vol. 48, no. 1, pp. 67–90, 2007.
2. B. Balamohan, P. Flocchini, A. Miri, and N. Santoro, "Time optimal algorithms for black hole search in rings," *Discrete Mathematics, Algorithms and Applications*, vol. 3, no. 04, pp. 457–471, 2011.

Tables and Figures

Algorithm		**Time**			**Move**	
	min	avg	max	min	avg	max
OptAvgTime	98	147	196	9702	9702	9702
OptTime	196	196	196	196	6512	9700
OptTeamSize	245	709	782	490	786	994
Divide	245	1048	1382	392	1121	1425
Group	100	159	198	2596	6927	8546

Table 1. Statistics after running each algorithm for every black hole position possible in a ring of size n=100

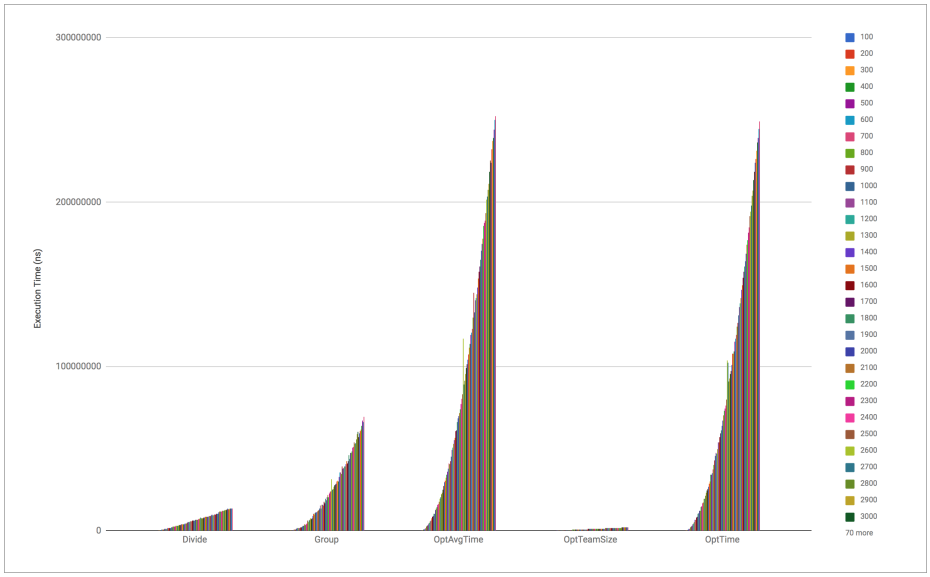

Fig. 1. Execution times from benchmarks, grouped by algorithm (from results.csv) with a ring size of 100 to 10000

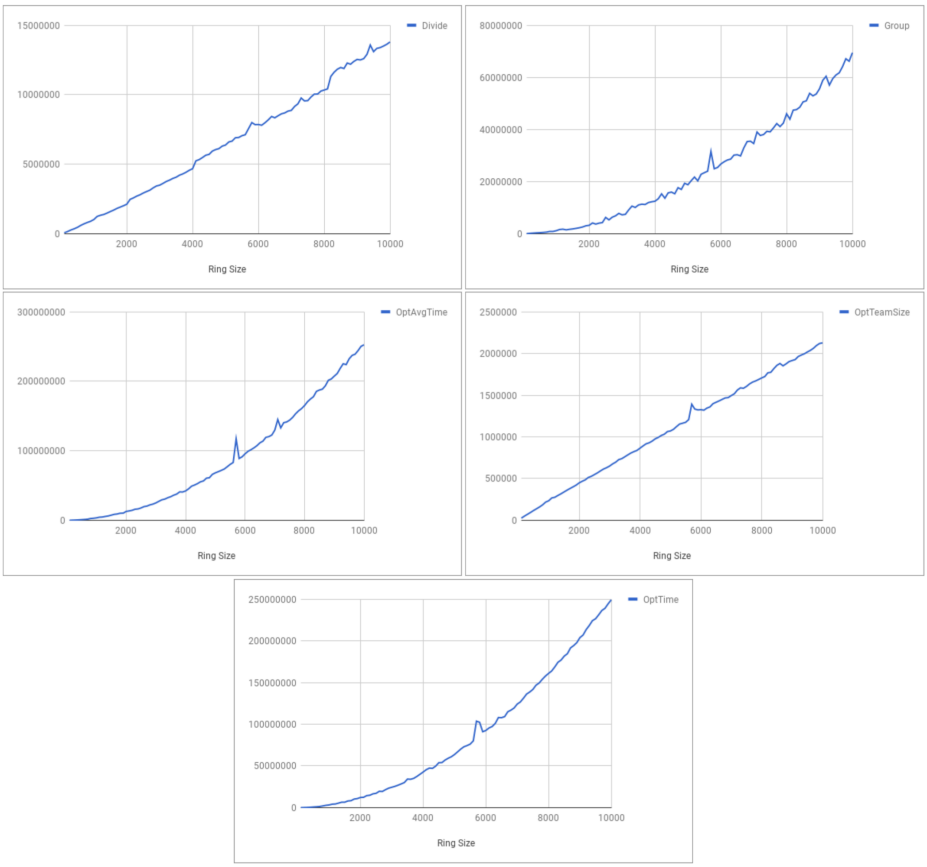

Fig. 2. Line charts of results.csv