CSI 5137A – AI-enabled Software Verification and Testing
Assignment 2, Autumn 2020

Due date: Nov 16th, 2020

The assignment files have to be submitted on BrightSpace by Nov 16th, 2020 midnight.

1 Background

In this assignment, you investigate and extend an implementation of a search-based test input generation approach. We use the AVMFramework (`http://avmframework.org`) for this assignment. The actual implementation of the framework is available at this github project page (`https://github.com/AVMf/avmf`). AVMf is a framework and Java implementation of the Alternating Variable Method (AVM), a heuristic local search algorithm that has been applied to several important software engineering problems. The framework includes several example engineering problems that can be automated using AVM. You can find the problem instantiations in the `examples` directory of the project. Among the different examples in the `examples` directory, we are interested in the test input generation problem implemented in `GenerateInputData.java`.

To become familiar with Alternating Variable Method (AVM), read the github project page and clone/download the project, and further, read the slides (avm-test generation.pdf) and the paper (avm-paper.pdf) that are attached to this assignment handout. In addition, to understand how the test input generation works, you can try to execute `GenerateInputData.java` for the three example input Java classes provided in the `inputdatageneration` directory: `Line`, `Triangle` and `Calendar`. Note that the instrumented versions of these three Java classes are provided in three classes named respectively `LineBranchTargetObjectiveFunction`, `TriangleBranchTargetObjectiveFunction` and `CalendarBranchTargetObjectiveFunction`. The instrumented classes help compute the standard fitness function for branch coverage for any branch in the original class. Further, the test inputs for these classes are, respectively, specified in `LineTestObject`, `TriangleTestObject` and `CalendarTestObject`. The information on how `GenerateInputData.java` can be used to generate test inputs for these three Java classes are available on the github page (`https://github.com/AVMf/avmf`) under the **Test Data Generation** section.

2 Your Tasks

For this assignment you need to answer the following questions and extend the above framework with a new search algorithm. The answer to questions should be submitted as a .pdf file. The required deliverables for the implementation task are specified below.

2.1 Question 1.

How does the Alternating Variable Method (AVM) work? How is this algorithm different from the Hill Climbing or Simulated Annealing algorithms we discussed in the class? Please try to describe the main ideas and the working of AVM in less than one page (using a few paragraphs).

2.2 Question 2.

Show the control-flow graph for the classify method of the Triangle class and label each branch with an id such that your branch ids are consistent with those specified in `TriangleBranchTargetObjectiveFunction`. Note that each branch id should be a number followed by T (for the true branch) or F (for the false branch).

2.3 Question 3.

Provide the smallest test suite that achieves branch coverage for the classify method of the Triangle class. For each test case in your test suite, specify the branches covered by the test case. Does this test suite achieve statement coverage as well?

2.4 Question 4.

Provide a test suite for the intersect method of the Line class that achieves statement coverage, but not branch coverage. Explain which branches of the intersect method are not covered by your test suite.

2.5 Question 5.

The fitness function to compare different candidate test inputs is defined based *approach level* and *branch distance* metrics. Explain how these two metrics are combined to compare different test input vector candidates and specify in which method of which Java class in the AVMf framework, this comparison is implemented.

2.6 Question 6.

As discussed in the class, one way to combine approach level and branch distance metrics is to first nomralize branch distance and then add it to approach level. Why do we need to normalize the branch distance metric but not the approach level?

2.7 Implementation Task.

Implement one of the local search algorithms we learned in the class (e.g., Hill Climbing and Simulated Anneal) in the AVMf framework and modify `GenerateInputData.java` to use your new local search algorithm to generate test input data. Do not modify the input interface of `GenerateInputData.java` though. That is, your modified version of `GenerateInputData.java` should still generate test inputs for branch "1T" of Triangle by passing `1T Triangle` to it as input. Note that you do not need to use the optional `search` as input to determine the type of search algorithm since `GenerateInputData.java` should call the new search algorithm that you have implemented by default.

For the implementation task, you should submit the following deliverables by the submission deadline:

- **Implementation:** The modified AVMf framework that generates test inputs using a new local search algorithm that you have implemented. Make sure that your submission is self-contained and compiles and executes without any problem.
- **Report:** A written report that contains a detailed description of your new search algorithm and how it is used for test generation.

AVM*f*: An Open-Source Framework and Implementation of the Alternating Variable Method

Phil McMinn
University of Sheffield, UK

Gregory M. Kapfhammer
Allegheny College, USA

Abstract. The Alternating Variable Method (AVM) has been shown to be a fast and effective local search technique for search-based software engineering. Recent improvements to the AVM have generalized the representations it can optimize and have provably reduced its running time. However, until now, there has been no general, publicly-available implementation of the AVM incorporating all of these developments. We introduce AVM*f*, an object-oriented Java framework that provides such an implementation. AVM*f* is available from `http://avmframework.org` for configuration and use in a wide variety of projects.

1 Introduction

The Alternating Variable Method (AVM) is a local search method that was first applied to a search-based software engineering (SBSE) problem — the automatic generation of numerical test data — by Bogdan Korel in 1990 [12]. Despite the application of, supposedly more robust, global search techniques to this problem (e.g., Genetic Algorithms (GAs)), the AVM has stood the test of time. In 2007, Harman and McMinn [7] reported its effectiveness and efficiency for a series of C programs, and combined it with a GA to provide a "best of" Memetic Algorithm approach [8]. It has since been implemented into tools to generate test data for C programs (e.g., IGUANA [17] and AUSTIN [14,15]); generate Java test suites with EvoSuite [3,4]; create relational database data with the *SchemaAnalyst* tool [9,18]; and combined with dynamic symbolic execution in Microsoft's Pex tool [16]. The AVM has also found application to additional problems, including decision ordering for software product lines [22], balancing workload in requirements assignment [21], solving reliability-redundancy-allocation problems [20], as well as test case selection [19] and test suite prioritization [2].

Since Korel's original work, the AVM has been extended and improved for problems in SBSE: now it can handle more variable types, including fixed-point numbers [7] and strings [9,18], and can leverage new strategies proven to speed up the search for certain common types of objective function landscape [10,11].

The AVM is therefore capable of handling a variety of search representations and locating solutions to SBSE problems in a very efficient manner. Yet, to incorporate it into a project, a developer has previously had to understand the different variants of the algorithm and produce a faithful implementation, or, attempt to adapt the open-source of a less general version specifically written for test data generation (e.g., [15]). Either of these options represents a potentially time-consuming and error prone task. To address this, we have developed AVM*f*, a general, open-source object-oriented framework that implements different variants of the AVM and its representations in Java. AVM*f* is available for download from `http://avmframework.org` for deployment in SBSE projects. It is fully documented and comes with a series of examples demonstrating its usage.

2 The AVM and Recent Improvements to the Algorithm

The Original AVM. The AVM optimizes a vector $\vec{x} = (x_1, \ldots, x_{len})$ according to some objective function by taking, in turn, each variable $x_i, 1 \leq i \leq len$ of the vector and subjecting it to an individual search process. The original AVM used a variable search process subsequently named "Iterated Pattern Search" (IPS) [10,11], shown by lines 1–7 of Figure 1. Here, we assume that $x_i \in \mathbb{Z}$, although later we explain how more complex types may also be handled by the approach. The initial part of the IPS algorithm involves making an increase and decrease of 1 to the value of the variable (lines 2–3), referred to as *exploratory moves*. If an exploratory move leads to an improvement in the objective value, a positive or negative "direction" is established for making further *pattern moves* (lines 4–6). Pattern moves of increasing size continue to be made while the objective value improves. When a pattern move fails to improve upon the objective value, the search has likely overshot an optimum, due to a pattern move that was larger than the difference between the current value of x_i and the optimal value. When this occurs, IPS loops back to the exploratory move process to re-establish a new direction. If exploratory moves do not lead to an improvement in objective value, IPS terminates and hands back control to the main loop, thus leading to the consideration of the next variable in the vector.

When all variables in the vector have been considered, the AVM wraps back to the first. When a cycle of all variables has completed without any improvement in the objective function, the AVM is lodged in a local optimum. At this point the search process can be restarted with a new (typically random) series of vector values. The AVM continues in this fashion until resources are exhausted (e.g., a maximum number of objective function evaluations or restarts have been expended, or a time limit has expired), or, the best outcome is attained — the optimal target vector is discovered. (For simplicity, these different termination criteria are not included as part of the algorithm definition in Figure 1.)

New Variable Search Algorithms. Kempka et al. [10,11] proposed two new variable searches for the AVM, as shown in Figure 1. Kempka et al. proved that these search techniques are more efficient than IPS for unimodal objective function landscapes. "Geometric Search" (GS) begins by performing exploratory moves followed by pattern moves like IPS. Unlike IPS, however, it does not iterate after overshooting the optimum. Instead it uses past moves to "bracket" the upper and lower limits of the variable in which the optimum must lie, performing a binary search to finally locate it (lines 8–15 of Figure 1). "Lattice Search" (LS) is a slightly faster alternative to GS where the unimodal assumption holds. LS converges on the optimum through moves that increase x_i from the lower value of the bracket through the addition of Fibonacci numbers (lines 16–22).

New Representations. Korel only demonstrated the original AVM with integer variables [12]. Harman and McMinn [7] extended this initial definition by allowing each variable to be specified with a set number of decimal places p, allowing fixed-point numbers to be handled. Exploratory moves correspond to the smallest possible increments and decrements of the variable (i.e., $\pm 10^{-p}$). Strings may also now be handled by the approach [9,18]. A string variable is essentially

```
while true do                                                              ▷ {IPS}
    if $obj(x-1) \geq obj(x)$ and $obj(x+1) \geq obj(x)$ return $x$         ▷ {IPS, GS, LS}
    if $obj(x-1) < obj(x+1)$ then let $k := -1$ else let $k := 1$           ▷ {IPS, GS, LS}
    while $obj(x+k) < obj(x)$ do                                            ▷ {IPS, GS, LS}
        let $x := x + k$, $k := 2k$                                          ▷ {IPS, GS, LS}
    end while                                                               ▷ {IPS, GS, LS}
end while                                                                  ▷ {IPS}
let $\ell := \min(x - k/2, x + k)$, $r := \max(x - k/2, x + k)$             ▷ {GS, LS}
while $\ell < r$ do                                                        ▷ {GS}
    if $obj(\lfloor(\ell + r)/2\rfloor) < obj(\lfloor(\ell + r)/2\rfloor + 1)$ then    ▷ {GS}
        $r := \lfloor(\ell + r)/2\rfloor$                                   ▷ {GS}
    else                                                                    ▷ {GS}
        $\ell := \lfloor(\ell + r)/2\rfloor + 1$                            ▷ {GS}
    end if                                                                  ▷ {GS}
end while                                                                  ▷ {GS}
let $n := \min\{n \mid F_n \geq r - l + 2\}$                                ▷ {LS}
while $n > 3$ do                                                           ▷ {LS}
    if $\ell + F_{n-1} - 1 \leq r$ and $obj(\ell + F_{n-2} - 1) \geq obj(\ell + F_{n-1} - 1)$ then    ▷ {LS}
        let $\ell := \ell + F_{n-2}$                                        ▷ {LS}
    end if                                                                  ▷ {LS}
    let $n := n - 1$                                                        ▷ {LS}
end while                                                                  ▷ {LS}
$x := \ell$                                                                ▷ {GS, LS}
```

Fig. 1. IPS, LS, GS algorithms for a variable $x \in \mathbb{Z}$. The function *obj* is equivalent to evaluating the objective function with a vector $\vec{x}$ with all components except x_i set to constants and x_i substituted by the free parameter x. F is the Fibonacci sequence starting from $F_0 = 0$. Each line is annotated to show the algorithm(s) it is a part of.

a sub-vector of the overall vector to be optimized. Their elements are characters that are individually manipulated by the local search routine. The length of this sub-vector is allowed to vary through a special sequence of moves that increase and decrease its size, supporting the optimization of variable-length strings.

3 The AVM Framework (AVM*f*)

The AVM Framework (AVM*f*) implements both the AVM algorithm and the subsequent enhancements to the original version proposed by Korel. The framework has been implemented with the aim of making the core algorithms as clear as possible, thereby closely matching the algorithmic definitions of Figure 1, while still adhering to well-accepted principles of good object-oriented design. AVM*f* is publicly available from `http://avmframework.org` as a Git repository for inclusion in SBSE projects where the AVM may be the core search algorithm, or, a component of a more complex technique (e.g., a Memetic Algorithm) involving calls to algorithms in the framework. Or, the code can simply be lifted from the repository and adapted to a project as developers see fit.

To enable its algorithms to be easily used in SBSE projects, AVM*f* provides a framework of Java classes, which we now describe in detail. Each aspect of the framework is practically demonstrated by the source code of a series of examples in the repository, the simplest of which are introduced at the end of this section.

Configuring an AVM Search. The primary class is the `AVM` class in the root (i.e., `org.avmframework`) package. In order to construct an `AVM` instance, the developer must supply an instance of one of the variable search methods — `IteratedPatternSearch`, `GeometricSearch` or `LatticeSearch` — which reside in the `localsearch` package. The developer must also construct the `AVM` instance using a `TerminationPolicy` parameter, an object that decides when the AVM should terminate if a solution cannot be found. Options include a maximum number of objective function evaluations, a maximum number of restarts, or a time limit. Finally, constructing the `AVM` instance further requires two objects of type `Initializer` that are used to initialize variable vector values at the start of the search and re-initialize them on a restart. Default values may be used that can be specified for each variable, or random values can be chosen (through instances of either `DefaultInitializer` or `RandomInitializer`, two classes that both reside in the `initializer` package). To support the generation of random numbers, AVM*f* requires a `RandomGenerator` from the `org.apache.commons` library that provides an implementation of the Mersenne Twister algorithm.

In order to initiate a search process, the `search` method of the `AVM` instance must be invoked with an instance of a `Vector` class and an `ObjectiveFunction`, respectively. The `Vector` class describes the representation of the problem (i.e., the types of variables in the vector to be optimized), while the `ObjectiveFunction` class describes how instances of those vectors should be rewarded with objective values during the search.

Representation. In order to configure the search representation, an instance of the `Vector` class (in the root package) must be created, and variables added to it through the `addVariable` method, which accepts an instance of a `Variable`. Since the `Variable` class is abstract, an instance of one of its concrete subclasses must be provided (i.e., one of `IntegerVariable`, `FixedPointVariable`, `CharacterVariable` or `StringVariable`). Each variable must be constructed with information such as its minimum or maximum value (maximum length for strings), number of decimal places for fixed-point variables, and a "default" initial value in the search space (e.g., an empty string or a zero value). These values are used to initialize vector variables when the `DefaultInitializer` provides a starting point for the search, as previously described in this section.

Objective Function. In contrast to the rest of the framework, which requires configuring instances of existing classes, an objective function must be supplied to the search process by overriding the abstract `ObjectiveFunction` of the `objective` package. This involves providing an implementation of the `computeObjectiveValue` method that takes a `Vector` as a parameter and returns an instance of the abstract `ObjectiveValue` class. Since the AVM only needs to know whether one entity has a "better" objective value than another, exact numerical values are not needed, and so this class requires the "`betterThan`", "`worseThan`" and "`sameAs`" methods to be overridden. The `objective` package also supplies the concrete `NumericalObjectiveValue` class for returning higher-is-better or lower-is-better numerical objective values as needed.

Reporting. The `search` method of the `AVM` class returns an instance of the `Monitor` class, which can be used to find out interesting statistics regarding the search. These include the best vector found by the search, its objective value, the number of objective function evaluations that took place, the number of restarts that happened and the amount of time that the search took (in milliseconds).

The `Monitor` class can also report the number of *unique* objective function evaluations. Employing the technique known as memoization, the objective function can make optional usage of a cache that maps previously observed vectors to objective values, avoiding the need to perform potentially costly re-evaluations.

Examples. AVM*f* comes with a series of examples demonstrating its use. Instructions on how to compile and run these examples are available in the project's `README.md` file located in the main directory of the code repository. The "`Quadratic`" example demonstrates the use of the AVM to solve a quadratic equation by finding one of its roots. "`AllZeros`" shows the optimization of an array of integers to zero values from arbitrary random values, while "`String`" optimizes a string value from an initially random string to a specified target.

Each example makes use of its own problem-specific fitness function, which forms part of its code definition. The following is taken from the `Quadratic` class, where the constants `A`, `B` and `C` correspond to the co-efficients of the equation (here, `A = 4`, `B = 10` and `C = 6`). The function obtains the value of `x` from the (single variable) vector, and computes the value of `y`. The objective value is then assigned as the distance between `y` and zero, since intuitively, the closer the value of `y` to zero, the closer the search is to finding one of the roots of the equation:

```
ObjectiveFunction objFun = new ObjectiveFunction() {
    protected ObjectiveValue computeObjectiveValue(Vector vector) {
        double x = ((FloatingPointVariable) vector.getVariable(0)).asDouble();
        double y = (A * x * x) + (B * x) + C;
        double distance = Math.abs(y);
        return NumericObjectiveValue.LowerIsBetterObjectiveValue(distance, 0);
    }};
```

The following shows the output of the search process and the discovery of one of the equation's roots, -1.5. Re-running the search from different starting positions leads to the other root, -1, also being found.

```
Best solution: -1.5
Best objective value: 0.0
Number of objective function evaluations: 80 (unique: 80)
Running time: 3ms
```

As part of future work, we plan to extend the example set with case studies showing how the AVM is being or can be applied to real SBSE problems, such as test data generation. These will be made available via the code repository.

4 Conclusions and Future Work

This paper introduced AVM*f*, an open-source implementation of the AVM and a framework supporting its use in SBSE projects. AVM*f* is capable of advancing the AVM in both industrial practice and in the SBSE research community. Using AVM*f*, possible future applications of the AVM include the following:

Automatically Generating Readable Test Data. Generating readable tests that humans can easily understand has been a recent interest of search-based testing researchers (e.g., rewarding inputs that obtain a high score from a language model [1]). In a recent study evaluating test generation tools, participants also requested more readable values [5,6]. Given that the AVM employs a local search, it could start with examples of human-generated inputs and adapt them to new coverage targets — all without losing the qualities of the original data.

Automatically Determining Optimal Software Configuration Values. Highly configurable software tools, such as the GCC compiler, may be tunable through the use of search-based techniques such as genetic algorithms or the AVM [13]. In large search spaces of parameters, the AVM's exploratory move phase equips it to quickly discover which particular variables are relevant to the problem, while its phase of pattern moves allows it to determine the optimal values of parameters. Again, as a local search technique, the AVM is also well suited to taking an existing known-good human solution and improving upon it.

Automated Bug-Fixing. Recent experiments reveal that real-world bugs can occur as a result of mistakes made when defining constant variables and setting values in configuration files [23]. As such, the AVM could search for appropriate values that could potentially form the basis of a "fix". During its exploratory move phase the AVM could, by performing a quick sweep of small changes through the values involved and seeing how the resulting fitness values are affected, quickly determine which constants are relevant to the fix.

References

1. Afshan, S., McMinn, P., Stevenson, M.: Evolving readable string test inputs using a natural language model to reduce human oracle cost. In: Proc. ICST (2013)
2. Arrieta, A., Wang, S., Sagardui, G., Etxeberria, L.: Test case prioritization of configurable cyber-physical systems with weight-based search algorithms. In: Proc. GECCO (2016)
3. Fraser, G., Arcuri, A., McMinn, P.: Test suite generation with memetic algorithms. In: Proc. GECCO (2013)
4. Fraser, G., Arcuri, A., McMinn, P.: A memetic algorithm for whole test suite generation. JSS (2015)
5. Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated white-box test generation really help software testers? In: Proc. ISSTA (2013)
6. Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated unit test generation really help software testers? a controlled empirical study. ACM TOSEM (2015)
7. Harman, M., McMinn, P.: A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation. In: Proc. ISSTA (2007)
8. Harman, M., McMinn, P.: A theoretical and empirical study of search based testing: Local, global and hybrid search. IEEE TSE (2010)
9. Kapfhammer, G.M., McMinn, P., Wright, C.J.: Search-based testing of relational schema integrity constraints across multiple database management systems. In: Proc. ICST (2013)
10. Kempka, J., McMinn, P., Sudholt, D.: A theoretical runtime and empirical analysis of different alternating variable searches for search-based testing. In: Proc. GECCO (2013)
11. Kempka, J., McMinn, P., Sudholt, D.: Design and analysis of different alternating variable searches for search-based software testing. TCS (2015)
12. Korel, B.: Automated software test data generation. IEEE TSE (1990)
13. Kukunas, J., Cupper, R.D., Kapfhammer, G.M.: A genetic algorithm to improve Linux kernel performance on resource-constrained devices. In: Proc. GECCO (2010)
14. Lakhotia, K., Harman, M., Gross, H.: AUSTIN: A tool for search based software testing for the C language and its evaluation on deployed automotive systems. In: SSBSE (2010)
15. Lakhotia, K., Harman, M., Gross, H.: AUSTIN: An open source tool for search based software testing of C programs. IST (2013)
16. Lakhotia, K., Tillmann, N., Harman, M., Halleux, J.: FloPSy — search-based floating point constraint solving for symbolic execution. In: Proc. ICTSS (2010)
17. McMinn, P.: IGUANA: Input Generation Using Automated Novel Algorithms. a plug and play research tool. Tech. Rep. CS-07-14, Dept. Computer Science, University of Sheffield, UK (2007)
18. McMinn, P., Wright, C.J., Kapfhammer, G.M.: The effectiveness of test coverage criteria for relational database schema integrity constraints. ACM TOSEM (2015)
19. Pradhan, D., Wang, S., Ali, S., Yue, T.: Search-based cost-effective test case selection for manual execution within time budget: An empirical study. In: Proc. GECCO (2016)
20. Qiu, X., Ali, S., Yue, T., Zhang, L.: Reliability-redundancy-location allocation with maximum reliability and minimum cost using search techniques. IST (2016)
21. Yue, T., Ali., S.: Applying search algorithms for optimizing stakeholders familiarity and balancing workload in requirements assignment. In: Proc. GECCO (2014)
22. Yue, T., Ali, S., Lu, H., Nie., K.: Search-based decision ordering to facilitate product line engineering of cyber-physical system. In: Proc. MODELSWARD (2016)
23. Zhong, H., Su, Z.: An empirical study on real bug fixes. In: Proc. ICSE (2015)

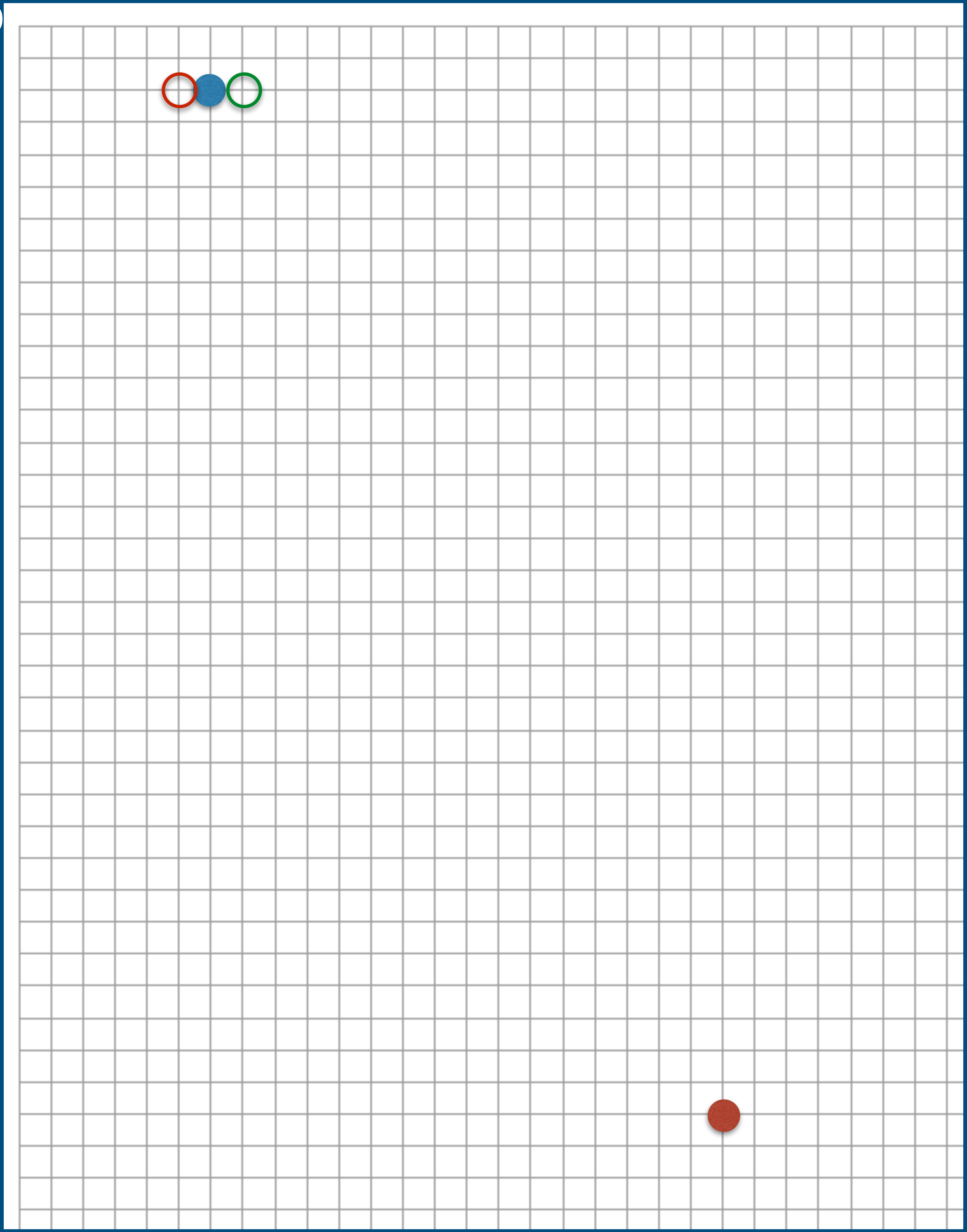
Alternating Variable Method (AVM)

- A type of Pattern Search: searches for an input vector that can maximise/minimise a given objective function
- It has two operation modes: exploratory move, and pattern move.
  - For each variable:
    - Use exploratory move to decide which direction results in fitter solutions
    - Use pattern move to accelerate to that direction

Alternating Variable Method

- Based on the known empirical results, AVM is one of the most effective algorithm for achieving C/C++ structural coverage
  - M. Harman and P. McMinn. A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007), pages pp. 73–83. ACM Press, July 2007.
  - M. Harman and P. McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. IEEE Transactions on Software Engineering, 36(2): 226–247, 2010.

(0, 0)



AVM: Exploratory Move

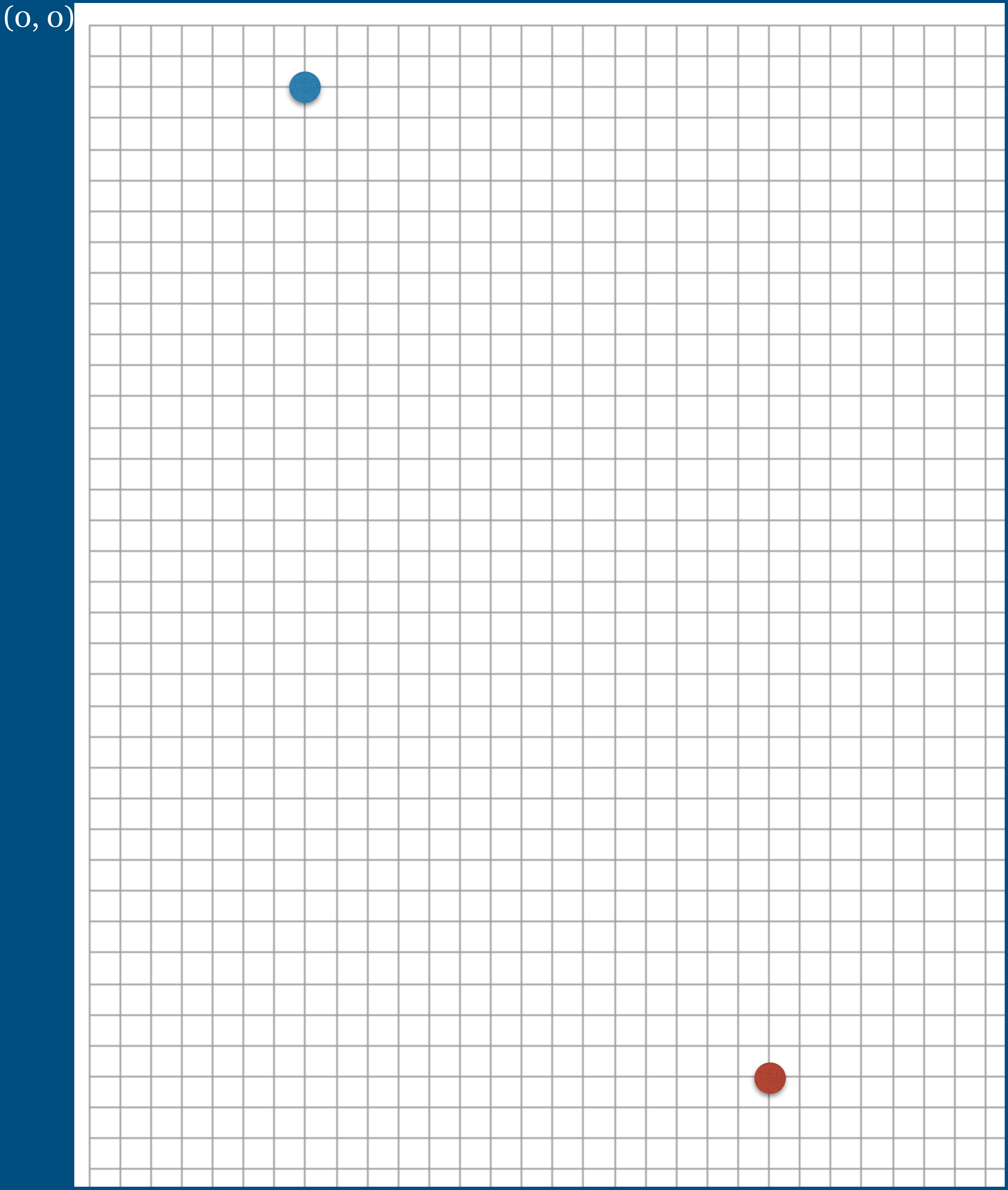
Starting from(6, 2), we want to search for the red dot at (22, 34). We can measure the distance to the goal.

First we try exploratory move for x: make the smallest change, and see which direction results in reduced distance. The initial distance is 35.77.

-1: (5, 2) **Increased** (36.23). X

+1: (7, 2) **Decreased** (35.34) O

Consequently, x needs to be increased at the moment.



AVM: Pattern Move

Now that we decided to increase x, try doubling the difference as long as the distance continues to decrease. At the beginning of the pattern move, x is equal to 7.

x = 9 (Δx=2): decrease (34.53)

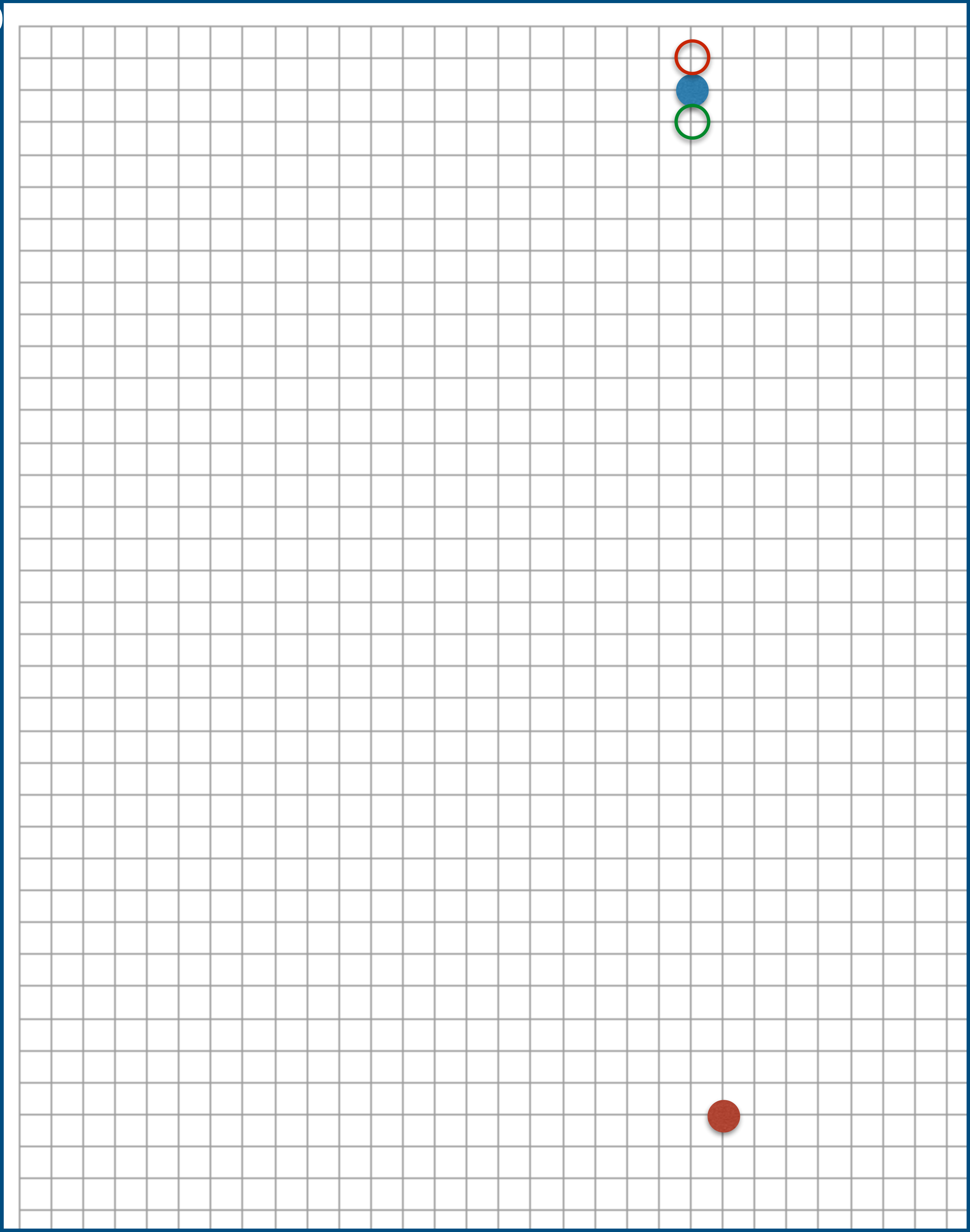
x = 13 (Δx=4): decrease (33.24)

x = 21 (Δx=8): decrease (32.01)

x = 37?(Δx=16): increase (35.34)

With increment of 16, the distance starts to grow: this is called overshooting. In this case, we cancel the last pattern move, and start the exploratory move for the next variable, y.

(0, 0)



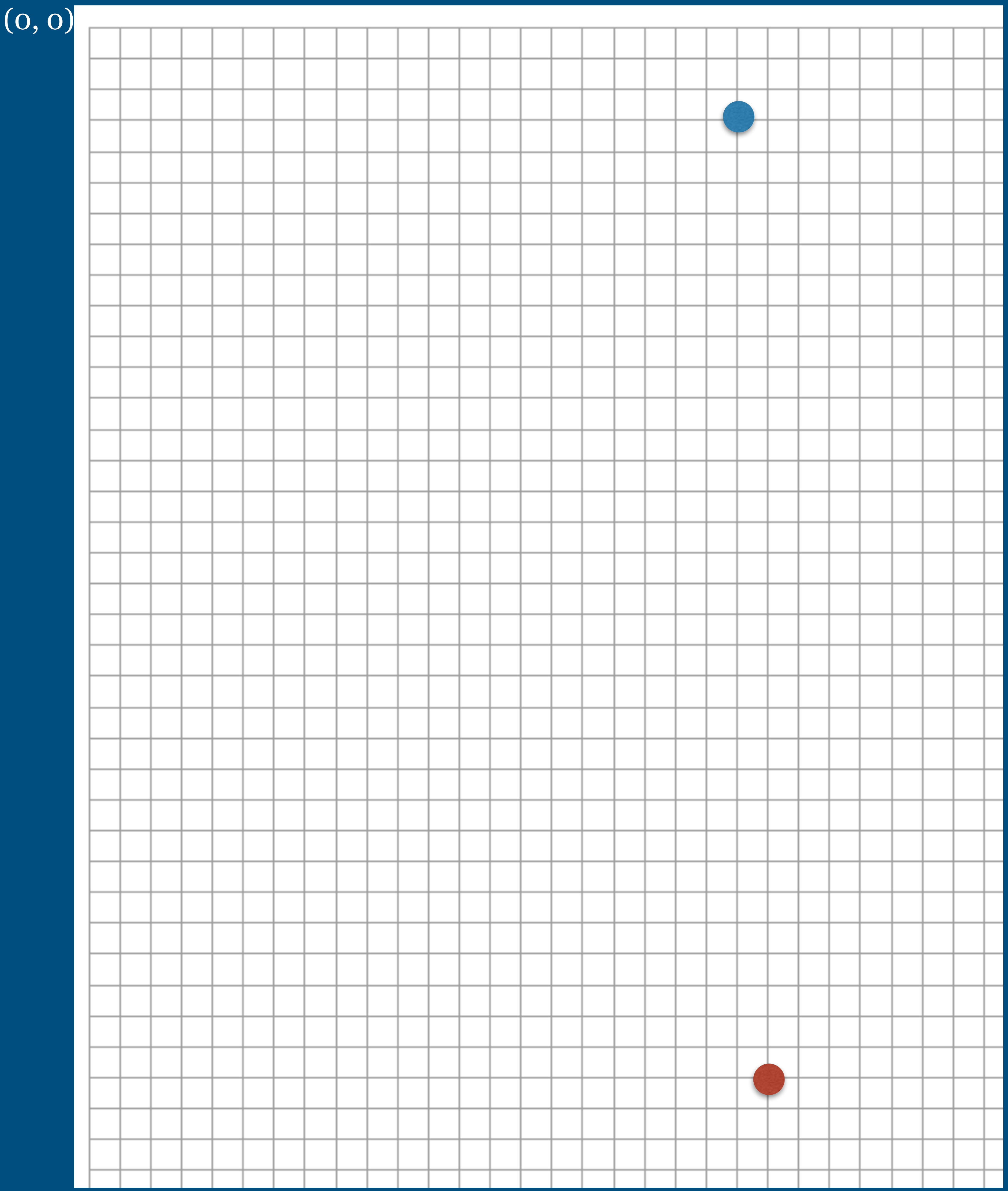
AVM: Exploratory Move

We now change y by 1 and decide the direction. The distance from the last location, (21, 2), is 32.01.

-1: (21, 1) **increase** (33.01). X

+1: (21, 3) **decrease** (31.01) O

So y needs to be increased.



AVM: Pattern Move

We increase the variable y with pattern moves now. Initially y is 3.

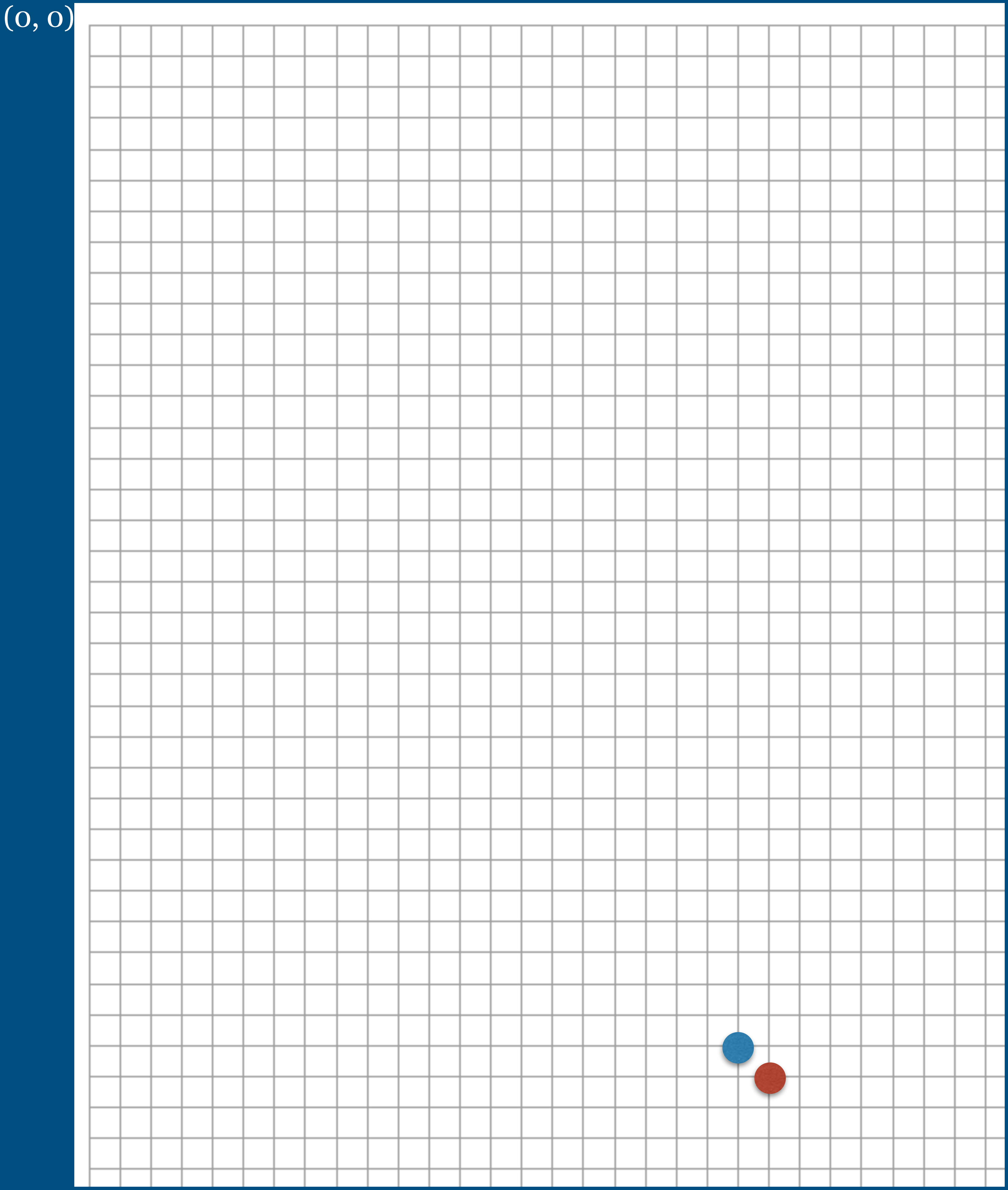
y = 5 (Δy=2): decrease (29.01)

y = 9 (Δy=4): decrease (25.01)

y = 17 (Δy=8): decrease (17.02)

y = 33(Δy=16): decrease (1.41)

y = 65(Δy=32): Overshooting!



AVM: Exploratory Move

After overshooting of y, we start the exploratory move for x. We decide to increase, but as soon as we try +2, it overshoots. After cancellation of this, we have the correct x.

After one more exploratory move for y, we reach the goal.

Alternating Variable Method

- For a reference implementation and basic applications, see: http://avmframework.org

- P. McMinn and G. M. Kapfhammer. AVMf: An open-source framework and implementation of the alternating variable method. In International Symposium on Search-Based Software Engineering (SSBSE 2016), volume 9962 of Lecture Notes in Computer Science, pages 259–266. Springer, 2016.