ssa-tool

A Utility for the Creation and Evaluation of Self-Stabilizing Algorithms

Sean Allred

14 April 2013

Abstract

In computer science, a self-stabilizing algorithm is a fault-tolerant problem-solving process that, when applied repeatedly to a system, will coerce the system into a 'correct state'. Such algorithms are incredibly useful in real-world situations such as network management, artificial intelligence, and other distributed systems. Given their parallel nature, algorithms of this class are more difficult to create and prove correct than more traditional, procedural algorithms. To aid in further research, this paper presents and defends an extensible utility for the representation, creation, and evaluation of self-stabilizing algorithms.

Contents

1 Introduction

1.1 Domain Introduction

In his seminal paper on the topic [5], Dijkstra introduced the concept of self-stabilizing algorithms. He visualized the realization of these algorithms as taking advantage of a very specific scenario:

> We consider a connected graph in which the majority of the possible edges are missing and a finite state machine is placed at each node; machines placed in directly connected nodes are called each other's neighbors. For each machine one or more so-called "privileges" are defined, i. e. Boolean functions of its own state and the states of its neighbors; when such a Boolean function is true, we say that the privilege is "present". In order to model the undefined speed ratios of the various machines, we introduce a central daemon — its replacement by a distributed daemon falls outside the scope of this article — that can "select" one of the privileges present. The machine enjoying the selected privilege will then make its "move"; i. e. it is brought into a new state that is a function of its old state and the states of its neighbors. If for such a machine more than one privilege is present, the new state may also depend on the privilege selected. After completion of the move, the daemon will select a new privilege. (Dijkstra, 1974)

1.2 Consequences and Further Research

This canonical definition is inherently limited, however; each node n only knows the state of its distance-1 neighbors $v \in N[n]$. Gairing et al. introduced an approach for simulating distance-2 information [10], which was later generalized to simulating distance-k information [12] by Goddard et al., necessarily introducing appropriate polynomial time and space complexities. Dijkstra's restrictive-yet-practical definition of a self-stabilizing algorithm was now *far* less restrictive, since Goddard et al. allowed *any* distance-k algorithm to be simulated in the original definition within a reasonable complexity increase.

Since then, many self-stabilizing algorithms have been put forward, unsurprisingly. Talks such as [6] present both the many applications of self-stabilization algorithms and the research done on the algorithms in general. Given their innately parallel nature, self-stabilizing algorithms are *incredibly* useful for accurately modeling and solving real-world problems.

1.3 Related Work

In the same vein, complex algorithms of this style — those that quickly become too complex for a human to keep track of — can be difficult to test without spending significant time actually *implementing* the algorithm from scratch in a programming language. This is a great barrier to some more mathematically-bent researchers in the field, given that they may have little to no experience in modeling mathematical systems as a program. Even for those that *are* comfortable with the practice, there are few tools that are readily available to assist in the work. One that seems to stand out from the rest is Python's NetworkX [14], an extensive graph representation and manipulation

library. While these libraries assist in modeling the raw graph, they provide little to no assistance in creating, testing, or maintaining the algorithm itself. These utilities are simply not designed for such a relatively specific task.

This is not to say that there aren't many excellent frameworks in which to *deploy* self-stabilizing algorithms. For example, `hpc/libcircle` [15] is "an API to provide an efficient distributed queue on a cluster". Using a blazingly fast algorithm described by [17], it is able to "quickly traverse and perform operations on a file tree which contains several hundred-million file nodes". This is a good tool for its purpose, but is unfriendly as far as self-stabilization research is concerned. The tool is written in and requires the use of C — an *excellent* choice of language in industries where speed is of the essence — but it would seem unwise to create the algorithm itself (and test it interactively) with such a language.

On GitHub, `fintler/balancemq` [9] provides another such API, this time written in Python, using an algorithm described by LaFon, Misra, and Bringhurst [17]. Again, while claiming to be an API for self-stabilizing algorithms, this (albeit useful) project does not provide true tools to create and work with fault-tolerant systems in more general, simpler cases. Nevertheless, these projects are excellent examples of the usefulness of these algorithms to the industry and the necessity of their existence and exploration.

1.4 Current Work

Currently presented is a Python framework within which to create, define, and test these algorithms. Complete with a (work in-progress) graphical interface which supports maintaining libraries of predicate and moves for later assembly, `ssa-tool` provides researchers with useful functionality to create and test algorithms in batch and by hand.

Necessity and Purpose While the tools above utilize self-stabilizing algorithms for specific applications, appealing themselves more to systems administrators than academics, `ssa-tool` aims to be general and robust enough to handle the wide variety of algorithms produced by scholarly research. `ssa-tool` represents self-stabilizing algorithms and their components as closely as possible to the available mathematical definitions. In this way, it aims to be a useful resource to quickly support published works and to provide the means to test algorithms *before* preparing publication, even to the point of creative construction.

Philosophy The tool is built upon a philosophy of accessibility; all core functionality is completely text-based. By using YAML [3] as the basic database structure and defining the algorithm steps with short Python snippets, `ssa-tool` maintains a simple file structure that is easily understandable and editable by a human hand without sacrificing straightforward manipulation and execution by a computer. This adherence to a lower level of operations and avoidance of higher-level functionality is deliberate; it is my experience that a sound engine, however bare it may be, is *essential* to build a functional, more user-friendly interface. By using industry-standard technologies and keeping the core system simple enough for a human to intuitively understand, it stands to reason that more feature-rich graphical interfaces may be laid atop this system by those more adept in developing them. It also reduces the chance and general opportunity for bugs to arise, ensuring a stable engine that can be easily extended.

Potential While the allotted term-time for development has drawn to a close, feature-freeze has set in. Considering the above philosophy, this is likely for the better. However, there are many features that `ssa-tool` provides the backbone for:

- The separation of engine from interface (even the rudimentary textual interface) allows for the painless implementation of better interfaces. see Task 9
- The use of NetworkX [14] provides a possible interface to popular graph visualization applications (such as Gephi [2]). see Task 15
- A complete log of changes is kept, feasibly allowing animation for output formats that support it (such as `gexf`).[1] see Task 16

See section 6 for a complete list of these tasks.

1.5 Graphical Interface

For those who wish to use it, `ssa-tool` also comes with a basic graphical interface (written with Tkinter [8]) for the construction and manipulation of 'bundles'—self-contained collections of algorithm descriptions with all necessary components (predicates/rules) needed to run them.[2] The use of this specific interface is documented in section 5.

As with all graphical interfaces, there is some functionality of `ssa-tool` that it does not support:

- It currently makes no provision for testing or running tests. see Task 8
- While it does provide templates upon predicate/move creation, it does not provide any form of syntax highlighting during their creation as is common with most editors. see Task 10
- When writing definitions for predicates/moves, there are no scrollbars and lines wrap, potentially confusing the user. see Task 11
- The prototype graph visualization tool (written with PyGame [19]) remains unintegrated into the manager. see Task 17

These tasks are left as goals for future versions to be considered for a truly public release. The current interface, as it stands, is a *prototype*; it is neither thoroughly tested nor fully representative of `ssa-tool`'s feature-set. See Task 3.

1.6 Further Work

There are many tasks that remain to be done for this project, and throughout this paper they will be referred to (e. g. Task 3). The complete list is available in section 6 on page 19.

[1] NetworkX does not yet support animated graphs. Outputting to this format will take quite a bit of work unless progress is made by the NetworkX project. See Task 16.

[2] This structure is detailed in Appendix A.

2 Mathematical Definitions

The core engine must be correct above all. To prove its correctness, all necessary definitions will be presented, cited as necessary, and referred to throughout the remainder of the paper.

2.1 Graph

A graph is a mathematical construct consisting of vertices V and edges E between them: $G = (V, E)$. We define its open neighborhood of $v \in G$ to be the set of all nodes u such that v is adjacent to u: $N(n) = \{v : (n, v) \in E\}$ and the closed neighborhood is defined as v together with its neighbors: $N[n] = N(n) \cup \{n\}$.

2.2 Self-Stabilizing Algorithm

As defined by Dijkstra [5], a self-stabilizing algorithm is a construct composed of predicates and corresponding moves. Both are functions of a single node and its neighbors.[3] 'Predicates' are functions to the Boolean values (1), where each 'move' updates the state of the system (2). When a predicate evaluates to True for any node n, it is said that n's privilege is present.

$$P\colon n, N(n) \to \{\text{True}, \text{False}\}, \tag{1}$$

$$M\colon n, N(n) \to n', N'(n). \tag{2}$$

A self-stabilizing algorithm can thus be seen as a collection of these predicate – move pairs. Formally, we may define such an algorithm S to be a discrete function from predicates to a random move functions:

$$S\colon P \to \text{RandomChoice}(\{M_1, M_2, \ldots, M_N\}) \tag{3}$$

It was defined in this way to clarify the logical representation of the algorithm; see subsection 3.2. To distinguish from other uses of 'algorithm', this particular data structure will be called the 'rule set'.

Two properties of the algorithm must be shown for a self-stabilizing algorithm to be proven correct: [1]

convergence The algorithm must complete using a finite number of moves. (S *converges*.)

closure If the algorithm completes, the system is in a correct state. (S is *closed*.)

A self-stabilizing algorithm converges if and only if $\exists N \in \mathbf{N} : \forall\, n > N, p \in P^n, v \in V,\ p(v) = 0$. That is, after some finite number of moves $N \in \mathbf{N}$, no predicate $p \in P$ privileges any vertex $v \in G$. If there are no privileged nodes, no node can make a move and the system is considered 'stable' [5].

A self-stabilizing algorithm satisfies closure if and only the absence of a privileged node necessarily implies a correct overall state:

$$\exists N \in \mathbf{N} : \forall\, n > N, p \in P^n, v \in V,\ p(v) = 0 \implies G \text{ is in a correct state.} \tag{4}$$

[3] Note that this is not the same as the closed neighborhood of a node. Consider these as functions $\lambda\colon n, N(n) \to \{?\}$.

Algorithm 1 Maximal Independent Set, definition from [12]

local f

ENTER
  if $f(i) = 0 \wedge \forall j \in N(i),\ f(j) = 0$
  then:
    $f(i) = 1$

LEAVE
  if $f(i) = 1 \wedge \exists j \in N(i) : f(j) = 1$
  then:
    $f(i) = 0$

Example Algorithm and Run A popular problem in graph theory is IndSet — the maximal independent set — and it has a particularly simple self-stabilizing algorithm (see Algorithm 1). For clarity, we define the maximal independent set as the set with the greatest cardinality within the power set of the vertex subsets $H \subseteq G$ such that no two nodes in H are adjacent in G.

Considering Algorithm 1, the first rule simply says if some node i is marked (where f is a node-indexed Boolean array) *and* no neighboring nodes j are marked, then we 'enter' the set by setting our value in f to 1. Similarly, the second rule ('leave') applies when, for some node i, the node is marked and a neighbor is as well.

The algorithm is thereby defined and is ready to be applied to some graph G and nodes i and neighbors j. Consider the graph in Figure 1. Nodes have been marked arbitrarily, representative of a system that is in a possibly incorrect state. We pick a node *arbitrarily*, in this case v_5, and apply the predicates from Algorithm 1 to the node to see which privileges are present. The only present privilege is 'enter', so we execute the move prescribed by the rule and mark v_5: $f(v_5) = 1$. The graph from Figure 1 is now Figure 2, and the process begins again, perhaps now v_4 is picked and satisfies the 'leave' predicate and executes its moves, leaving Figure 3. Since no other predicates apply, we say that 'no privileges are present' and the graph is considered 'stable'.

The fact that we have reached an iteration where no privileges are present gives evidence that IndSet converges. Since no other node can be marked and no two marked nodes are adjacent to each other, this is indeed a maximal independent set. Since the algorithm did what it said it was going to do, there is evidence that IndSet is closed.

Obviously these properties would require *formal* proof, but many examples of an algorithm succeeding could convince a researcher to invest the time in its pursuit.

Different Models The paradigm given above is known as the central-daemon model [5]. At least one other model exists where each node acts of its own accord distinct from any central daemon [7]. To simplify initial development, the daemon model was implemented as opposed to any truly distributed model (see subsection 3.3). An extension may later be added — perhaps using the Python variant Stackless [22] — that implements an alternative run-mode that uses distributed techniques.

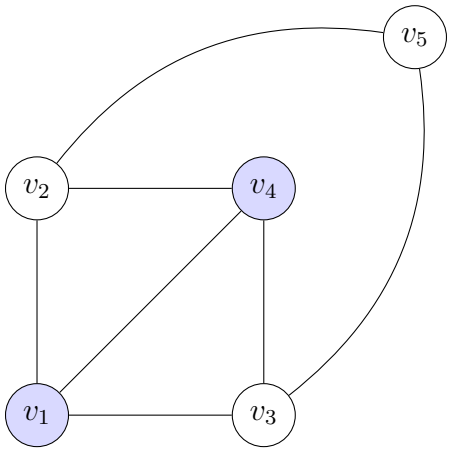

Figure 1: An arbitrarily marked graph G at time $t = 0$.

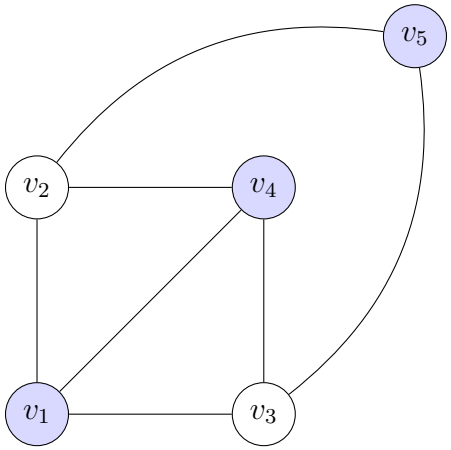

Figure 2: The graph from Figure 1 at time $t = 1$.

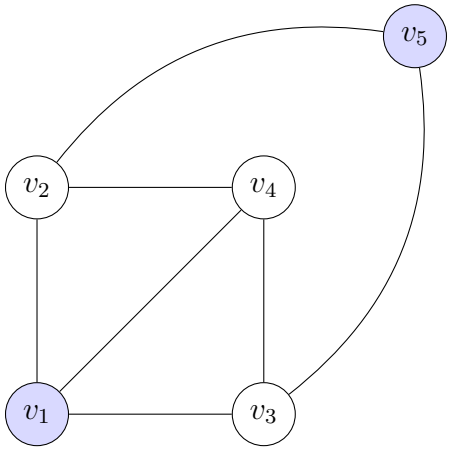

Figure 3: The graph from Figure 1 at time $t = 2$. No more privileges are present for any node v_i, so G is now called 'stable' and the marked nodes represent a maximal independent set.

Listing 1: A model definition for a predicate: $\forall\, n \in N[v],\ n$ is not marked.

```
if v['marked']:
  return False
for neighbor in N:
  if n['marked']:
    return False
return True
```

3 Logical Representation

The entirety of this tool is written in and tested with Python 3.3.5. The following listings serve to introduce the organization of the library and to serve as a reference to be used when extending it.[4]

3.1 Predicate and Moves

According to Equation 1 and Equation 2, both predicates and moves are defined as simple functions of a node and its neighbors, where each predicate returns a Boolean value and each move returns an updated node and neighborhood. However, in order for the tool to be useful in a collaborative setting, there needs to be both documentation and at least a few pieces of other metadata attached to these basic elements. The population of this metadata is the responsibility of two distinct classes — 'Predicate' and 'Move' — that provide fields for the following information:

author author(s) of the function

date timestamp for editing

name short summary of function[5]

TEXnical documentation mathematical definition

description long form discussion of the function

Definition and Loading As a bundle is designed to be a self-sufficient representation of one or more algorithms, all predicate and move definitions are stored locally as Python 3 function bodies within the `predicate` and `move` directories under root. For example, if we wished to create a predicate to determine if a node could mark itself (considering the IndSet problem), we could put Listing 1 in `predicates‣my-pred.py`. The contents of this file will be evaluated as a function of three arguments:

`self` The current entitiy. It is not recommended that this variable is used, but it exists and is a reserved word. Do not alter this variable unless you are confident in your actions.

[4] All development is tracked as a literate program on GitHub at `http://www.github.com/vermiculus/ssa-tool`.

[5] Must be unique; see Appendix A.

Listing 2: Predicates and moves are defined by named files in the bundle. See also Task 12.

```
def load_definition(self, path, obj):
  if hasattr(obj, 'filename'):
    <<inherit class variables for YAML export>>

    # read definition file
    with open('/'.join([path, obj.ssa_folder, obj.filename])) as f:
      lines = f.readlines()

    # store the definition in a 'hidden' variable
    obj._definition = lines

    # prepare a string to execute as a definition
    lines = ['def␣temp(self,␣v,␣N):\n'] + \
            ['␣␣␣␣' + l for l in lines]

    # execute the string, defining the 'temp' function
    exec(''.join(lines), locals())

    # set this function as the behavior for the singleton class
    ssa_obj.__class__.__call__ = locals()['temp']
```

v The current node (v for 'vertex'). Its data may be accessed dictionary-style as in Listing 1.

N The neighborhood of v as a Python 3 `set`.[6] Each member may be viewed as the above.

Warning: Do not alter the values of v or N in a predicate function. In future versions these values may be copied before they are made available to the function body (see Task 5), but no such provisions are currently being made. Altering them will introduce race conditions in execution.

Warning: You may view the contents of `self` if you so desire (with the recognition that this potentially invalidates the definition of a self-stabilizing algorithm), but under no circumstances should you *ever* alter its contents. Doing so will almost certainly compromise the logical consistency of the bundle and could irreversibly destroy all unsaved work in the current bundle.

Warning: Since Python is a whitespace-sensitive language, all function body definitions *must* be indented using four spaces. The use of tabs (or inconsistent use of spaces) will cause the parser to crash at Listing 2. When using the interface, the tab key will insert these four spaces.

[6] In reality, this type is the return of the `values()` method of a Python dictionary.

Listing 3: Each rule must be resolved *after* all predicates and moves have been defined.

```
mapping = {entity.name if hasattr(entity, 'name')
                       else repr(entity)          : entity
           for entity in entities}
for rule in self.rules:
    rule.predicate = mapping[rule.predicate]
    rule.moves = [mapping[m] for m in rule.moves]
```

Listing 4: An example structure to use as an 'Algorithm''s `ruleset`

```
ruleset = {
    some_predicate: [some_move, some_move],
    some_predicate: [some_move, some_move, some_move],
    some_predicate: [some_move]
}
```

3.2 Self-Stabilizing Algorithm

According to Equation 3, a self-stabilizing algorithm is defined as a mapping of these predicates to a set of their potential moves. These must be provided as the constructor argument `ruleset`, as in Listing 3.

In an 'Algorithm', the `ruleset` is assumed to have dictionary-like structure and behavior. A single predicate maps to a `list` or `set` of potential moves, as in Equation 3. Refer to Listing 4 for an example of this data structure. In addition to basic constraints on overall structure, every single basic component in the structure (each predicate and each move) must be directly callable with two arguments (a vertex and its neighborhood). If both of these constraints are met, this data structure can logically be used as part of the algorithm's definition.

3.3 Central Daemon

This implementation's testing model is based of the concept of a 'central daemon'. (This was the first model explicitly introduced by Dijkstra [5]; see also subsection 2.2.) The 'Algorithm' class plays double-duty to fulfill this role as a default behavior, but perhaps this should be altered. The overall implementation and interface is available in Listing 5, but we will look at each part individually.

The daemon begins by finding assembling a list of privileged nodes. To do this, it *must* go through each node and check it for the presence of privilege.

Warning: I consider this the greatest single bottleneck in the entire project. It is feasible that this can be altered into a distributed/threaded algorithm, but I haven't the knowledge to do so. The fact that predicates should *never* write to a node or its neighbors allows this distribution of work among threads. This should be implemented in a future iteration—see Task 23.

Within each iteration of the loop through the nodes in the graph (see Listing 6), we check to see if the node is privileged or not (see Listing 7). If it is privileged, we add the privileged node to a dictionary called `privileged_nodes`.

Warning: We can also increase performance here with threading (see Task 23). Be careful of the difference, however: in the above we were simply reading information and running it through a predicate function. (In fact, we are doing the same here, but the above is a separate idea that can be sorted out.) The difference here is that we are altering the `privileged_nodes` data structure. The branch in Listing 7 introduces a race condition: if two predicates P_1, P_2 simultaneously succeed for an as-yet unprivileged node n at time t, time $t+1$ will result in only one unknown predicate P_i being written to `privileged_nodes`. Altering the data structure used for `privileged_nodes` may remove this limitation.

4 Testing Interface

Testing is currently only supported via the text interface (see Task 8). See Listing 10 for an example run of INDSET on a star graph. Valid graph input formats are `gexf`, `gml`, and `yaml`.

5 Interface

While the core code-base has been designed and implemented to be easily accessible to those who wish to use it directly or extend it, the expected everyday interface to this project is the graphical tool presented below. It fully supports the creation and documentation of self-stabilizing algorithms, rules, predicates, and moves using a combination of textual and graphical interfaces. Since the bundle format is self-contained, all of these objects — predicates, moves, and algorithms — can be saved and distributed to colleagues that are using the same or derivative software.

5.1 Predicates and Moves

As reflected in the logical representation (see subsection 3.2), self-stabilizing algorithms persist as a collection of predicates and moves. As such, this tool supports the convenient creation of these basic components.

Predicates and moves can be created using the graphical interface, but it should be noted that the *definition* of these is left as plain-text input. (See also Task 10 and Task 11.) (This definition will be written verbatim to disk; see subsection 3.1.) To create them, navigate to the Predicates or Moves tabs and click the `add` button beneath the list on the far left (Figure 4). This will add an empty item — in this case, a predicate — to the list. Editing its Predicates » Name field will update its name in the list accordingly. Fill in the Predicates » Author, Predicates » Date, and Predicates » TEX fields as appropriate.

Warning: When giving any entity a name, be sure to edit it character-by-character starting from the end, deleting using *only* the backspace ← key. No other behavior has been

Listing 5: A generalized run of a self-stabilizing algorithm.

```
def run(self, graph, count=1):
    """Run the algorithm `count` times.

    <<run documentation>>
    """
    assert count >= 0

    history = list()
    while count > 0:
        <<run once>>
        count -= 1
    return history
```

Listing 6: Finding privileged nodes.

```
for node in graph:
    neighborhood = Algorithm.neighbor_data(graph, node)
    <<determine if node is privileged>>
```

Listing 7: Getting the privileges of a single node.

```
for predicate in self.ruleset:
    if predicate(graph.node[node], neighborhood.values()):
        if node in privileged_nodes:
            privileged_nodes[node] += predicate
        else:
            privileged_nodes[node] = [predicate]
```

Listing 8: Picking a random, satisfied predicate.

```
node = random.choice(list(privileged_nodes.keys()))
neighborhood = Algorithm.neighbor_data(graph, node)
satisfied_predicate = random.choice(privileged_nodes[node])
```

Listing 9: Applying a random move enabled by the satisfied predicate.

```
old_node = copy.deepcopy(node)
old_node_data = copy.deepcopy(graph.node[node])
old_neighborhood = copy.deepcopy(neighborhood)

next_move = random.choice(self.ruleset[satisfied_predicate])
next_move(graph.node[node], neighborhood)
```

Listing 10: A model testing session using the command-line utility.

```
$ ./ssa-tool run "Independent Set" \
            from examples/ind-set.ssax \
              on test.gml \
                 --non-interactive
Welcome to SSA-Tool, version 1.
Running:
  Algorithm: "Independent Set"
       from: "examples/ind-set.ssax"
         on: "test.gml"
    (format: "gml")
Read Graph:
[(0, {'id': 0, 'label': 0, 'marked': 0}),
 (1, {'id': 1, 'label': 1, 'marked': 1}),
 (2, {'id': 2, 'label': 2, 'marked': 0}),
 (3, {'id': 3, 'label': 3, 'marked': 0}),
 (4, {'id': 4, 'label': 4, 'marked': 0})]
History:
[{'move': move 'mark node',
  'neighbors': {0: {'id': 0, 'label': 0, 'marked': 0}},
  'new node': 4,
  'node': (4, {'id': 4, 'label': 4, 'marked': 0})},
 {'move': move 'mark node',
  'neighbors': {0: {'id': 0, 'label': 0, 'marked': 0}},
  'new node': 2,
  'node': (2, {'id': 2, 'label': 2, 'marked': 0})},
 {'move': move 'mark node',
  'neighbors': {0: {'id': 0, 'label': 0, 'marked': 0}},
  'new node': 3,
  'node': (3, {'id': 3, 'label': 3, 'marked': 0})}]
Stable Graph:
[(0, {'id': 0, 'label': 0, 'marked': 0}),
 (1, {'id': 1, 'label': 1, 'marked': 1}),
 (2, {'id': 2, 'label': 2, 'marked': True}),
 (3, {'id': 3, 'label': 3, 'marked': True}),
 (4, {'id': 4, 'label': 4, 'marked': True})]
```

tested, but consistent crashes occur when attempting to delete multiple characters via a selection. See Task 2.

When you are ready to provide a definition, simply enter it in the large text area beneath the Predicates » File and Predicates » Description fields. The definition must be in Python 3 syntax [23].

Warning: Due to the imperfect implementation of the interface, the last line of the definition string is *always* lost. Unless you terminate the definition with a newline character,

the last line you give *will be lost*. For example, if the definition is only one line (say `return n['marked']`), this entire definition *will be lost*. See Task 4.

Warning: When you have finished editing the definition for a predicate or move, the file is immediately written to logical. This action cannot be undone and is unaffected by the File Manager » Save Bundle operation. This can be mitigated by Task 13.

5.2 Documentation

All entities — predicates, moves, rules, and algorithms — support documentation in various manifestations. In the graphical interface, these fields include the author, a name,[7] a description, and TEXnical documentation.

TEXnical documentation When exporting an algorithm as a PDF (processed via TEX/Ti*k*z),[8] the TEXnical documentation for each entity is given special treatment. As bundles should be self-sufficient, there is an easy syntax to use to denote the properties of a node. Considering INDSET, the quality of being 'in the set' may be denoted as `"marked"(n) = 1`. Note the literal double quotes (the single character `"`) around the function name; in addition to clearly describing the purpose of a value, this notation will be replaced as appropriate by the more verbose (but correct) `\operatorname{marked}`.

It should be noted that TEX export is not yet functional in the core tool; this syntax is merely set in place to guide further development. See Task 18.

5.3 Algorithms

Algorithm assembly is fully supported using the graphical interface. To create an algorithm, simply click the Algorithms » add button underneath the list of algorithms to the right. (This list will be empty in a new bundle.) Once this has been done, giving the Algorithm a name (via the Algorithms » Name field) will update its text in the listing accordingly.

The process is similar to add new rules to the algorithm. To create a new rule, simply press the Algorithms » Rules » add button under the list of rules. Provide a predicate via the Algorithms » Rules » Predicate (which is color-coded by the list of predicates in the bundle) and transfer moves between the master list on the left (similarly populated) and the rule's list on the right using the angle bracket buttons (> and <) between them. The predicate text will be red if the predicate does not exist and will become black when it is recognized as a predicate.

Special Instructions

Unfortunately, this tab is especially buggy (see Task 1). Special care must be taken to ensure that the interface does not crash. For reference, consider the following script when creating these entities.

[7] Required and must be unique. See Appendix A.

[8] Not yet supported via the graphical interface. See Task 18.

This script creates the algorithm IndSet as part of a new bundle.

1. Open the interface from the terminal: `python3 gui.py`
2. Navigate to the Algorithms tab by clicking on it.
3. Create a predicate to test if a node should mark.
   (a) Navigate to the Predicates tab.
   (b) Click Predicates » add to create a new predicate. Select the predicate when it appears in the list box.
   (c) Click Predicates » Name and, using either the arrow keys or the mouse, navigate to the end. Be careful not to select any text; this has not been tested (and in some cases, causes an exception). See Task 2.
   (d) Pressing backspace ← to delete the existing text, rename the predicate to "node should mark".
   (e) Give the entity additional metadata to the predicate through the Author, Date, TeX, Description, and File Name fields.
   (f) Provide the definition provided in Listing 13, taking care to add the terminal newline (see Task 4).
4. Create a predicate to test if a node should unmark. Repeat the above for a predicate named "node should unmark", with the definition provided in Listing 14.
5. Select the previous predicate to save the current one. (See Task 13.) After this, the interface should look similar to Figure 4.
6. Create a rule to mark a node. Repeat the process for a move (under the Moves tab) named "mark node" with the definition given in Listing 11.
7. Create a rule to unmark a node. Repeat the process for a move named "unmark node" with the definition given in Listing 12.
8. Select the previous move to save the current one. (See Task 13.)
9. Create an algorithm entity called "Independent Set".
   (a) Click Algorithms » add to create an algorithm entity.
   (b) Edit Algorithms » Name using the same procedure as before. Call the algorithm "Independent Set".
   (c) Repeat this process for Algorithms » Author and Algorithms » Date, being careful to give Date in the correct format (see Task 7).
10. Add a rule to mark a node.
    (a) Click Algorithms » Rules » add to create a new rule.

(b) Repeat the process above to give the rule the name "ENTER", taking care to edit the correct field (Algorithms » Rules » Name).

(c) Edit Algorithms » Rules » Author as appropriate.

(d) Edit Algorithms » Rules » Predicate to "node should mark". When the name given matches the name of an existing predicate, it will turn color from red to black.

(e) In the listbox on the left, select "mark node" and click Algorithms » Rules » >.

11. Add a rule to unmark a node. Repeat the above for a rule named "LEAVE" with the predicate "node should unmark" and the single move "unmark node".

12. Select the previous rule to save the current one. After this, the interface should look similar to Figure 5.

13. Navigate to File Manager.

14. Give the a full path of the file you wish to write (or a path relative to the working directory the interface was started from) in File Manager » Path.

15. Click File Manager » Save Bundle.

16. Close the interface. The bundle now exists at the path specified in File Manager » Path.

Warning: When 'overwriting' a bundle, the operation is union-with-overwrite. This means that the tool will not overwrite files that do not exist in the bundle it wishes to write, but it will overwrite those it has a record for. This means that the bundle description document *will* be overwritten, but perhaps other files that existed previously will remain. If this is a possibility, it is a good idea to load the bundle first so that different names can be given to entities.

5.4 Creating, Loading, and Saving Bundles

To save and load bundles, navigate to File Manager tab. Enter the *full*, *absolute* path of the bundle you wish to load and, taking special care not to press File Manager » Save Bundle (as this will overwrite it), press File Manager » Load Bundle. Successively loading bundles will aggregate them under a union operation to allow for distributed work toward larger and larger bundles.

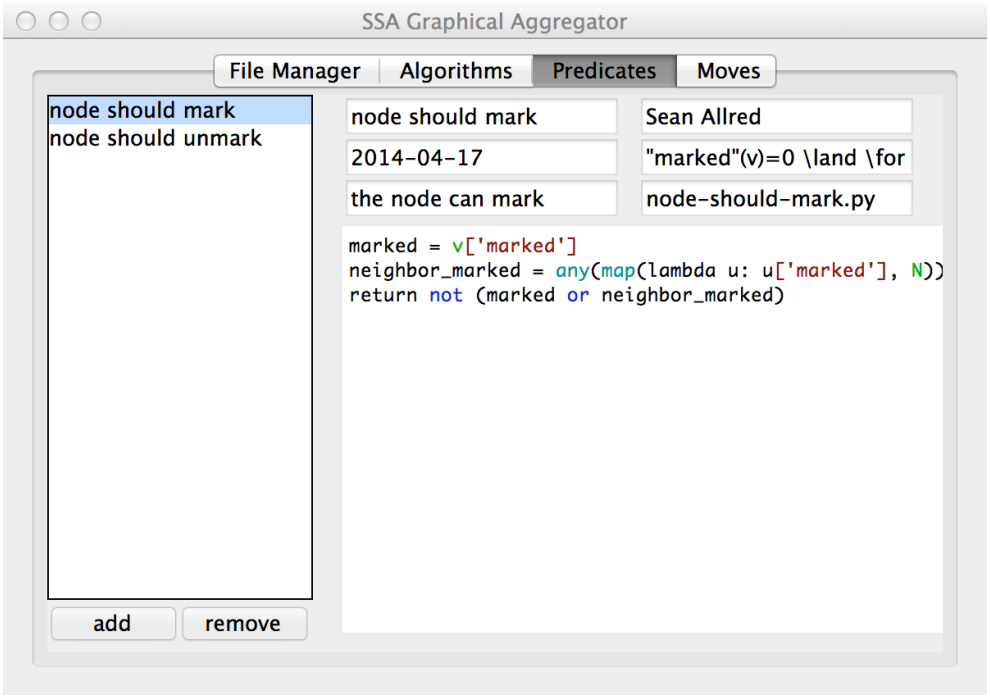

Figure 4: The interface looking at a completed predicate.

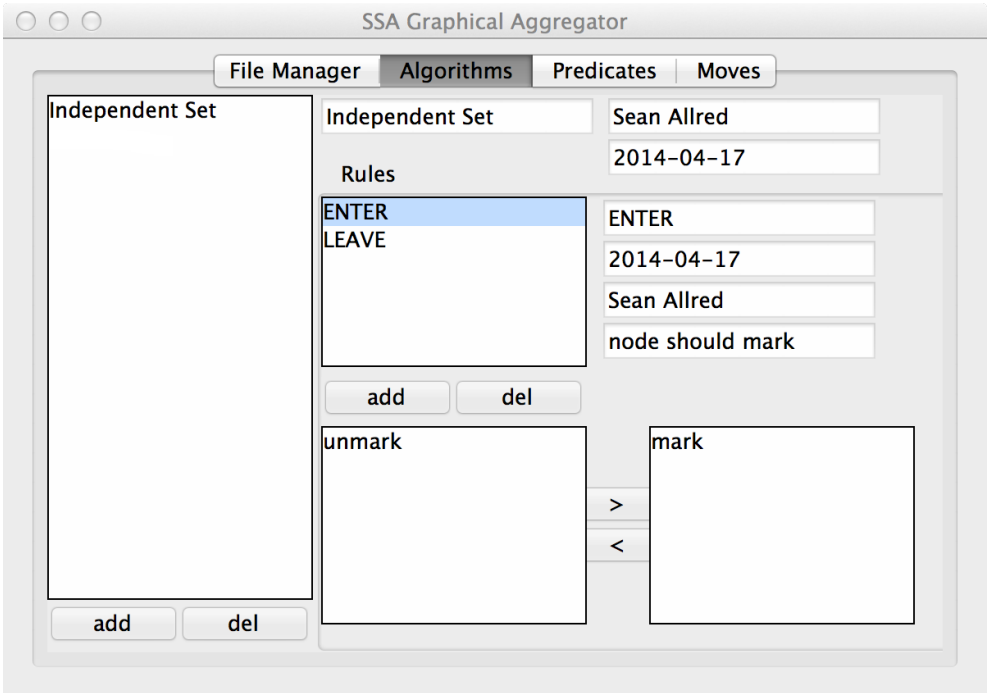

Figure 5: The interface looking at a completed algorithm.

6 Further Work

There are many things that, due to the sheer time constraints of the project, have been left undone. Some are tasks that I consider to be bugs or areas of severe lacking, but some are critical bugs of the interface that should be taken into account when using it.

6.1 Remaining Tasks and Bugs

Critical Bugs

Task 1. The Algorithms tab should be completely re-written. Due to time constraints, very poor decisions (with respect to best practice) were made in the interest of a working product. Crashes are common if and when the user deviates from the script prepared in section 5.

Creation of predicates and moves via Predicates and Moves works fine, with the exception of the bug discussed in Task 4.

Task 2. When typing in a new name for any entity (algorithm, rule, predicate, or move), consistent crashes occur when multiple characters are selected and replaced together. The cause of this bug is unknown.

Non-Critical Bugs

Task 3. Leverage the Nose system [18] to automate tests. Randomized graph construction tools have been developed in the repository (under `simulation‣generators`); these will be helpful. Several tests have already been written for the generator itself — it would be best to follow the example set forth there as tests hope to be efficient with the larger data structures the they would create. This task should be completed with Task 9.

Task 4. In the interface, when switching the active predicate for editing (from P_1 to P_2, say), the last line of the definition of P_1 will be lost. This may be solved by changing how the definition is saved in the core objects from a list of lines to a single string. This will make the interface with the `Text` widget from Tkinter much smoother.

Task 5. Prohibit the permanent modification of graph entities when applying a predicate. This can be effected as part of Task 11.

Task 6. On the Predicates and Moves tabs, rudimentary syntax highlighting has been implemented, but does not refresh itself when the text is edited. This should be made more reliable. See Task 10.

Task 7. Dates have a very strict format: `YYYY-MM-DD`. A date-time parsing library should be leveraged to consistently parse information provided in a variety of formats.

Essential Extensions and Re-Designs

Task 8. There is currently no graphical interface to the algorithm testing functionality. This should eventually be married with the visualizer (Task 17), but even a simple interface to select a graph and run the test — piping output on-screen — would suffice.

Task 9. Continue work on and thoroughly test the interface. There are likely many issues that have not yet been discovered, and there are certainly best-practices in human–computer interaction that are not being followed. This task should be completed with Task 3.

Task 10. The predicate/move definition area in the graphical interface should have reliable syntax highlighting. Code may be borrowed from the IDLE tool for Python, which aside from syntax highlighting, also supports interactive documentation. Such interactive documentation may potentially take the place of the template comments. See Task 6.

Task 11. Within the Predicates and Moves tabs, the definition area needs some rethinking. I imagine two ways it can be made more usable: • providing a second, resizable dialog that consists solely of the definition area, or • providing horizontal (and vertical) scroll bars to navigate the definition. I would personally hope definitions for such basic components to remain small (in favor of working with Task 12), but this obviously cannot be assumed.

Task 12. Currently, a new class is being created for every predicate and move object loaded from a bundle in order to accommodate their ability to be called. This can be simplified by loading the definition into the temporary function and then setting a 'hidden' local function-variable (say, `_def`) to this definition. The class-level `__call__` may then simply pass the call onto this member variable.

Task 13. Under the current interface, frequent saves must be made in order to maintain state. This can by mitigated with the automated use of temporary files. These saves are effected by switching the entity in question; i. e. to save a predicate, switch to a new predicate. This will write the old one to memory. (Alternatively, this can be solved by writing to logic as a callback hooked onto an `Entry`'s edit event — in reality, this would be a trace on the string variable.)

6.2 Potential Extensions

Task 14. In the bundle description document (Appendix A) and the graphical interface (section 5), a date is a timestamp of the last edit. This can be automated, but is not currently. It requires tracking changes made to the bundle during the current session, but shouldn't be too difficult. Currently, all dates are required to be in `YYYY-MM-DD` format, otherwise the parser crashes; see Task 7.

Task 15. Gephi [2] is a very popular tool used in graph theory research. Equipped with an extraordinarily powerful visualizer, it is well-suited to run simulations using advanced graphical methods that may not be easily achievable within PyGame [19], what would be the standard visualizer.

Gephi is written in Java and some work will need to be done to glue the two together, but it is

perhaps a very useful tool to have integrated into an already-pervasive utility.

Task 16. The core utility keeps a history of each run of an algorithm on any graph. Unfortunately, there are no obvious interfaces (within NetworkX [14] or otherwise) with which to export this to a standardized format.

One such format is `gexf`: the Graph Exchange Format. Notation is available to represent change over time with respect to a node, but there is no facility within NetworkX's exporters to use this.

To some extent this is to be expected; the model that NetworkX uses to represent a graph does not seem to allow any representation of change. As such, a custom exporter will need to be written (perhaps using [11]) that can take a graph and a list of deltas and can output the correct notation. This notation is detailed in the official documentation [13].

Task 17. A prototype visualizer has been written for use with the tool, but is not yet ready for release. It is still available within the GitHub repository at `vermiculus/ssa-tool`. It leverages the graph representation and layout facilities of NetworkX [14] and uses PyGame [19] to draw the graph on-screen. Given the nature of Python reference types, any and all changes to the graph will be immediately reflected on-screen. This task is merely concerned with the integration of these two tools.

Task 18. It would be exceptionally useful to have printed output from the tool that details the definition of an algorithm or set of algorithms within a bundle, perhaps even typeset histories (using Ti*k*z) of a run. TEXnical documentation is being kept to allow this, but serious work in text processing will need to be done in order to do this cleanly.

Task 19. In order to fully facilitate reproducible research (as reproducible as it could be, considering the arbitrary nature of selection), graphs and corresponding tests should be included as part of bundles.

Task 20. Implement a history browser within the graphical interface. This is a simple parsing of the data-structure described in section 3, and may also be integrated into the visualizer from Task 17.

Task 21. PyYAML [20] is able to read and write arbitrary YAML attributes to and from Python objects, but there is no way to explore any additional attributes that might exist within the file. It may be useful to have a sort of 'data explorer' that can browse and edit these attributes.

Task 22. Some predicates and moves may eventually be defined as the aggregation of others. Design and implement a syntax to refer to predicates and moves defined within the bundle within the definition of the predicate or move itself. I would suggest Noweb syntax [21], but I am far from attached to it (*especially* if the syntax is used elsewhere in Python 3).

Task 23. The tool currently implements only the centralized daemon model of execution, but other models exist that are far more distributed. (See for example [7].) There are many real-world simulation possibilities when using a truly distributed system, but the current implementation is

single-threaded.

The project may be tweaked and re-targeted toward the Stackless variant of Python [22]. Stackless supports multi-threading as a core goal, and this task would likely be easiest to complete with it.

This runs the risk, however, of breaking NetworkX [14], PyYAML [20], or PyGame [19].

6.3 Further Research Now Possible

Task 24. Given how it represents algorithms as a population of predicates, moves, and rules, `ssa-tool` could feasibly be used to devise genetic algorithms that generate self-stabilizing algorithms. To do so effectively would require a quick and completely autonomous evaluation of effectiveness as a fitness function — perhaps how quickly the algorithm converges or if it converges at all.

7 Reflection

This project has been long and difficult. After starting soon after my return to campus in the fall of 2013, the project is finally coming to a close at the time of writing. There are many things left to be done (see section 6), but there are also many things that have been accomplished. Some of these accomplishments are in and of the project itself, but the greater accomplishments — those that will remain with and influence me — are those practices and perspectives that I've acquired through the course of this capstone experience.

7.1 Restraints per Constraints

Like several of my peers during this project, the first great challenge was perhaps initial restraint and discipline. Brooks described in his book *The Mythical Man-Month, Anniversary Edition: Essays On Software Engineering* a phenomenon known as the 'second-system effect'. The phenomenon describes the woes of even the careful the software architect,

> This second is the most dangerous system a man ever designs. […] The general tendency is to over-design the second system, using all the ideas and frills that were cautiously sidetracked on the first one. The result, as Ovid says, is a "big pile". ([4])

My reflection on my general project-management decisions bring this passage to mind not because I added such frills, but because I fell victim to my own reckless hubris in believing I knew enough about the domain to begin modeling it in a program. Logically representing the system without a full — or even working — knowledge of the current research and definitions set the project back at least a month, possibly longer. Once I relented to *reading* before I *wrote*, the project accelerated to the point of noticeable and measurable progress.

This naturally led to the second great challenge: self-motivation and discipline (again). As I began to research the topic in earnest, my time was consumed for a short time with reading and absorbing

information. This was a short-lived period of near-frenzied research — in print, online, and in-person — where I read a wide variety of different articles that utilized or talked about fault-tolerant systems. (This frenzy was started by reading "Distance-k Information in Self-stabilizing Algorithms" by Goddard et al. for an assignment in a separate class.) Before other time constraints won over with the sheer pressure of impending due dates, I was able to more clearly see the project's place in the overall arena of research. This clarity of purpose — coupled with the understanding offered by the research itself — unsurprisingly gave me the necessary tools and motivation to fully design the concepts and relationships of the underlying system.

Thus, the core engine — `ssa-tool` itself — was written and re-written time and time again. Each successive iteration took very little time when considered on its own, but each was significant in marking an increase in understanding. The final engine — including the time for all previous iterations and less the persistent aspects — was working within a month. As happy as I was to complete this milestone, what the coming months had in store would overturn the perspectives I had assumed in creating this pre-release.

7.2 Thinking Ahead

The project's initial goal was 'simple':

> Create a graphical tool for the simple creation, manipulation, and evaluation of self-stabilizing algorithms while minimizing the purely technical requirement of the user.

The project was to manifest itself in a fully-featured graphical user interface.

I had created a *very* robust system that worked without a graphical interface. It was (and still is) my assumption that a good system relies on an even better engine, and the command-line interface was that engine. Unfortunately, it turned out that this system was not very conducive to interfacing with a real-time window system. A few minor redesigns needed to be made to glue the two together, and the interaction is still largely imperfect (as the task list in section 6 is painfully indicative of).

The more fundamental problem was the very choice of language. For a command-line interface, Python was an excellent choice. It was expressive and concise, file operations were simple and plain-text serialization was well-supported with PyYAML [20]. Being comfortable on the command-line and a big fan of plain-text data, this stage of development was — for lack of a more perfect word — wonderful.

Graphical interfaces in Python, however, are *not* so wonderful.

If I had considered the goals of the project more seriously (and perhaps recognized that I — with my own preferred uses — was not the primary user), I would have begun the implementation in C# (given the wonderful interface design tools available with Visual Studio). This would have absolutely *thrashed* portability, but would have made the tool far more stable in its prescribed use case. There are several tools that would require re-implementation (such as the graph drawing capabilities I had written for PyGame [19]; see Task 17), but more research and simple resolve would have solved this issue far quicker and more effective than what has come to pass.

7.3 The Grand Distraction

Everyone loves well-documented software. My experience with Graph# — and the eventual migration to a more convenient platform — encouraged me to take on the largest project of my academic career in the most foreign — but *useful* — actual software development paradigm that I had encountered to date. Writing the document as a literate program [16] forced me to write good documentation for the project, but perhaps this level of documentation was not the best use of my time in the beginning stages of the project. The first commit activity in the source code dates to 10:30pm Christmas Eve, more than halfway through the year-long project. While I will never regret the use of literate programming in this project — it is this use that largely taught me what the project *was* — I should have focused far more on the actual project rather than the format in which it would be written and maintained.

Additionally, the desire to produce write a 'perfect' project — following all Python conventions — took a large portion of time. Working in a literate document certainly did not require this. If I so chose, it would have been inconsequential to tangle to one file and one file only. The desire to make the tool easily navigable outside of this format — while a fine desire for a second iteration of this prototype — was a significant time-sink.

It so happened that — in the production-frenzy of the final weeks of term — I began to work solely from what began as a 'test' file. This was not a literate environment. While it allowed me to work far quicker in some cases — there was no intermediate tangling step — it was a *nightmare* to keep everything organized. A better approach would have been to work initially from these 'test' files — no literate programming or even significant documentation — and then transfer and 'untangle' them into the literate document, writing the essay-documentation for the current component while it was still familiar to me. Without a literate backbone to encourage it, the project's source lacks the high degree of documentation and structure that I pride myself in producing for my other projects. Before the project enters my portfolio of work that I am *proud* of, this is first and foremost.[9]

7.4 Special Thanks

There are many resources, both online and personal, that have made this research project possible. First and foremost is my advisor, Dr. Alan Jamieson, who suggested this project and has continually guided its development through clear explanations of the underlying concepts. Second is the research community surrounding Python, producing such useful tools as NetworkX [14] (logical representation), PyYAML [20] (persistent representation), and PyGame [19] (visual representation). Finally are the many friends who graciously listened to my explanations of this research and often asking questions of me that I would not have asked myself, always enhancing my understanding of the research domain and its audience.

[9] As a sidenote, another hindrance — completely unnecessarily — was the lack of GitHub highlighting for the literate document. This has thankfully been resolved as `github/markup#253`.

A Bundle Description Document Specification

The file format that `ssa-tool` natively works with is actually an OS X-style 'package' — it is a standardized directory with a specific extension. The recommended extension to use is `ssax`, but the tool itself does not require it. An example directory structure can be seen in Figure 6, but the following sections will formally describe each portion.

A.1 Predicates and Moves

Predicates and moves are stored under the separate directories `predicates` and `moves` as in Figure 6. Each file in these directories is a (potentially very short, see Listing 1) Python 3 script that defines the body of a templated function which is hardcoded as in to Listing 2.

Within the bundle description document, predicates and moves must be set off with the YAML 'tags' `!Predicate` and `!Move`, respectively. The recommended attributes[10] for these objects — *required* for the graphical interface — are as follows:

name a unique name (with respect to class)[11] for the component

description a description of this component (e. g. this predicate will be true if and only if the node is unmarked and isn't adjacent to any marked nodes).

author the author(s) of this component

date the date of the last edit (not automatically tracked; see Task 14)

filename the name of the file (without the directory) containing the definition for the component. That is, `mark.py` rather than `moves‣mark.py`.[12]

Per Task 18, there is a final *recommended* attribute:

tex TEXnical documentation for the component

A.2 Rules

Rules are declared as items (with the `!Rule` tag) under the `rules` attribute of an algorithm. They are given much the same attributes as predicates and moves, but with additional attributes of `predicate` and `moves` that map to the name of a predicate and names of moves within the bundle.

A.3 Algorithms

Algorithms (tagged as `!Algorithm`) are simply the aggregation of their rules, with additional attributes for `name`, `author`, and `date`.

[10] See Task 21.

[11] That is, I could have a predicate named `marknode` and a move named `marknode`, but I can't have two of either.

[12] Filenames — as long as they are consistent — are arbitrary. The core tool has not been tested with atypical filenames (see Task 3). It is recommended that filenames are kept to alphanumeric characters and hyphens.

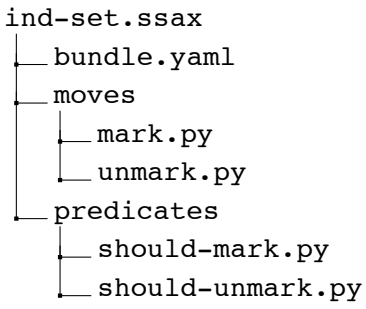

Figure 6: The layout of an example algorithm, IndSet

Listing 11: `moves‣mark.py`

```
v['marked'] = True
```

Listing 12: `moves‣unmark.py`

```
v['marked'] = False
```

Listing 13: `predicates‣unmarked-and-neighbors-unmarked.py`

```
marked = v['marked']
neighbor_marked = any(map(lambda u: u['marked'], N))
return not (marked or neighbor_marked)
```

Listing 14: `predicates‣marked-and-neighbor-marked.py`

```
marked = v['marked']
neighbor_marked = any(map(lambda u: u['marked'], N))
return marked and neighbor_marked
```

Listing 15: `bundle.yaml`

```yaml
--- !Move
name: mark node
description: mark this node
author: Sean Allred
date: 2014-05-17
tex: '"marked"(n) = 1'
filename: mark.py
--- !Move
name: unmark node
description: unmark this node
author: Sean Allred
date: 2014-05-17
tex: '"marked"(n) = 0'
filename: unmark.py
--- !Predicate
name: marked and neighbor marked
filename: marked-and-neighbor-marked.py
description: two adjacent nodes are marked
author: Sean Allred
date: 2014-05-17
tex: '"marked"(n) = 1 \land \exists v \in N(n) : "marked"(v) = 1'
--- !Predicate
name: unmarked and neighbors unmarked
filename: unmarked-and-neighbors-unmarked.py
description: All nodes in N[n] are unmarked
author: Sean Allred
date: 2014-05-17
tex: '"marked"(n) = 0 \land \forall v \in N(n) : "marked"(v) = 0'
--- !Algorithm
name: Independent Set
author: Sean Allred
date: 2014-05-17
rules:
  - !Rule
    name: unmark if touching
    description: if a neighbor is marked, unmark
    author: Sean Allred
    date: 2014-05-17
    predicate: marked and neighbor marked
    moves: [ unmark node ]
  - !Rule
    name: mark if able
    description: if no neighbors are marked, mark
    author: Sean Allred
    date: 2014-05-17
    predicate: unmarked and neighbors unmarked
    moves: [ mark node ]
```

Listing 16: A standardized widget creator

```
def new(cls, widget_dictionary, name, **kwargs):
    print('Creating␣widget␣{0:<14}␣under␣{1}'.format(cls.__name__, name))
    return cls(widget_dictionary[name][1], **kwargs)
```

Type	File Manager	Algorithm	Predicate	Move
Variables	`fmv`	`agv`	`pdv`	`mvv`
Functions	`fmf`	`agf`	`pdf`	`mvf`
Widgets	`fmw`	`agw`	`pdw`	`mvw`

Table 1: Standardized interface dictionaries, indexed by description.

B Logical Organization of Graphical Interface

The graphical interface is logically organized into several dictionaries based on tabs (see Table 1). The keys of these dictionaries are string-descriptions of the purpose of the entity, and the value is either a single variable or function (in the cases of `??v` and `??f`) or a pair (as a `tuple`) of positional coordinates and the widget itself (within `??w`). The coordinates are relative to the upper-left corner of the containing frame. If the first element of the tuple is `None` (instead of a pair), the widget will simply be 'packed' instead of 'placed'. To create the widgets, a utility function is declared that will assemble the widget and log any information about it that is necessary, providing a centralized interface to manage data bindings (see Listing 16).

For example, to add some text entry field for a predicate's name, I would call `new` from Listing 16 with the arguments `(Entry, pdw, 'tab', textvariable = pdv['predicate name']).`

Bindings between graphical elements and their logical counterparts are maintained in a dictionary called `bind`, primarily indexed by class and secondarily indexed by the object's name. This dictionary is reset upon calling `refresh`, which purges data from the interface and rebuilds it using the `bundle` data structure.

About the Dictionary-Based Approach The choice to use dictionaries was spurred by what has proven to be a common problem in my own software implementations. Within the dynamic functions of the interface (e. g. the function to transfer moves to and from a rule), it is always possible that the widget the function is altering has not yet been created. By using dictionaries, resolution is deferred until the interface is completely built, thereby side-stepping the need to carefully order the creation of widgets and instead order them in a way that makes sense or is aesthetically pleasing.

References

[1] A. Arora and M. Gouda. "Closure and convergence: a foundation of fault-tolerant computing". In: *Software Engineering, IEEE Transactions on* 19.11 (Nov. 1993), pp. 1015–1027. ISSN: 0098-5589. DOI: 10.1109/32.256850.

[2] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. *Gephi: An Open Source Software for Exploring and Manipulating Networks*. 2009. URL: http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154.

[3] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. *YAML Ain't Markup Language Version 1.2*. Oct. 1, 2009. URL: http://www.yaml.org/spec/1.2/spec.html.

[4] F. P. Brooks. *The Mythical Man-Month, Anniversary Edition: Essays On Software Engineering*. Pearson Education, 1995. ISBN: 9780132119160. URL: http://books.google.com/books?id=Yq35BY5Fk3gC.

[5] Edsger W. Dijkstra. "Self-stabilizing Systems in Spite of Distributed Control". In: *Commun. ACM* 17.11 (Nov. 1974), pp. 643–644. ISSN: 0001-0782. DOI: 10.1145/361179.361202. URL: http://doi.acm.org/10.1145/361179.361202.

[6] Shlomi Dolev, Anish Arora, and Willem-Paul de Roever. "Self-Stabilization". online. A collection of abstracts from a talk. 1998. URL: http://www.dagstuhl.de/Reports/98/98331.pdf (visited on 03/01/2014).

[7] Shlomi Dolev and Ted Herman. "Dijkstra's Self-Stabilizing Algorithm in Unsupportive Environments". English. In: *Self-Stabilizing Systems*. Ed. by Ajoy K. Datta and Ted Herman. Vol. 2194. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 67–81. ISBN: 978-3-540-42653-0. DOI: 10.1007/3-540-45438-1_5. URL: http://dx.doi.org/10.1007/3-540-45438-1_5.

[8] Effbot.org. *An Introduction to Tkinter*. 2005. URL: http://effbot.org/tkinterbook.

[9] fintler. *BalanceMQ*. Version b99dc7c5d6329d2906012cae8147cac223bf8a6e. Apr. 29, 2013. URL: http://github.com/fintler/balancemq (visited on 04/10/2014).

[10] Martin Gairing et al. "Distance-two information in self-stabilizing algorithms". In: *Parallel Processing Letters* 14.03n04 (2004), pp. 387–398.

[11] Paul Girard. "PyGEXF". Sept. 10, 2013. URL: http://github.com/paulgirard/pygexf.

[12] Wayne Goddard et al. "Distance-k Information in Self-stabilizing Algorithms". In: *Structural Information and Communication Complexity*. Ed. by Paola Flocchini and Leszek Gąsieniec. Vol. 4056. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 349–356. ISBN: 978-3-540-35474-1. DOI: 10.1007/11780823_27. URL: http://dx.doi.org/10.1007/11780823_27.

[13] GEXF Working Group. *GEXF File Format*. Version 1.2. 2009. URL: `http://gexf.net/format/schema.html` (visited on 04/10/2014).

[14] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring network structure, dynamics, and function using NetworkX". In: *Proceedings of the 7th Python in Science Conference (SciPy2008)*. Pasadena, CA USA, Aug. 2008, pp. 11–15.

[15] HPC. *libcircle*. Version f8714e61742e7262918c568a76022a5b13f3fade. Feb. 13, 2014. URL: `http://github.com/hpc/libcircle` (visited on 04/10/2014).

[16] Donald E Knuth. "Literate programming". In: *The Computer Journal* 27.2 (1984), pp. 97–111.

[17] Jharrod LaFon, Satyajayant Misra, and Jon Bringhurst. "On Distributed File Tree Walk of Parallel File Systems". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 87:1–87:11. ISBN: 978-1-4673-0804-5. URL: `http://dl.acm.org/citation.cfm?id=2388996.2389114`.

[18] Jason Pellerin et al. *Nose: Nice Testing for Python*. Version 1.3.1. 2014-03-10. URL: `http://github.com/nose-devs/nose` (visited on 04/10/2014).

[19] Pygame. *Pygame Documentation*. 2014. URL: `http://pygame.org/docs/ref` (visited on 04/10/2014).

[20] PyYAML. *PyYAML: a YAML parser and emitter for Python.* Version 3.11. Mar. 26, 2014. URL: `http://pyyaml.org/wiki/PyYAML` (visited on 04/10/2014).

[21] Norman Ramsey. "Noweb: A simple, extensible tool for literate programming". 2011. URL: `http://www.cs.tufts.edu/~nr/noweb`.

[22] Stackless. *Stackless Python*. 2014. URL: `https://stackless.readthedocs.org/en/latest` (visited on 04/10/2014).

[23] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Paramount, CA: CreateSpace, 2009. ISBN: 1441412697, 9781441412690.