Exercise 1, Artificial Intelligence

Heuristic search using A*

Martin Persson
Computer Systems Engineering 3
Halmstad University 2006

d02mape@stud.hh.se

Foreword

This report is the documentation for the exercise part of the course “Artificial Intelligence” taken by the author at Halmstad University during spring 2006. This first exercise focuses on the development of a simple A*-based path-finding agent. Since the author has little experience using the Python language, he has taken this assignment as an opportunity to increase his skills in the said programming language, as well as try it out as an alternative to the languages he is already proficient in.

Since Python seems to be a somewhat less widely used language among the rest of the students, a brief introduction to the language is included, as well as an overview of some of the practices commonly used in conjunction with it. The authors thoughts on the programming language is also part of the report.

How to use the main program

The script implementing the agent is launched from the command line like any other program. One argument is required, the map filename that the agent should solve. Example:

```
$ ./ai_exercise_1.py maps/default.map
```

This will launch the program with the map “default.map”. You can then choose to run it interactively or fully automatic. You choose by entering the number that corresponds to the choice you want to make. In interactive mode, you can see each step of the solving process, complete with the nodes in the open and closed list. Open nodes are marked by a plus sign (“+”), and closed nodes by a minus sign (“-”).
Once a solution has been found, the final path will be drawn with “o” characters to visualize it.

If you would like to skip ahead a certain number of iterations (watching each expansion gets old kinda quick), just enter the number you want to skip and press `enter`. The solving process is then advanced at full speed for the specified number of steps. It then stops and displays the map again. If a solution is found before the requested number of steps have transpired, the simulation is halted and the solution is shown.

How to use the map conversion tool

Included in this package is a small tool called map_convert.py. It is an extremely simple script that is used to generate a *weighted* map from a standard *binary* type map. It replaces all whitespace characters with a random integer between 1 and 9, thus giving us a “terrain map”.

The tool is used from the command line, using the following syntax:

```
$ ./map_convert.py source.map target.map
```

Where `source.map` is the filename of the original binary map, and `target.map` is the filename of the new, weighted map. Using a weighted map as an input file is illegal, and the result unspecified.
The map conversion tool is flawed in its design, see the section titled “Handling weighted maps”.

Table of Contents

Foreword....i
How to use the main program....i
How to use the map conversion tool....i
Overview....1
About Python....1
    Python basics....1
    Python classes....2
The A*-algorithm....3
The path-finding agent script....4
    Class overview....4
        The “Position” class....4
        The “Graphnode” class....4
        The “Worldmap” class....5
        The “Scenario” class....5
        The “Agent” class....5
    The main script....6
Implementation and development details....6
    The map parsing task....6
    Handling weighted maps....8
    Generation of straighter paths....8
Using Python for the assignment....9
    Python programming practices implemented....9
        The __slots__ directive....9
        Loose typing....9
        Aggressive dictionary usage....10
    Python versus traditional languages....10
References....12
Appendix A....i
Appendix B....i
Appendix C....i

Overview

As part of the course Artificial Intelligence at Halmstad University, we are required to implement an agent that uses the well-known A*-algorithm to solve a simple maze problem. We must implement a program (in a language of our choice) that can load a simple text file containing the description of a maze, consisting of different squares (passable, impassable, possibly with varying “occupational cost”) as well as a start and goal location.
The agent must then be able to find it's way to the goal node using the A*-algorithm. The agent's progress must be visualized, and the user should be able to watch each iteration of the problem solving process on the screen. In addition, the agent must be able to generate paths that are as straight as possible at the user's discretion.

About Python

I have chosen to implement this agent in the Python language. It is a interpreted, dynamically typed language suitable for rapid prototyping. It has automatic garbage collection, all access is performed “by reference” and it has support for object-oriented programming.

Some of it's more unusual properties includes the use of indentation (as opposed to the more widespread use of curly braces) to define blocks of code. In addition, variable names are created on-the-fly, without the need of prior, explicit declaration. In this respect, it behaves like a “slack” scripting language, such as PHP. On the other hand, Python does not require the user to prefix all variable names with a special symbol, as do most scripting languages.

Many features in Python is implemented by special naming conventions. For example, the overloading of an operator for a class is done by a creating a regular method with a special, reserved name, typically starting and ending with a pair of underscore marks. For example, the addition operator (+), is implemented by defining a method called `__add__()`.

Python basics

Standard container classes in Python includes the `list` (an ordered, sequential-access collection), the `dictionary` (associative, unordered container, similar to Perl's `hash`), the `set` container and its immutable cousin, `frozenset`. In addition to those, there exists a number of support container types such as iterators and range (enumeration) types, but those are seldom or never created explicitly by the user, but rather by language constructs. Python also has a native string class, dubbed `str`.

The Python equivalent of `NULL` (C, Java), `NIL` (Perl) and `0` (C++) is `None`.

Strings in Python can be enclosed in either single or double quotes. Triple-qouted strings can span multiple source lines. Strings are concatenated using the plus sign (+). Formatted string output is usually performed using the `%` operator, which takes a `printf`-style style string as left hand operand, and a tuple (an immutable list) of values as the right.

In Python, functions are declared by the def keyword. Then comes the argument list (enclosed in

brackets, terminated by a colon), and after that the function body.

```
def my_function (arg1, arg2):
    """A simple test function."""
    print "Argument 1: " + str(arg1)
    print "Argument 2: " + str(arg2)
    return True
```

Text 1: A simple function definition.

The above example shows the use of the documentation string in order to describe the function for other users. Python uses this in addition to regular comments to make it easier to maintain code documentation, much in the spirit of `javadoc`.

The return value is never declared in a Python function, it is determined at run-time.

Python classes

```
class MyClass (object):
    """This is a completely fake class."""

    def __init__ (self, name = "Mr Fooblesworth):
        self.name = name

    def doStuff (self):
        pass
```

Text 2: A simple "new-style" class with a constructor and an empty method.

Classes are declared by the class keyword, followed by the class name, and any parent classes enclosed in brackets. The line is terminated by the colon. Then the body of the class follows, which consists mainly of method bodies.

Python classes require no explicit member declaration. Members are created when referenced, much like regular variables do. The convention is to create them all in the "constructor" method, called `__init__()`, with valid invariants. Please note that the term "constructor" isn't really valid for the `__init__()` method, since it has no real part in the construction of the actual object in memory.

This is shown in the example above, along with the use of a default argument and the `pass` keyword, which is used to implement stub methods (empty methods) in classes, as well as empty functions (in structured programming). Also note the ever-present `self` keyword, which must be the first argument to any *bound* method in a class, as well as prefixed to any member identifier. If the self keyword is not present as the first argument, a method is determined to be *unbound*, it is not associated with an object instance, but rather with the class itself. This is equivalent of Java/C++ `static` methods.

As of Python 2.2, a new way of declaring classes emerged, commonly referred to as *new-style classes*. A top-level new-style class is required to inherit from the default `object` base class. New-

style classes are able to implement more of the specialized methods which works as operators, and have more functionality.

Python class members and methods are public by default. There exists limited support for hiding certain data in never versions. This is done by naming the members and methods with two leading underscores, and at most one trailing underscore.

A useful method that one can implement in ones classes is the `__str__()` method. It's purpose is much like Java's `toString()`; to provide a user-readable string representation of the object. Python will automatically use this method whenever it needs a string representation of ones object, such as when you try to print it to the terminal.

I will further discuss Python's properties later in this document. There is a lot to be said about it...

The A*-algorithm

The search algorithm that we will implement in this exercise is the well-known A*-algorithm (pronounced “A-star”). It is a simple *best-first* algorithm, based which performs *informed searches*. Best-first simply means that the algorithm expands the *seemingly* best node first, based on information that it (partially) gets from the *heuristic function* that is a core component of the algorithm.

The inclusion of this function is what makes A* an informed (heuristic) search algorithm. It means it can measure how close it is to archive it's goal for any given node in the search graph.
The algorithm uses this heuristic in order to guess which nodes it should expand first in order to find the solution as fast as possible. A simper, non-informed search algorithm such as *Dijkstra* have no clue weather a given node is even in the general direction of the goal node, and simply tries every possible combination of nodes until it finds a solution. This behavior is also known as the *brute-force* approach.

An interesting aspect of A* is it's *backtracking behavior*. Backtracking is the process of abandoning the currently expanded path (usually because the path cost has exceeded a given limit) and looking elsewhere for a solution. A* can abruptly abort the expansion of a path and start expanding nodes at another location, only to return to the path later. Other algorithms often expand a given subtree to a certain depth before reevaluating it's choices. A*, however, re-evaluates *all* available nodes at each iteration, always choosing the best one,

This can be observed in the running algorithm whenever it has to find a way around an obstacle that blocks the optimal straight-line path to the goal node. A* will hit the target, and try to find a way around it by “sliding” along the obstacle. At a certain point, the cost of sliding along the obstacle will become so great that A* decides a better path might exist elsewhere, and will expand another node which isn't in the path it just tried to plot around the obstacle. It might even choose to go back all the way to the root node, given the right circumstances.

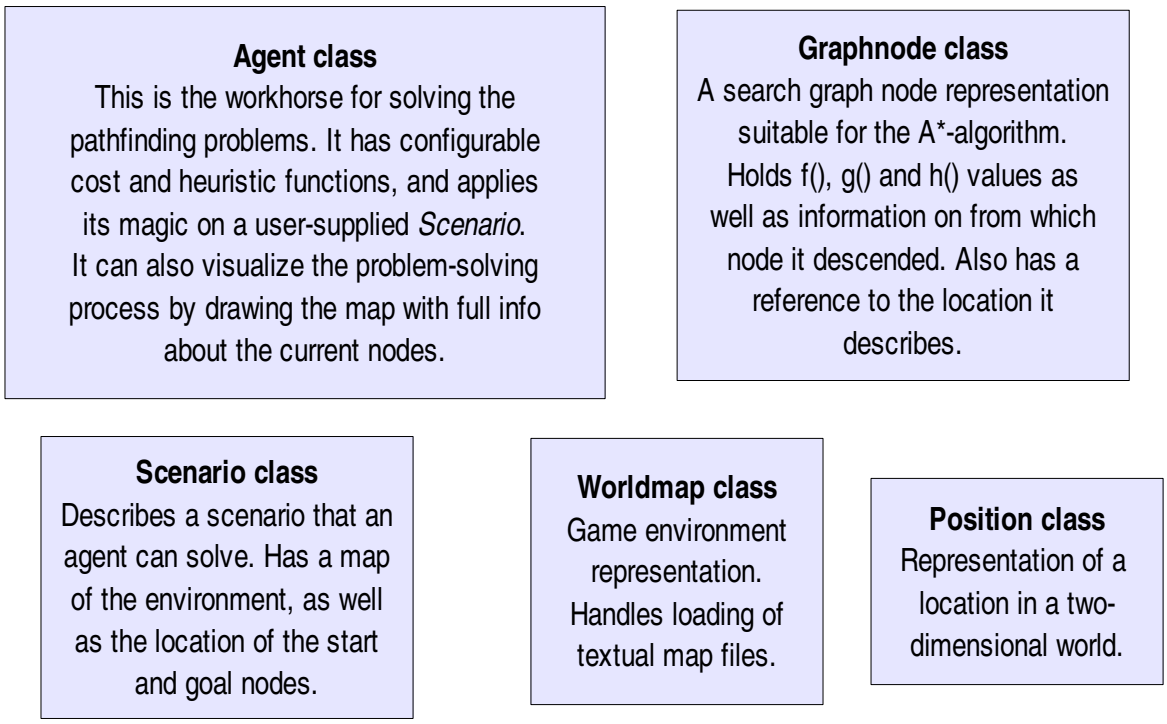
This behavior is what causes what seems like brute-force nod expansion around difficult obstacles.

The path-finding agent script

This part of the document will give a brief description of the parts that constitutes the complete agent program solution. The intent of this text is to relive the reader of having to read through the entire source code in order to understand how the agent is implemented.

Class overview

The A* path-finding system is built out of several classes, shown here.



The "Position" class

This is a simple class which represents a 2-dimensional location on the map. It has an `x` and an `y` member, representing the horizontal and vertical location on the map. A position can be compared (it implements the `__eq__()` and `__ne__()` methods, which is the Python way of overloading "==" and "!=", respectively. They can even be added, subtracted, multiplied and divided with each other (with the help of `__add__()`, `__sub__()`, `__mul__()` and `__div__()`, respectively). The observant reader might have noticed the likeness with your average vector class by now?

The "Graphnode" class

A `Graphnode` is what the core algorithm works with. It associates a map position (named by the member `location` with an f(), g() and h() value (the `f`, `g` and `h` members), as well as a parent `Graphnode`, creatively named `parent`.

Initially, `Graphnode` instances were comparable for equality using the `__eq__()` and

`__ne__()` methods. The method would return true for any two nodes who designated the same map location. The problem was that `Graphnode` also implements the `__cmp__()` method, which is used to make collections of `Graphnode` instances sortable based on their f() value, so the agent would be able to select the best node for expansion. Having two comparison methods that compared different member data seemed to confusing, so it was dropped in favor of manually comparing the `location` member.

The "Worldmap" class

The map is stored in the Python equivalent of a matrix, a list of lists. The matrix is column-major, with the outer list (the first index) corresponding to the vertical (y) position, and the inner lists (the second index) being the horizontal position. The reason for this somewhat unorthodox layout is simply because the file is simpler to implement the loader this way.

To remedy any possibility of the user confusing the crap out of himself because of this, an easy-to-use access method, called `get_data()`, exists. It takes a `Position` object denoting a map location, and returns the contents of that location. Since almost all locations are denoted by `Position` objects, this makes the code cleaner that using two separate arguments for vertical and horizontal values.

The `load_map()` method of the class is used to load a map from a file, along with the start and goal locations. Those are not stored in the `Worldmap` class, but rather in the `Scenario` class, which is why the `load_map()` method returns a dictionary object with two key/value pairs, where the keys are the two strings "start" and "goal", and the associated values are two `Position` objects.

The "Scenario" class

The `Scenario` class describes a problem that an agent should solve. It basically just a `Worldmap` instance, but also stores the start and goal locations needed to describe a full path-finding problem.
Except for the `__init__()` method, the only method in this class is `setup()`, which is used to load a scenario from a map file.

The "Agent" class

The top-level class is the `Agent` class in the current design. The user assigns the agent a task via the `setup()` method, and solves that task by calling the `think()` method until it returns either a failure or a solution. Each call to `think()` advances the process one step towards the solution (if one exists).
The Agent has a method, called `show()`, that illustrates the progress by drawing the entire map to the screen, and shows the currently expanded and soon-to-be-expanded nodes as "-" and "+" signs. These corresponds to the nodes on the closed and open lists, which is members of the `Agent` class.

The agent has a number of different heuristic and cost functions at its disposal, as well as functions for determining the penalty for creating dwindling paths. The user selects what functions should be called by assigning the corresponding function objects to the two members named `cost_function` and `heuristic_function`. They work much the same as function pointers

would in a C implementation.

The `open` list is a standard Python list, consisting of `Graphnode` objects. As mentioned earlier, this class has a `__cmp__()` method, which roughly corresponds to a class implementing the `Comparable` interface in Java lingo. This is how the agent sorts the open list.

In an earlier design (which was subsequently thrown away), the agent was simply member of a higher-level container class. The agent's `think()` function returned a `Successor` object, which was stored in the containing object, which was responsible for the visualization of the agent's progress.
This design (which at first glance seems like a more proper representation) had serious trouble with object-to-object interaction, often requiring intimate two-way interaction between objects and its encapsulated members. The class hierarchy was therefore re-engineered to the current, simple, top-down approach.
The current design is not very elegant one (the most prominent design flaw is that the agent class does the visualization), but because of time constraints it was never redesigned. Since it worked...

The main script

The main script is located at the end of the file. It poses several questions to the user, allowing him/her to choose how to run the simulation, as well as determine what heuristic functions to use etc.
It then creates a `Scenario` instance, and loads the map file (supplied on the command line) into it. The an `Agent` is created, and configured with the functions chosen by the user, as well as appointed to the `Scenario` task.
Finally, the `think()` method of the agent is called repeatedly until the path-finding problem has been either solved or deemed insolvable. After each iteration, the map might be drawn, if the user so wishes.

Implementation and development details

Here we will outline different tasks that had to be accomplished in order to get the software ready. Also included are the problems encountered and solutions that we developed to overcome them.

The map parsing task

The first problem that was tackled was the issue of map loading. The map files consists of ASCII text files, with the two first rows containing key/value pairs (`height` and `width`, respectively), followed by an equality sign ("="), and then the value in decimal digits. Then follows the map data, which is represented by ASCII characters, organized in rows whose width matches the `width` entry.
The number of rows is equal to the value of the `height` value.

Never having done anything like this in Python before, the easiest solution seemed to be to use the `readline()` method to retrieve a line from the input stream. The key/value-pairs could be parsed by reading the first two lines one-by-one and perform a `split()` string operation on the retrieved

line.
The string parts are “trimmed” (or “chomped” in Perl lingo) with the help of the `strip()` string method, which removes any whitespace at either end of the string.

If the “key” part of the string seems correct (should be either “height” or “width”), the right-hand part of the string (supposedly the value part) will be parsed as a signed integer. The proper Python way to do this is with the following line:

```
self.height = int(string)
```

In older Python versions, this was done much like in C, with the `atoi()` function (part of the `string` module). Since release 2.0 of Python, the use of explicit type conversions is deprecated in favor of the object-oriented method shown above. Upon failure, a `TypeError` exception is emitted. The explicit conversion functions are scheduled to be completely removed with release 3.0 of Python.

An unexpected problem that arose was inconsistent character case in the map data symbols. Simply put, some characters were in lowercase, some in uppercase. This was probably a data file bug, so an extension to the loader was written, allowing it to ignore map character case. This behavior can be controlled via the `case_sensitive` data member. If False, all characters are transformed to uppercase prior to examination.

The map loader also deals gracefully with any kind of line breaks, be they either Unix, Windows or Mac-style. Newlines and carriage-return symbols are stripped from the input stream, and the length of the valid symbol line is verified so that it matches the `width` value parsed earlier.

The `Worldmap` class isn't really concerned with the location of the start and goal nodes. It simply stores map information, not path-finding task data. However, the location of the start and goal nodes *are* stored in the map file which only the `Worldmap` class can parse. Formally that information would should only be stored within the class as long as a map file is being parsed, or it might as well have be a data member altogether. The solution chosen works in such a way that when a map has been loaded from disk, the `load_map()` method returns a dictionary which contains the location of the start and goal nodes. The caller can then use that data at it's discretion.

A dictionary was chosen as the container since it offers a more descriptive storage than say, a list of `Position` objects. With that solution, we would have to know which item in the list that corresponds to which entity (the start or the goal node). This would force the user to either look it up in the documentation, or in the source code. Also, a dictionary's keys (which are easily retrievable) serves as metadata (data that describes the content and purpose of other data) for the values. This is especially important when the number of returned items grows.

It is because of the above mentioned reasons that the open 3D-modeling software package *Blender* has switched of the container objects used in its Python API to dictionaries from the lists they used earlier (prior to the 2.41 release). It also serves to improve the readability of the source code, since a textual string key is generally easier to comprehend than a integer offset/index.

Handling weighted maps

Another compulsory feature that the agent should have was the ability to navigate a weighted map, a terrain height-map. We did not receive any such example maps, so one of the first task was to build a simple generator for these kinds of maps. This is what the script `map_convert.py` is for (the sourcecode kan be found in appendix B).

The current version is very simplistic, and deeply flawed. It simply opens a map, and replaces all occurances of whitespace characters in the input stream with a random decimal integer between 1 and 9. Thus, the terrain map is simply noise, and not a true terrain. The application cannot generate a continous terrain since it only have access to a limited set of map data at any given point. A better approach would be to load the entire map into memory and use a fractal algorithm such as midpoint displacement in order to get a true terrain.

In order to extend the system to handle weighted maps, a number of changes needed to be made. First, a way of determining the map type was developed. It is implemented in the `load_map()` method of the `Worldmap` class. The first map character loaded which isn't a blocked square (ie. not an “X”) is used to determine the map type. If it is a whitespace character, the map type is set to binary, and if it is a positive integer in the range from 1 to 9, the map type is set to “weighted”.

A new cost function were added, called `cost_weighted()`, which resides in the Agent class. It returns the sum of the delta values between two consecutive nodes in the current path.

An issue that arose is the question of what value to use for the weighted maps start and goal locations. In a binary map, those are treated as accessible nodes, but what should we use in a weighted map? We did not receive any explicit instructions on what to do, so the current implementation uses the average of the surrounding accessible squares for those locations.

Generation of straighter paths

As part of the assignment, we were asked to implement functionality that would allow our agent to generate smoother, straighter paths than the dwindling, creeping paths that your typical bare-bones A*-algorithm generates.

The solution we created is simple. As soon as at least one move has been made (the agent has expanded at least one node), two vectors are constructed from the old position and the current position, as well as the current position and each one of the possible new locations.
If the two vectors match, no penalty is applied since they describe the same direction.
If they don't match, a user-configurable penalty is added to the cost for reaching the node.
The higher the cost, the more nodes A* will expand in order to find the cheapest path.

Using Python for the assignment

The author had very little experience with Python for application development, so this assignment seemed like a good opportunity to gain a deeper knowledge of Python. This section of the report is dedicated to how Python was used to complete the task, and what the repercussions were.

Python programming practices implemented

A number of interesting Python-specific techniques were implemented in this project.
We will take a brief look at them in this section, and explain the reasons for using them.

The __slots__ directive

This directive is used to limit the members that a class can have, and prevent new ones from being added at runtime. You add it to your class's body and supply it with a list (or even better, a tuple) of strings containing the valid identifiers for the class members.

```
class Person (object):
    __slots__ = (“name”, “age”, “sex”)
    __init__ (self):
        pass
```

Text 3: A restricted class.

When I first started using Python, I tried to program it the same way I usually program i C++, which is very strict. I used the __slots__ directive on every class I created, as a means to mimic my stricter coding style. This is *not* how the aforementioned directive is intended to be used. It's purpose is to allow for efficient storage of large number of class instances in memory for tasks that require large amounts of objects (such as search graphs).

Regular classes have a full hash table for each instance, which uses a lot of memory. What the `__slots__` directive does is to create a less flexible data structure with just enough space for the requested attributes. It is not a recommended practice, since it works against the principles of Python, and as such should only be used sparingly, under certain circumstances.

Loose typing

Another habit that I carried over from my previous C++ programming experience was the habit of performing rigorous checking of any argument sent to a function. You are not supposed to do very much checking, but let the internal data structure do much of the sanity checking.

The first implementation had a lot of type checking for certain methods, as well as value-range error checks done in user code. After several counseling sessions in the #python irc channel, the author was finally convinced of the folly of doing this.
Many common programming errors are caught by common Python data structures, such as as indexing errors, and many value errors. Since Python has a strong bias towards prototyping, the idea

is that the user shouldn't have to code such failsafe mechanisms himself, but should rely on the underlying Python systems to throw an exception, which can be caught.

This is the reason that the final code has so little error-handling code in it.

Aggressive dictionary usage

Creative Python programming involves hefty usage of the dictionary container type native to Python. It is one of the most powerful language constructs, offering fast look-up times as well as the ability to perform arbitrary *mappings* (associations) between objects. This ability is a key component in the implementation of many algorithms, as well as a being very convenient multi-purpose container for *pairs* of things.
An example of this kind of dictionary usage can be seen in the `get_adjacent()` method in the `Worldmap` class. The method is used by the agent in order to find out what moves it can make from a certain map square. At first glance, returning a list of accessible squares (represented by `Position` instances) might seem like a good idea. But wait, if we are working with weighted maps, we'll need to know what value each square has as well, won't we?

To solve this problem, we could either query the contents of each square with the `get_data()` method, or we could create a new class, perhaps called `MapSquare`, to convey this information. The first method is inefficient (the map data will be read twice, once by the `Worldmap` class, once by the `Agent` class when it queries the locations), it is cumbersome (since we'll need plenty of code in the `Agent` class to perform the querying) and somewhat ugly (`Agent` will be very tightly tied with `Worldmap`, and the interface will be cluttered).

The solution taken in the code (which, in the authors opinion, is more elegant) is to return a dictionary object, whose keys are the accessible locations (`Position` instances), and whose corresponding values are the map symbol for each square. The user may now use the keys from dictionary directly and even retrieve the map symbols easily with the same keys.

Another common use for dictionaries in Python is as a replacement for the common `switch` statement, present in most implicit computer languages. Python lacks this language construct, so another solution is needed to avoid excessively long else-if chains in the code.
The solution is to use a dictionary whose keys represents the values for each “case”.
The corresponding value is then either a scalar value or a function object, depending on circumstances. An example of this can be found in the code for the simple menu system, where the user can choose parameters for the simulation.

A more complex example can be found in the `map_load()` method in the current version of the program. Compare the implementation to the older one found in appendix B, and notice the more compact and more flexible implementation using the dictionary.

Python versus traditional languages

The practice of using indentation for defining where blocks of code start and end seems like an idea that looks good on paper, and works worse in reality. First of all, this design forces everyone to use a single coding style, since the code cannot be formatted to ones personal preference, which in

essence is a restriction of ones personal freedom. It also makes it hard for people which uses regular (non-graphical) consoles to work with the code, since it cannot be properly formatted for their screens. It forces everyone that uses the code to bend themselves and their working environment to fit Python's needs, instead of the other way around.

Secondly, how smart is it really to use INVISIBLE characters for something as common and error-prone as code block definition? A single line with the incorrect indentation level will seriously change the way the code will execute, and it cannot even be spotted in the editor unless the programmer have chosen to display the formatting characters, something that will clutter up the editor screen more than any braces-based language ever would.

Another peculiar design decision with Python is the way that it forces the programmer to concentrate a lot of effort on basic tasks that most other languages have the compiler do for him.
In Python, there is no way to enable a 'stricter' mode, such as in Perl (the 'use strict' directive).
If the programmer ever misspells a variable name, there is a great chance that he won't notice until the program doesn't behave the way it is supposed to, since Python silently creates the “new”, misspelled variable. There is no way to circumvent this behavior, save for the use of external code-checking tools.
The defense for this behavior is often stated as “Python assumes the programmer is well-disciplined”, which is a really questionable design in this authors opinion.

Why should the programmer focus effort on boring tasks that can easily be done by the compiler?

Since the argument for using Python is often that “more time can be spent on improving the application”, one must really question this design unless of course the programmer is perfect, and never makes mistakes... Which is something we all know isn't true.
Add to this the fact that bugs will have to be found at run-time instead of compile-time, and apply the common knowledge (from the C community) which states: *Any bug found during the compilation phase saves ten times the time debugging the application for run-time errors.*

This peculiar design exists even in the object-oriented part of Python. For example, whenever the programmer extends an existing class, the “constructor” (the `__init__()` method) of the new, extended class will have to manually call the base class's constructor, or the base class will remain uninitialized. And since all Python class members are created on-the-fly when assigned to, the base class won't even *have* any members. Is this even proper object-oriented programming?

Access control is another interesting Python area. There is no such thing. It's not even possible to declare a constant in Python, and don't even think about non-mutating (like `const` or `final`) methods in your classes. While we are on the subject, skip the concept of `private` or `protected` data as well. There is no such thing. Again the programmer is responsible for “not doing something he shouldn't”, which in itself might be reasonable if you are the only person that uses the code. But isn't most professional development carried out in teams? Oh, but yes, in the perfect world everything is of course documented using the fantastic `Pydoc` tool!

Sarcasm aside, it is this authors personal opinion that Python is not a good language. It is very slack in areas of the language which is critical to finding and eliminating bugs, and extremely strict in

areas where programmers traditionally could exercise their own, personal preference without impacting functionality at all. Personally, I'd not call it a programming language, but rather view it as a scripting language with neat syntax, and treat it as such.

References

Stuart Russel, Peter Norvig. *Artificial Intelligence: A modern approach*, second edition.
Pearson Education, Inc. 2003

Alex Martelli. *Python in a nutshell*, first edition.
O'Reilly & Associates, Inc. 2003

Appendix A

Full source code to the program.

```
#!/usr/bin/python

import sys
import string
import math

# Constants (as close as we get in Python anyway)
SOLUTION_PENDING = 0
SOLUTION_FOUND = 1
NO_SOLUTION_EXISTS = 2

def get_user_integer (msg):
    """Custom input handler, allows input of non-zero positive integers."""
    userstring = raw_input(msg)
    try:
        number = int(userstring)
    except TypeError:
        number = 1
    except:
        number = 1
    if number <= 0:
        number = 1

    return number

class Position (object):
    """ Representation of a 2-dimensional location."""

    __slots__ = ("x", "y")

    def __init__ (self, x = 0, y = 0):
        self.x = x
        self.y = y

    def __eq__ (self, cmp_to):
        """Method called for equality testing '=='."""
        return self.x == cmp_to.x and self.y == cmp_to.y

    def __str__ (self):
        """'Stringification' method."""
        return "Position: (x, y) = [%d, %d]" % (self.x, self.y)

    def __add__ (self, rhs):
        """Method called for the '+' operator."""
        return Position(self.x + rhs.x, self.y + rhs.y)

    def __sub__ (self, rhs):
        """Method called for the '-' operator."""
        return Position(self.x - rhs.x, self.y - rhs.y)

    def __mul__ (self, rhs):
        """Method called forMartin Persson System och Design the '*' operator."""
        return Position(self.x * rhs.x, self.y * rhs.y)

    def __hash__ (self):
        """Method for hashing a Position object (used by dictionaries)."""
        return hash((self.x, self.y))

class Graphnode (object):
    """Comparable, sortable based on f(), has a parent and children nodes."""

    __slots__ = ("parent", "f", "g", "h", "location")

    def __init__ (self):
        self.parent = self
        self.f = 0
        self.g = 0
        self.h = 0
```

```
        self.location = None

    def __str__ (self):
        return "Graphnode: (f, g, h) = [%f, %f, %f]\n\t%s" % (self.f, self.g, self.h,
str(self.location))

    def __cmp__ (self, cmp_to):
        """This method is called by lists in order to sort their elements.
        Since we want our agent to sort the nodes by cost, in a raising
        manner, we implement comparision in terms of f()."""
        if self.f < cmp_to.f:
            return -1
        elif self.f > cmp_to.f:
            return 1
        else:
            return 0

class Worldmap (object):
    """Represents a square world of tiles of varying sort.
    Worldmap is loaded from a textual file by the load_map()-method."""

    def __init__ (self):
        self.filename = ""
        self.width = 0
        self.height = 0
        self.case_sensitive = False
        self.map_data = []
        self.map_type = None

    def load_map (self, filename):
        """This is a simple map parser (hack is a more proper name).
        Based on a read-by-line approach. It can handle both DOS, UNIX and MAC-
        formatted files, and tries to work out if the user is supplying us
        with malformed data. It returns a dictionary with the loaded map's start
        and goal positions."""

        self.filename = filename
        inputfile = open(self.filename, "r")

        # Create the map object (just a list). We set it to 'None' if no
        # map has been loaded (in the constructur, '__init__').
        self.map_data = []
        stuff = { }

        # These tables contains the valid values and substitutions for maps.
        table_binary = { " ":" ", "X":"X", "S":" ", "E":" " }
        table_weighted = { "X":"X", "S":" ", "E":" ", "1":"1", "2":"2", "3":"3",
"4":"4", "5":"5", "6":"6", "7":"7", "8":"8", "9":"9" }
        table = table_binary

        for i in range (0, 2):
            # Read a line, split by '=', clean up and interpret.
            line = inputfile.readline()
            parts = line.split('=')
            clean_parts = []
            for j in parts:
                clean_parts.append(j.strip())

            if clean_parts[0] == "height":
                self.height = int(clean_parts[1])
            elif clean_parts[0] == "width":
                self.width = int(clean_parts[1])

        num_ignored = 0 # Use this to keep track of ignored symbols.
        for i in range(self.height):
            line = inputfile.readline()
            newline = []

            for j in range(len(line)):

                # 'i' indicates the current row, 'j' indicates the current
                # column. We'll assign the current symbol (or "token") to
                # 'mapsymbol' for convenience.
                mapsymbol = line[j]

                # Strip out any newline symbols, but keep a note of it.
```

```
                if mapsymbol == '\n' or mapsymbol == '\r':
                    num_ignored = 1 + num_ignored
                    continue

                # If we are not sensitive about case in the input stream,
                # just transform all tokens to uppercase. The internal map
                # is always in uppercase.
                if not self.case_sensitive:
                    mapsymbol = mapsymbol.upper()

                # When we get the first map symbol which isn't an blocked
                # square, start or goal square, we can determine map type.
                if self.map_type == None:
                    if mapsymbol != "X" and mapsymbol != "S" and mapsymbol != "E":
                        if mapsymbol in table_binary:
                            self.map_type = "binary"
                            table = table_binary
                        elif mapsymbol in table_weighted:
                            self.map_type = "weighted"
                            table = table_weighted
                        else:
                            raise ValueError("Map type could not be detected.")

                # Extract the appropiate substitution symbol, and also make
                # sure that the symbol is valid.
                try:
                    newsymbol = table[mapsymbol]
                except KeyError:
                    raise ValueError("Invalid symbol \"%s\" found in map!" %
mapsymbol)

                if mapsymbol == "S":
                    stuff["start"] = Position(j, i)

                elif mapsymbol == "E":
                    stuff["goal"] = Position(j, i)

                newline.append(newsymbol)

            self.map_data.append(newline)

        # Whine a little, the users deserve it you know...
        print "Warning: %d symbols in the input stream were ignored (newlines?)." %
num_ignored
        inputfile.close()

        if self.width != len(self.map_data[0]):
            raise ValueError("Map width does not match matrix contents.")

        if self.height != len(self.map_data):
            raise ValueError("Map height doesn not match matrix contents.")

        print "This appears to be a " + self.map_type + " type map."

        # Calculate the values for the start and goal nodes for a non-binary
        # map type. This must be done since we don't know the value for those.
        # The value is calculated by averaging the value of all adj. squares.
        if self.map_type == "weighted":
            sum = 0
            nodes = self.get_adjacent(stuff["start"])
            for mapsym in nodes.itervalues():
                sum = sum + int(mapsym)
            sum = sum / len(nodes)
            self.set_data(stuff["start"], str(int(sum)))
            print "Averaged the value for the start location to: %d" % sum

            sum = 0
            nodes = self.get_adjacent(stuff["goal"])
            for mapsym in nodes.itervalues():
                sum = sum + int(mapsym)
            sum = sum / len(nodes)
            self.set_data(stuff["goal"], str(int(sum)))
            print "Averaged the value for the goal location to: %d" % sum

        # Return the locations of the 'start' and 'goal' nodes.
        return stuff
```

```
    def valid_map_location (self, pos):
        """Validates a given map location.
        The position is valid if within map bounds an does't contain an 'X'."""
        if pos.x >= self.width or pos.x < 0:
            return False
        if pos.y >= self.height or pos.y < 0:
            return False
        if self.map_data[pos.y][pos.x] == "X":
            return False

        return True

    def get_data (self, pos):
        """Convinience map data access method.
        Less error-prone than manual access, since matrix is (y, x) ."""
        return self.map_data[pos.y][pos.x]

    def set_data (self, pos, val):
        """Easy-to-use mutator method for changing map data."""
        self.map_data[pos.y][pos.x] = val;

    def get_adjacent (self, pos):
        """Return all the accessible locations based on the supplied one.
        Return value is a dictionary with Position instances as keys, and
        the map symbols as values. Only valid locations are returned."""

        adjacent = { }

        # Evaluate what directions that constitutes valid moves.
        test = Position(pos.x, pos.y + 1)
        if self.valid_map_location(test):
            adjacent[test] = self.get_data(test)

        test = Position(pos.x, pos.y - 1)
        if self.valid_map_location(test):
            adjacent[test] = self.get_data(test)

        test = Position(pos.x + 1, pos.y)
        if self.valid_map_location(test):
            adjacent[test] = self.get_data(test)

        test = Position(pos.x - 1, pos.y)
        if self.valid_map_location(test):
            adjacent[test] = self.get_data(test)

        test = Position(pos.x + 1, pos.y + 1)
        if self.valid_map_location(test):
            adjacent[test] = self.get_data(test)

        test = Position(pos.x + 1, pos.y - 1)
        if self.valid_map_location(test):
            adjacent[test] = self.get_data(test)

        test = Position(pos.x - 1, pos.y + 1)
        if self.valid_map_location(test):
            adjacent[test] = self.get_data(test)

        test = Position(pos.x - 1, pos.y - 1)
        if self.valid_map_location(test):
            adjacent[test] = self.get_data(test)

        return adjacent

class Scenario (object):
    """A problem that an agent must solve.
    It has a worldmap, a start and a goal location."""

    def __init__ (self, filename):
        self.world = Worldmap()
        locations = self.world.load_map(filename)
        self.start = locations["start"]
        self.goal = locations["goal"]

    def setup (self, filename):
        self.world = Worldmap()
```

```
        locations = self.world.load_map(filename)
        self.start = locations["start"]
        self.goal = locations["goal"]

class Agent (object):
    """An path-finding A*-based agent."""

    def __init__ (self):
        self.open = []
        self.closed = []
        self.task = None
        self.location = None
        self.iterations = 0
        self.path = []
        self.cost_function = self.cost_simple
        self.heuristic_function = self.heuristic_manhattan
        self.smooth_path = False
        self.dir_penalty_cost = 30

    def setup (self, task):
        """Initializes the agent to work with the supplied task."""
        self.open = []
        self.closed = []
        self.task = task
        self.location = task.start
        initial_node = Graphnode()
        initial_node.location = self.location
        initial_node.parent = initial_node
        self.open.append(initial_node)
        self.iterations = 0
        self.path = []

        # If it is a weighted map, use cost_weighted, else cost_simple
        if self.task.world.map_type == "weighted":
            self.cost_function = self.cost_weighted
        else:
            self.cost_function = self.cost_simple

        self.heuristic_function = self.heuristic_manhattan
        self.smooth_path = False

    def heuristic_manhattan (self, loc):
        return math.fabs(self.task.goal.x - loc.x) + math.fabs(self.task.goal.y -
loc.y)

    def heuristic_straight_line (self, loc):
        return math.hypot(self.task.goal.x - loc.x, self.task.goal.y - loc.y)

    def cost_simple (self, node):
        """Returns the length of the current path to the start (root) node."""

        cost = 0
        tmp = node
        while tmp.location != self.task.start:
            tmp = tmp.parent
            cost = cost + 1

        return cost

    def cost_weighted (self, node):
        """Returns the cost of the current path, for a weighted map."""
        cost = 0
        tmp = node
        while tmp.location != self.task.start:
            cur_par = int(self.task.world.get_data(tmp.parent.location))
            cur_val = int(self.task.world.get_data(tmp.location))

            cost -= cur_par - cur_val
            tmp = tmp.parent

        return cost

    def direction_penalty (self, cur_node, test_node):
        """Return the directional penalty for using 'node' as a successor.
        This method creates a vector A from the current and it's parent node,
        as well as a vector B from the current node and 'node'. The returned
```

```
        cost is the dot product of those two vector."""

        if test_node.location == cur_node.location:
            raise ValueError("Testing a node against itself.")

        if cur_node.parent == cur_node:
            return 0

        # Create the vectors
        vec_a = cur_node.location - cur_node.parent.location
        vec_b = cur_node.location - test_node.location

        # Hacked up dot product.
        #if (vec_a.x * vec_b.x + vec_a.y * vec_b.y != 0):
        #    return 3;
        #else:
        #    return 0;

        if vec_a != vec_b:
            print "penalty!"
            return self.dir_penalty_cost

        print "no penalty!"
        return 0

    def get_path (self, node):
        """Builds a list of all nodes from the root node to the referred."""

        path = []
        tmpnode = node
        while tmpnode.location != self.task.start:
            path.append(tmpnode)
            tmpnode = tmpnode.parent

        return path

    def think (self, verbose = False):
        """The main solving iteration method.
        Call this repeatedly in order to solve the current task.
        The return value tells you wether the task is currently unsolved,
        successfuly solved, or unsolvable."""

        # If the open list is exhausted, we have failed utterly...
        if len(self.open) == 0:
            return NO_SOLUTION_EXISTS

        # Sort the candidate nodes depending on f()-value.
        self.open.sort()
        if verbose == True:
            print "The open list:"
            for i in self.open:
                print i

        cur_node = self.open.pop(0)

        if verbose == True:
            print "The closed list:"
            for i in self.closed:
                print i

        # If the node is the goal node, return success after storing path.
        self.location = cur_node.location
        self.closed.append(cur_node)
        if self.location == self.task.goal:
            self.path = self.get_path(cur_node)
            return SOLUTION_FOUND

        # Get all adjacent nodes from the map.
        adjacent = self.task.world.get_adjacent(cur_node.location)

        # Encapsulate each one in a Grapnode, and add the costs and heuristics.
        nodes = []
        for adj_loc in adjacent:
            new_node = Graphnode()
            new_node.parent = cur_node
            new_node.location = adj_loc
```

```
            if self.smooth_path == True:
                new_node.g = self.cost_function(new_node) +
self.direction_penalty(cur_node, new_node)
            else:
                new_node.g = self.cost_function(new_node)

            new_node.h = self.heuristic_function(adj_loc)
            new_node.f = new_node.g + new_node.h
            nodes.append(new_node)

        for node in nodes:

            in_open = False
            in_closed = False

            # If this node is already in the open list, ignore it.
            for o in self.open:
                if o.location == node.location:
                    in_open = True
                    break

            # If in 'closed' and new is cheaper, update 'parent' and 'g'.
            for c in self.closed:
                if c.location == node.location:
                    in_closed = True
                    break

            if not in_closed and not in_open:
                self.open.append(node)

        self.iterations = self.iterations + 1

        return SOLUTION_PENDING

    def show (self):
        """Prints the current state of the agent's task solving progress."""

        print "Iteration #%d" % self.iterations

        # Print lines one by one, check if the location has any data about it
        # in any of the lists, or is a special location (start, goal, agent).
        # If it is any such location, print a special symbol.
        for y, row in enumerate(self.task.world.map_data):

            line = ""
            replacement = ""
            for x, symbol in enumerate(row):
                replacement = symbol
                cur_pos = Position(x, y)
                if cur_pos == self.task.goal:
                    replacement = "E"
                elif cur_pos == self.task.start:
                    replacement = "S"
                elif cur_pos == self.location:
                    replacement = "A"
                else:

                    for i in self.open:
                        if i.location == cur_pos:
                            replacement = "+"
                            continue

                    for i in self.closed:
                        if i.location == cur_pos:
                            replacement = "-"
                            continue

                    for i in self.path:
                        if i.location == cur_pos:
                            replacement = "o"
                            continue

                line = line + replacement

            print line
```

```
        # Print explanatory string to please the user.
        print "+ (open)  - (closed)  o (final path)"


### Main script start ###

try:
    print "Using map file: \"%s\"" % sys.argv[1]

except IndexError:
    print "Error: Please supply a map filename as argument to this script."

try:
    myscenario = Scenario(sys.argv[1])

except IOError:
    print "Error: Unable to open map: \"%s\"" % sys.argv[1]

myagent = Agent()
myagent.setup(myscenario)

print "\nSelect simulation speed/visualization:\n"
print "1. Interactive: Redraw and wait after each succession."
print "2. Automatic: Full speed, no drawing until solution has been found."

mode = get_user_integer("\nWhat mode do you wish to use? (default: Interactive): ")

# Allow the user to select what heuristic function the agent should use.
print "\nSelect the agent's heuristic function:\n"
print "1. Manhattan distance"
print "2. Straight line distance"
heuristics = { 1 : myagent.heuristic_manhattan, 2 : myagent.heuristic_straight_line }
heur = get_user_integer("\nWhich heuristic do you want to use? (default: Manhattan):
")
myagent.heuristic_function = heuristics[heur]

# Allow user to select optional features.
print "\nSelect extra features to use:\n"
print "1. Default (no extra features enabled)"
print "2. Straight path generation"

feat = get_user_integer("What features do you want to enable? (default: None) ")

if feat == 2:
    myagent.smooth_path = True
    # Allow custom directional weighting.
    myagent.dir_penalty_cost = get_user_integer("Enter the penalty for a nonstraight
path (default = 1): ");

print
myagent.show()
raw_input("\nPress [enter] to begin simulation...")

# Loop until we have solved the stuff, or it b0rks completely.
ret = 0
usercounter = 0
while ret == SOLUTION_PENDING:
    ret = myagent.think()
    if mode == 1 and usercounter == 0:
        myagent.show()
        usercounter = get_user_integer("\nNumber of iterations to run non-
interactively (default = 1)... ")
    usercounter = usercounter - 1

myagent.show()

if ret == NO_SOLUTION_EXISTS:
    print "A solution could not be obtained for this problem."
elif ret == SOLUTION_FOUND:
    print "A solution was found after %d iterations." % myagent.iterations
    print "Final path has %d nodes." % len(myagent.path)
```

Appendix B

The source code to the map conversion program.

```
#!/usr/bin/python

import sys
import random

inputfile = open(sys.argv[1], "r")
outputfile = open(sys.argv[2], "w")

line = ""
for line in inputfile.readlines():
    newline = ""
    for j in range(len(line)):
        mapsymbol = line[j]

        if mapsymbol == " ":
            newsymbol = str(random.randint(1,9))
        else:
            newsymbol = mapsymbol

        newline += newsymbol

    outputfile.write(newline)

inputfile.close()
outputfile.close()
```

Appendix C

Source code excerpt from older version of program.

```
def load_map (self, filename):
    """This is a simple map parser (hack is a more proper name).
    Based on a read-by-line approach. It can handle both DOS, UNIX and MAC-
    formatted files, and tries to work out if the user is supplying us
    with malformed data. It returns a dictionary with the loaded map's start
    and goal positions."""

    self.filename = filename
    inputfile = open(self.filename, "r")

    # Create the map object (just a list). We set it to 'None' if no
    # map has been loaded (in the constructur, '__init__').
    self.map_data = []
    stuff = { }
    weights = ("1", "2", "3", "4", "5", "6", "7", "8", "9")

    for i in range (0, 2):
        # Read a line, split by '=', clean up and interpret.
        line = inputfile.readline()
        parts = line.split('=')
        clean_parts = []
        for j in parts:
            clean_parts.append(j.strip())

        if clean_parts[0] == "height":
            self.height = int(clean_parts[1])
        elif clean_parts[0] == "width":
            self.width = int(clean_parts[1])

    num_ignored = 0 # Use this to keep track of ignored symbols.
    for i in range(self.height):
        line = inputfile.readline()
        newline = []

        for j in range(len(line)):

            # 'i' indicates the current row, 'j' indicates the current
            # column. We'll assign the current symbol (or "token") to
            # 'mapsymbol' for convenience.
            mapsymbol = line[j]

            # Strip out any newline symbols, but keep a note of it.
            if mapsymbol == '\n' or mapsymbol == '\r':
                num_ignored = 1 + num_ignored
                continue

            # If we are not sensitive about case in the input stream,
            # just transform all tokens to uppercase. The internal map
            # is always in uppercase.
            if not self.case_sensitive:
                mapsymbol = mapsymbol.upper()

            # When we get the first map symbol which isn't an blocked
            # square, start or goal square, we can determine map type.
            if self.map_type == None:
                #print "symbol \"%s\"" % mapsymbol
                if mapsymbol != "X" and mapsymbol != "S" and mapsymbol != "E":
                    if mapsymbol == " ":
                        self.map_type = "binary"
                    elif mapsymbol in weights:
                        self.map_type = "weighted"
                    else:
                        raise ValueError("Map type could not be detected.")

            # This is kinda ugly at first sight, but it's very flexible
            # if I'd want to change the format of the internal map.
            # This is a full substitution/action table for input stream
            # tokens.
```

```
            if mapsymbol == "X":
                newsymbol = "X"
            elif mapsymbol == " ":
                newsymbol = " "
            # S denotes the agent's starting position. For a binary-type
            # map, store the position and replace it with an empty square
            # in the internal map.
            elif mapsymbol == "S":
                if self.map_type == "binary":
                    newsymbol = " "
                else:
                    newsymbol = "-"
                stuff["start"] = Position(j, i)
            # E denotes the agent's goal position. Replace with an empty
            # square in a binary-style map.
            elif mapsymbol == "E":
                if self.map_type == "binary":
                    newsymbol = " "
                else:
                    newsymbol = "-"
                stuff["goal"] = Position(j, i)
            elif mapsymbol in weights:
                newsymbol = mapsymbol

            else:
                raise ValueError("Invalid symbol found in map file.")

            newline.append(newsymbol)

        self.map_data.append(newline)

    # Whine a little, the users deserve it you know...
    print "Warning: %d symbols in the input stream were ignored (newlines?)." %
num_ignored
    inputfile.close()

    if self.width != len(self.map_data[0]):
        raise ValueError("Map width does not match matrix contents.")

    if self.height != len(self.map_data):
        raise ValueError("Map height doesn not match matrix contents.")

    print "This appears to be a " + self.map_type + " type map."

    # Calculate the values for the start and goal nodes for a non-binary
    # map type. This must be done since we don't know the value for those.
    # The value is calculated by averaging the value of all adj. squares.
    if self.map_type == "weighted":
        sum = 0
        nodes = self.get_adjacent(stuff["start"])
        for mapsym in nodes.itervalues():
            sum = sum + int(mapsym)
        sum = sum / len(nodes)
        self.set_data(stuff["start"], str(int(sum)))
        print "Averaged the value for the start location to: %d" % sum

        sum = 0
        nodes = self.get_adjacent(stuff["goal"])
        for mapsym in nodes.itervalues():
            sum = sum + int(mapsym)
        sum = sum / len(nodes)
        self.set_data(stuff["goal"], str(int(sum)))
        print "Averaged the value for the goal location to: %d" % sum

    # Return the locations of the 'start' and 'goal' nodes.
    return stuff
```