UNIVERSITY OF SASKATCHEWAN

Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephone: (306) 966-4886, Facimile: (306) 966-4884

CMPT 317
Fall 2018
Introduction to Artificial Intelligence

Assignment 1 – Solutions and Grading

Searching

Date Due: 01 October 2018, 11:59pm **Total Marks: 50**

Overview

In this assignment we'll explore the ideas in AIMA Chapter 3 using a problem defined as follows.

Inverse Arithmetic Problem

. You are given a finite list of integers, L, and a target integer, T. The problem is to determine an integer arithmetic expression E, using some or all of the integers in L, so that if E is evaluated according to the rules of integer arithmetic, the answer is T.

Integer expressions will be constructed using only the operations $+$, $-$, $\times$, and integer division / (integer division means we ignore the remainder, and use only the quotient, e.g. $5/2 = 2$).

For example, the given information could be as follows:

- $L = [1, 2, 3, 4, 5, 7]$
- $T = 47$

There are several possible expressions we can construct:

- $((2 \times 4) \times 5) + 7$
- $((3 + 4) \times 7) - 2$
- $(7 \times 7) - 2$
- $(3 \times 4) + (5 \times 7)$

For pragmatic reasons, we will look for linear expressions only, e.g., $((2 \times 4) \times 5) + 7$, and we will not be trying to find non-linear expressions like $(3 \times 4) + (5 \times 7)$. Restricting the shape makes your job easier, and including non-linear expressions does not increase the insight you gain from this exercise. For this assignment, assume that each value from L can be used at most once, but the operators can be used as many times as you need.

We're interested in finding the expression that uses the fewest operations. Most problems will not have a single unique shortest expressions, and almost all problems will have many much longer expressions.

Programming

You'll have to do some programming here, and I'll provide strong guidance about how to get things organized. You can use any programming language you like, as long as it's Python, Java, or C/C++. If you want to use a different language, please let me know before proceeding.

The arithmetic expressions are probably best represented by text strings. In Python, you have access to the function `eval()`, which is quite powerful, but we can use it to calculate the value of any expression represented as a string. If you're using Java, you are permitted to use a third-party library to evaluate expression strings; there are a few I found by Googling.

You'll also be running your programs on a collection of examples. You'll be expected to report on aspects of this exercise.

Execution instructions

The markers may or may not wish to run your program, to verify your results. To help them, you should provide brief instructions on what to do to get your program running. Include:

- Programming language used (including version, e.g., Java 8 or Python 3)
- A simple example of compiling and/or running the code from a UNIX shell would be best.

Keep it brief, and name it with the question number as the following example: `a1q2_EXECUTION.txt`. If your assignment uses third party libraries, they have to be included in your submission.

Question 1 (10 points):

Purpose: To practice specifying problems, and to lay the groundwork for later questions.

Degree of Difficulty: Easy

AIMA Chapter(s): 3.1, 3.2

Specify the problem. Define:

1. The initial state.
2. The goal state.
3. The actions that can be applied to a given state.
4. The result of applying an action to a given state.
5. The path cost.

You should draw at least a portion of the problem space defined by the initial state and the actions in the specification, just to get a sense of what you're dealing with. But you don't need to hand your drawing in.

Implement your specification using your programming language of choice. In particular, implement:

1. Problem State class
2. Problem class

Be sure to test your implementations thoroughly.

What to Hand In

- A file named `a1q1.txt` containing your definitions.
- A file named `a1q1.LANG` containing your implementation. Use the file extension appropriate for your choice of programming language, e.g., `.py` or `.java`. **If you have more than one file to submit, you should use `a1q1_` to prefix your filenames.**

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 5 marks: You defined the initial state, the goal state, the actions, and results, and the path cost. Your file `a1q1.txt` describes:
  - The components of the data structure (or class) that stores the problem state.
  - The method `is_goal()`.
  - The method `actions()`, and how you represented your actions.
  - The method `result()`.
  - The path cost of sequences of actions (or sequences of states).

  You may copy/paste block comments from your source code, if you've documented it well. Point-form is fine. Full marks will be given if the description is short, and if it is complete.
- 3 marks: Your implementation follows the suggested interface guidelines. Specifically:
  - You have a function or method `is_goal(state)` that returns a Boolean.
  - You have a function or method `actions(state)` that returns a list of actions.
  - You have a function or method `result(state, action)` that returns a new state.
- 2 marks: Your implementation is well-documented. Specifically:
  - Every function or method has at least a brief description of its purpose.
  - Your name, NSID, and student number are in the file.

Solution: A couple of solutions can be found on Moodle. I'll discuss the simple version.

- A State simply stores the expression (a string), the value of the expression (an integer), and the remaining choices, a list.
- The initial state is simply a state with an empty expression, `''`, the value 0, and a list of choices.
- A goal state is recognized by looking at the stored value of the expression. If that's equal to the target, we are at a goal state!
- An action is a tuple consisting of a string, and a number.

  For the initial state, the only action is `'load'`, which basically means *start with the number given*. Every possible number in choices is included among the load actions.

  For all other states, actions are tuples with strings indicating arithmetic (add, mult, etc). All of the four operations are combined with all of the remaining choices, producing a longish list of possible actions.
- The result method creates a new state from a given state and an action. Importantly, it has to calculate the value of the new expression, and it has to make a copy of the remaining choices.
- The path cost for this problem is counted by the size of the expression. Each operation adds 1 to the path cost.

Storing the value of the expression means we only have to calculate it once, which is a big deal.

```
class State(object):
    def __init__(self, value, expr, choices):
        self.value = value
        self.choices = choices
        self.expr = expr

    def __str__(self):
        if self.value is None:
            return '< Initial state,' + str(self.choices) + '>'
        else:
            return '<'+str(self.value)+' '+self.expr+' '+str(self.choices)+'>'

    def used(self, c):
        """Check if value c appears as a choice."""
        return c not in self.choices

    def expression(self):
        return self.expr
```

The Problem class defines the problem-solving interface. We need to remember the target, and the list of choices:

```
class Problem(object):
    """The Problem class defines the transition model for states.
    """

    # Referring to these will reduce string method costs!
    action_start = "load "
    action_add = "add "
    action_mult = "mult "
    action_subtr = "sub "
    action_div = "div "

    def __init__(self, target=None, choices=None):
        self.target = target
        self.choices = choices
```

The function to check for a goal state does not evaluate the expression. This saves a lot of time, and we pay for it with some space. The space is non-trivial, because we need a lot of states!

```
    def is_goal(self, a_state:State):
        """The target value is stored in the Problem instance."""
        return a_state.value == self.target
```

The function to list actions creates tuples to store the operator and the operand separately. Using strings is easier as a first version, but all those string methods add up, so it's faster to avoid using string concatenation, and slicing, etc.

```
    def actions(self, a_state:State):
        """Returns all the actions that are legal in the given state.
            Here, an action is represented as a tuple (string, integer)
            The strings are the ones defined above as class attributes,
            so that we only ever have one copy of each of them.  Every
            action tuple refers to one of these strings.
        """
        actions = []
        if a_state.expr is "":
            for c in self.choices:
                actions.append((self.action_start, c))
        else:
            for c in self.choices:
                if not a_state.used(c):
                    actions.append((self.action_add, c))
                    actions.append((self.action_mult, c))
                    actions.append((self.action_subtr, c))
                    actions.append((self.action_div, c))

        return actions
```

The function to create resulting states follows. Note that the expression is composed, and the value calculated once and for all.

```
    def result(self, a_state:State, an_action):
        """Given a state and an action, return the resulting state.
           An action is a tuple (string, integer).
           To recognize an action, we compare string references using 'is'.
           Presumably, the string refers to one of the action strings defined
           above.  This guarantees that the comparison is fast.
        """
        the_op = an_action[0]
        operand = an_action[1]
        if the_op is self.action_start:
            new_expr = str(operand)
            new_value = operand
        elif the_op is self.action_add:
            new_expr = '(' + a_state.expr + ' + ' + str(operand) + ')'
            new_value = a_state.value + operand
        elif the_op is self.action_subtr:
            new_expr = '(' + a_state.expr + ' - ' + str(operand) + ')'
            new_value = a_state.value - operand
        elif the_op is self.action_mult:
            new_expr = '(' + a_state.expr + ' * ' + str(operand) + ')'
            new_value = a_state.value * operand
        elif the_op is self.action_div:
            new_expr = '(' + a_state.expr + ' // ' + str(operand) + ')'
            new_value = a_state.value // operand
        else:
            # it's not one of the known strings.  Uh oh.
            new_expr = None
            new_value = None

        new_choices = a_state.choices.copy()
        new_choices.remove(operand)
        return State(new_value, new_expr,new_choices)
```

There is a second, more efficient, version on Moodle which has the identical interface, but the data structures are different. Instead of storing the expression, the action is stored. This avoid string concatenation. Instead of storing a list of choices, each State stores the value chosen, and a reference to the parent State, effectively a linked tree of States. Copying all those lists of choices consumes a lot of memory! The more efficient version simply walks up the tree to the root, and checks the choices. This takes a bit more time, but it saves a ton of space. For large problems, this matters!

Notes for markers:

- This question doesn't have an executable requirement. In other words, there needs to be nothing to run.
- Check the file `a1q1.txt`.
  - It should outline the main ideas that are in the code. Give full marks if the explanation is more or less clear.
  - You are not marking for efficiency.
  - Students' work does not need to match the model solutions to get full marks.
  - Deduct marks if the actions function does not return all actions (i.e., if it makes choices about which actions to return), or if it returns states.
- Interface: Check the implementation for the State and Problem classes.
  - Give full marks if the three functions have the required interface.
  - This is all or nothing.
- Internal Documentation: Check the implementation for the State and Problem classes.
  - Give full marks if each of the required functions has a comment indicating what it does.
  - The documentation has to be reasonable, but need not be long. A little more terse than the model solution is fine.
  - Give zero marks if there is no internal documentation at all.

Question 2 (10 points):

Purpose: To build and apply uninformed search algorithms to the problem.

Degree of Difficulty: Moderate

AIMA Chapter(s): 3.3, 3.4

Using the guidelines provided in the textbook, and in lectures 03 and 04, implement TreeSearch, with the following search strategies:

- Breadth-first
- Depth-first
- Depth-limited
- Iterative-deepening

Remember that the first two strategies differ only in the Frontier ADT implementation (queue, stack). Depth-limited search discards search nodes that exceed the depth-limit; iterative-deepening calls depth-limited search with increasing depth limits until a solution is found.

Test your algorithms by applying them to the example problems found in the file `simple_examples.txt`. Question 5 gets you to put your algorithms to a more intensive test. Each line in the example files gives a target followed by the list of integers (separated by spaces).

Demonstrate by copy/paste from your output that your program generates solutions to a few interesting problems, showing that the expression returned does evaluate to the target.

What to Hand In

- A file named `a1q2_EXECUTION.txt` containing brief instructions for compiling and/or running your code. See page 1.
- A file named `a1q2.txt` containing a demonstration of the output of your program, which is copy/paste from a console, or output file. You only need to show a few examples for each search strategy. If this file is missing, the marker will assume your implementation is incomplete or not working.
- A file named `a1q2.LANG` containing your implementation of the search algorithms. Use the file extension appropriate for your choice of programming language, e.g., `.py` or `.java`. You may submit multiple files, provided that the filenames begin with `a1q2_`

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 2 marks: Your file `a1q2.txt` contains a few examples of output from your program.
- 4 marks: Your implementation follows the suggested interface guidelines. Specifically:
  - You implemented tree search with a FIFO queue, i.e., breadth-first search.
  - You implemented tree search with a LIFO queue, i.e., depth-first search.
  - You implemented a version of depth-first search that limits the depth of search , i.e., depth-limited search.
  - You implemented iterative-deepening search using repeated calls to a depth-limited search.

  The implementations may be separate functions, or they can be implemented by reusing a generalized treesearch algorithm.
- 2 marks: Your treesearch algorithm(s) use the methods `is_goal()`, `actions()`, and `result()` to interface with the Problem class.
- 2 marks: Your implementation is well-documented. Specifically:
  - Every function or method has at least a brief description of its purpose.
  - Your name, NSID, and student number are in the file.

Solution: A model solution can be found on Moodle, in the files `UninformedSearch.py` and `Frontier.py`. In the file `UninformedSearch.py`, there are a bunch of classes related to search:

- SearchNode: contains the state, a parent node, and the path cost.
- SearchResult: contains a Boolean flag indicating success, a reference to a SearchNode, and various statistics about the time and space consumed by the search.
- Search: contains the search algorithms
  - The `_tree_search()` function is the basic search algorithm. It interacts with the Problem class only through the standard interface (see A1Q1). It interacts with a Frontier through a standard interface (see below).
  - The algorithms for DFS, BFS, and Depth-limited simply set up the frontier, and then call `_tree_search()`.
  - The IDS algorithm calls Depth-limited search a number of times.

The file `Frontier.py` defines several variants of the Frontier, using a common interface.

- FrontierFIFO implements a FIFO queue, and is used for BFS.
- FrontierLIFO implements a LIFO queue, and is used for DFS.
- FrontierLIFO_DL implements a LIFO queue, but ignores SearchNodes that are too deep. This one is used for depth-limited search (and indirectly by IDS).

Notes for markers:

The model solution is quite carefully engineered for reusability, and it's not reasonable to expect that students obtain the same level of design. It is going to be very likely that each of the search strategies will have its own version of the search loop. That's perfectly acceptable.

- This question has an executable requirement. Check the file `a1q2_EXECUTION.txt`. Markers are not required to run the code, but may choose to do so. If a marking decision could be decided by running the code, and if the execution instructions are incomplete, or not helpful, the grade item in question can be given zero.
- Check the file `a1q2.txt`.
  - It should demonstrate the uninformed search strategies on a few examples.
  - Full marks unless the file is missing, or if it does not provide a demonstration.
- Check the implementation files beginning with `a1q2_`.
  - If the prefix `a1q2_` is not used, give zero marks.
  - Check the search algorithms for the following basic ideas:
    - DFS and Depth-limited search must use a LIFO stack.
    - BFS must use a FIFO queue.
    - Iterative deepening must call Depth-limited search.
  - All search algorithms use the Problem interface: `is_goal(s)`, `actions(s)` and `result(s,a)`.

- Internal Documentation: Check the implementation files.
  - Give full marks if each of the required functions has a comment indicating what it does.
  - The documentation has to be reasonable, but need not be long. A little more terse than the model solution is fine.
  - Give zero marks if there is no internal documentation at all.

Question 3 (10 points):

Purpose: To practice the design of heuristic evaluation functions for informed search.

Degree of Difficulty: Moderate

AIMA Chapter(s): 3.5, 3.6

Recall that a heuristic evaluation function is a measure attached to a given problem state; this measure estimates the cost of the path from the problem state to the goal state. In the text, it's represented by the function $h(n)$, where confusingly n is a search node containing a problem state. Recall also the lesson in AIMA Chapter 3, which is that one way to derive a heuristic function is to relax one or more constraints in the problem, and use the true future cost in the relaxed problem as an estimate for the future cost in the problem you're trying to solve.

Design a heuristic for the Inverse Arithmetic Problem. Give pseudo-code for calculating it, based on your problem state, and the goal state. In your description, explain how your function comes from a relaxed problem, and give two distinct examples of the heuristic evaluation function applied to a few simple problem states. Address the question of whether or not your heuristic function is admissible.

Notes:

- It's more fun if your heuristic is good, but it doesn't have to be good to get full marks. Make sure you can make a case that it estimates cost to goal, even if not very well.
- You are not required to find an admissible heuristic, but you need to know whether it is or not!

What to Hand In

- A file named `a1q3.txt` containing a brief description of your heuristic evaluation function, mentioning the points described above.

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 2 marks: Your description is clear and generally well-written.
- 3 marks: Your pseudocode description gives a clear picture of the heuristic function.
- 2 marks: You gave at least two examples of the heuristic function to real problem states.
- 3 marks: You addressed the question of admissibility clearly demonstrating that you understand the concept. It does not matter if your heuristic function is or is not admissible; just be sure you know which!

Solution: There are a few key aspects that all heuristic evaluation functions should have.

- It must estimate the remaining path cost. If the $h(n)$ value is not comparable to $g(n)$ in terms of scale (or "units"), then one of two things could happen:
  1. If the $h(n)$ values are much bigger than $g(n)$ values, then A* starts to behave like GBFS; none of the caution that $g(n)$ is intended to create has an effect.
  2. If the $h(n)$ values are much smaller than $g(n)$ values, then A* starts to behave like UCS of DFS; the heuristic provides almost no guidance at all.
- It should be efficient to calculate.

The heuristic that I used is defined by a very simple function:

```
def calc_h(self, target, value):
    """This function computes the heuristic function h(n)
    """
    return int(round(math.log10(1+abs(target - value))))
```

The heuristic looks at the difference between target and value. The bigger the difference, the more steps. However, to convert to steps, I use $\log_{10}$. The use of absolute value allows for negative differences, and adding 1 means $\log_{10}$ won't die if value equals the target. Rounding gets us to an integer number of actions.

We can understand the heuristic by considering a "relaxed problem" is one in which every action is "Multiply by 10".

This heuristic is not admissible. To demonstrate that it's inadmissible, we need only recognize that the function will overestimate the number of steps on a few examples.

- $T = 500$, expression is `'2 * 5'`. The heuristic estimates 3 steps to a solution, but if the list contains the value 50, there's a solution in a single step.
- $T = 500$, expression is `'(3 * 25) * 10'`. The heuristic estimates 2 steps to a solution, but if the list contains the value 2, there's a solution in a single step.

It's also possible to show that the above heuristic value is not consistent. Suppose we have a target of 500, and an expression whose value is 10. The heuristic calculates an estimate of 3 steps to a solution. But suppose the action `'Multiply by 50'` is available. The next state has an expression whose value is exactly 500, and heuristic estimates zero cost to a solution (appropriately). The condition for consistency is clearly violated:

$$h(10) = 3 \geq 1 + h(500) = 1 + 0 = 1$$

Notes for markers:

The key aspects for grading are as follows:

- A clear statement about what the heuristic does is presented.
- Pseudocode for the heuristic is included in the description.
- There are examples in the description.
- The description includes a discussion about admissibility (or consistency). Heuristics are not expected or required to be admissible for full marks.

Question 4 (10 points):

Purpose: To build and apply informed search algorithms to the problem.

Degree of Difficulty: Easy

AIMA Chapter(s): 3.5

Using the guidelines provided in the textbook, and in lecture, implement TreeSearch, with the following search strategies:

- Uniform-cost search (UCS)
- Greedy best-first search (GBFS)
- A* Search

Remember that these three strategies differ only the measure that the frontier ADT uses to organize the search nodes. UCS uses $g(n)$ only; GBFS uses $h(n)$ only; A* uses $g(n) + h(n)$; here, $g(n)$ is the path cost from the initial state to the state contained in node n, and $h(n)$ is your heuristic evaluation function from the previous question.

Test your algorithms by applying them to the example problems found in the file `simple_examples.txt`. Each line in the example files gives a target followed by the list of integers (separated by spaces).

Demonstrate by copy/paste from your output that your program generates solutions to a few interesting problems, showing that the expression returned does evaluate to the target.

What to Hand In

- A file named `a1q4_EXECUTION.txt` containing brief instructions for compiling and/or running your code. See page 1.
- A file named `a1q4.txt` containing a demonstration of the output of your program, which is copy/paste from a console, or output file. You only need to show a few examples for each search strategy. If this file is missing, the marker will assume your implementation is incomplete or not working.
- A file named `a1q4.LANG` containing your implementation of the search algorithms. Use the file extension appropriate for your choice of programming language, e.g., `.py` or `.java`. You may submit multiple files, provided that they filenames begin with `a1q4_`

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 2 marks: Your file `a1q4.txt` contains a few examples of output from your program.
- 4 marks: Your implementation follows the suggested interface guidelines. Specifically:
  - You implemented tree search with a priority queue.
  - You implemented UCS using treesearch with a priority queue, taking path-cost $g(n)$ to order the priority queue.
  - You implemented GBFS using treesearch with a priority queue, taking estimated cost to goal $h(n)$ to order the priority queue.
  - You implemented A* using treesearch with a priority queue, taking estimated solution cost $f(n) = g(n) + h(n)$ to order the priority queue.

  The implementations may be separate functions, or they can be implemented by reusing a generalized treesearch algorithm.
- 2 marks: Your treesearch algorithm(s) use the methods `is_goal()`, `actions()`, and `result()` to interface with the Problem class.
- 2 marks: Your implementation is well-documented. Specifically:
  - Every function or method has at least a brief description of its purpose.
  - Your name, NSID, and student number are in the file.

Solution: A model solution can be found on Moodle, in the files `InformedSearch.py`, and `InformedFrontier.py`.

The file `InformedSearch.py` subclasses `UninformedSearch.py` and provides the three informed search algorithms, UCS, GBFS, and A*. The only real difference is that these methods use a Frontier based on a priority queue, and that can be found in `InformedFrontier.py`.

The file `InformedFrontier.py` defines several variants of the Frontier, using a common interface.

- FrontierPQ implements a priority queue, and is a base class for informed Frontiers.
- FrontierUCS orders nodes according to path cost g, and is used for UCS.
- FrontierGBFS orders nodes according to heuristic estimate h, and is used for GBFS.
- FrontierA* orders nodes according to path cost and the heuristic estimate, $g + h$, and is used for A*.

The last piece of the puzzle is that the heuristic value for a Problem State is stored in the state itself. This means that the heuristic value is ready to be used simply by looking into the problem state. An accessor would be safest, but a public attribute works just fine.

```python
class InformedState(State):
    """We add an attribute to the state, namely a place to
       store the estimated path cost to the goal state.
    """
    def __init__(self, value, expr, choices, hval):
        super().__init__(value, expr, choices)
        self.hval = hval
```

A new class for informed search subclasses the Problem class defined above.

```python
class InformedProblem(Problem):
    """We add the ability to calculate an estimate to the goal state.
    """
    def __init__(self, target=None, choices=None):
       super().__init__(target, choices)
```

The heuristic value is calculated by a new method in the InformedProblem class:

```python
    def calc_h(self, target, value):
        """This function computes the heuristic function h(n)
        """
        return int(round(math.log10(1+abs(target - value))))
```

The heuristic value is calculated in the call to result(), as follows:

```python
    def result(self, a_state:State, an_action):
        """Given a state and an action, return the resulting state.
           An action is a tuple (string, integer).
           We add the heuristic value to the informed state here.
        """
        result = super().result(a_state, an_action)
        result.hval = self.calc_h(self.target, result.value)
        return result
```

Notes for markers:

The model solution is quite carefully engineered for reusability, and it's not reasonable to expect that students obtain the same level of design. It is going to be very likely that each of the search strategies will have its own version of the search loop. That's perfectly acceptable.

- This question has an executable requirement. Check the file `a1q4_EXECUTION.txt`. Markers are not required to run the code, but may choose to do so. If a marking decision could be decided by running the code, and if the execution instructions are incomplete, or not helpful, the grade item in question can be given zero.
- Check the file `a1q4.txt`.
  - It should demonstrate the uninformed search strategies on a few examples.
  - Full marks unless the file is missing, or if it does not provide a demonstration.
- Check the implementation files beginning with `a1q4_`.
  - If the prefix `a1q4_` is not used, give zero marks.
  - Check the search algorithms for the following basic ideas:
    - The informed searches require a Frontier based on a priority queue.
    - For UCS, the priority queue is ordered by path cost g.
    - For GBFS, the priority queue is ordered by heuristic estimate h.
    - For UCS, the priority queue is ordered by $g + h$.
  - All search algorithms use the Problem interface: `is_goal(s)`, `actions(s)` and `result(s,a)`.
  -
- Internal Documentation: Check the implementation files.
  - Give full marks if each of the required functions has a comment indicating what it does.
  - The documentation has to be reasonable, but need not be long. A little more terse than the model solution is fine.
  - Give zero marks if there is no internal documentation at all.

Question 5 (10 points):

Purpose: To apply your algorithms to more difficult problems, and draw conclusions from the results.

Degree of Difficulty: Moderate

AIMA Chapter(s): 3 (all)

In the files `moderate_examples.txt` and `harder_examples.txt` you'll find problems that should challenge some of your implementations, and will give you an idea about how well your heuristic evaluation function (Question 3) works.

Run your algorithms (uninformed and informed) on these problems to gauge the quality of your implementations. Give a report on what you found. You might discuss all or any of the following criteria for empirical evaluation:

- Which algorithms used the most time to solve the problems?
- Which algorithms used the most memory to solve the problems?
- How often did your algorithms find an optimal solution?
- Did any algorithms run so long that you terminated them before they returned an answer? If so, which algorithms did you terminate most frequently?

To answer this question, you could consider any of the following modifications to your search algorithms:

- Add code to time the search. Use a clock twice: just before starting the search loop, and just before you return a solution.
- Add a time threshold to your search algorithms, to terminate long-running searches. Set the threshold to short values until you are sure everything is working, and then set it longer to collect your data.
- Count the number of nodes that your search generates, and the maximum size of your frontier over the course of solving a single problem. Both of these statistics can be insightful!
- Don't calculate average times (or memory) across the set of problems. An average isn't really meaningful here, because the variation in problem difficulty is too high. Instead, count the number of times one algorithm is faster than another, and by how much.
- How well does your heuristic function from Question 3 work, compared to uninformed algorithms?

Comment on your findings. Include tables, plots, etc to support your findings.

What to Hand In

- A file named `a1q5.txt` (other formats, DOC, DOCX, PDF, are acceptable) containing a discussion of your findings in this question.

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 5 marks: Your description is clear and generally well-written.
- 5 marks: Your description is based on data (tables, plots), and not simply vague impressions you obtained while coding.
  - You summarized the behaviour of the different uninformed and informed search strategies on the collections of example problems provided.
  - Your summary mentioned issues related to completeness, optimality, as well as performance in terms of time (clock, or nodes expanded), and memory (maximum size of the frontier).
  - Your description included a discussion of the quality of your heuristic function from Question 3, by comparing your informed results to the uninformed results.

Solution:

To summarize the performance of the search algorithms, I timed every example (Time), counted how many nodes were expanded (Expanded), and recorded the maximum size of the frontier at any point during the solving of a single example as an indicator for space usage (Space). The time and the nodes expanded values should be strongly correlated. The summary gives the median statistic for these data. That means there were as many examples above the median as below the median. An average would be overly influenced by extreme values. I also counted the success rate: how many problems were unsolved (Unsolved, lower is better), and I determined the maximum depth of any solution (Depth); this is an indication of optimality.

I ran all the algorithms on the **simple problems**, limiting the runtime to 100 seconds for each problem.

Search	Time	Expanded	Space	Unsolved	Depth
DFS	0.104	12176	73	0/40	7
BFS	0.033	892	10632	0/40	4
IDS	0.046	1262	49	0/40	4
UCS	0.052	892	10632	0/40	4
GBFS	0.003	47	534	0/40	6
A*	0.002	24	447	0/40	5

DFS generally took the longest, A* was fastest. UCS and BFS were essentially the same (as expected), with BFS taking slightly less time. BFS and UCS used a lot more space. The depth statistic reveals that DFS, GBFS, and A* are not optimal, which is expected.

Of the uninformed methods, IDS was clearly superior, requiring the least amount of space. Informed search using the A1Q3 heuristic really worked well, and for these small problems GBFS and A* were comparable.

I ran all the algorithms (except DFS, because it's terrible, and UCS, because it's the same as BFS) on the **moderate problems**, limiting the runtime to 100 seconds for each problem.

Search	Time	Expanded	Space	Unsolved	Depth
DFS	-	-	-	-	-
BFS	7.31	28929	1037732	54/198	5
IDS	2.21	27346	121	5/198	5
UCS	-	-	-	-	-
GBFS	0.010	66	2172	26/198	15
A*	0.012	72	3014	0/198	6

There were enough easy problems that BFS was not a complete disaster. The informed search strategies GBFS and A* were two orders of magnitude faster than IDS. It looks like GBFS was fastest, but keep in mind that A* solved the 26 problems that GBFS could not solve in 100 seconds. By the Depth statistic, we can see that GFBS found at least a few relatively poor solutions, whereas A* was quite close to optimal; it's hard to say if the Depth value of 6 is due to a sub-optimal solution, or a solution to a problem that none of the optimal solutions could solve. The effective branching factor for A* on these problems was about 2.9.

I ran BFS, IDS, GBFS and A* on the **harder problems**, limiting the runtime to 100 seconds for each problem.

Search	Time	Expanded	Space	Unsolved	Depth
DFS	-	-	-	-	-
BFS	100.0	83364	3922581	158/192	5
IDS	103.9	758073	160	97/192	6
UCS	-	-	-	-	-
GBFS	100.0	442771	13815439	154/192	13
A*	0.234	1307	47290	8/192	9

For all but A*, the median time is 100 seconds or more, which means that most of the problems were unsolved, and cut off as close to 100 seconds as possible. My implementation of IDS might not get an opportunity to cut off the search right at 100 seconds. GBFS fails here because the heuristic is not enough to guide the search to a solution; there will be very many states with roughly the same heuristic value, and GBFS is beginning to look like BFS. A* solved almost all of the problems, and most of them in less than a second. The combination of a half-way decent guess, h, and the caution of thinking about paying high costs, g, were very effective. The effective branching factor for A* on these problems was about 4.1.

Conclusions: IDS is the best uninformed method. It does a lot of work, but its space use is so low that it's at least plausible for harder problems. A* performed best of all, and the heuristic was informative enough to provide high quality solutions in a reasonable amount of time for almost all problems. In general an instance of the Inverse Arithmetic problem has many sub-optimal solutions, and probably a few optimal ones (all with the same g cost), so that increases the chance that an informed search algorithm can find a solution. But there are instances (e.g., anything involving the product of all the numbers) that have very few optimal solutions, and solving those will be more challenging.

Notes for markers:

- Check the file `a1q5.txt`.
- Assess the quality of the descriptions:
  - Point form is perfectly fine.
  - Give full marks unless it is particularly difficult to understand.
- Assess the content of the document. For full marks, it must:
  - Describe some aspect of how well the algorithms worked. It is to me expected that IDS out-performs uninformed methods. Some students might not have a great heuristic evaluation function, so if their result s for A* are not great, that's fine. The document should make a comment either way.
  - Comment on issues of completeness, optimality, time and space usage. We expect IDS to be complete and optimal, if given enough time; IDS might halt before finding a solution if time is limited severely. We expect that DFS and IDS use least space.
  - Comment on the effectiveness of the heuristic from A1Q3. The quality of GBFS and A* depends on the heuristic. It could be better than IDS, or it could be more like BFS.
  - Support the comments with data, tables, or lists of values, or plots. The tables don't have to be exactly like mine, conclusions cannot be drawn without any data.

The discussion above gives the flavour of what could be done, but it shouldn't be used as an expectation of what should be done.

Extra work for the ambitious

1. Implement the GraphSearch algorithm, and compare your results to the TreeSearch algorithm results.
2. Drop the assumption that a number can be used at most once. Ask Mike for example problems that are created by allowing multiple uses of some numbers. Compare your results to the other variation.