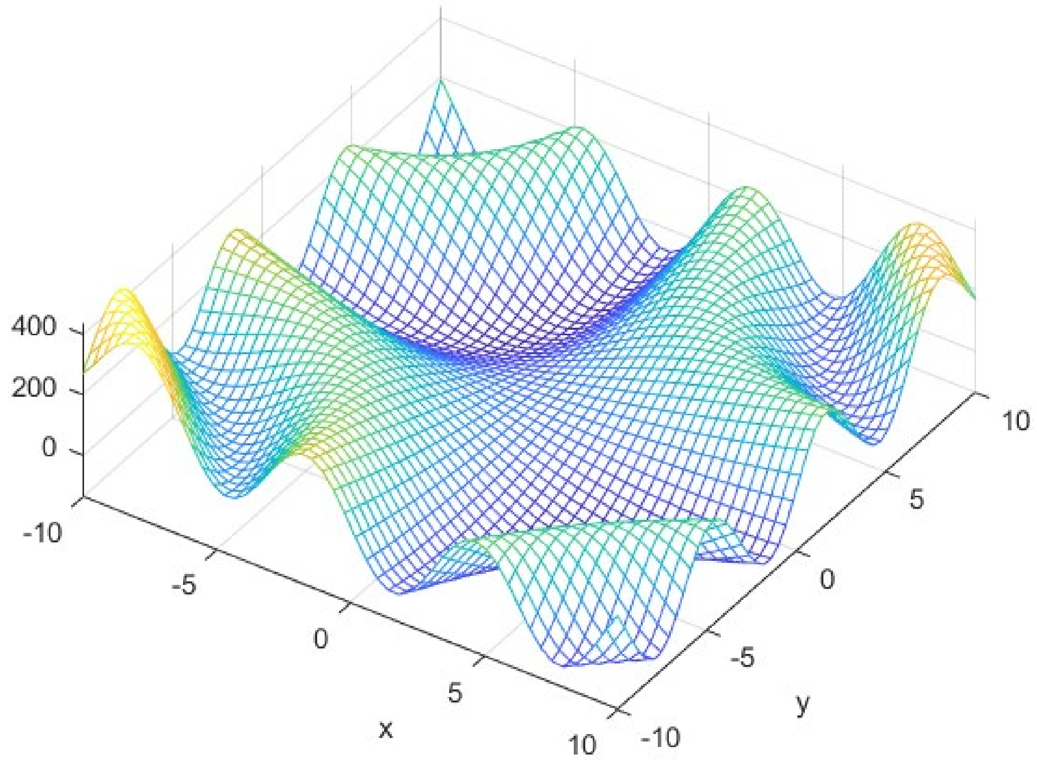
A2

Part I must be done individually, remember to show your work for all quantitative questions. Part II may be done with a pair programmer or individually.

Part I – Questions (20 points each unless noted otherwise)

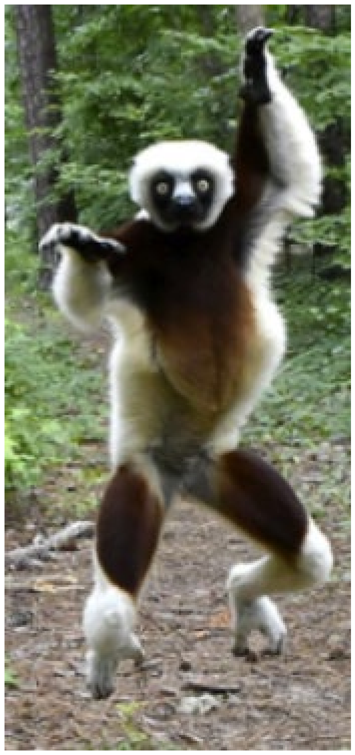
1. Prove that the Manhattan distance is a consistent heuristic in an N-puzzle solution search for any constant state transition cost ≥ 1.

2. An agent is navigating the following fictional landscape on planet Kepler 22-b that corresponds to the function: $f(x, y) = 150 sin(xy/10) - 2y - 3x + xy + x^2 + y^2$:



The agent is looking for the lowest point in the landscape (goal: find potential surface water). a) What is the gradient function? b) Given position (-.6, 7), what would the new position be after one update if the update step size is 0.01? c) Verify that this decreases the objective function by computing the objective function at the original and new locations.
Reminder: $\frac{d}{dx} sin(kxy) = ky\ cos(kyx)$

3. A zoologist is managing three populations of critically endangered Coquerel's sifaka (*Propithecus coquereli*, image credit Duke Lemur Center). The sifaka have a natural diet of: leaves, flowers, fruit, bark, and dead wood. The zoologist is currently feeding the populations accordingly:



	Units of food provided				
population	leaves	flowers	fruit	bark	dead wood
A	4	4	5	7	5
B	5	6	5	3	4
C	5	5	5	5	4

The lemurs' fitness is defined as the squared deviation from their optimal weight. Treating the diet as a list, e.g. [4,4,5,7,5] for population A, write Python-like pseudocode functions for *mutation(diet)* and *crossover(diet1, diet2)*.

(Note that while sifaka really are endangered and this is what they eat during part of the year, this simplified mockup does not take into account things that a real world modeling problem would need to consider such as nutritional value, etc. Otherwise said, if you find yourself managing a population of lemurs 😉, do not do this.)

4. Consider the fairly simple problem of selecting 2 matching cards from a deck of cards with pictures on them. The pictures consist of moon, sun, and stars, and there are two instances of each card (6 cards in total). Legal moves are pick a card that has not been picked, and the goal state is matching the first card drawn. Sketch an and-or search tree for this problem. You do not need to draw the full tree, just enough to make it clear that you understand what the full tree would look like.

Part II: Coding

1. (40 points) Write a Python module newtonraphson.py that contains functions to find the roots of an equation for a polynomial in x using the Newton-Raphson method. Your solution should have the following signature:

```
def NewtonRaphson(fpoly, a, tolerance = .00001):
    """Given a set of polynomial coefficients fpoly
    for a univariate polynomial function,
    e.g. (3, 6, 0, -24) for 3x^3 + 6x^2 +0x^1 -24x^0,
    find the real roots of the polynomial (if any)
    using the Newton-Raphson method.

    a is the initial estimate of the root and
    starting state of the search

    This is an iterative method that stops when the
    change in estimators is less than tolerance.
"""
```

coefficients should be an iterable (tuple or list) of coefficients for the powers of x. As an example:

$$7x^4 + 3x^3 - 5x^2 + 32x - 7x^0 = 0$$

would be represented as $[7, 3, -5, 32, -7]$. Suggested examples for trying:

NewtonRaphson([7, 3, -5, 32, -7], 5) and
NewtonRaphson([7, 3, -5, 32, -7], -50)

Note that only real roots will be returned when we start the search with a real value. **You may not use any library functions for taking derivatives or evaluating polynomials.** Write auxiliary functions (also in newtonraphson.py):

```
def polyval(fpoly, x):
    """polyval(fpoly, x)
    Given a set of polynomial coefficients from highest order to x^0,
    compute the value of the polynomial at x.  We assume zero
    coefficients are present in the coefficient list/tuple.

    Example:  f(x) = 4x^3 + 0x^2 + 9x^1 + 3 evaluated at x=5
    polyval([4, 0, 9, 3], 5))
    returns 548
    """
```

```
def derivative(fpoly):
    """derivative(fpoly)
    Given a set of polynomial coefficients from highest order to x^0,
    compute the derivative polynomial.  We assume zero coefficients
    are present in the coefficient list/tuple.

    Returns polynomial coefficients for the derivative polynomial.
    Example:
    derivative((3,4,5))  # 3 * x**2 + 4 * x**1 + 5 * x**0
    returns:  [6, 4]     # 6 * x**1 + 4 * x**0
    """
```

2. N-Puzzle Search (120 points)

Write a generic graph search routine that can be used to conduct breadth-first, depth-first, and A* search by varying the parameters given to it. Several classes are provided to you in package basicsearch_lib02 to help you in this endeavor and you are required to use them. Do not support multiple solution states for this assignment, and your goal state is the ordered tiles with the blank at the bottom right of the puzzle.

Skeleton code is provided in the following files that can be accessed from Canvas:

tiledriver.py – The driver module contains function driver that is executed if tiledriver.py is the main entry point of the Python program. It should create 31 different tile board puzzles. For each of these, a solution will be searched using breadth first, depth first, and A* search using a Manhattan distance heuristic. This means that you are searching on the same puzzle three times each and there are 31 different puzzles. Keep track of the number of nodes expanded and the amount of time used for each of these. Upon completion, print a table summary that indicates the mean (average) and standard deviation of the following measurements:

- length of plan (number of steps to solution)
- number of nodes expanded
- elapsed time in seconds.

Running the 31 trials will take a little bit of time, so use a much smaller number when debugging. One execution of the 31 trials on an Intel i7-9700U took a little under 19 minutes without exploiting multiple cores (duration depends on the difficulty of the randomly generated puzzles) and it is recommended that you ensure that each search type is working before you try to start looping on solutions.

npuzzle.py – Partially completed class to represent the problem, derived from a generic Problem representation in basicsearch_lib02.searchrep. Complete the skeleton code that is provided for you. When you create a problem to be searched, you should create an instance of this class which will contain an instance of TileBoard. The parent class, Problem, accepts two keyword arguments that you will need to use (do not forget to call super): g and h. g is a cost function and h is a heuristic function. See descriptions in modules searchstrategies specified below. You can pass a function handle by using the name of the function, e.g. g=BreadthFist.g.

problemsearch.py – Implement function graph_search. It takes an instance of NPuzzle and flags for controlling verbosity and debugging (see file for details) and conducts a search. When instantiating the NPuzzle, be sure to use parent class's g and h arguments to set the path cost and heuristic functions which will govern the type of search that is conducted. Function graph_search returns a 3 tuple consisting of:

- a plan (list of actions)
- the number of nodes that were expanded.
- the amount of time the search took in seconds (use the provided Timer class in timer.py to time execution).

You will need to use a priority queue and search nodes (see below)

searchstrategies.py – Implement classes BreadthFirst, DepthFirst, and Manhattan. These are classes that provide implementations for the cost to node (g) and cost from node to goal heuristic (h) functions. Note g and h are class methods (see details on class methods in the comments of the provided code) and that when you pass them to the NPuzzle constructor, you need to pass the function handles to the constructor rather than invoking the function. Concretely, if you wanted to pass BreadthFirst's function handle for g to NPuzzle, you would call NPuzzle(num_tiles, g=BreadthFirst.g, …). NPuzzle's parent class stores g in a publicly accessible instance variable and other code (e.g. the Node class discussed below) will invoke the functions to evaluate search nodes. Pay close attention to the expected signatures. For any search node *child* (class Node) generated as a child of node *parent*, we will call:

g(cls, parent, action, child) – Computes the cost to arrive from start to child.

h(cls, child) – Estimates the cost from child to the goal.

The definitions of these will depend on the search type as we discussed in class. Properties of the search node such as depth, problem state, etc., can be accessed from the node, see the implementation of class Node for details.

explored.py – Implement class Explored. Apart from the zero argument constructor, it has two methods: exists(state) and add(state). Both of these expect state tuples from a TileBoard (see TileBoard.state_tuple()) and will use a hash table to determine whether a state has been seen before (exists) and to add new states as they are removed from the frontier set (add).

The module basicsearch_lib02 contains several functions that you should use in your implementation. Do not make any changes to these:

basicsearch_lib02.queues.py – Class PriorityQueue should be used to maintain the order of the queue. Remember, we defined g and h in a way that priority queues can be used all search types and f(node) = g(node)+h(node). Order your queue by f(node).

basicsearch_lib02.searchrep.py – This module contains classes that will be useful in representing search problems and trees.

The Problem class represents a generic problem class that should be the parent of your NPuzzle class, and as stated has constructor arguments for specifying the g and h functions. Class Node should be used to construct a node in the search space, and these are the objects that should be generated in your search. To create the first node, call the constructor with instances of the problem representation, NPuzzle, and the initial state (a

TileBoard instance). To obtain the list of new search states obtainable from a search node, call expand with an instance of the problem. It will use the problem's actions method to determine the possible actions that can be generated and derive new child nodes. Node's get_f function will be useful for maintaining the order of your priority queue. While you need not modify anything in the library, be sure that you understand how this is working. Step through the code with a debugger if you cannot follow it by reading. Finally, searchrep provides function print_nodes that takes a list of search nodes and prints out their puzzle representations on a single line. You may find this useful for debugging, but otherwise the function serves no purpose.

What to turn in:

- Part I questions must be submitted as a Word or PDF document to Canvas and include an affidavit that the work is your own and that you understand the penalties for cheating.
- Part II: Submit
    - newtonraphson.py
    - tiledriver.py – Place single or pair programming affidavit in the comments at the top of this file.
    - npuzzle.py
    - problemsearch.py
    - explored.py. and
    - log.txt, a text file containing the output of running tiledriver.py.