Announcements	Syllabus	Administration	Assignments	Resources

CS B551, 2015

Elements of Artificial Intelligence

Homework 2. Due by 4pm on Friday, Sept 25

Search-Based Problem-Solving

Goals

The goals of this assignment are to understand general Graph Search, its implementation, and how it can be applied to real problems. The task domain will be controlling 2-D motions of a robot hand, as described below. For this world you will implement five queue-modification algorithms for use with a general Graph Search procedure we provide:

1. Breadth First Search
2. Depth First Search
3. Best First Search (with a heuristic you define)
4. Best First Search (with another heuristic you define)
5. A* Search

Definitions

A 'problem instance' is defined by three things, which together provide everything needed to define what the system must solve: (1) the board dimensions, (2) the initial state, and (3) the goal state.

The Robot Hand World

1. There is a rectangular grid with cells that are squares whose side is of the unit length. The width and height of the grid are whole number multiples of this unit length. There is no third dimension. The cells are indexed by 0..(*length* - 1) and 0..(*width* - 1).

2. Every block is a unit square. No two blocks may be in the same position. The blocks are considered distinct from one another, so if a goal state involves one block and another to its left, it matters which of the two blocks is where.

3. There is a single robot hand which can move around the grid. The hand may be in an open state or a closed state. If and only if the hand has no block in it and it is in an open state, it may occupy the same cell as a block. In this situation, the robot hand may do a CLOSE action and pick up the block. When the robot hand does an OPEN action, it is no longer holding the block. The hand may move left, right, forward, or backward with the same cost ***C*** whether or not it is holding a block (no atomic diagonal moves of cost ***C***). The OPEN and CLOSE actions cost C/10. (Note that if these had no cost, uniform cost search might favor open/close loops.) Moving right is the positive y direction, moving left is the negative y direction, moving forward is the positive x direction, and moving backward is the negative x direction. (This choice of language follows the conventional 3D axis, and the code defining this RobotWorld actually accommodates three dimensions, should you want to try out a harder domain.)

4. In all problem instances, the robot hand starts in the open state and in the bottom left corner of the grid, coordinate (0,0). Each problem instance has its own initial arrangement of blocks.

5. Each problem instance has exactly one goal state, which consists of an arrangement of blocks. The location of the robot hand is not relevant to the goal state. Again, the blocks are considered distinct so it matters which block is where.

6. In the world, there is no 'time' beyond that actions are ordered. Discrete actions punctuate a change in the world. The time the agent spends thinking is not relevant to the world, although it is relevant to you testing your work!

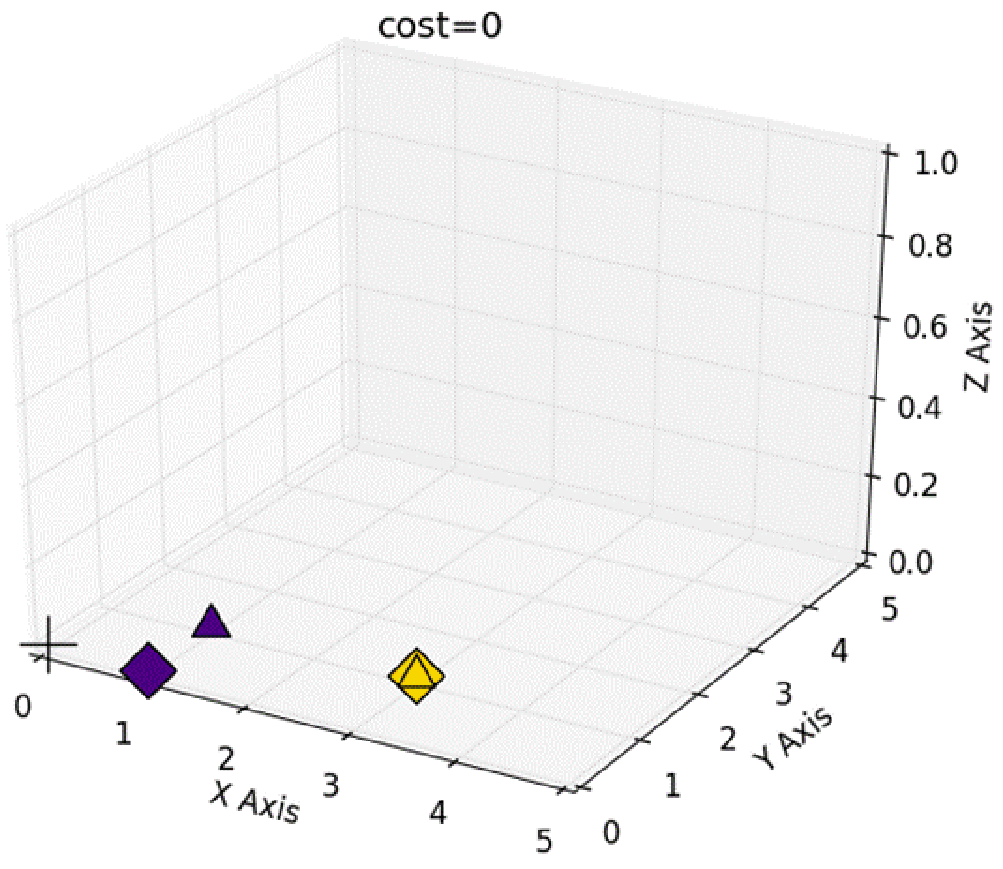

Figure 1: This is a sample initial state

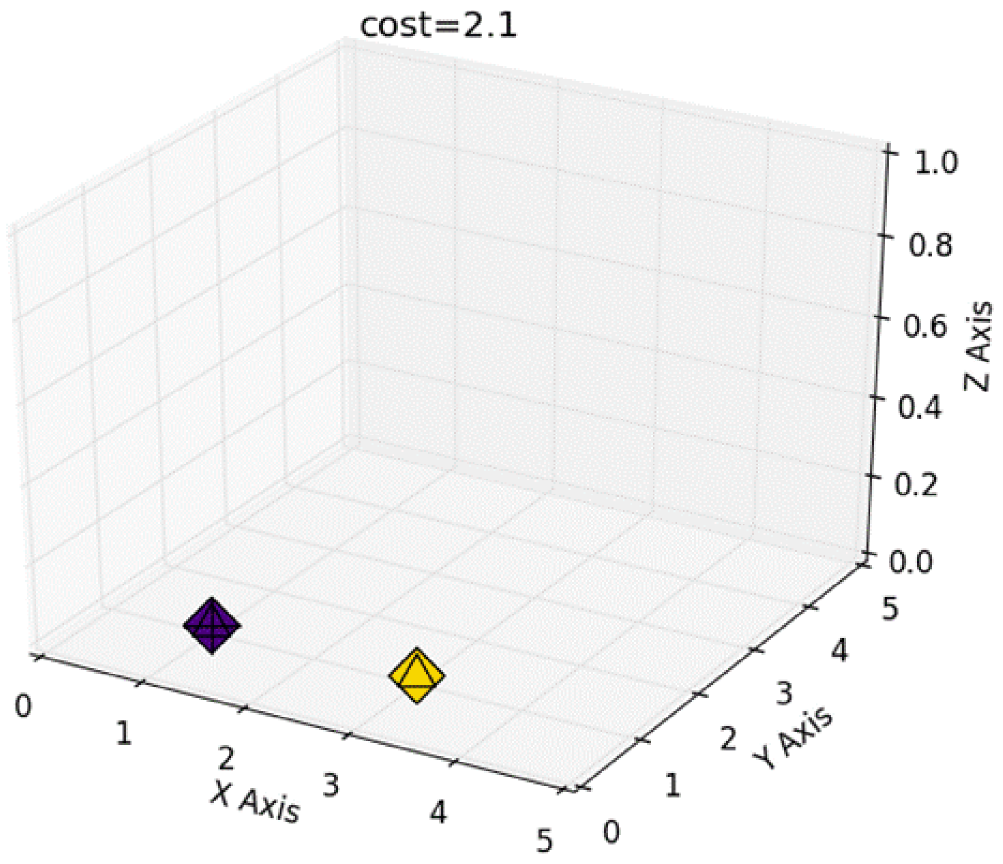

Figure 2: The cross indicates the robot hand holding the block. 'Cost' here indicates the sum of costs of actions leading up to the configuration you see.

What Code Is Provided and Where to Run it

Because the provided code requires libraries which may not be present on other machines, we encourage all students to code and test on the CS linux machines such as the Burrow and Sharks systems.

We provide a general graph search procedure and other support code. This includes:

1. A definition of a world state: the RobotWorld class, which includes data structures which keep track of block and

hand state, accumulated cost, the goal state, and allowed actions given a configuration. Although it may be possible to use only a single RobotWorld object per search, we recommend making copies before doing actions, and storing these as immutable (after that one mutation) states of the world in your queue.

2. A number of problem instances: presolved, easy, medium, hard for testing your code and demonstrating its performance.

3. Visualization code for displaying a block and hand state. When graphsearch finds a solution it automatically calls this on each state of the solution and places a set of serially numbered images in the snapshots directory.

4. A python implementation of Graph Search: graphsearch. We strongly recommend using this rather than implementing your own because it has appropriate error messages, pleasing output, terminates the search at the appropriate time, updates the visited set for you, and initiates the visualizations after searches complete.

   Graphsearch takes three arguments: a queue, a queue modification function, and a timeout limit, which is the number of nodes to expand before terminating.

   The queue_modification argument is a function expands a single node from the queue and updates the queue (according to the algorithm you have in mind). This function takes a queue and a collection of visited RobotWorlds and returns a 2-tuple whose elements are: the updated queue and the RobotWorld the function chose to expand. (You may be asking yourself now,"wait, isn't an additional parameter that A* and Best First need to know about their heuristic?". Yes, and this is explained in the next item.)

5. A function run which takes an RobotWorld, a title for the run (a string for the sake of meaningful output), and a collection (v.i.z python iterable) of heuristics. To use this function, your A* implementation should be called 'aStar', your Best First 'bestFirst', yourBreadth First 'breadthFirst', and your Depth First 'depthFirst'. It doesn't matter what you call your heuristics, because the set of heuristics gets passed as an argument to 'run'."

Note: You do not need to understand the internal details of the code for run. If you read the code, you may be initially be confused by its use of partially applied functions. This is a technique in which some arguments of a function are fixed, to produce a new function of smaller arity. It is useful but you are not responsible for it for the class. Here we have an A* definition with three parameters (one of which is a heuristic function), we have a number of heuristic functions, and we have an A* definition of two parameters for each heuristic.

Tips, Questions and Answers

We'll try to anticipate some possible questions about the RobotWorld here:

1. The `RobotWorld` class is the datatype of elements of the queue.

2. Use python's `heapq` as your priority queue implementation. Note that if you store RobotWorld objects in a heapq, you may wish to make copies of RobotWorlds, modify them, and store the modified version in the queue. If so, you'll want to use `deepcopy` rather than copy.

3. **We strongly encourage examining the code sample `dummyExample` in the support code.** Note that dummyExample is never called; it is just an example of how to correctly use some of the idioms mentioned here, such as pushing items onto the queue, looping over a RobotWorld's allowed actions, and testing whether a state is a goal state.

4. Because graph search involves enumerating possible neighbors to the current state, a RobotWorld provides a `allowedActions` method which returns a set of strings. Each of these strings can be passed as input to the do method, and the corresponding action will happen (and have effects on the robot hand's position, and possibly the state of a block.)

5. If you use the provided `graphsearch` function, you will get a sequence of visualizations of each expanded RobotWorld. These visualizations look like Figure 2.

   - The blocks are colored according to their identity (exactly one color per block). These are persistent over time, so a block with a certain name in one frame of the animation will have the same color in all other frames.

- The triangles of the same color as a block indicate where the block's goal location is.
- The title 'cost=<number>' at the top indicates the cumulative cost of actions so far to bring the RobotWorld into its current state.
- The cross indicates the location of the robot hand.

It is not part of the assignment, but if you wish you could turn these frames into an animation with animation software to enable you to watch the search process. An easy way is to use WinSCP or Filezilla to get all your images to your system (if you're running on a remote machine), and then go to http://gifmaker.me/ and make a free gif for your images.

There is also a FAQ section at the end of this document for more specific questions.

Allowed resources

In writing your code, you may use any material in Russell & Norvig without acknowledgement, but may **not** use other material or look at anyone else's code, including code available online. You may discuss the assignment at a high level with others in the class, but must acknowledge those discussions in the comments.

Submitting the assignment

The assignment will be submitted via Canvas. It will be graded on the CS linux machines, which run Python 2.6.6, so must run there, regardless of where you choose to develop it (be aware that Python 3 code may not be compatible). Be sure to test your code there prior to submission. As it is impossible for graders to test code that doesn't run, maximum credit for non-running code is 50%.

What to Submit

1. The Five search implementations mentioned above. You may write these so that they can interface with graphsearch, or you may write your own general graphsearch procedure and use that. These should be in a python file named search.py. Please be sure to include your name and email address as comments at the top of the file.

2. Output from each of your five search implementations running on all the provided problems. All your methods should solve the presolved and easy worlds. At least A* with your best heuristic should also solve medium and hard worlds. Coming up with a heuristic for solving the hard world may be challenging; the majority of points will be given even if you do not solve this. This can be output from the provided run function. You don't need to be concerned if your system exceeds its time limit for other cases. The output should be in a text file named hw2_output.txt. The linux command "script" captures a transcript until you enter control-D.

FAQ (to be updated if we receive additional general-interest questions)

Why doesn't the code run when I haven't implemented the search algorithms yet? Specifically, what is this "NoneType is not iterable" error I see when I try to do so?

The reason it says "NoneType is not iterable" is because the unimplemented depthFirst function in the support code is a stub that simply returns "None", and the code tries to unpack that None value into a tuple (queue, expanded). It will not be a problem after you define your own search algorithm that interfaces correctly with 'graphsearch'.

Why should I use 'graphsearch' rather than incorporating that functionality into my own search algorithms?

Because it incorporates behavior in common to all the search algorithms. Namely, terminating when a goal state is reached, and adding the expanded vertex to the "visited" set, keeping track of whether the search has timed out, and producing the desired output of the search in a standard format. You write your search algorithms so that they take a visited set and a queue as arguments and return the modified queue and the expanded vertex. You can do this because python, unlike languages like C++, can treat functions as values that can be passed to functions. Namely, the 'queue_modification' argument to graphsearch is where your search algorithms go.

What is this "run" function?

It takes a world and the heuristics you define, and it calls graphsearch with each of your search algorithms. It also takes care of the fact that your A* and Best First implementations have three parameters, whereas graphsearch expects search functions of two parameters. This is done by binding the heuristic internally, and making it no longer a 'parameter'.

What is this line "hs = {}"?

The curly braces indicate a set. The values that this set holds should be the heuristic functions you write, that is functions which take a RobotWorld and return a number. Just as python can consider sets of numbers, like {1,2,3}, it can consider functions like {f,g,h}. The name "hs" is an abbreviation for "heuristics", and this variable is for use by the the 'run' function.

How many heuristics do I need to define?

Two.

I'm using Microsoft Windows and I'm having difficulty with matplotlib, what should I do?

Develop by sshing onto CS linux machines, like burrow. If something really weird is still happening, talk to an AI. Or you could even delete the code related to visualization, because it is not integral to the search process.

Do I need to make an animation?

No, if you choose to make an animation it is for the sake of your own understanding of how your search is working. We will grade based on the textual output of graphsearch.

Additions as of 9/19/15:

What should I do if I'm getting timeouts?

You could raise the timeout value. It is also expected that anything other than A* with a good heuristic may get timeouts on the hard and medium worlds, so don't worry about your DFS timing out on medium or hard.

When using 'run', there seems to be a lot of time between searches? Is something the matter?

Probably not. Visualization takes about as much time as the actual search process, so you can comment out the visualization code if you are trying to run it as fast as possible.

What is 'experiment'?

It is code that automatically calls 'run' on the worlds, and plots the results of the searches. The plot is saved as 'experiment.png'.

It seems like most of the time is spend doing visualization rather than searching. What's up with that?

Yes, visualization does take more of the time. In the newest version of the code, it is turned off by default for this reason. In the definition to graphsearch, simply change "visualization=False" to "visualization=True" if you want visualization.

How do I know if I’m using the newest version of the code?

No changes in the code since the original posting of the assignment should affect the functionality for interacting with the system you write, except that the new version turns off visualization by default (it’s fine either way; see below for why and how to turn it back on if you wish). Consequently, if you've started modifying the code you don't need to use a newer version. If you haven't haven’t made any of your own modifications yet, you are using the latest version of the code if the header file is dated Sept 19. The md5 checksum is 8223fcd6aece5452f1d13a63bb943af7 (you can verify this with the linux md5sum command).

Please let us know if you have questions not covered here!