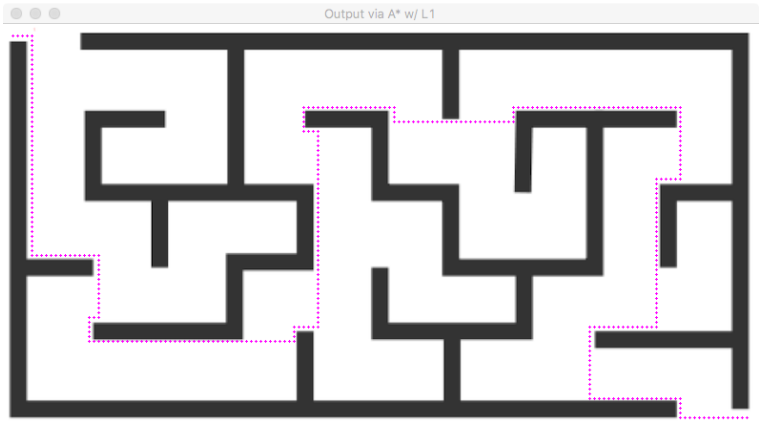
Northeastern University
CS4100 – Artificial Intelligence
Fall 2017, Derbinsky

Solve a Maze via Search

By the end of this project you will have built an application that applies graph search to solve a maze, supplied as an image (see example below). Along the way you will ...

- learn to effectively use OpenCV[1] to both debug intermediate steps in an AI pipeline, as well as usefully output final results;
- compare different representations of a problem in terms of solution quality and solving time;
- as well as compare algorithmic performance of uninformed and informed search algorithms on reasonably sized problems.



1 Software

For this project you will need Python 3 with OpenCV, as well as starter code on which to base your solution.

1.1 Python & OpenCV

To begin, download and install Anaconda (using Python 3) for your platform[2]. Anaconda is a convenient packaging of Python with libraries/tools that are commonly used in data science.

Once installed, open a command prompt with the Anaconda environment[3]. Now type the following command to install OpenCV[4]: `conda install -c menpo opencv3`

After installation, run `python`, confirm it is using Anaconda, and then type `import cv2`; if you receive no errors, you are ready to roll!

[1] Open Source Computer Vision Library (`http://opencv.org`)
[2] `https://www.anaconda.com/download`
[3] Windows: run "Anaconda Prompt"; Mac/Linux: run Terminal
[4] On Mac/Windows you may have to run `conda install python=3.5` before this step.

1.2 Starter Code

Download `starter.zip`, which accompanies this document. It includes Python starter code, as well as mazes both in image and text form.

2 OpenCV

If you are not familiar with OpenCV, now is a good time to learn some basics. Here are some useful resources[5]:

- `http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_tutorials.html`
- `https://pythonprogramming.net/loading-images-python-opencv-tutorial/`
- `http://docs.opencv.org/3.0-beta/modules/refman.html`
- `http://modelai.gettysburg.edu/2010/set/gettingSetWithOpenCV.html`

Before continuing, you should feel comfortable loading/showing images, drawing shapes on images, and accessing/manipulating pixels.

3 Solve a Maze, Visualize Results

Before processing an input image, you should begin by solving an easier problem: given a two-by-two grid of colors (where each distinct color has an associated cost), as well as start and end locations within the grid, apply the GraphSearch algorithm to find a solution path.

For example, the following comes from the `txt/easy_water_50.txt` problem file...

```
(4, 0)
(4, 13)
KKKKKKKKKKKKK
KWWWWBBBBWWWK
KWWWWBBBBWWWK
KWWWWBBBBWWWK
WWWWWBBBBWWWW
KWWWWBBBBWWWK
KWWWWBBBBWWWK
KWWWWWWWWWWWK
KKKKKKKKKKKKK
```

The goal is to get from row 4, column 0 (starting top-left; 0 indexed) to row 4, column 13. For reference, this is a representation of the `img/easy_water.png` file, where each character represents the darkest pixel in a 50x50 patch of the original picture[6]. The character codes, as encoded in the `COST_KWRGB` constant of the `vizutil.py` file, represent...

K Black (∞ cost; i.e. walls)

W White (1 cost; i.e. unimpeded path)

[5] Note that OpenCV is written in C++, and so references and tutorials will often require some additional searching/Python-izing.

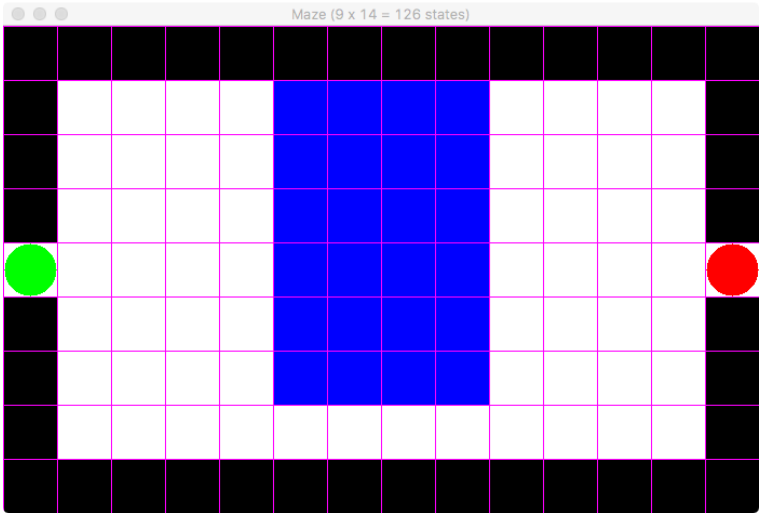
[6] Termed "max pooling" (see `http://ufldl.stanford.edu/wiki/index.php/Pooling`)

R Red (100 cost)

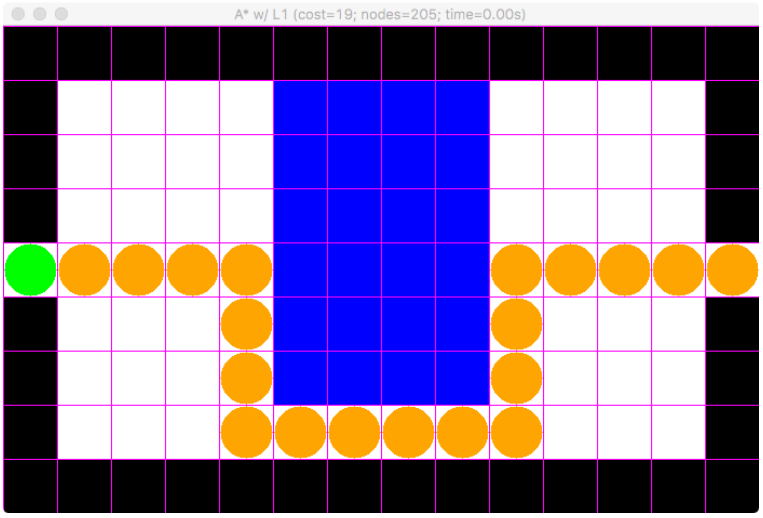
G Green (5 cost)

B Blue (3 cost)

Visually the problem is represented as follows (green circle=start, red circle=finish)...



and once solved using A* search (i.e. GraphSearch using a PriorityQueue with an L1, or Manhattan/Taxi-Cab, heuristic: $f(x) = g(x) + h(x)$, $h(x) = |x|$...



Once you have completed this part of the project, your Python code will produce these two images given a file that has the contents above.

3.1 Visualize a Maze

First, open `viz.py` and implement the `mazeMat` function. To test your function, simply execute the following commands at the prompt...

```
python search.py txt/easy_water_50.txt 50
```

The first argument is the path to the text file, and the second is how big a square each color code should produce visually. Each supplied problem file has embedded in the name an intended output size, and each comes from images in the `img` directory.

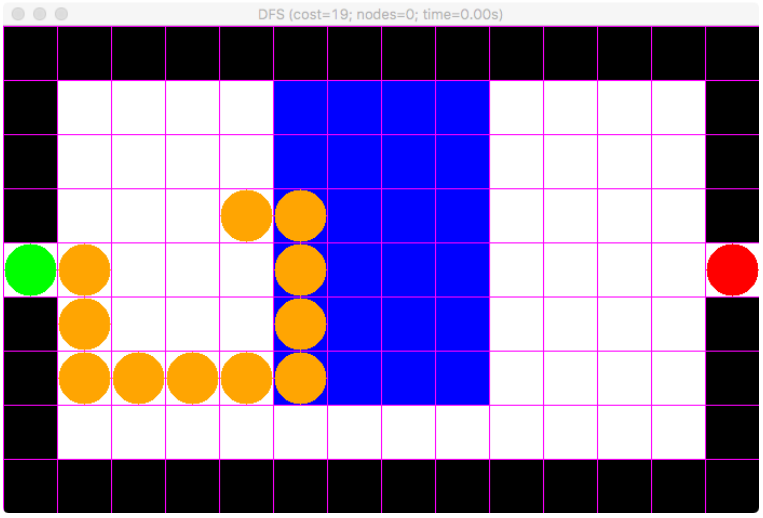
When you run the command a window will pop up and will stay open until you press a key. Afterwards, Python will exit with an error (this is expected, and we will fix this in the next part). An important lesson to learn in many AI-related fields: it's hard to debug what you can't see, and so we are focusing on seeing our problem first.

When you think you have the code working, try other sizes of the easy water maze. Then move on to other mazes. Remember to check that each character has a one-to-one correspondence with a square in the resulting visualization, and that your start/finish locations are accurate.

3.2 `MazeProblem` – with Visualization

Now it's time to implement the `MazeProblem` class in the `search.py` file. You will need to implement functions that provide a start state, goal test, and successor function. You may also add additional constructor functionality, but that isn't necessary.

To test some of this visually, implement the `mazeAddPath` function in the `viz.py` file. This function should use the `whereNext` function of the supplied `MazeProblem` to draw circles representing the supplied path. You will find a commented-out line in the `solveMaze` function of the `search.py` file that should result in the following output when run (`python search.py txt/easy_water_50.txt 50`)...



This thoroughly tests `mazeAddPath`, and some of `getSuccessors`. Don't move on until you've got at least this much working.

3.3 Search!

It is now time to implement the `graphSearch` function in the `search.py` file. Once correctly implemented, simply run the program again to illustrate solving the maze using Depth-First Search (DFS), Breadth-First Search (BFS), Uniform-Cost Search (UCS), and A*. Each solution is presented visually, and will wait for you to press a key to continue.

3.4 Analysis

Use your functioning maze solver to respond to each of the following questions. You will need to submit data and possibly visualizations to support your responses.

1. Submit a spreadsheet with nodes pushed, solve time, solution length, and solution cost for each of the mazes in the `txt` folder using each of the search methods.
2. Which of the methods are not optimal? For any such method, cite a specific piece of data you collected as evidence.
3. The differently sized mazes (e.g. easy_water_1 vs. easy_water_10 vs. easy_water_50) are actually just different representations of the same underlying image (i.e. just pooling more pixels in the original image into a single character code). What effect does the multiplier size have on solving the maze?

Note: if you wish to run a search using only a single search method, you can run the `search.py` program with a final argument, which is the name of that method (see the `METHODS` constant in `search.py` for all legal labels), for example...

```
python search.py txt/easy_water_50.txt 50 "A* w/ L1"
```

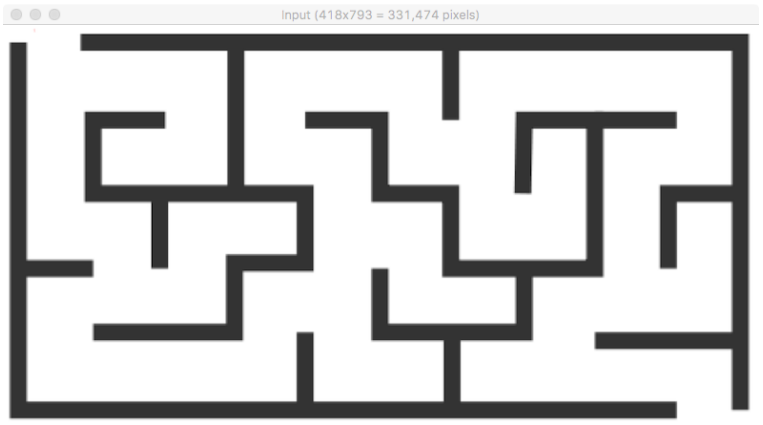
This can come in handy for debugging, as well as re-collecting data without wasting time on slower algorithms.

4 Image Processing with OpenCV

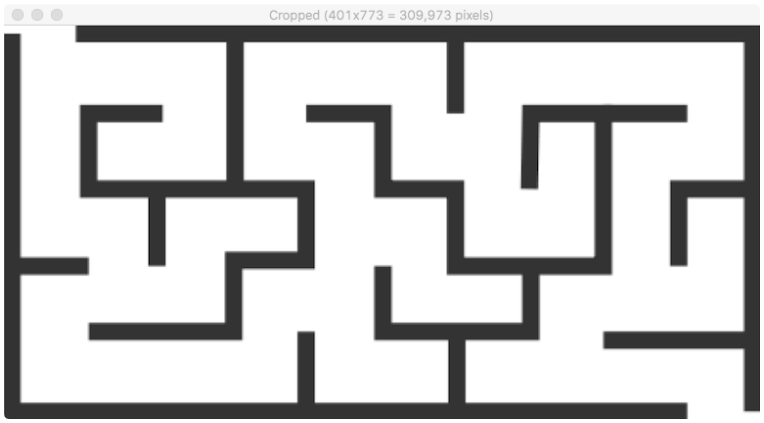
Now that you have written (and understand!) code to solve a maze in a friendly format, it's time to extract a maze from an image. The goal will be for you to apply simple OpenCV functionality to a few image pre-processing steps, utilize the `solveMaze` function you just finished to solve the result, and then super-impose the resulting path on the original image. For example, executing the following command...

```
python image.py img/small.png 10 "A* w/ L1"
```

Would produce something like the following...

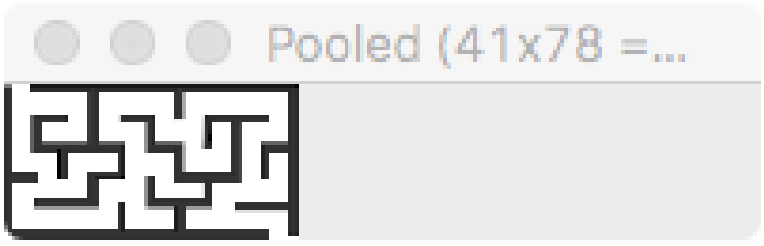


First, the original image (primarily for debugging purposes).

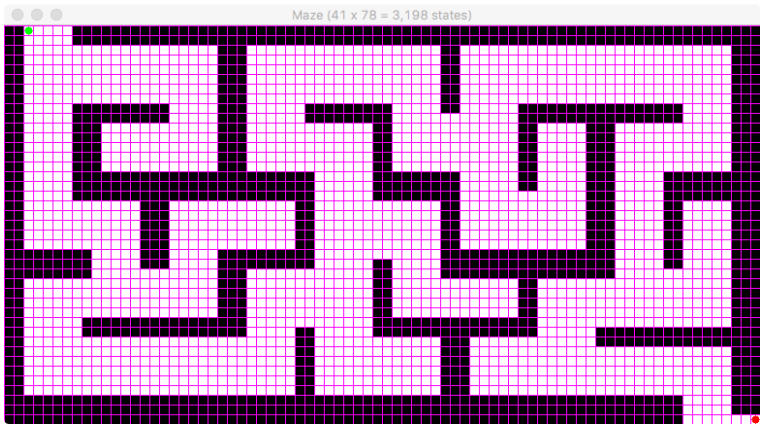


Next, a "cropped" image. For this project we will assume that the maze is "axis-aligned" (meaning, not rotated), rectangular, and surrounded by a border of a fixed color. So your code should exclude any white space surrounding the border.

In real images there may be some noisy pixels that prevent a good crop. A reasonable strategy would be to convert the image to grey scale, perform a Gaussian Blur[7], and then consider only pixels that are above a threshold[8].



Now you should "pool" the image – that is, sequentially look in squares of length `multiplier` (in this case 10), extract the pixel with the darkest pixel within that square, and that becomes the pixel in the pooled image.

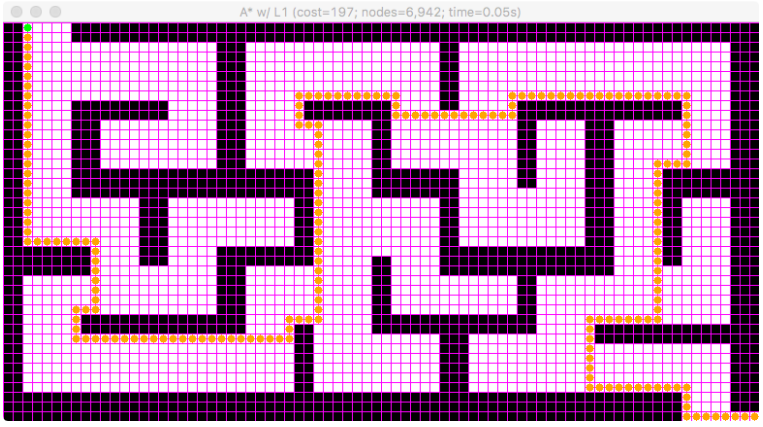


Now to make the maze! First, create the friendly row-of-rows representation from the last part of the

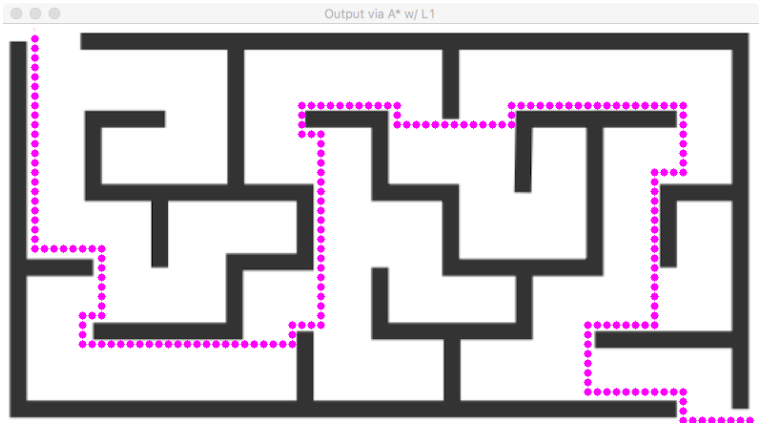
[7] `https://en.wikipedia.org/wiki/Gaussian_blur`
[8] Look at the `cv2.threshold` function for this part.

project. Importantly, pixels in the original image may not correspond perfectly to those in the map of colors to costs (`COST_KWRGB`) – so a reasonable method is, for each pixel, simply find the closest[9]

You will also need to find start/finish locations within the maze. For this project we will make the simplifying assumption that entry/exit locations are on the border, and are white. So a reasonable method would be to collect such locations and find the farthest apart – the "start" is then the location closest to the top left.



Now use the `solveMaze` function you implemented in the `search.py` file. This will plot a solution to the pre-processed maze – you now need to map this solution back to the original image...



It is recommended that you implement this within the `origAddPath` function of the `viz.py` file (just to be clean/organized). Importantly, you will need to take into account the result of cropping (i.e. the degree of offset of the top-left corner of the maze from the top-left corner of the original image), as well as pooling (i.e. enlarge, via the multiplier, path locations from the small pooled maze to the original enlarged image).

You will find a breakdown of these steps in the `main` function of the `image.py` file. You are free to implement additional helper functions in this file as you see fit.

[9] The `cv2.norm` function helps with this part.

4.1 Analysis

Use your functioning image-based maze solver to respond to each of the following questions. You will need to submit data and possibly visualizations to support your responses.

1. Characterize how the max-pooling multiplier affects search time and memory[10].
2. Look at the super-imposed solution of easy_water using a multiplier of 50 and A* search – is this solution optimal? Discuss.
3. Is there an upper bound on pooling size? For example, consider a maze with thin walls...

Note: within the `img` folder you will find a PowerPoint file that can be used to easily construct mazes for your program – have fun!

5 Extensions

You now have a basic application for solving image-based mazes. However, we made several simplifying assumptions. For extra credit, pursue one or more of the following directions...

- Handle differing maze shapes. This will likely require a different `MazeProblem` representation, as well as detection/cropping of the outer border[11].
- Handle rotations. Start with 2D (e.g. use the PowerPoint file to slightly rotate the "easy" mazes). Now consider that you capture an image of a maze from a sheet of paper that is tilted away from a camera – i.e. a 3D rotation of a 2D maze. Look into homography to detect the maze[12].
- To scale to live video[13], there need to be several performance enhancements, in addition to the extensions above. One possible avenue is employing "erosion" to thin-out paths in the maze (i.e. instead of uniformly reducing the size of the image, as with pooling, use content-specific thinning)[14]. Another path is to convert Python loops into more optimized OpenCV calls: depending on your implementation, this may include optimization of the color binning[15] and/or the pooling[16].

[10] Collect data while solving. Windows: Task Manager; Mac: Activity Monitor; Linux: `top`

[11] See `https://www.pyimagesearch.com/2016/02/08/opencv-shape-detection/`

[12] `https://www.learnopencv.com/homography-examples-using-opencv-python-c/`

[13] `http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_video_display/py_video_display.html`

[14] `http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html`

[15] `https://codereview.stackexchange.com/questions/143529/opencv-3-using-k-nearest-neighbors-to-analyse-rgb-image`

[16] `https://stackoverflow.com/questions/38179797/numpy-max-pooling-convolution`