---

title: "Java"

excerpt: "Lecture Notes"

header:

teaser: /assets/images/Java.png

mathjax: true

feature\_row:

- image\_path: /assets/images/Java.png

# alt: "Gameplay 2048"

# title: "Gameplay 2048"

- image\_path: /assets/images/Java.png

# alt: "Gameplay 2048"

# title: "Gameplay 2048"

- image\_path: /assets/images/Java.png

# alt: "Gameplay 2048"

# title: "Gameplay 2048"

---

{% include feature\_row %}

{% include toc %}

This course is currently in progress. Check back later for more updates.

{: .notice--warning}

<!--

## Review

### Course Link

<a href="https://www.edx.org/course/artificial-intelligence-ai-columbiax-csmm-101x-0">Artifical Intelligence - Ansaf Salleb-Aouissi</a>

### Info

This course on artificial intelligence mirrors much of the syllabus from the book <a href="http://aima.cs.berkeley.edu/">Artificial Intelligence: A Modern Approach" - Stuart Russel and Peter Norvig</a>. The course is well articulated by Prof. Ansaf Salleb-Aouissi, University of Columbia, in EdX. The class has 5 programming assignments which translates the theory learnt to solving real-life problems.

### Syllabus

Uninformed/heuristic/adversarial search, constraint satisfaction, perceptron, k-NN, neural networks, support vector machine, decision trees, Markov decision processes, reinforcement learning, logical/propositional agent, applications to natural language processing and vision

### Repository

The [repository](https://github.com/Adaickalavan/Artifical-Intelligence-Ansaf-Salleb-Aouissi-Columbia-University-EdX) consists of the following:

\* Lectures - Notes

\* Projects - Instructions and Python codes

### Verified Certificate

See [certificate](https://courses.edx.org/certificates/f0b21c7e213f46af83b1f5c50b7a8d7e)

## Week 2 Project: Search Algorithms

We are to implement a breadth-first search (BFS), depth-first search (DFS), and A\* search to solve an 8-puzzle game. The pseudo codes for all 3 algorithms are given in the course lecture notes. To complete the assignment, a simple implementation of the given pseudocode is sufficient.

In order to meet the execution time constraints, there are several key points to note as follows.

1. Use `class tree` to represent a tree which depends on nodes created from `class node`. This keeps the code clean and efficient.

1. Use `string` to store the state of each node, instead of lists or arrays. For example, store `stateStr = "21345670"` as a string to denote a node state. String comparison is faster than list/array comparison.

1. Store the explored states' strings in a set : `exploredStates = set()`. Perform `exploredStates.isdisjoint({childStateStr})` to check whether the current child state string `childStateStr` exists within the explored states' strings.

1. Use string based states for comparison and a list based states to reorder state elements when moving to another state. For example, use lists to change `[2, 1, 3, 4, 5, 6, 7, 0] --><!-- [1, 2, 3, 4, 5, 6, 7, 0]`. It is easier to manipuate state elements in a list form. It may be useful to know how to iterate between a string and a list in your code.

```python

#Convert list to string: StateList ---> StateStr

stateStr = ''.join(stateList)

print(stateStr)

#Convert string to list: StateStr ---> StateList

stateList = list(stateStr)

print(stateList)

```

1. A memory and time efficient implementation of priority queue is needed for A\* algorithm. I have implementaed my own below which uses a dual representation of the priority queue in the form of a list and a dictionary. It has two functions, namely, push and pop to place and get items from the queue, respectively. The priority queue:

\* Orders and re-orders items according to their priority

\* Removes duplicate entries with lower priority upon the addition of higher priority entries

\* Uses the first entry if there are duplicate entries with the same priority value

My priority queue implementation, with comments.

Complete code snippet will be uploaded at a later date.

{: .notice--warning}

```python

class priorityQueue():

def \_\_init\_\_(self):

def pushObject(self, priority, item):

def popObject(self):

```

## Week 4 Project: Adversarial Search and Games

We are to solve the 2048 problem using the minimax algorithm with alpha-beta pruning. The algorithm is given in course lecture notes. You may try playing the 2048 game <a href="https://gabrielecirulli.github.io/2048/">here</a> and practice the alpha-beta pruning algorithm <a href="inst.eecs.berkeley.edu/~cs61b/fa14/ta-materials/apps/ab\_tree\_practice/">here</a>.

Although this programming assignment is claimed to be the toughest in this course, it appears the be pretty starightforward.

1. Implement the pseudocode from the course lecture notes

2. Design a heuristic to evaluate the utility

Some good heuristics for solving 2048 are discussed at this <a href="https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048">thread</a>. Key to devising a good heuristic is to ensure the heuristics do not have significant overlap or have low degrees of correlation. Keeping the heuristic simple is really helpful to this end. In my project, I used only two heuristics, namely, number of remaining free tiles and smoothness.

\* Higher number of remaining free tiles imply

\* increased count of tile merges, thus higher valued tiles

\* longer gameplay, thus leading to higher valued tiles

\* Smoothness is the sum of absolute value difference between adjacent tiles. Minimizing the difference results in tiles being arranged in a monotonic order (sprouting from a corner) making tile merges more likely. In fact, smoothness is the <em>king</em> in heuristics for the 2048 game. For experimental results proving this claim, refer <a href="artent.net/2014/04/07/an-ai-for-2048-part-4-evaluation-functions/">here</a>

It is important to normalize the 2 heuristics first. The number of remiaing free tiles was divided with the total number of tiles (i.e., 4x4 = 16) and the total smoothness value was divided by the total sum of all tiles on grid. Hence, we have our final heuristic $$h$$ function to be

$$

\begin{align\*}

h\_1 & = \frac{number\\_of\\_free\\_tiles}{total\\_number\\_of\\_tiles} \\

h\_2 & = \frac{sum\\_of\\_all\\_differences}{sum\\_of\\_all\\_tile\\_values} \\

h & = h\_1 - h\_2 \\

\end{align\*}

$$

A max depth search of 4 levels and a time limit of 0.18 seconds is used, since a time limit for each move is imposed by the assignment instructions.

Below is my execution result for my heuristics, achieving full marks of 100%.

```

[Executed at: Sun May 28 4:14:32 PDT 2017]

Results of 10 trials:

Trials with max tile of 2: 0 %

Trials with max tile of 4: 0 %

Trials with max tile of 8: 0 %

Trials with max tile of 16: 0 %

Trials with max tile of 32: 0 %

Trials with max tile of 64: 0 %

Trials with max tile of 128: 0 %

Trials with max tile of 256: 0 %

Trials with max tile of 512: 10 %

Trials with max tile of 1024: 10 %

Trials with max tile of 2048: 60 %

Trials with max tile of 4096: 20 %

Trials with max tile of 8192: 0 %

Trials with max tile of 16384: 0 %

Trials with max tile of 32768: 0 %

Trials with max tile of 65536: 0 %

```

Code describing my heuristics implementation is as follows, which is only 13 lines long!

Complete code snippet will be uploaded at a later date.

{: .notice--warning}

```python

return h1 - h2

```

## Week 7 Project:

We are to implement machine learning agorithms using SciKit-Learn library

## Week 9 Project:

We are to implement

I represent a constraint as Cx != Cy (value in cell x not equal to value in cell y),

#where a cell is one of the 81 squares on the board. Every square on the board

#must have a different value than the squares in its row, its column, and its

#box (the local 3x3 grid). Therefore, each cell has 20 constraints with other

#cells (3x8 = 24, but 4 are duplicates, leaving 20 unique ones). Therefore,

#there are 81\*20 = 1620 constraints in total.

Also, iterating through dictionary.items() would be faster than iterating

#through keys and accessing elements by them. And a dict comprehension would be

#faster than adding each new value separately.

```python

def visualizeBoard(sudokuAssign):

#Prints a sudoku grid containing the assigned variables.

#This function accepts a dictionary of the form:

#sudokuAssign = {'A1':1, 'A2':4, 'A3':8, 'A4':0, 'A5':0, 'A6':7, 'A7':5, 'A8':0, 'A9':3,

# 'B1':3, 'B2':0, 'B3':2, 'B4':0, 'B5':4, 'B6':0, 'B7':9, 'B8':0, 'B9':1,

# :

# :

# 'I1':2, 'I2':8, 'I3':4, 'I4':0, 'I5':6, 'I6':0, 'I7':0, 'I8':3, 'I9':9,}

#'0' values indicate unassigned variables and will not be printed.

s = ""

line = "-------------------------------------\n"

s += line

for row in "ABCDEFGHI":

s += "|"

for col in "123456789":

if sudokuAssign[row + col] != 0:

s += ("%3d" % sudokuAssign[row + col]) + "|"

else:

s += ("%3c" % ' ') + "|"

s += "\n" + line

print(s)

```

The testcases used by the vocareum grader were obtained in the report provided when I submitted my code for grading. The report, including the testcases, is reproduced below.

```

[Executed at: Thu Jul 13 10:58:36 PDT 2017]

003020600900305001001806400008102900700000008006708200002609500800203009005010300 [5/5]

500400070600090520003025400000000067000014800800600000005200000300007000790350000 [5/5]

000000000047056000908400061000070090409000100000009080000000007000284600302690005 [5/5]

000000000000530041600412005900000160040600002005200000000103200000005089070080006 [5/5]

090060000004000006003000942000200000086000200007081694700008000009510000050000073 [5/5]

000070360301000000042000008003006400004800002000003100005080007200760000000300856 [5/5]

000102900103097000009000070034060800000004500500021030000400000950000000000015307 [5/5]

800000090075209080040500100003080600000300070280005000000004000010027030060900020 [5/5]

000002008401006007002107903007000000065040009004000560000001000008000006910080070 [5/5]

006029000400006002090000600200005104000000080850010263000092040510000000000400800 [5/5]

000000000010720000700014826000000000006000900041906030050001000020097680000580009 [5/5]

005100026230009000000000000000900800590083000006500107060000001004000008853001600 [5/5]

680400000000710009013000000800000300000804090462009000000900037020007108000000026 [5/5]

000900007020007061300810002000078009007300020100040000000000050005000003010052078 [5/5]

000000060000130907900200031002000000004501703010006004046000020000010000200605008 [5/5]

000000000000002891080030507000000000047001085006427003000000000030005070719000204 [5/5]

010050000362010005070206400000005070005090600900000000700001008000374900601000000 [5/5]

000001086076300502000009300007000060900000800054000207008035900030900000000407000 [5/5]

307009000000003060006400001003100094025040803060300002000000006000200900580000040 [5/5]

021000050000000708000400020000600035060000000083020600059002086030001000006904200 [5/5]

```

## Week 11 Project: