CS 380 Artificial Intelligence, Fall 2021
Sect: 001/002
Homework 3 - Informed Search

by Jeffrey L Popyack, Ehsan Baradaran Khosroshahi

Due Date: Friday, November 5, 2021, 11:55 PM

NOTES:

- **Assignment Due Date extended by one day to Friday, Nov. 5**
- **Your program should be named Rubik_2x2x2.py**
- **Inadvertent reference to Depth-First Search removed (now says Best-First)**

PURPOSE:

This assignment provides experiences with Graphsearch, heuristics and A*, and also preliminary experiences with adversarial search and the Minimax algorithm.

THE ASSIGNMENT:

This assignment consists of two sections:

1. Written problems to be turned in.
2. A program to be completed.

Click here to return to top of this page.

---

1. **Written Problems to be turned in**:
   A. **Algorithm A* (8-Puzzle)** ***(20 points)***

Consider the familiar 8-Puzzle problem – a 3x3 sliding block puzzle featuring 8 numbered pieces and a blank cell is to be rearranged to place the tiles in a specified order. We have examined two heuristics (referred to below as heuristics h_1 and h_2). We introduce some other heuristics appropriate for the 8-Puzzle.

$h_0(node)$: 0 $h_1(node)$: number of tiles out of place $h_2(node)$: sum of distances out of place $h_3(node)$: 2**DT(node)* – defined below $h_4(node)$: $h_2(node)$ + 3**S(node)* – see below $h_5(node)$: $h_1(node) + h_3(node)$ $h_6(node)$: $h_2(node) + h_3(node)$ $h_7(node)$: maximum of all *admissible* heuristics in $\{h_1(node), h_2(node) \ldots h_6(node)\}$	2 3 4 8 1 6 7 5 ***Sample A***	1 2 3 8 4 7 6 5 ***Goal State***	2 1 3 8 4 7 5 6 ***Sample B***

DT(node) counts "direct tile reversals" – in which two adjacent tiles must be exchanged to be in their respective proper positions (*DT(node)=2* for sample state B, since 1-2 and 5-6 are direct reversals.)

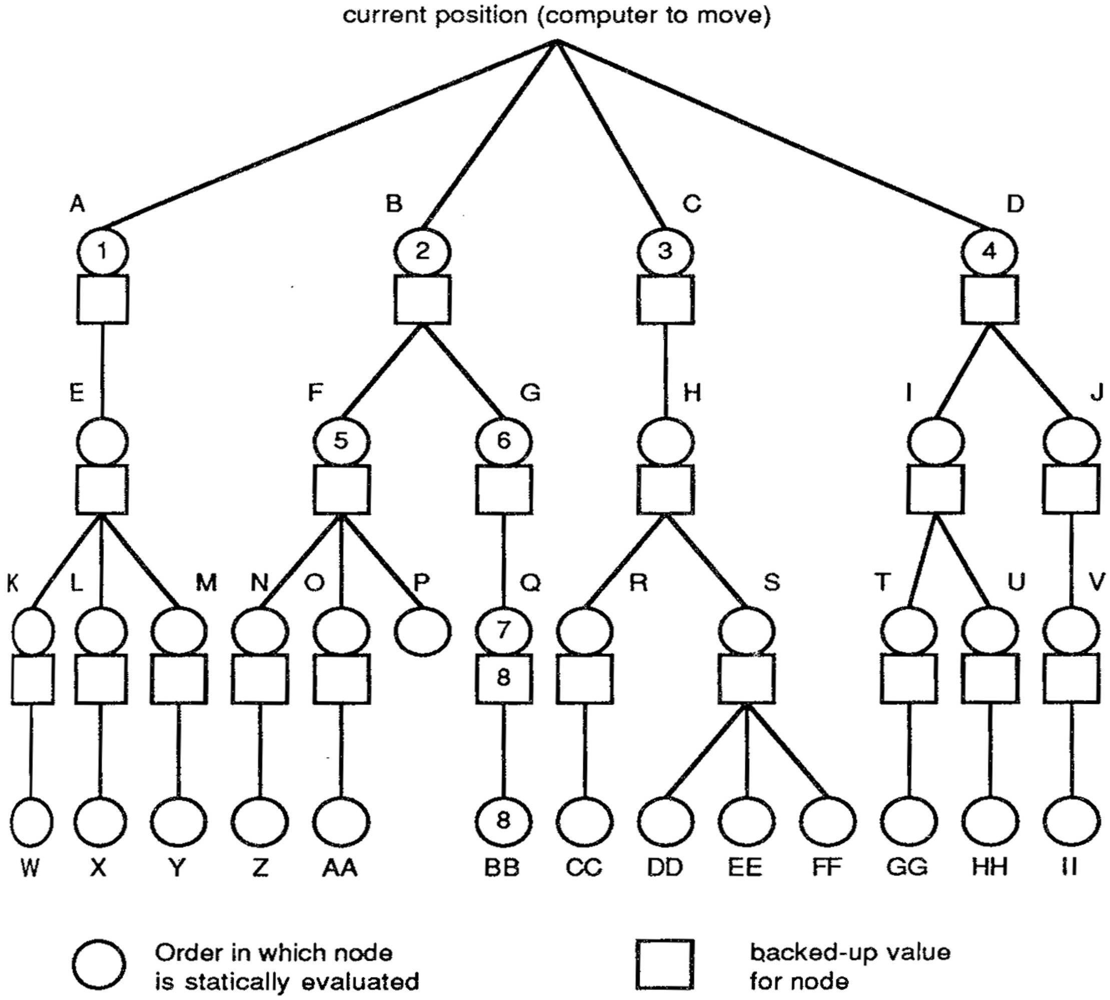
S(node) is a "sequence score" – for every tile around the edge, count 2 for every tile not followed by its proper successor and 0 for every other tile. If there is a piece in the center, add 1.

For each of these heuristics, determine whether it is *admissible*, and justify your answer.

For the *admissible* heuristics above, show the precedence from least to most informed, e.g., $h_i(node) \le h_j(node) \le \ldots$

B. **Minimax, Alpha-Beta** ***(from Tanimoto) (20 points)*** **:** The game tree in the figure below illustrates the possible moves, to a depth of 4, that can be made from the current position (at the root) in a hypothetical game between a computer and a human. The evaluation function is such that the computer seeks to maximize the score while the human seeks to minimize it. The computer has five seconds to make its move and four of these have been allocated to evaluating board positions. The order in which board positions are evaluated is determined as follows: The root is "searched." A node which is at ply 0, 1, 2, or 3 is *searched* by

- generating its children,
- statically evaluating the children,
- ordering its children by static value, in either ascending or descending order, so as to maximize the probability of alpha or beta cutoffs during the searches of successive children,
- searching the children if they are not in ply four, and
- backing up the minimum or maximum value from the children, where the value from each child is the backed-up value (if the child is not in ply four) or the static value (if the child is in ply four).


current position (computer to move)
A
1
B
2
C
3
D
4
E
F
5
G
6
H
I
J
K
L
M
N
O
P
Q
7
8
R
S
T
U
V
8
W
X
Y
Z
AA
BB
CC
DD
EE
FF
GG
HH
II
Order in which node is statically evaluated
backed-up value for node

Static values for nodes: A 4, B 15, C 13, D 10, E 20, F 9, G 8, H 10, I 10, J 8, K 5, L 20, M 3, N 7, 0 6, P 0, Q 9, R 12, S 10, T 15, U 10, V 9, W 7, X 22, Y 2, Z 7, AA 5, BB 8, CC 15, DD 12, EE 13, FF 13, GG 20, HH 22, II 18.

The computer requires 1/7 seconds to statically evaluate a node. Other times are negligible (move generation, backing up values, etc.). The computer chooses the move (out of those whose backed-up values are complete) having the highest backed-up value.

a. Give the order in which nodes will be statically evaluated. Hint: the first eight nodes have been done for you. Be sure to skip the nodes that alpha-beta pruning would determine as irrelevant. Indicate where cutoffs occur.
b. Determine the backed-up values for the relevant nodes. Node Q has been done for you.

c. Keeping in mind that it takes 1/7 seconds per static evaluation, what will be the computer's move?
d. Now assume that it takes 1/8 seconds per static evaluation. What will be the computer's move?

Click here to return to top of this page.

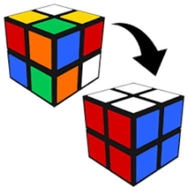
2. **Programs to be turned in**:
   A. ***GraphSearch: Rubik's Cube (60 points)***:

For this assignment, you will implement **Graphsearch**, use it to solve Rubik's Cube, and compare results with **Backtrack**, which was implemented in the previous assignment (`https://www.cs.drexel.edu/~jpopyack/Courses/AI/Fa21/assignments/HW2/index.html#Prog`).

Rubik's Cube is a classic 3-Dimensional combination puzzle created in 1974 by inventor Erno Rubik. The standard version consists of a 3×3×3 cube. See Online Rubik's Cube Simulator (`https://ruwix.com/online-puzzle-simulators/?d=3`) .

The cube looks like this:

There are six faces that can be rotated 90 degrees in either direction. The goal is to rearrange the colors to make each face monochromatic by rotating slices.



One important fact about Rubik's Cube is that it has a large **state space**, with approximately 4.3 × 10^{19} different possible configurations for a 3×3×3 cube. By comparison, the 8-puzzle state space has 362,880 different configurations of the eight tiles and blank space, which is a much smaller number. For this reason and for the sake of practicality, we tackle a 2×2×2 Rubik's Cube in this assignment which has around 3,674,160 possible states, still a large number.

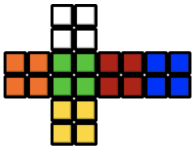
The code for this assignment should be written in Python 3 to run on tux.cs.drexel.edu.

Implementation Setup

The various parts of this assignment will require a shell script that can pass an argument to your code—thus allowing you to use **python3** while allowing us to be able to run your code with a consistent interface. However, **you may only use built-in standard libraries (e.g., math, random, etc.); you may NOT use any external libraries or packages** (if you have any questions at all about what is allowable, please email the instructor).

State Representation

For this assignment, you first need to create a representation for the state of the cube. Let's assume that we can represent the cube as a two-dimensional representation, which we can print like this:



```
   WW
   WW
OO GG RR BB
OO GG RR BB
   YY
   YY
```

Each letter represents a color, and each 2×2 square represents a face in the cube. The goal is for each face to have all its tiles of the same color.

Write a class **Cube** that takes a string representation for a cube with the **proper size**. We will assume that the input string will have the following format: (make sure to export errors in case the input string is wrong)

```
"WWWW RRRR GGGG YYYY OOOO BBBB"
```

The indexing order that maps this string to the 2-dimensional representation of the cube is as follows:

```
          0  1
          2  3
 16 17    8  9    4 5   20 21
 18 19   10 11    6 7   22 23
         12 13
         14 15
```

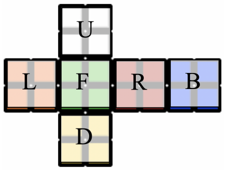
If the state representation argument is not provided here and in any of the future commands, please use the board above as the default.

Identifying Solutions

Write a method that determines whether the given cube is at a goal state. This should be very easy: if all the faces have the same color, then you have reached the solution state. Then, augment the possible commands to accept a "**goal**" command that returns "**True**" or "**False**" depending on whether the given board is at the solution state or not.

Move generation

The notation used for Rubik's Cube moves is **F**, **R**, **U**, **B**, **L**, and **D** capital letters. Each of these letters marks one 90 degree clockwise rotation of one face of the cube – in the order **F**ront, **R**ight, **U**p, **B**ack, **L**eft, and **D**own (below figure). And if the letter is followed by an apostrophe, that means a counterclockwise rotation. For a visual simulator of the moves, visit Online 2×2×2 Rubik's Cube Simulator (`https://ruwix.com/online-puzzle-simulators/2x2x2-pocket-cube-simulator.php`)



For rule (move) generation, implement a function **'applyMove'** that given a state and a move, performs the move in the state.

Important note: each move described above is actually a permutation of the indices of the colors. For example, after applying **U** the position of the colors will change as below:

```
[2,0,3,1,20,21,6,7,4,5,10,11,12,13,14,15,8,9,18,19,16,17,22,23]
```

That is, the colors originally in positions **2,0,3,1** are moved to positions **0,1,2,3**; the colors originally in positions **20,21** are moved to positions **4,5**; the colors originally in positions **6,7** remain in positions **6,7**, etc. For your convenience, we have provided a program skeleton (`https://www.cs.drexel.edu/~jpopyack/Courses/AI/Fa21/assignments/HW3/Rubik/Rubik_2x2x2_base.py`) containing the permutation for each move.

Graphsearch

Graphsearch should be written so that it uses Best-First Search with Algorithm A*, i.e., with a heuristic of the form *f(node) = depth(node) + h(node)*, where *h(node)* does not overestimate the distance to a goal. Note that using *h(node)=0* for all nodes amounts to Breadth-First Search. For Best-First Search, you will need to develop an admissible heurisitc *h(node)*. Use your insight and ingenuity for this. Note that you should ast least be able to tell the difference between a node that is potentially 1 move away from a goal, and one that is not - therefore, a very simple admissible heuristic can assign either a 1 or 2 to a state, and should prove itself to be more informed than Breadth-First Search.

For both Breadth-First and **Best-First** Search, output should print the following:

- the entire sequence of moves and resulting states to solve the changed cube,
- the number of nodes that were generated,
- the number of nodes that were expanded,
- and the time that the function took to find the solution.

You should adapt your Backtrack algorithm from the previous assignment so that it does an Iterative Deepening Backtrack. Note that like Breadth-First and Best-First Search with Algorithm A*, Iterative Deepening Backtrack should find a shortest path to the goal.

For Iterative Deepening Backtrack, output should print the following for each new value of **MAX_DEPTH**:

- the entire sequence of moves and resulting states to solve the changed cube,
- the number of calls to backTrack,
- number of failures before finding the solution.,
- the time that the function took to find the solution.

You should also print the totals for all depths tried.

The notation used for Rubik's Cube moves is **F**, **R**, **U**, **B**, **L**, and **D** capital letters. Each of these letters marks one 90 degree clockwise rotation of one face of the cube – in the order **F**ront, **R**ight, **U**p, **B**ack, **L**eft, and **D**own (below figure). And if the letter is followed by an apostrophe, that means a counterclockwise rotation. For a visual simulator of the moves, visit Online 2×2×2 Rubik's Cube Simulator (`https://ruwix.com/online-puzzle-simulators/2x2x2-pocket-cube-simulator.php`)

WHAT TO SUBMIT:

All homework for this course must be submitted electronically using Bb Learn. ***Do not e-mail your assignment to a TA or Instructor!*** If you are having difficulty with your Bb Learn account, ***you*** are responsible for resolving these problems with a TA, an Instructor, or someone from IRT, before the assignment it due. It is suggested you complete your work early so that a TA can help you if you have difficulty with this process.

For this assignment, you must submit:

- A PDF document with your answers to the "Written problems to be turned in".
- Source code, documentation and results for your program. **Your program should be named Rubik_2x2x2.py**.
- A PDF document with written documentation containing a few paragraphs
  - explaining your program,
  - analyzing the time and space complexity of algorithms and comparison (one or two paragraph is enough),
  - explaining the heuristic you used for A*,
  - results showing testing of your routines.
  - and if you did anything extra as well as anything that is not working.

Click here to return to top of this page.

ACADEMIC HONESTY:

You must compose all program and written material yourself, including answers to book questions. All material taken from outside sources must be appropriately cited. If you need assistance with this aspect of the assignment, see a consultant during consulting hours.

Click here to return to top of this page.