1 CandyGrab Continued

Today we get to bring the CandyGrab agents from lecture to life! Download and unzip:

- https://www.cs.cmu.edu/~15281/recitations/rec1/candygrab.zip

This code is meant for you to mess around with during recitation. Don't worry about breaking things, as you can always redownload the original code. Inside the CandyGrab directory you'll find:

- `candygrab.py`: The game environment
- `agent.py`: The Agent base class. All agents inherit from this simple base class.
- `agent*.py`: Implementations of various agents. We saw some of these in lecture.
- `play_one.py`: Simple script to have two agents play one round of the game
- `play_many.py`: Simple script to have two agents play many rounds of the game

Feel free to edit any of these files or make copies of `play_*.py` or `agent*.py` to create your own battles or agents. But first, we'll have you just run and inspect the code to get an idea of how things are working in this simple game. You can start by running the following:

- `python36 play_one.py`
- `python36 play_many.py`

Take a look through the source code and answer the following questions:

(a) How is the state represented in the code?

The number of remaining pieces

(b) What is Agent005's strategy?

Choose a random action

(c) What is Agent003's strategy?

Copy the opponent's action

(d) How do the AgentRL stats change when playing against an Agent008 rather than an Agent007? What difference in the two agents do you think causes this? (You should modify `play_many.py`)

With an Agent008, fewer of the table entries have N/A's. This is because Agent008 will act randomly when neither of its actions is optimal (it can't force its opponent to have mod 3 pieces). Because of this randomness, the RL agent will see more game states than it might have versus an Agent007.

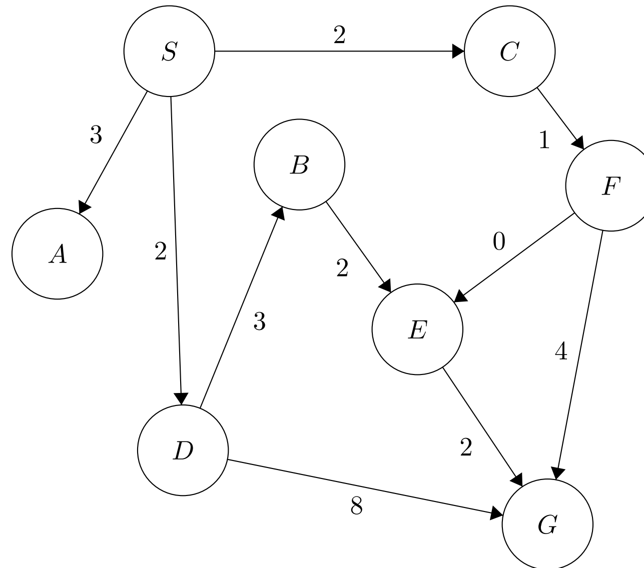
(e) When we run the games 1000 and 10000 times, how do the AgentRL stats change when playing against another AgentRL? Which states seem to be good states to take? (You should modify `play_many.py`)

The stats do not change that much since the two agents are both RL agents under explore mode (random actions), and neither agent is exploiting their table of statistics to try to win more games. For example, looking at AgentRL table of statistics, action 2 from state 5 has a much higher win probability (75%) than action 1 from state 5 (35%). However, because explore mode defaults to true, AgentRL will continue choosing random actions from state 5, rather than the more attractive action 2.

(Bonus) AgentRL has an `explore_mode` setting that defaults to `True`. What happens when you play 500 games, turn off explore mode (`bb8.explore_mode = False`), and then play 500 more games?

The `explore_mode` indicates whether or not the agent should explore - act randomly, so that it can record as many different game states into its internal table as possible, or exploit - where it uses the information it has learned to play as well as it can. Since `explore_mode` is initially true, the RL agent is only acting randomly. However, once it uses its table, it begins to do much better, and win a lot more of its games. This is a concept we'll come back to when we learn about reinforcement learning (RL) :)

2 Search Algorithms



Using each of the following graph search algorithms presented in lecture, write out the order in which nodes are added to the explored set, with start state S and goal state G. Break ties in alphabetical order. Additionally, what is the path returned by each algorithm? What is the total cost of each path?

(a) Breadth-first

Explored set: S, A, C, D, F, B
Frontier: ~~S~~, ~~S-A~~, ~~S-C~~, ~~S-D~~, ~~S-C-F~~, ~~S-D-B~~, S-D-G, S-C-F-E
Path: S-D-G
Path cost: 10

(b) Depth-first

Explored set: S, A, C, F, E
Frontier: ~~S~~, S-D, ~~S-C~~, ~~S-A~~, ~~S-C-F~~, S-C-F-G, ~~S-C-F-E~~
Note: S-C-F-E-G is never added to the frontier because G is already on the frontier.
Path: S-C-F-G
Path cost: 7

(c) Uniform cost

Explored set: S, C, D, A, F, E, B
Frontier: ~~(S, 0)~~, ~~(S-A, 3)~~, ~~(S-C, 2)~~, ~~(S-D, 2)~~, ~~(S-C-F, 3)~~, ~~(S-D-B, 5)~~, ~~(S-D-G, 10)~~, ~~(S-C-F-E, 3)~~, ~~(S-C-F-G, 7)~~, (S-C-F-E-G, 5)
Note: Horizontal strike-through means the node was replaced on the frontier. Specifically, ~~(S-D-G, 10)~~ was replaced by ~~(S-C-F-G, 7)~~, which was, in turn, replaced by (S-C-F-E-G, 5)
Path: S-C-F-E-G
Path cost: 5
Note that here we arbitrarily broke ties alphabetically by *the last state in the path* (so S-D-B was explored before S-C-F-E-G because B is before G). We also could have done so by alphabetical order of the entire path, in which case S-D-B would not be explored.

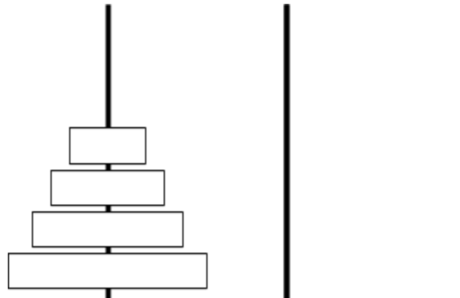
(d) Iterative deepening

depth = 0:
Explored set: S
Frontier: ~~S~~

depth = 1:
Explored set: S, A, C, D
Frontier: ~~S~~, ~~S-D~~, ~~S-C~~, ~~S-A~~

depth = 2:
Explored set: S, A, C, F, D, B
Frontier: ~~S~~, ~~S-D~~, ~~S-C~~, ~~S-A~~, ~~S-C-F~~, ~~S-D-B~~, S-D-G
Path: S-D-G
Path cost: 10

3 Tower of Hanoi



The Tower of Hanoi is a canonical puzzle studying problem solving and formulation. The puzzle starts with n disks of different sizes stacked in order of size (see picture above) on a peg, along with two empty pegs. We can move disks freely between the pegs, but larger disks cannot be stacked on top of smaller ones. The goal is to move all disks to the third peg.

We will attempt to formulate Tower of Hanoi as a search problem.

(a) How could we represent this puzzle as a problem, ie. what would the states be?

Possible answer: store 3 lists tracking the disks stacked on each peg. Enumerate the disks 1 to n.

For the following questions, assume that you have used your state representation in your answer above.

(b) What is the size of the state space in terms of n?

Assume that a disk number can appear on any of the three lists, independently of where the other numbers are. This means that there are 3^n possible combinations of the three lists. We don't need to factor in the order of those lists, because the only legal order is sorted from smallest to largest. For example, with only two disks, the number of combinations is 3^2. Specifically:

- `([1, 2], [], [])`
- `([1], [2], [])`
- `([1], [], [2])`
- `([2], [1], [])`
- `([], [1, 2], [])`
- `([], [1], [2])`
- `([2], [], [1])`
- `([], [2], [1])`
- `([], [], [1, 2])`

(c) What is the starting state?

`([1, 2, ..., n], [], [])`

(d) From some given state, what legal actions are there?

We can pop the first integer from any list and push it to the front of any other list, given that this integer is less than the current first integer of the target list (or that the target list is empty).

(e) What is the goal test? Remember that this determines whether a given state is a goal state, `goalTest(state)`.

`goalTest` should check that `state == ([], [], [1, 2, ..., n])`.