NB: This program was written with Python 3.12.2. It might not run on previous versions due to changes in Python typing and the standard library.

1. See `possible_actions` in `switch.py` for implementation. See `python main.py test1` for test coverage.

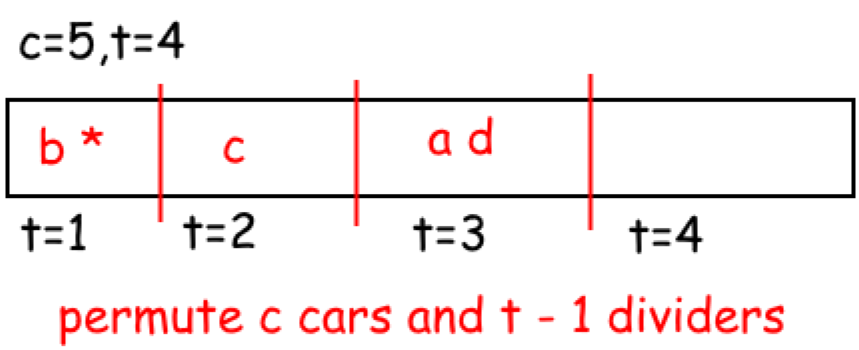
2. See `result` in `switch.py` for implementation. See `python main.py test2` for test coverage.

3. See `expand` (and similarly `expand_with_actions`) in `switch.py` for implementation. See `python main.py test3` for test coverage.

4. For my blind tree search method, I used iterative deepening. ID is complete (as `possible_actions` returns a finite list) and optimal (as this problem has a constant step cost). During ID's execution, the fringe contains only nodes from the initial state down to the node currently being searched, so it takes $O(bd)$ space, where b is the maximum number of new states from some state and d is the length of the shortest action path. Additionally, it executes in $O(b^\wedge d)$ time, proportional to BFS's execution time. For these reasons, the lectures claim "[ID] is the best uninformed method for large search spaces with uniform action cost", of which this problem fits the criteria.

5. Given c cars (including the engine) on t tracks, the size of the search space is $(c + t - 1)! / (t - 1)!$. To solve this, I used a visual combinatorics trick I learned from MATH 350. This problem is equivalent to sorting c objects into t bins, then ordering each t bins. We can represent this by placing each t bin horizontally and counting all arrangements of c objects, including the $t - 1$ walls between bins.

For example, if we have $c = 5$ and $t = 4$,



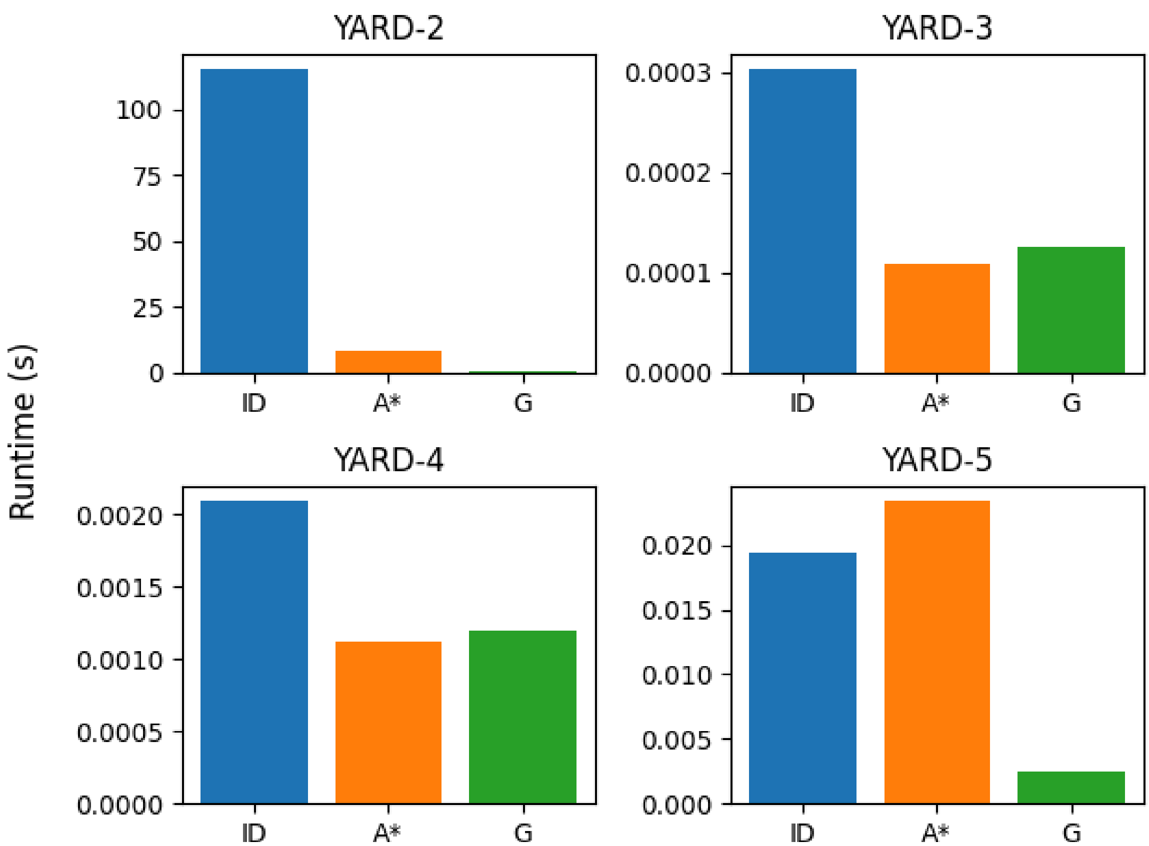
To permute $c + t - 1$ objects, we take the factorial $(c + t - 1)!$. How the $t - 1$ dividers between bins are arranged isn't important to the original problem, so we divide out from the above factorial to get our final result, $(c + t - 1)!\,/(t - 1)!$.

6. The heuristic I have chosen is number of cars take number of misplaced cars as compared to the goal state. Originally this was take number of cars on track 1, but messages in `#481_projects` on the Discord make me think this might not be the case generally. This is clearly admissible, since if n cars aren't in their final position, it will take at least n actions to correctly move them, since only one car moves at a time.

The algorithm I chose to implement this heuristic with was A*. A* is complete and, since the chosen heuristic is admissible, optimal. I note from the lectures that tree search should generally use IDA*, but I noticed it ran slower than vanilla A*. Not sure what that says about my implementations, but A* works fast enough.

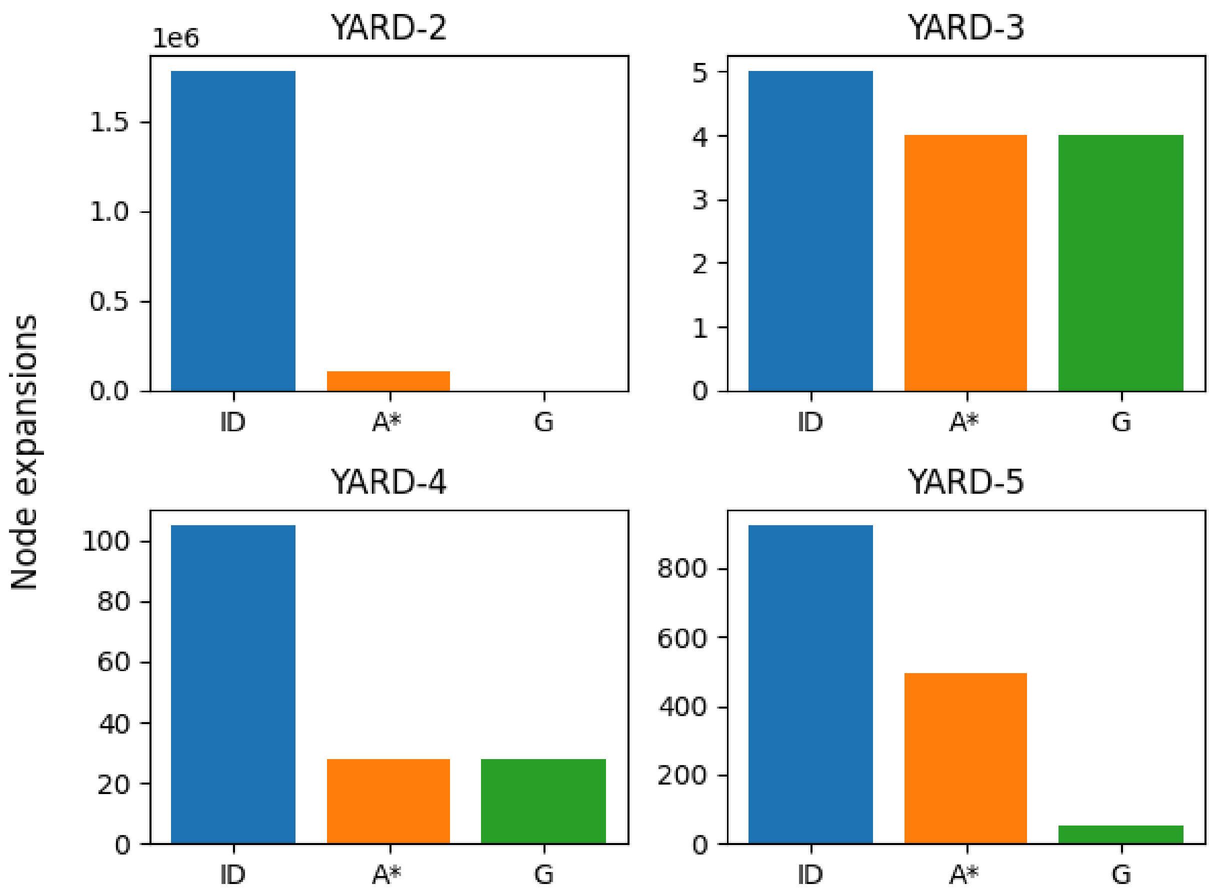
See the below writeup.

7. As expected, graph search performs way better than A*, which performs way better than ID. In fact, ID and A* can't even solve YARD-1. Below are the runtimes, taken as an average of three runs.



Extrapolating this data with an exponential curve fit, we could expect YARD-1 runtimes on A* to take 2,443 seconds, and ID to take 671,282 seconds. Graph search took 4.476 seconds.

Below are the node expansion counts.



On YARD-2, ID made 1,771,261 expansions, while graph search made just 2,987. This data parallels the runtime charts pretty well; YARD-5 seems to be in a sweet spot where it's just large enough that low-overhead algorithms like ID perform better than a more intensive algorithm like A*, but both are still leagues worse than graph search.

Below is the raw data for these visualizations.

	Runtime (s)			Node expansions		
	ID	**A***	**G**	**ID**	**A***	**G**
YARD-1	—	—	4.476050	—	—	41564
YARD-2	114.819105	8.471539	0.220113	1771261	106524	2987
YARD-3	0.000302	0.000108	0.000125	5	4	4
YARD-4	0.002096	0.001126	0.001201	105	28	28
YARD-5	0.019321	0.023390	0.002474	924	492	53

Program Usage

Basic instructions on how to use this program can be found by running `python main.py`. You can run blind tree search (ID), heuristic tree search (A*), or graph search on both the provided and custom-written yards by specifying YARD-1, YARD-2, YARD-3, YARD-4, YARD-5, or a filename after. The file should be plaintext where the first three lines are Lisp definitions of the yard, initial state, and goal state. You can check the `examples` directory for examples.