Rubik's Cube

The Rubik's Cube problem in Problem Set 6 maps all possible configurations of the Rubik's cube as vertices in a graph, and uses edges to represent valid twists of the cube. Given the node of the starting state, you can use Breadth-First Seach (BFS) to find a sequence of edges (cube twists) that will "solve" the cube, by getting into a desirable configuration.

Both configurations and moves are represented using permutations, so we start off by reviewing permutations, and then we look at the graph structure.

Permutations Review

Informally, a permutation describes an ordering of a set's elements. Formally, an N-element permutation is a one-to-one mapping of the set $1, 2 \ldots N$ to itself. A permutation f can be represented by the list of N numbers $f(1), f(2) \ldots f(N)$. Here is an example of a 5-element permutation:

$$\pi = (3\,4\,1\,5\,2)$$

Permutations are sometimes described using the explicit notation below. This notation is useful for understanding the inner workings of permutations.

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 1 & 2 & 4 \end{pmatrix}$$

The example above suggests that, in order to apply π to a 5-element list, we should first output the third element, then the fourth element, then the first element, and so on. Given the list $[a, b, c, d, e]$, we can apply π to it (permute its elements by π) and obtain $[c, d, a, e, b]$. Figure 1 illustrates the process of applying a permutation.

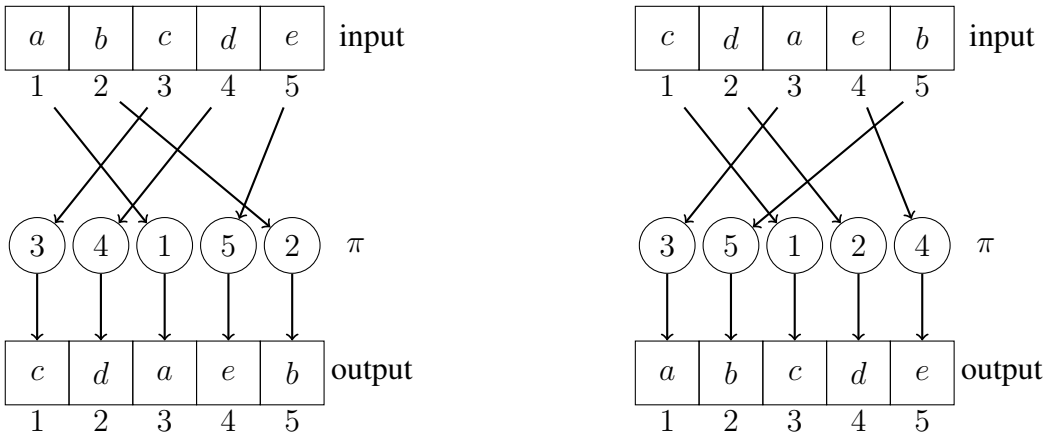

Figure 1: The result of applying π to $[a, b, c, d, e]$. Applying π^{-1} to this result produces the original list.

When programming, we usually prefer to use 0-based indexing for representing permutations, so $\pi = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 1 & 5 & 2 \end{smallmatrix}\right)$ would be represented as $[2, 3, 0, 4, 1]$ in Python.

The **inverse of a permutation** "undoes" the effects of applying the permutation to a list of elements. For example, by applying π^{-1} (the inverse of π) to $[c, d, a, e, b]$, we should obtain the original list $[a, b, c, d, e]$. Remember that we obtained $[c, d, a, e, b]$ by applying π to $[a, b, c, d, e]$.

A permutation's inverse can be computed by observing that $\pi^{-1}(\pi(i)) = i$ for $1 \leq i \leq N$. Intuitively, if π moves the third element in the input to the first position in the output, π^{-1} must take the first element in that output (which becomes its input) and move it to the third position in its own output, because π^{-1}'s output must match π's input. Figure 2 illustrates the process of computing π^{-1}.

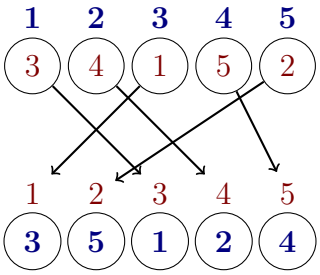

Figure 2: Computing π^{-1}. $\pi(1) = 3$, so $\pi^{-1}(3) = 1$. $\pi(2) = 5$, so $\pi^{-1}(5) = 2$. $\pi(3) = 1$, so $\pi^{-1}(1) = 3$. $\pi(4) = 2$, so $\pi^{-1}(2) = 4$. $\pi(5) = 4$, so $\pi^{-1}(4) = 5$.

The **identity permutation** I_N is a "no-op", and applying it to a list of elements yields the same list. From the definition, it follows that $I_N(i) = i$ for $1 \leq i \leq N$. For example, $I_5 = (12345)$. Applying a permutation's inverse to the permutation yields the identity permutation: $\pi \cdot \pi^{-1} = I_5$.

Cube State

A plastic 2x2 Rubik's cube is made out of 8 plastic "cubelets". Each plastic cubelet has 3 visible faces that are colored, and 3 faces that are always face the center of the big cube, so we never see them, and we ignore them from now on. Therefore, a plastic 2x2 Rubik's cube has 24 (8x3) colored plastic faces belonging to the 8 cubelets.

The code represents plastic faces using constants named as follows: `yob` is the **y**ellow face of the cubelet whose visible faces are **y**ellow, **o**range, and **b**lue. The code also numbers the 24 plastic faces from 0 to 23, and these numbers are the values of the constants named according to the convention above.

One way of representing the Rubik's cube configurations is to reflect the process of building a physical cube by pasting the 24 colored plastic faces on a wireframe of a cube. The left side of Figure 3 shows a wireframe 2x2 Rubik's cube.

The cube's wireframe has 8 cubeletes wireframes, each of which has 3 visible hollow faces where we can paste a plastic face. We refer to the wireframe faces as follows: `flu` is the **f**ront face of the **f**ront-**l**eft-**u**pper cubelet in the wireframe. Wireframe faces are also associated numbers from 0 to 23.

A cube configuration describes how plastic faces are pasted onto the wireframe cube, so it maps the 24 plastic faces 0-23 to the 24 wireframe faces 0-23. This means that a configuration is a

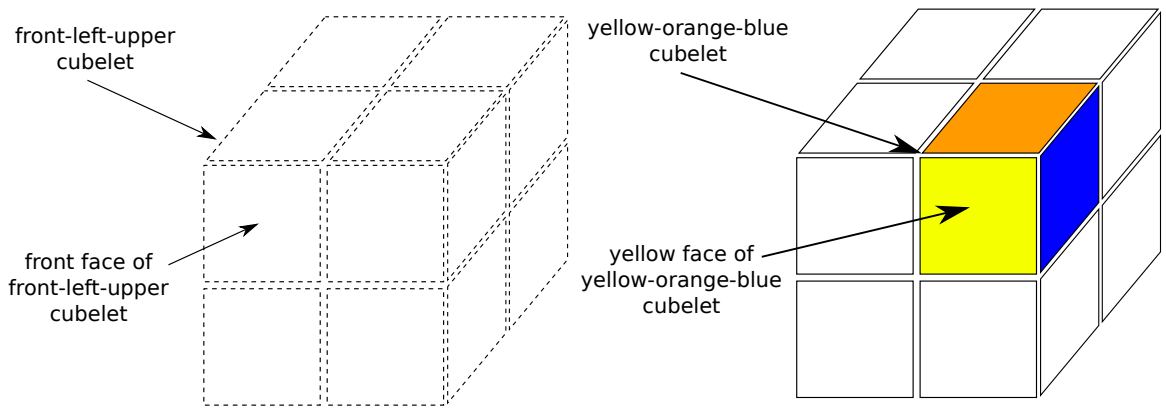

Figure 3: The wireframe Rubik's 2x2 cube is at the left, and the plastic cube is at the right. The plastic cube is made out of plastic faces, and the wireframe cube has positions where the plastic faces can be pasted.

permutation, which can be stored in a 24-element array.

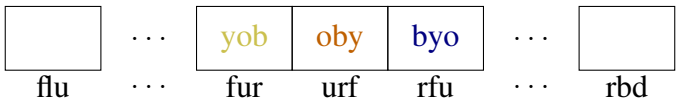

Figure 4: A configuration of the Rubik's cube is represented as a 24-element array, mapping the 24 plastic faces to the 24 wireframe faces. The configuration above has the `yob` plastic cubelet mapped to the `fru` wireframe cubelet, like in Figure 3.

Implicit Graph Representation

Given the representation above, there are $24!$ possible configurations. Some configurations are outright impossible. For example, mapping two faces of the same plastic cubelet to faces of different wireframe cubelets will clearly result in an impossible configurations, because we're not allowed to break apart the cube in order to solve it.

A graph with $24!$ configurations won't fit into a normal machine's RAM, so we can't use the straight-forward approach of generating the graph first, and then running Breadth-First Search (BFS) on it. Instead, we will code up an implicit representation of the graph, which will allow us to generate the vertices and edges that the BFS visits, on-the-fly, as we run BFS.

In order to run BFS, we need the vertices corresponding to our stating state and to the winning state, and an implementation of NEIGHBORS(V), which returns all the neighbors of a given vertex v.

Cube Twists

In order to implement NEIGHBORS(V), we need to analyze the configuration changes caused by cube twists, and code them up inside the NEIGHBORS(V) method.

Twisting a cube changes the cube's configuration, by changing the position of the plastic faces onto the wireframe. A twist moves the cube's wireframe so, for example, rotating the front face clockwise will always move the plastic face that was pasted onto `flu` (the front face of the front-left-upper cubelet) to `rfu` (the right face of the front-right-upper cubelet). Figure 5 illustrates the effects of rotating the front face clockwise.

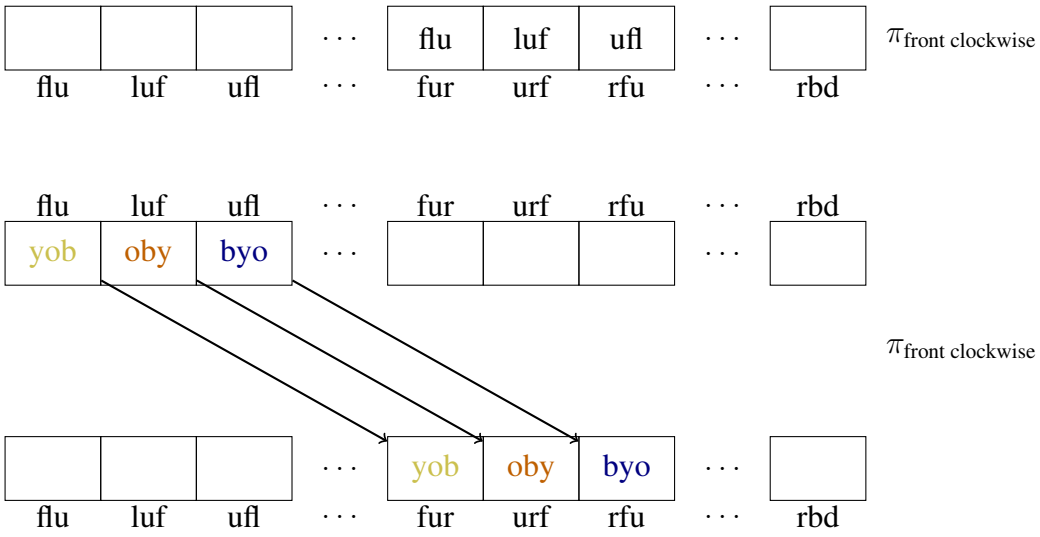

Figure 5: The configuration changes and permutation representing a clockwise twist of the cube's front face.

Due to the property above, we can represent cube twists as permutations. Applying a twist's permutation to the list describing a configuration permutation produces the list describing the new configuration permutation. So we can implement NEIGHBORS(V) by hard-coding the permutations corresponding to cube twists, and applying them to the configuration corresponding to v.

The inverse of a twist's permutation represents the move that would undo the twist.

StarCraft Zero

StarCraft is a very popular real-time-strategy (RTS) game produced by Blizzard. It has a dedicated TV channel in South Korea, and passionate players have spent countless hours modeling its rules and perfecting strategies for defeating their opponents. This problem analyzes a 6.006-exclusive version of the game called StarCraft Zero, and asks you to compute the optimal "build order" (opening strategy) for a StarCraft Zero player using the Zerg race.

Game Rules

Note: *The model below has many rules, to showcase the power of solving games by understanding the underlying configuration graphs. In the interest of time, your recitation instructor might have chosen to use a simplified model during section.*

In StarCraft Zero, Zergs have the units and buildings listed below. The costs associated with each unit and building are shown in Figure 6.

1. a **Drone** (unit) can mine minerals at the rate of 8 minerals / second, or it can mutate into a building, in 1 second
2. an **Overlord** (unit) allows you to exert control over 8 non-Overlord units; if you have O Overlords, you cannot build more than a total of $8O$ non-Overlord units (Drones and Zerglings)
3. a **Hatchery** (building) can produce units at the rate of 1 unit / second; initially, a **Hatchery** can only produce Drones and Overlords
4. a **Zergling** (unit) can attack enemy units
5. a **Spawning Pool** (building) allows all Hatcheries to produce Zerglings; once you build a Spawning Pool, you may build Zerglings from any Hatchery
6. an **Evolution Chamber** (building) allows you to reseach an **attack upgrade** and a **defense upgrade** that make all your Zerglings more powerful; once researched, each upgrade applies to all your units

StarCraft Zero time flows in discrete 1-second quantas. At the beginning of a 1-second quantum, players receive the minerals collected by drones, and issue build orders. At the end of the quantum, the build orders complete. For example, assuming a player has \$450 at the beginning of second 0, she can build a Drone starting at second 0 (minerals drop to \$400). The Drone would be ready at the end of second 0, so it would generate \$50 at the beginning of second 1 (minerals rise to \$450). The player can order it to mutate into a Hatchery at the beginning of second 1 (minerals drop to \$0),

Your control ability is limited, so you can control at most 200 units, even if you have more than 25 overlords. ($25 = \frac{200}{8}$)

At the beginning of a match, a StarCraft Zero player has 1 Hatchery, 1 Overlord, and 6 Drones.

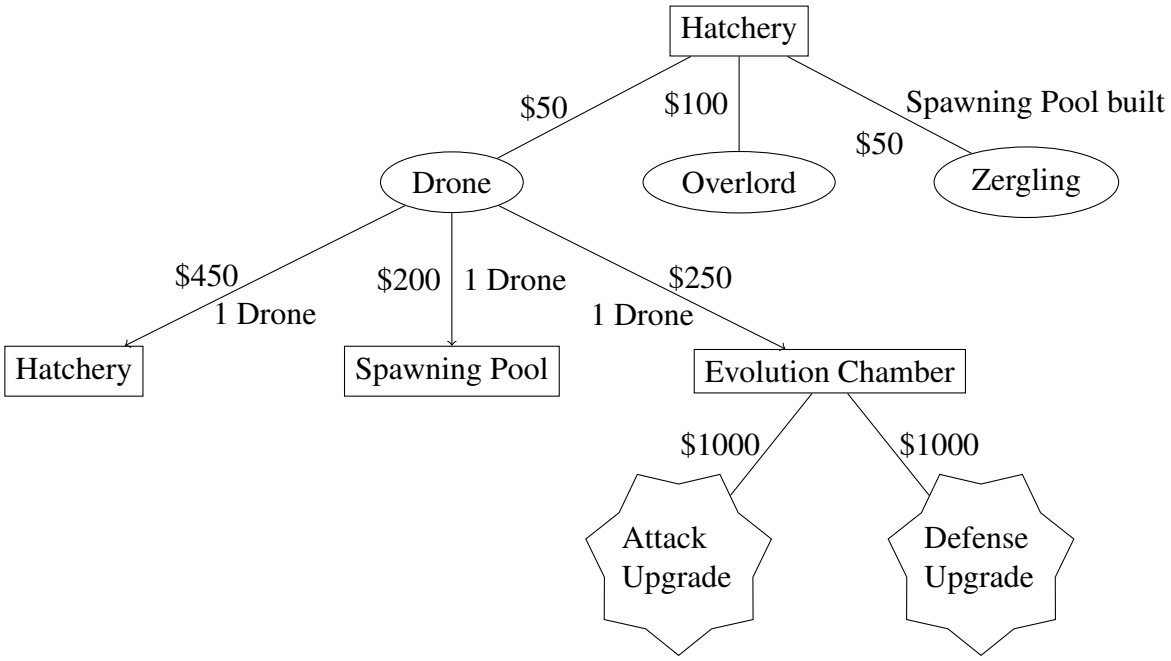

Figure 6: StarCraft Zero tech tree for the Zerg race

Zergling power	No attack upgrade	Upgraded attack
No defense upgrade	1	1.33
Upgraded defenese	1.2	2

Table 1: The effect of upgrades on Zergling power in StarCraft Zero

Table 1 shows the effect of researching upgrades on a Zergling's attack power. For example, a player would start out with Zerglings with a power of 1, then research the attack upgrade, which would give all Zerglings a power of 1.33, then research the defense upgrade, so that all the Zerglings have an attack power of 2.

Build Order Problem

The end goal of any build order is to amass a large army of powerful Zerglings as fast as possible, and use them to attack and destroy the enemy.

The main goal is to "reach the control cap" (build 200 units) as quickly as possible. Both Zerglings and Drones count for the control cap, but only the Zerglings can attack, so we want at most 50 Drones, which translates to at least 150 Zerglings, when we hit the cap. We want our army to be as powerful as possible, so the goal includes researching both upgrades.

Our strategy also has a secondary goal of keeping the enemy "in check". In order to do this, at every minute $2M$ (time $2M * 60$ seconds), we have to harass the enemy by attacking them with a group of Zerglings that have a total force of at least $\lfloor 6 \log(1 + M) \rfloor$. The Zerglings sent to harass the enemy will all die. For example, if we have no upgrades at minute 2, we have to send in 6

Zerglings. If we had researched the defense upgrade (but not the attack upgrade) by minute 2, we would have only had to send in 5 Zerglings.

Assume we have an infinite APM (actions-per-minute, a metric which counts how fast we can issue commands in the game), so we can issue as many orders as we wish simultaneously. We're also willing to make the simplifying assumption that any time we build things, our minerals will drop to 0. This means that our strategy will loop around the following steps:

- Assume we have 0 minerals.
- Wait for our Drones to collect enough minerals.
- Spend (almost) all the minerals building things.

Hint: it might help to assume at first that you can only do one action at a time, and you will wait for the action will complete. This means your strategy will loop around the following steps:

- Assume we have 0 minerals.
- Wait for our Drones to collect enough minerals.
- Spend (almost) all the minerals building one unit or building, or researching one upgrade.

Solution: Graph Model

The problem can be solved by building a graph where all the game configurations are represented as vertices, and the moves allowed in the game are represented as edges. Each edge connects the vertex of the initial starting configuration to the vertex of the configuration that results from performing the edge's move on the initial configuration. Depending on the graph structure, finding the best strategy can be achieved by a Breadth-First Search, or by a minimum-cost path algorithm (Dijkstra for non-negative edges, Bellman-Ford for the general case).

The choice of how we represent a state has a major impact on the size of the graph, which in turn impacts the running time of the solution.

Our solution represents each game state after the end of the 1-second quantum when a build order is issued. A state is a tuple (h, d, z, o, S, E, A, D), which means we have h Hatcheries, d Drones, z Zerglings, and o overlords. Additionally, we have a Spawning Pool if S is true, an Evolution Chamber if E is true, we have researched the attack upgrade if A is true, and we have researched the defense upgrade if D is true.

The initial state is $h = 1, d = 6, z = 0, o = 1$. An edge between two vertices u and v means "accumulate just enough minerals, then issue orders to get from u to v". An edge's weight is the time spent waiting for the minerals needed to issue the orders that will transition the game state from u to v.

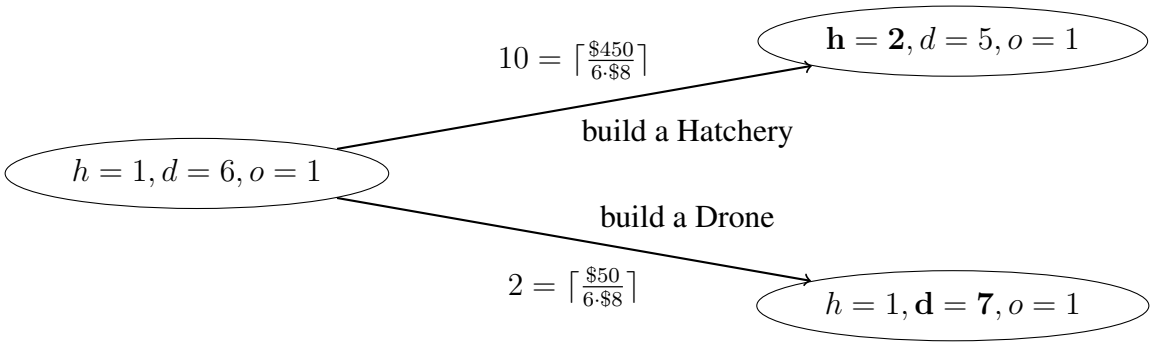

Figure 7: StarCraft Zero tech tree for the Zerg race

Solution: Modified Dijkstra

Edge weights are all non-negative, so we can run Dijkstra's algorithm to find a minimum path between the initial state and all possible states, and stop when we hit a desirable target state where $z \geq 150$ and S, E, A, D are all true.

Representing build actions as edges in the graph is reasonably straight-forward, but representing the harassment requirement is a bit tricky. Our solution takes advantage of the fact that we're operating on an implicit representation of the graph (initial vertex and NEIGHBORS(v) function), and on the inner workings of Dijkstra's algorithm.

At the time when Dijkstra visits a vertex v and calls NEIGHBORS(v), the minimum distance to v is already known. In our case, this translates to knowing the time at which we enter the game state corresponding to v. When generating edges and their destination vertices in NEIGHBORS(v), we check each edge to see whether it crosses a 2-minute boundary – we can do that, because we know the starting time, and we can compute the waiting time for the edge. If an edge does cross a 2-minute boundary, we do the following.

1. Compute the number of Zerglings required to satisfy the harassment requirement, and call it z_h. We can do that because we have encoded A and D in the state associated with each vertex, so we know what upgrades we have.

2. If $z < z_h$, we don't have enough Zerglings to harass the enemy, so we're going towards a dead end. We remove the edge from NEIGHBORS(v)'s return value, since it is not actually a valid move.

3. If $z \geq z_h$, we adjust the edge to reflect the harassment order, which will result in losing z_h Zerglings. To do this, we modify the edge's destination – if it previously pointed to a state where the player has z' zerglings, we make it point to a state that is identical in every way except the number of Zerglings is $z' - z_h$, instead of z'. This accounts for the Zerglings that are lost while harassing the enemy.

Figure 8 shows an example of processing edges that cross a 2-minute mark in NEIGHBORS.

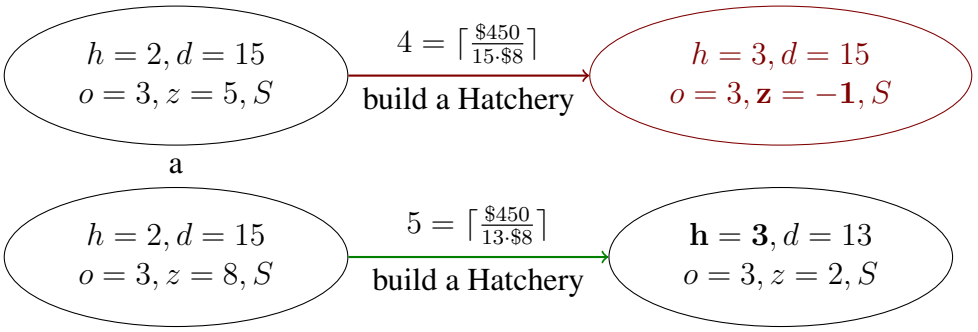

Figure 8: Filtering edges processed by NEIGHBORS to account for the harassment requirement. Both edges above cross the 2-minute mark, so the player needs to send a group of Zerglings to harass the enemy. In both cases, the player has no upgrades, so she needs to send in 6 Zerglings. The initial configuration for the top edge doesn't have enough Zerglings, so the edge is removed from the output of NEIGHBORS. The final configuration for the bottom edge is adjusted to reflect the loss of 6 Zerglings during the harassment.

Further Optimizations

The running time can be optimized by eliminating vertices that would clearly lead to a sub-optimal strategy. For example, it always makes sense to research the attack upgrade before the defense upgrade, so we can ignore all the nodes where A is false but D is true. Also, it never makes sense to build more than 25 Overlords, so we can eliminate all vertices where $o > 25$.