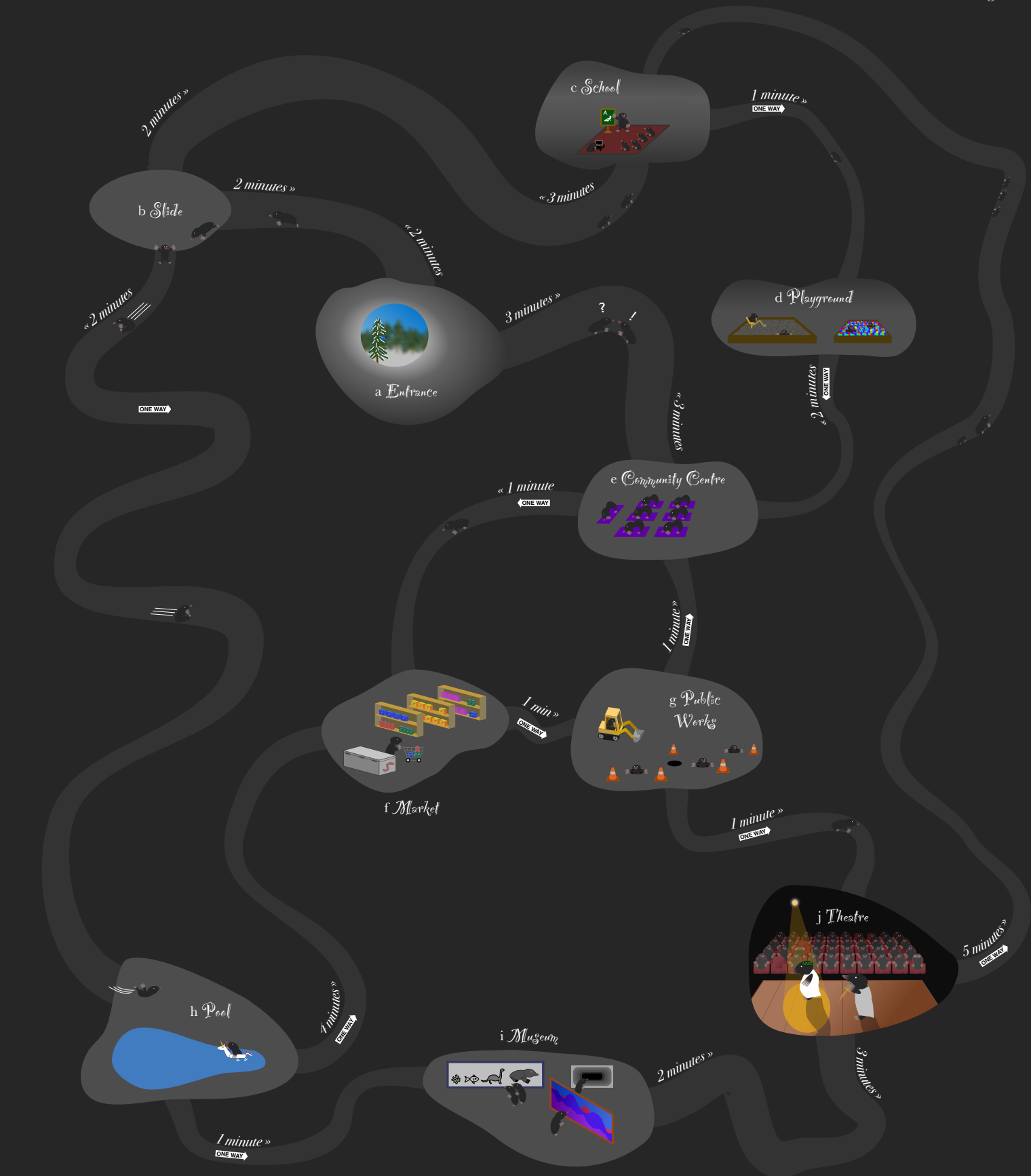
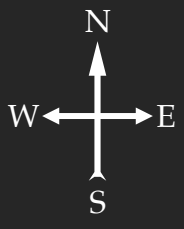


The Moleville Ministry of Tourism would like to provide an app to facilitate navigating their tunnel system. The app must find the shortest route between any two locations (a through j on the map above).

Locations connect via tunnels whose entrances are specified by compass directions. Average travel times are shown for each tunnel, and may depend on the direction taken. There are also some one-way tunnels. Travel time within a location is assumed to be negligible.



Your job is to write the back-end algorithm implementing the `MoleGuide` interface provided by the ministry. The interface files are contained in an `interface` directory in the provided project. An unfinished implementation is provided in the `work` directory for you to start with.

Do **not** modify any files in the `interface` directory.

Do **not** use any third party libraries.

Use only the programming language in which this project is written. An alternate project may be provided if the language you request is supported.

Task 1

Implement the `findRoute()` method from Listing 1. Given start and finish locations, the method returns the fastest route between them. A route contains a list of waypoints leading from one location to the next until the destination is reached.

For example, the route from *Entrance* to *Public Works* contains the following waypoints:

```
{Entrance, East}
{Community_Centre, West}
{Market, East}
{Public_Works, FINISH}
```

Note that the final waypoint always has a direction of `FINISH`.

Listing 1

```
/**
 * Finds the fastest route from the starting location to
 * the finishing location.
 *
 * @param start     The location to start from.
 * @param finish    The location to finish at.
 * @return the route found.  If multiple routes are possible
 * then no route shall be shorter than this one.  If the
 * destination is unreachable then the result shall be a
 * null-route.  The result will be overwritten on the next
 * call to findRoute().
 */
const Route& findRoute(Location start, Location finish);



// Definitions

struct Route {
    int numWaypoints;
    int travelTimeMinutes;
    Waypoint waypoints[NUM_LOCATIONS];
};

struct Waypoint {
    Location location;
    Direction direction;
};

typedef enum {
    Entrance, Slide, School, ... Theatre
} Location;

typedef enum {
    North, South, West, East, FINISH
} Direction;
```

Listing 2

```
/**
 * Reports the collapse of a tunnel.  Subsequent calls
 * to {@link #findRoute()} will avoid this tunnel.
 *
 * @param nearby    A location nearby the collapsed tunnel.
 * @param direction The direction of the collapsed tunnel from the
 *                  location.
 * @returns zero if successful, non-zero if there is no tunnel at the
 * specified location and direction, or the specified tunnel cannot be
 * entered from that direction.
 */
int reportCaveIn(Location nearby, Direction direction);

/**
 * Reports that a collapsed tunnel has been repaired.  Subsequent
 * calls to {@link #findRoute()} will no longer avoid this tunnel.
 *
 * @param nearby        A location nearby the repaired tunnel.
 * @param direction     The direction of the repaired tunnel from the
 *                      location.
 * @returns zero if successful, non-zero if there is no tunnel at the
 * specified location and direction, or the specified tunnel cannot be
 * entered from that direction.
 */
int reportCaveInCleared(Location nearby, Direction direction);
```

Task 2

Implement the maintenance methods from Listing 2. Tunnels are occasionally closed due to collapse, so the API includes methods to report closures and openings.

Closed tunnels cannot be traversed in either direction and must be bypassed by the `findRoute()` method. Note that tunnel closure may make some locations unreachable.

Task 3

In the `work` directory of the project, create a `README.md` file containing:

1. Assumptions you have made (if any).
2. A brief overview of your solution and how you verified it.
3. Explain how you verified your solution.
4. A list of any enhancements you would make if you had more time.

Task 4

Zip up your completed project and submit it to us.

Good luck, and thank you for taking the time to apply to Kardium!

Building your program

You can use whatever build system you want, but the project includes a Gradle build for your convenience. Gradle commands are invoked from the project's root directory.

To build and test the project: `gradlew build`

To clean the project: `gradlew clean`

The build places a test report in `work/build/reports`.

The build also places a console-based application in `app/build/install`. You can launch it by running the `app` script (you may have to drill down through build type and architecture subdirectories).