Software Testing 2025/2026 – Testing Evidence (LO3)

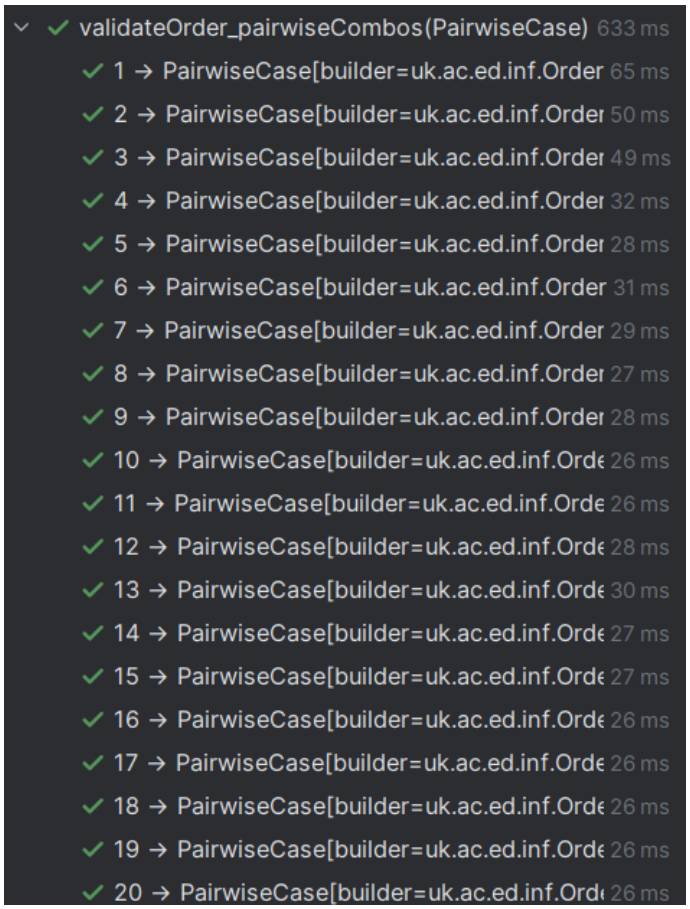
R1.1

This requirement has been tested mainly by pessimistic systematic partition testing. The input space was divided into each type of validation code, and sub-divided into their possible causes e.g. a value being too high, too low or non-numeric. Through this, each error code and its causes were tested independently. Additionally, various semantic issues were tested for, such as malformed JSON and empty inputs.

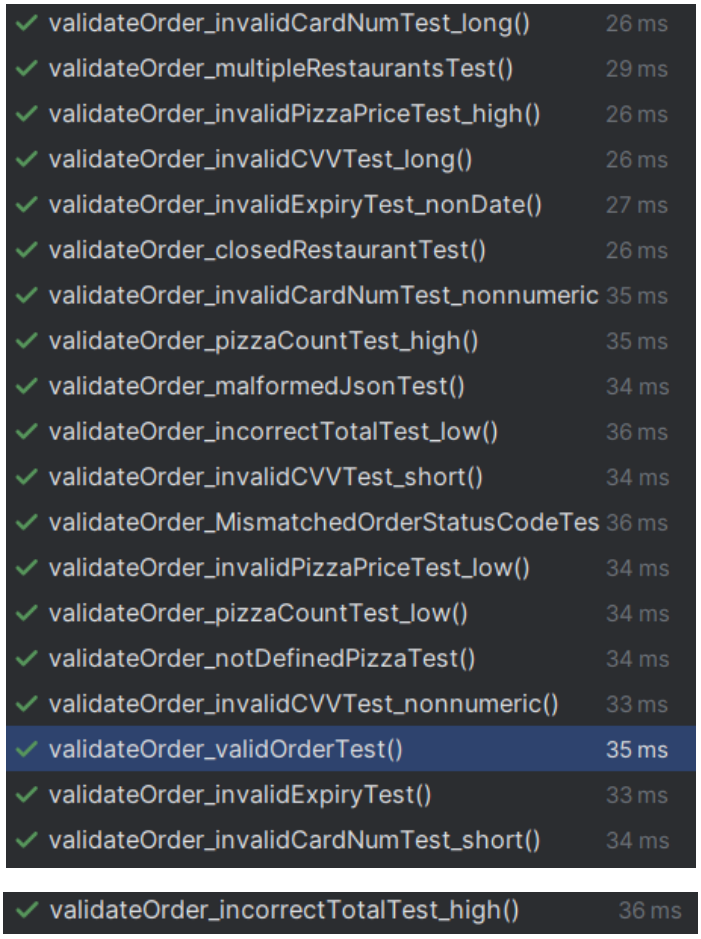
Instrumentation was added to create orders that met the criteria for each error, as well as a mock restaurant for comparison.

Pairwise combinatorial testing was carried out, combining pairs of errors (sub-divided into causes) to ensure that simultaneous errors can be handled. Instrumentation was added to reduce the pairs to the minimum number of pairwise combinations, accelerating testing.

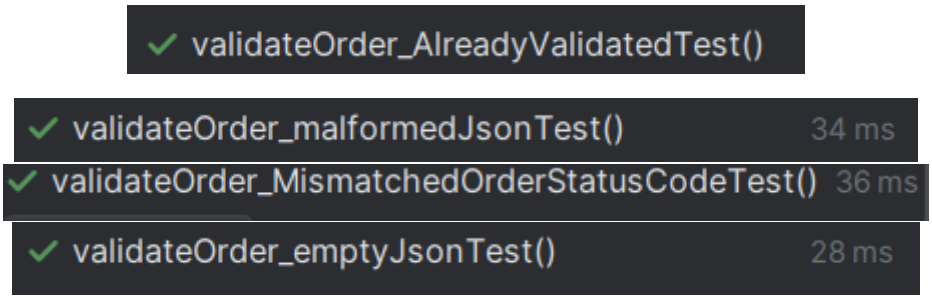
The 42 tests created for this requirement all pass and achieve a combined 100% statement and branch coverage in the validation method, ensuring that every case is covered at least once and catches the matching error.



Pairwise combinations tests



Individual error tests



Semantic error tests

Final results

```
            List<OrderValidationCode> codes = new ArrayList<>();

            LocalDate orderDate = LocalDate.parse(order.getOrderDate());
            YearMonth orderDateYM = YearMonth.from(orderDate);

            //check for all errors and add responses to list
            codes.add(order.getCreditCardInformation().validateCreditCard(orderDateYM));
            codes.add(order.validatePizzas(validator, restaurants, orderDate));

            //as only 1 error can occur, check all codes and take the 1 that is NOT undefi
            for (OrderValidationCode code : codes) {
                if (code != OrderValidationCode.UNDEFINED) {
                    result.setValidationCode(code);
                    result.setStatus(OrderStatus.INVALID);
                    break;
                }
            }

            //if no errors were found, set it to valid and no error
            if (result.getOrderValidationCode() == OrderValidationCode.UNDEFINED) {
                result.setValidationCode(OrderValidationCode.NO_ERROR);
                result.setStatus(OrderStatus.VALID);
            }
        }
    }
    return ResponseEntity.ok(result);
}
```

```
@PostMapping("/validateOrder")  shleem0 *
public ResponseEntity<OrderValidationResult> validateOrder(@RequestBody String orderStr) thr

    //initialise local variables
    Order order;
    OrderValidationResult result = new OrderValidationResult(OrderStatus.UNDEFINED, OrderVal

    //get restaurants from rest service
    StringBuilder resBody = getDataFromREST(restaurantURL);
    List<Restaurant> restaurants = mapper.readValue(resBody.toString(), new TypeReference<Li

    //check input isn't null
    if (validator.inputStringValidator(orderStr)) {

        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(null);
    } else {

        try { //read json into order class
            order = mapper.readValue(orderStr, Order.class);

        } catch (JsonProcessingException e) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(null);

        }
        //if order validation is already done, assign the existing values
        if (order.getOrderValidationCode() != OrderValidationCode.UNDEFINED && order.getOrde
            result.setStatus(order.getOrderStatus());
            result.setValidationCode(order.getOrderValidationCode());
        }
```

100% method statement and branch coverage (green)

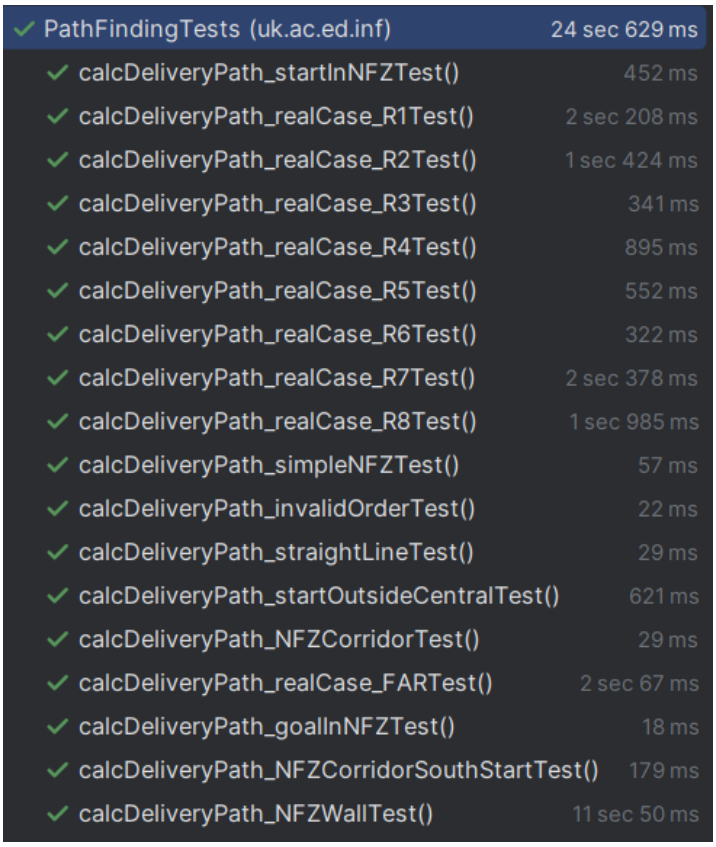
R1.2

This requirement was tested by a model-based approach. Stubs for the HTTP endpoints were used to fetch custom no-fly zones (NFZs), central areas and restaurants (for starting point) via the *TestMaps* class. This, along with path generation using the real endpoint data, thoroughly tests its integration. Integration with other system components was tested as well, with invalid orders not returning a path, showing their combined effectiveness.

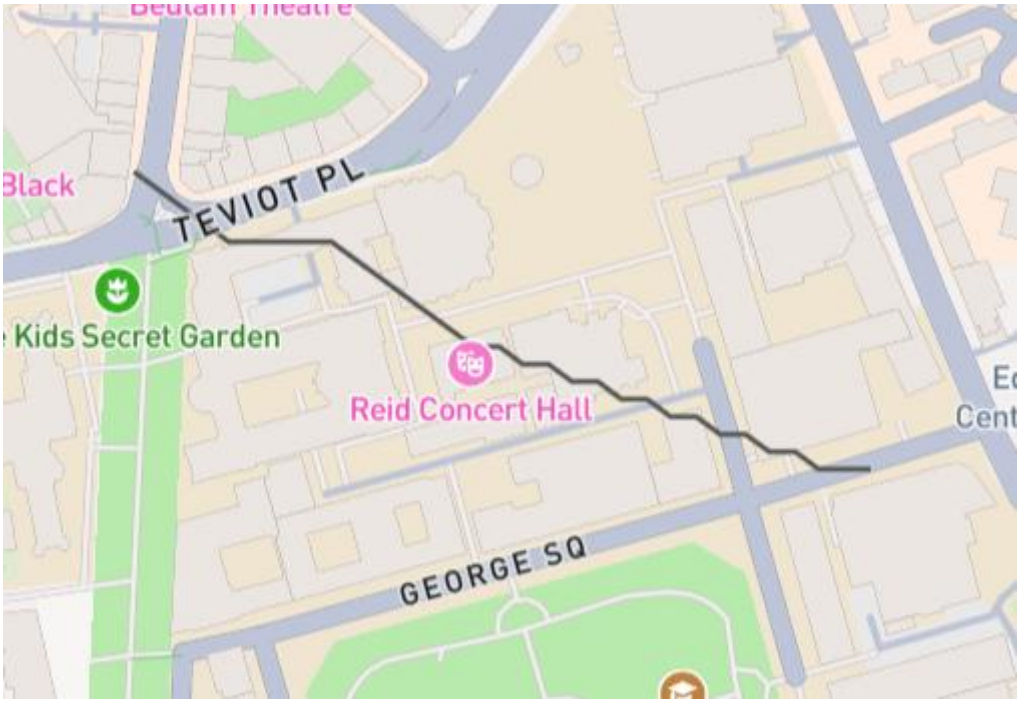
Path validity was checked by external instrumentation in the form of a routing oracle. This independently checks that all pathing constraints are followed in each test case.

The testing approach for this requirement is optimistic; partitioned cases including starting within/outside the central area, various NFZ positions & combinations and starting points were tested and all create valid paths. A small amount of negative testing, such as the start or goal points being placed in NFZs, was carried out and was achieved successfully. These cases, along with testing all real endpoint data, allows us to assume with confidence that the path generator will always produce a valid path.

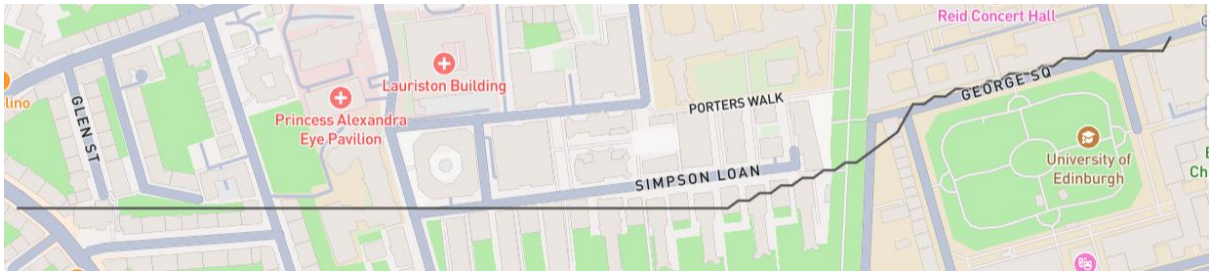
All tests pass and produce a valid GeoJSON that we can manually check for reassurance.



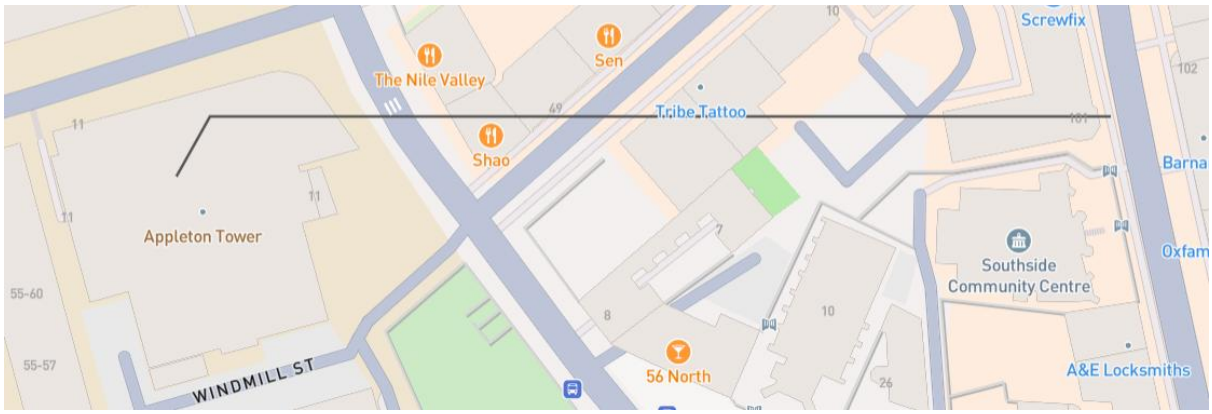
All test cases



R1 path

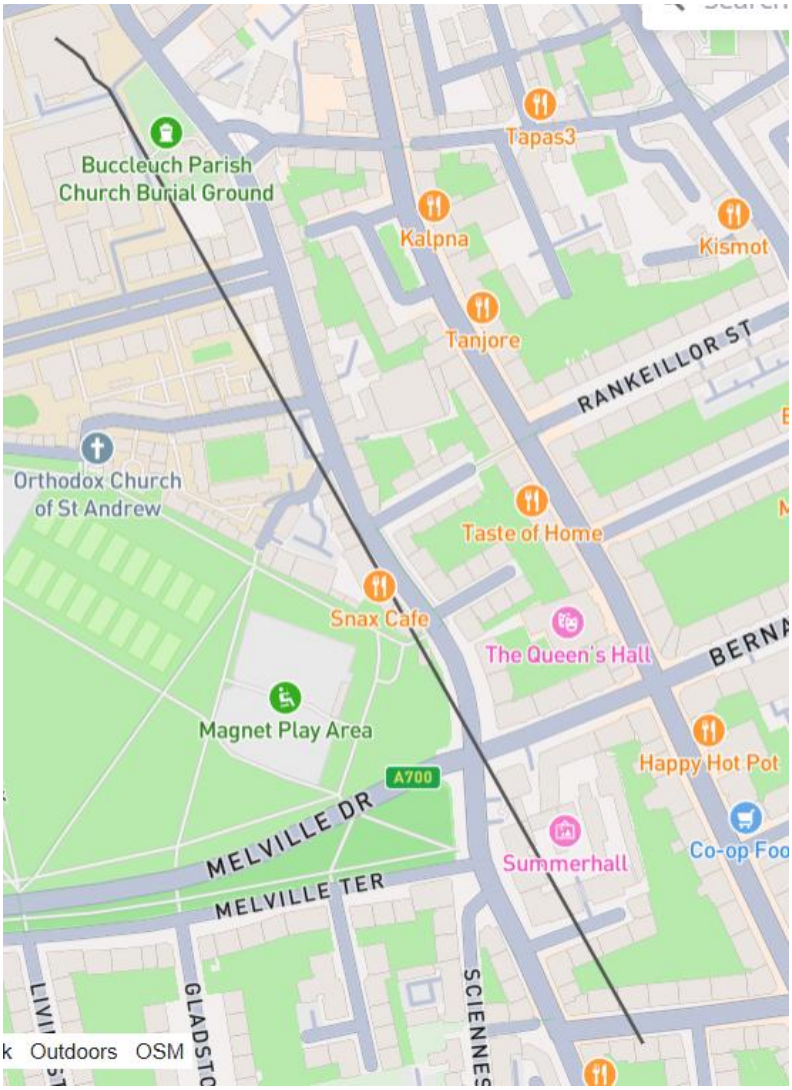


R2 path



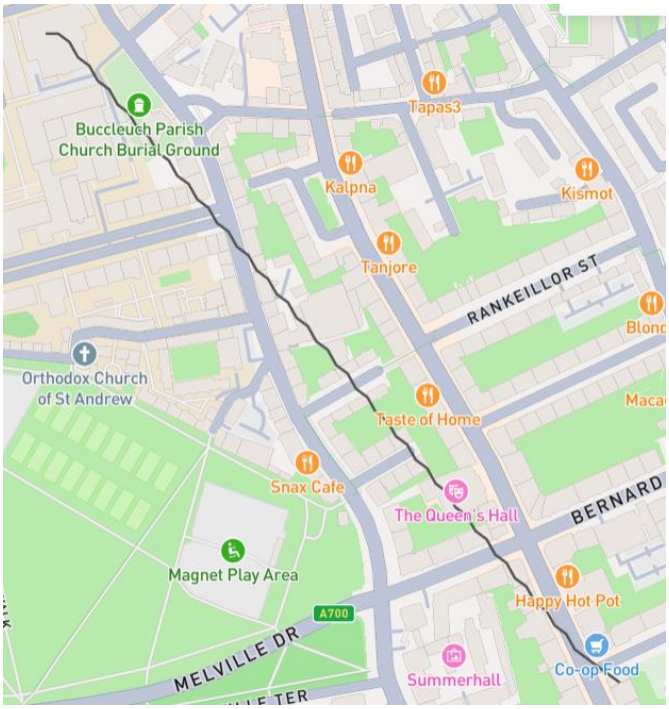
R3 path

R4 path



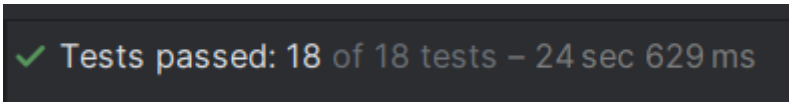
R5 path

R6 path



R7 path

R8/FAR path



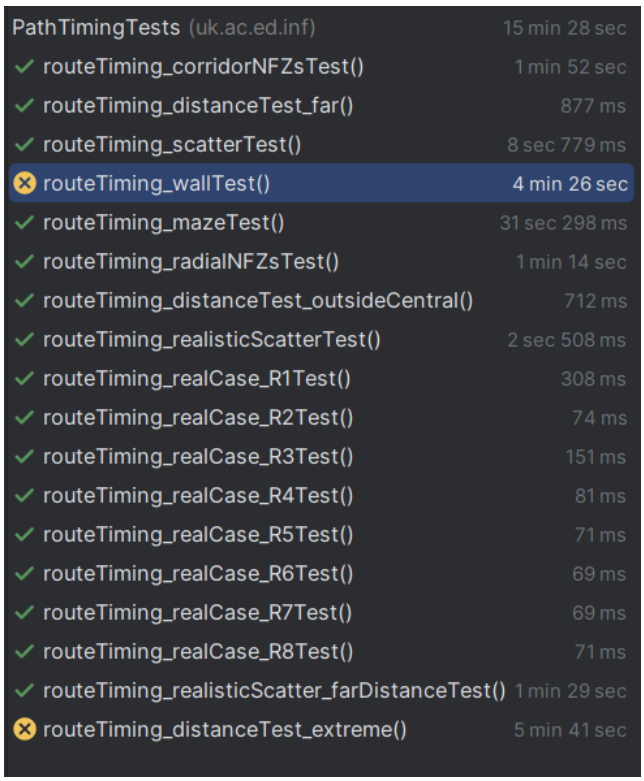
Final results

R2.1

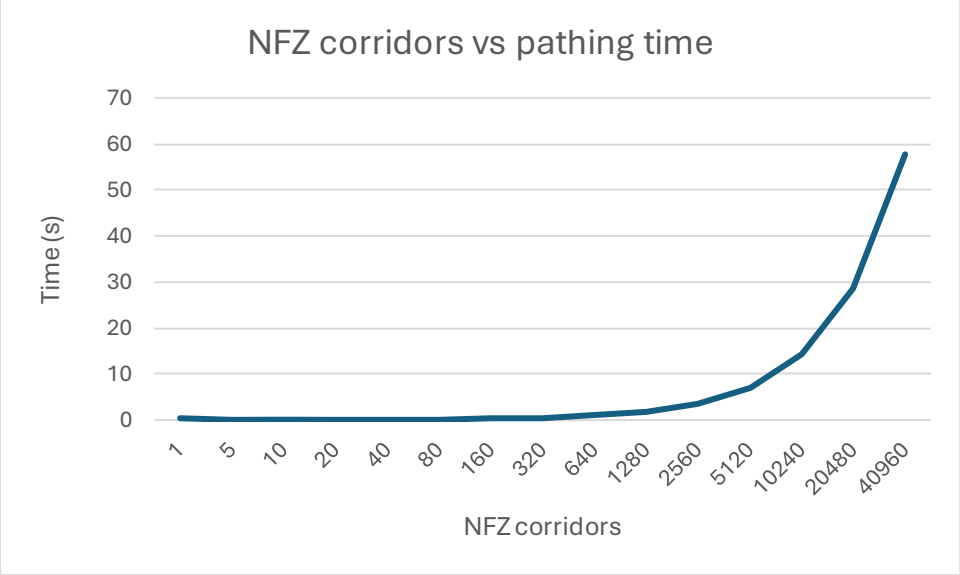
This requirement was tested via performance-based system testing. Repeated testing of various arrangements of no-fly zones and distances from the goal were tested and the time taken was recorded. The NFZs and restaurant positions are received by mock stubs for consistent performance. The instrumentation used in testing of R1.2 was used here, along with other processes to create multiple NFZs in a parameterised fashion. While times and number of NFZs were recorded, the number of nodes visited, and path length were not; this reduces the depth of analysis we can carry out using these tests.

Most realistic cases run in under 60 seconds. Some notable performance details were:

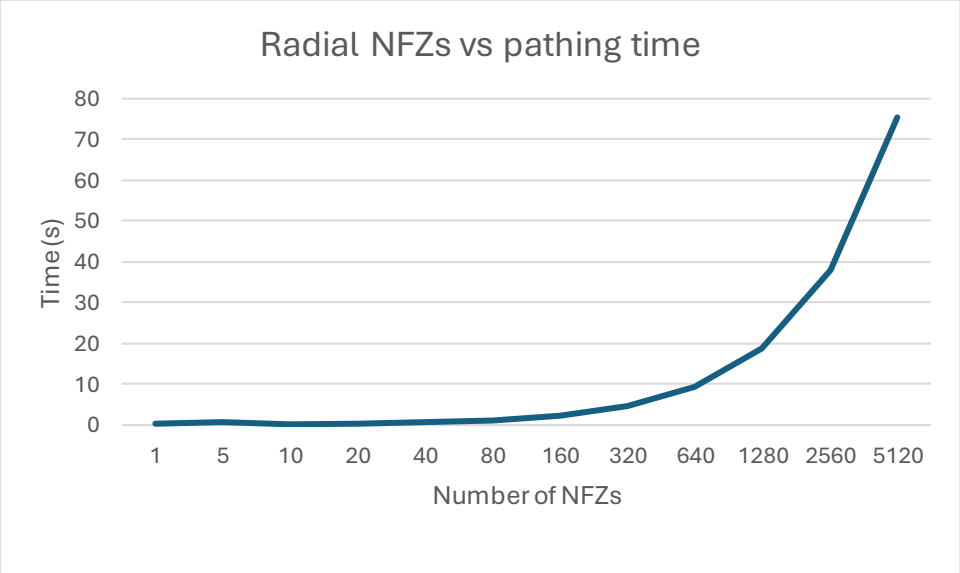
- In the NFZ corridor cases, runtime increases linearly with number of NFZs and reached approximately 60s at 40960 NFZs. As the real endpoints provide 4 NFZs, this is an acceptable boundary of ~10240%
- In the radial NFZ cases, runtime surpassed 60s at 5120 NFZs. This gives an acceptable boundary of ~1280%
- In the NFZ wall cases, runtime spikes beyond 60s at a length of 12 movement length NFZs directly between goal and endpoint. This indicates improvements required in the pathing algorithm to avoid this specific case
- Random scatters of increasing numbers of NFZs do not notably affect performance
- For a start point within Edinburgh and a small scatter of NFZs, runtime does not exceed 5s, showing the system operates well in intended cases
- Over extreme distances out of the system's scope (e.g. starting in London), runtime can take upwards of 5 minutes



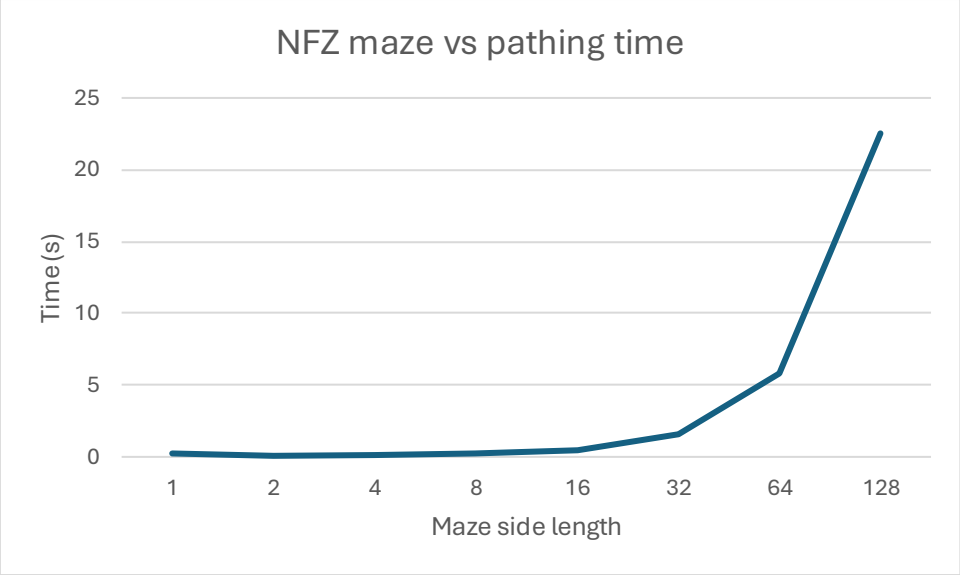
All tests



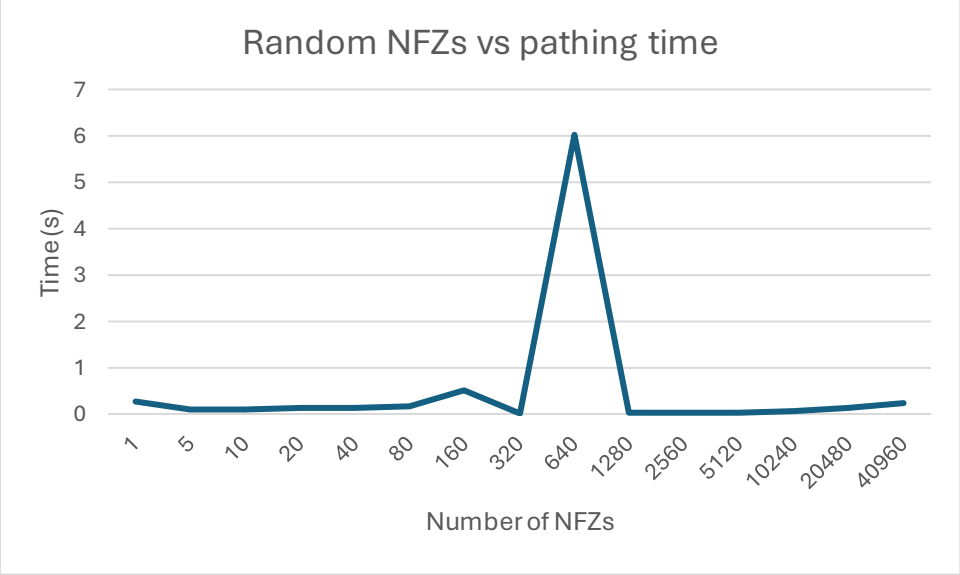
Time taken to navigate continuous NFZ "corridors"



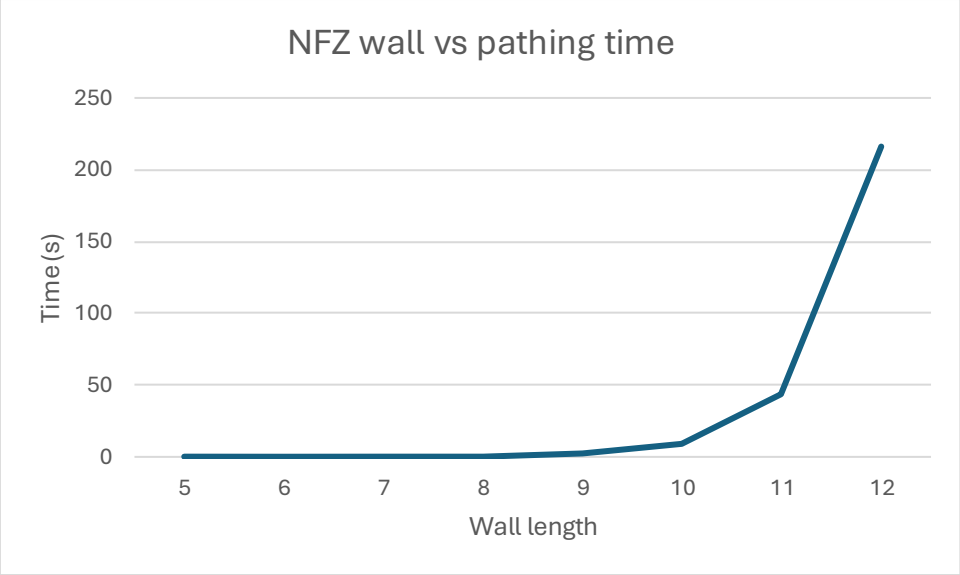
Time taken to navigate NFZs in a radius around goal



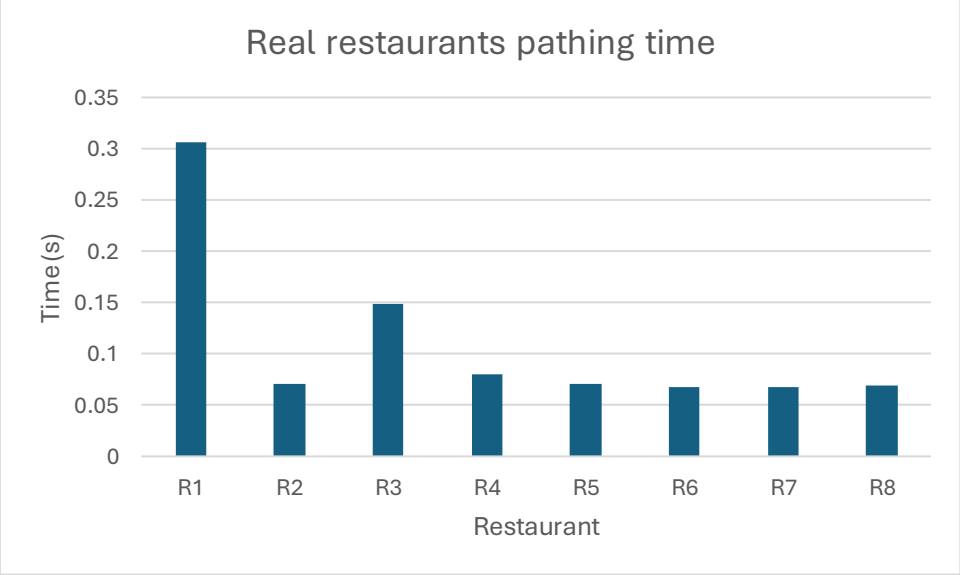
Time taken to navigate an NFZ maze



Time taken to navigate a random scatter of NFZs around goal



Time taken to navigate NFZ wall between start and goal



Time taken to reach goal using real endpoint data