Code Jam 2012 - World Finals

Analysis: Zombie Smash

Firstly, it is worth noting that the small dataset with only 8 zombies can be solved simply by evaluating each possible permutation for the order in which to smash zombies and keeping the best one. For each permutation, simply attempt to smash the zombies in the order given, skipping any zombies that cannot be reached on time. This simple approach has exponential time complexity and clearly will not scale for the large data set.

For the more efficient approach, let us start by considering the game state represented in an inefficient way and see how it can be made more efficient. We can represent the game state at any point in time with the following tuple:

- (Time, Location, Set of zombies already smashed)

This certainly works - given this state, we essentially have the snapshot of the game at any point in time, but this is very verbose. Since we want to smash as many zombies as possible, we want to smash zombies as soon as possible, i.e. we want to arrive at the grave they pop out of as soon as possible. With that in mind, we can make the following assumption: we will arrive at the grave of the next zombie as soon as possible, and potentially stand around waiting until that zombie can be smashed. Once smashed, we will move to the next grave as quickly as we can and repeat the process. That way, it becomes unnecessary to keep track of any states where we are transiting between two graves because those states can be derived from the states at the origin and destination graves. We can change the state to be:

- (Time last zombie was smashed, Last zombie smashed, Set of zombies already smashed)

This is an improvement, but now the problem is that we keep track of all possible sets of zombies already smashed. Letâ€™s see what we can do about that.

Consider the state (T_1, Z_1, {Z_1 â€¦}) where we have just smashed zombie Z_1 at time T_1. The set of zombies already smashed contains Z_1, and possibly a bunch of other zombies. Suppose that Z_0 is a zombie that we have smashed earlier. There are two cases: either Z_0 appears at an interval overlapping with Z_1, or it has already appeared before Z_1.

1. If Z_0 has already appeared before Z_1 then by T_1 it can no longer be at its grave (even if we havenâ€™t smashed it) and explicitly tracking that it has already been smashed is unnecessary.
2. Otherwise, if Z_0 appears in an interval overlapping with Z_1, is it possible that we will attempt to smash Z_0 again, if we donâ€™t keep track of it? Suppose that Z_0 was smashed at T_0, since the Zombie Smasher needs to recharge twice, Z_0 will be already gone because it stands around for 1000ms and it takes 1500ms for the Zombie Smasher to recharge twice. Again, it is not necessary to explicitly track the set of zombies smashed to avoid smashing the same zombie twice.

In light of the above observations we can simplify the state to be:

- (Time last zombie was smashed, Last zombie smashed, Number of zombies already smashed)

It is easy to see that we prefer earlier times over later times for smashing a zombie - the sooner we smash a zombie, the sooner we can move on to the next one, so we are only interested in

states with minimal time possible. Let us model the state transitions as a graph and minimize on time.

The game starts at time 0 and location (0, 0). Based on this information, we can generate the initial frontier of zombies that can be reached and smashed on time. Given this frontier, the times and the locations at which zombies will appear, we can apply a modified Dijkstraâ€™s Algorithm to find the set of game states that are reachable. Once we know those, we can simply return the maximum number of zombies smashed in a reachable state. Here is the pseudo-code:

```
solve():
  all_states = Q = generateStates()
  while Q is not empty:
    s = Q.popMin()
    if s.time = infinity:
      break;

    for each zombie z such that z ≠ s.zombie:
      earliest_arrival_time = s.time + max(750,
                                           dist(s.zombie, z))
      if earliest_arrival_time ≤ z.appearance + 1000:
        earliest_smash_time = max(z.appearance,
                                  earliest_arrival_time)
        Q.update(earliest_smash_time, z, s.smashed + 1)

  // Scan for states with time < infinity, keeping the maximum
  // number of zombies smashed to get the final answer.
  return best_reachable_state(all_states)


generateStates():
  states = {}
  states.Add(0, nil, 0) // Include the initial state.
  for each zombie z:
    for zombies_killed from 1 to Z:
      // For other reachable states this will be revised later.
      earliest_smash_time = infinity
      if zombiles_killed = 1:
        earliest_arrival_time = dist((0, 0), z)
        if earliest_arrival_time ≤ z.appearance + 1000:
          earliest_smash_time = max(z.appearance,
                                    earliest_arrival_time)
      states.Add(earliest_smash_time, z, zombies_killed)
  return states
```

Crude worst-case complexity analysis of the above: `generateStates()` will produce O(**Z**^{2}) states as each element of the pair (Last zombie smashed, Number of zombies already smashed) can vary from 0 to **Z** independently. Each state will be iterated over at most once by the outer while loop of `solve()`, and the inner for loop of `solve()` will run over all zombies, costing another O(**Z**), assuming an efficient heap is used, giving O(**Z**^{3}), which is fast enough for the large dataset where **Z** = 100. Lastly, a few contestants solved this problem with dynamic programming, keeping a 2D table with zombie index in one dimension and time since that zombie has popped up in the other dimension, maximizing on the total number of zombies smashed.