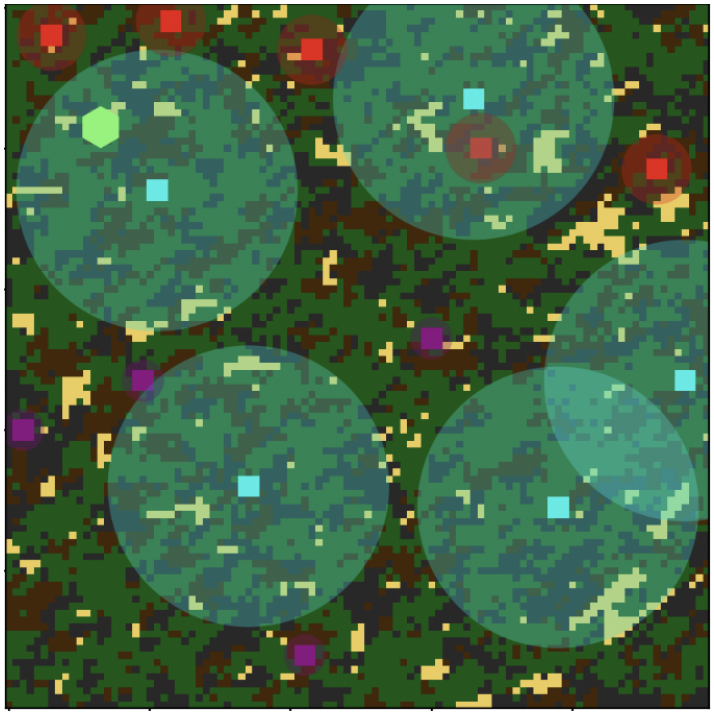
CS 262 Final Project: Adelphon

Christy Jestin, Charles Ma, Ash Ahmed

Github: https://github.com/golfcm6/adelphon
Engineering Notebook: https://github.com/golfcm6/adelphon/blob/main/notebook.md



Overview

Adelphon is a game where a group of individuals, each either a runner or relayer, navigate an unknown map filled with animals and dangerous terrain in search of treasure. In constructing the game, we leveraged a variety of distributed system concepts revolving around effective communication between individual machines (and how to handle complex situations where individuals reconcile different pieces of information, are limited by how much data they can transmit, etc.). Directions on running the game are included in the repo's README.

Game Implementation

Design

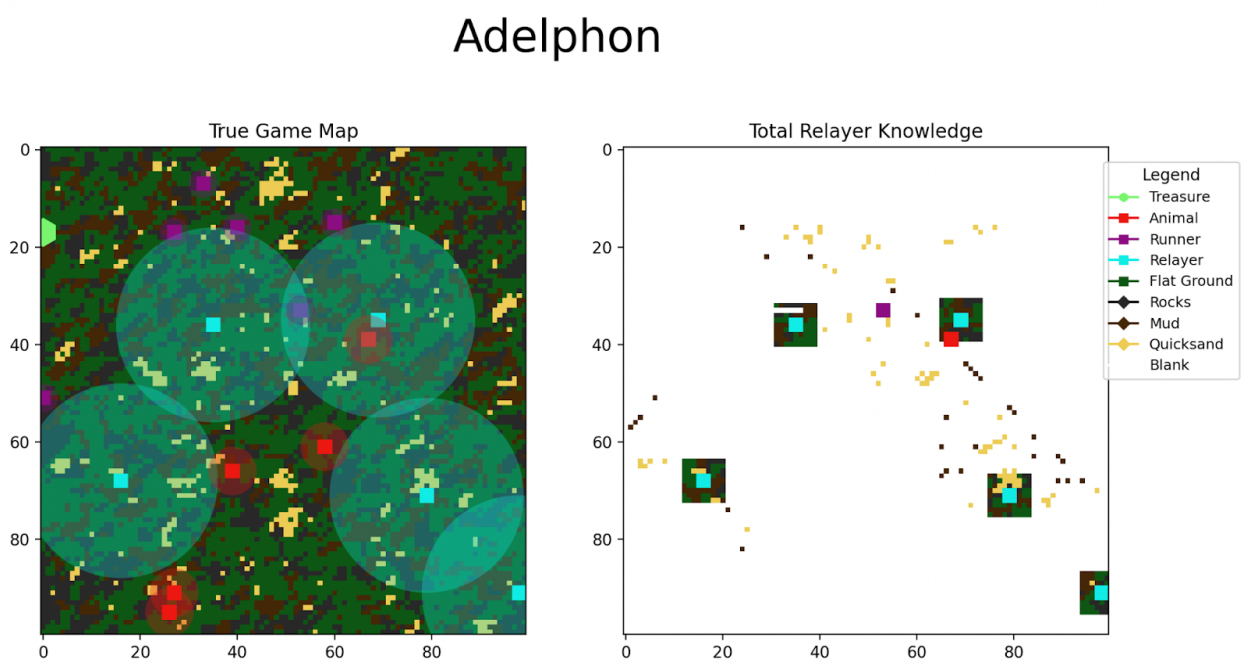
The game design for Adelphon involves two core roles: runners and relayers. Runners are the core explorers of the map, who can move based on their own decision-making and information fed to them by nearby relayers. They can be killed by animals that roam the map and slowed

down by different terrains that they encounter. Relayers cannot move (or be killed), and receive information from nearby runners, can share that information with the other relayers, and ultimately send nearby runners relevant, specific information about where to go. There are constraints on the size of messages that can be transmitted, forcing all parties to think carefully about what to send under what conditions.

The rules are as follows: some number of runners and relayers are initialized on a game map, consisting of different types of terrain, animals, and one location with the treasure. The goal of runners and relayers is to find the treasure before all runners die. The game proceeds through discrete timesteps, where at each timestep:

- Runners:
    - Make a move on the map, collecting novel information (terrain, animals, treasure found or not, alive or dead)
    - Send that information to nearby relayers, and wait for a response
- Relayers:
    - Gather information from all nearby runners, updating their individual map/information
    - Upon receiving updates from all alive runners, share with every other relayer and learn what other relayers know
    - Upon learning from all other relayers, send relevant specific information back to your nearby runners

The game ends when all runners are dead or a runner has found the treasure.



Implementation Details

We found that the cleanest way to design this game was to have 4 classes: Game, Runner, Relayer, and Visualizer. Game setup, done through *spawn.py*, involves starting one process with the visualizer thread, then a process for each of the runners and relayers (each of whom runs their own copy of the same game instance, with everyone taking in the same random seed). This avoids unnecessary sockets between one “master game instance” and each runner/relayer, while ensuring that all game information is the same for everyone.

The Visualizer class contains the logic for generating two plots: one for the “true game map,” and another containing the map that the relayers see based on the information they’ve aggregated. It accepts connections from relayers and runners to update the plots of the game as runners move and relayers acquire more info.

The Game class is initialized with a random seed and defines the map fully (types of terrain at each coordinate, initial animal locations/logic for how they move, treasure location, initial runner and relayer locations). Game contains important constants that can be customized to create different game conditions and used by different classes, including the number of different types of players, the viewing radius of different players, movement, etc. . Its primary function is *query*, called by runners at each time step; it takes in the next planned move of a runner and returns new game information for the runner’s “local view”.

The Runner class is initialized with a method *one_step* that implements the logic for the actions a runner takes at each timestep including querying the game state and moving when applicable, updating its local game instance, sending information to relayers and getting guidance in response, and finally the logic for choosing its next move given information from the relayer.

The Relayer class contains logic for receiving communication requests between runners and other relayers, as well as the strategy for advising where runners should go based on existing information. The class contains data structures to compile information sent by the runners and other relayers, as well as the strategy function *find_target* that takes in a runner’s location and outputs an ideal target coordinate for the runner to go to.

We’ve defined multiple constants and methods shared across different classes in *common.py*, including defining different communication codes, ports, map dimensions, and a default seed if none is provided.

Distributed Systems

In Adelphon, there is a “truthful” version of the game that is determined by a seed, but each runner and relayer have a different understanding of the game depending on their interactions over time. There is no single database or game instance that every player/machine refers to; instead they all have a local understanding of the game and are each fed instructions by other players/machines as the game progresses. As such, many concepts of distributed systems are critical to Adelphon:

a) using sockets and defining a wire protocol for runners and relayers to accurately communicate with each other
b) keeping each machine in sync across each timestep so that runners can communicate to the relayer new information they have acquired while relayers can aggregate new runner information to more intelligently direct runners on their next steps
c) Replication of the base game instance across each machine, while also being able to limit each machine to their personal view of the game

We'll cover how we implemented these three concepts in the following sections.

Communication and Wire Protocols

Setting up Communication

To have runners and relayers communicate with each, as well as communicating with the visualizer instance, we established a protocol for different players to both create and listen to previously created sockets depending on each player's id. There are two guiding principles: we need to generate sockets for all runners to communicate with each relayer, but not amongst each other. We also need sockets for all relayers to communicate with each internally as well as talk back with the runners.

In *common.py* you'll find constants shared across codefiles delineating the ports for the visualizer, the spawn process (this is the file we run to instantiate the visualizer and subsequent relayers and runners with the seed you've provided), and the base port to programmatically generate port numbers based off relayer ids (BASE_PORT + id = RELAYER_id_PORT).

As defined in our spawn implementation, all relayers will be instantiated before runners, and in increasing order of ids. Thus each relayer instance holds three types of socket instances: a new listening socket binded to a port facing all runners, a new listening socket binded to a different port facing all relayers *with a higher id than itself*, and a list of socket instances that are connected to sockets generated by relayers *with a lower id than itself*.

All runners thus need to only connect to a list of relayer ports that already have socket instances listening to them by the time runners are instantiated. Finally, all runners and relayers have two additional variables in their class holding a socket instance to the visualizer and the spawn process.

Wires

There are four conventions of communication: one that is shared among runner → relayer communication as well as intra-relayer communication, a second for relayer → runner communication, a third for relayer → visualizer communication, and a fourth for runner → visualizer communication.

One note on communication between relayers and runners: while they share very similar conventions, they have different constraints on the *amount* of information they can share. As defined in *common.py*, there are two constants for the amount of info a runner can share/receive over the wire, as well as one for a relayer. The runner's constant is strictly less than the relayer, but in both cases, the characters have to make prioritized decisions over what info to send at each timestep given the constraints.

1. runner → relayer, relayer ←→ relayer

CODE | ID | LOCATION | TREASURE | ANIMAL_COORDS | TERRAIN_INFO, where:

a) CODE refers to either a RELAYER_CODE or RUNNER_CODE (defined in *common.py*) that tells the recipient what type of player the sender was
b) ID refers to the players id
c) LOCATION refers to a list of player coordinates, which includes just one element when sent from runners (its own location) but could include multiple elements when sent by a relayer (tracking a set of all runner locations within its range)
d) TREASURE refers to the location of the treasure, which is an empty string if not observed or a tuple with coordinates if observed
e) ANIMAL_COORDS refers to a series of tuples of animal coordinates, with runners sending all within their view, and relayers sending all animals aggregated from runner reports in their range
f) TERRAIN_INFO refers to a series of tuples of terrain coordinates and terrain_type (x, y, terrain_type), with information similarly aggregated by relayer and independent by runner like in ANIMAL_COORDS

Units in the wire that contain a series of information (LOCATION, ANIMAL_COORDS, TERRAIN_INFO) use the delimiter "!" to separate each subunit of info. See the *prepare_info* method in *common.py* for the full implementation.

2. relayer → runner

When responding to runners, the goal of relayers is to let runners know the target coordinate that runners should proceed to based off the aggregated info of the relayers at that time step, as well as relevant terrain and animals. The wire is

ID | TREASURE | TARGET | ANIMAL_COORDS | TERRAIN_INFO, where:

a) ID refers to relayer id
b) TREASURE is the coordinates to the treasure if it is known, else an empty string
c) TARGET refers to a target coordinate calculated by relayer knowledge and a specific runner location

Cf. the previous wire for specifications on ANIMAL_COORDS and TERRAIN_INFO.

3. relayer → visualizer

The wire is as such:
CODE | ID | TREASURE | ANIMAL_COORDS | TERRA | RUNNER_LOCATIONS

a) CODE refers to the relayer's unique code
b) ID is the relayer's id
c) TREASURE, ANIMAL_COORDS, RUNNER_LOCAITONS share the same schema as implementations in previous wires
d) TERRA refers to a list of tuples containing all coordinates and terrain types the relayer is aware of at this moment

4. runner → visualizer

The wire is as such: CODE | LOCATION

a) CODE refers to the runner's unique code
b) LOCATION refers to the specific runner's location

Ending the Game and Graceful Exits

We encountered some challenges in effective communication to all participants when the game ended (either a runner has won/all runners are dead, or the user hits control C), and we used some interesting socket designs and libraries to aid with this. We decided that the protocol for who/when things were killed generally involved informing the visualizer (or the spawner for control Cs) that the game had been won or lost, and then those instances would send "game over" style messages to everyone else, who would parse them and exit themselves. We used os.kill and the subprocess library functions to actually kill child processes in some situations.

Replication

As mentioned earlier, there is no one shared game instance that the relayers are referring to. Instead, each relayer controls and updates its own local game instance based on timestep syncs. In this way, the relayers act similarly to replicated server instances/databases, where there is no one point of failure for the game state for relayers, yet at every sync, the game state that exists independently on each relayer is identical to the other game states on other relayers because they each communicate and aggregate the same information shared by runners (and all players of the game share the same initial seed).

Protocol for Responses and Miscellaneous Codes

At each timestamp, the runners make a move and initiate communication to all relayers. If they are not within range of a certain relayer, the runner is only able to send a "heartbeat" code (*TOO_FAR_AWAY* in *common.py*) just to let the relayer know it isn't within range and to document that this specific runner has made a move. We need all runners to communicate individually with all relayers because there is no other way for relayers to know when to start syncing with the other relayers unless all runners have communicated. Once runners have sent info to all relayers, they block until hearing back from the relayers.

Once relayers have received responses from all runners, they sync with the rest of the relayers, communicate their view to the visualizer, and respond to the runners, using the wires defined in the section above. Upon the completion of this cycle, the next time step occurs and runners independently make their moves.

In addition to the above description of runner ←→ relayer responses, relayers and runners both communicate with the visualizer in each timestep: relayers upon syncing with other relayers and before communicating with runners, and runners right after communicating with relayers but before hearing back from relayers. In this case, the visualizer responds with a simple code delineating the message was received.

There are miscellaneous codes used in special cases, including a code to let the spawn process know that a subprocess is up and running, and codes to delineate special games states like a runner dying or a runner finding the treasure (thus winning the game).

Challenges

Designing the game in a way that balanced coordination between runners/relayers while strategically adding challenges was very interesting. We employed tactics at the game level: varying the number of animals and their kill radius, terrain, and number of relayers/runners were obvious ways to change the difficulty of the game but didn't have much to do with distributed systems. We played around with multiple concepts like crashing relayers, making players turn malignant (send incorrect info), but ultimately decided that adding a wire throttle was most apt for our game. By restricting how much info different players could send across the wire, this was most skeuomorphic to a real life version of the game, where runners would have light gear and communication tech that wouldn't be as capable as the tech possessed by relayers who camp out at their locations.

Another challenge we encountered was getting the communication set up among relayers and runners. We struggled to determine how many sockets we'd need and how to handle a protocol for creating and connecting the sockets. The solution was actually not in the socket setup itself, but in our ordering for how different players were created. By strictly requiring that relayers are spun up in increasing id order before all runners, we could programmatically create new sockets for any relayer id > than the current relayer being set up, and confidently connected to sockets created by lower id relayers. Additionally, we realized we only needed to generate n sockets for relayer ←→ runner connections (n being the number of relayers). All runners would connect to the runner facing socket of each relayer; when those requests came in, the relayer could grab that specific connection socket and store it in a data structure, indexed by id, holding all runner facing connections.

Another challenge we faced was algorithmically determining the movements of runners and animals. We had a bug that resulted in animals consistently congregating in the top left corner of the map, but was caused by a faulty helper function that didn't include the full range of motion

when randomizing movement. More importantly, we thought a lot about how relayers could relay important information to runners after aggregating info, and how runners should intelligently act on their responses. The relayer has the advantage of having a holistic map that shows which blocks have been explored for treasure and which have not. The strategy on the relayer end then became to first always return the coordinate of the treasure if it was known, but most of the time when that wasn't the case, to increment the L-infinity norm and check for coordinates nearby to each runner that haven't been explored, returning the closest coordinate. On the runners end, we developed a simple algorithm to figure out where the recommended coordinate from the relayer was relative to its own position, and move in that direction given no harmful terrain or animals were there. Evaluating this, this was one of the weaker aspects of the project in that we could've implemented more advanced algorithms for recommending and making moves as ours were intelligent but basic. Fortunately, this wasn't core to any of the distributed systems problems we were trying to solve with our project.

One last challenge was that *spawn.py* would occasionally move too fast when creating subprocesses (runner and relayer instances), such that before connections were established for one process, it would try to connect with the next, resulting in errors. Our solution was to have the spawner wait to receive a connection heartbeat from a process (done when it is initialized by calling *alert_spawn_process()* in common) before it moves onto initializing the next process.

Testing

Testing for our project is a bit different from design projects we've done in this class. Unlike those distributed systems, where we needed to write unit tests that checked for edge cases/proper behavior involving different input types (from users, between processes, etc.), there is no "varying input" used in our game: everything involves simulating a game according to pre-defined logical workflows and rules. There is randomness for movements, map generation, etc., but the key thing to test is if our logic is being followed as expected. Thus, most testing involves running the simulation, observing the outcome, and reconciling any differences between our expectations and the results.

Another difference is that there was no focus on performance optimization like we have had in previous projects; we intentionally wanted to set limits on communication size, and restricted the level of detail in logic (e.g., simple movement logic for relayers vs Dijkstra's) to see how simulations would play out. The philosophy was very much "can you organize your thoughts effectively and make the best of limited information", and testing verified that expected behavior was carried out.

For testing our connections between different machines, we avoided building out a lot of core game behavior until we were confident in our connections/distributed systems logic. Concretely, we first verified that our relayers had the ability to connect to every runner (and vice versa) through print statements instead of actual game logic. Since our setup involved more intelligent connection mechanisms than simply creating sockets for every scenario (e.g., we have relayer syncs occur by a relayer reaching out to other relayers to send information, and receiving a

response with their current knowledge), as we added game logic we verified that the right sockets were being used in the right situations. Notably, we did not anticipate needing to create a lot of socket logic for gracefully handling different forms of the game ending (user killing through control C, game being won, or game being lost) until we observed these scenarios resulting in hanging processes/messy Connection Errors in the terminal.

On the game logic side of things, testing also involved running the game, observing if the behavior we expected actually occurred, and reconciling any differences. One notable example was that we debugged an annoying error where animals kept gravitating towards the top left corner of the map (when our logic just has them move randomly); we ran the game, observed this, and caught a missing + 1 for the second parameter in numpy random number generation (inclusive exclusive format).

Further Exploration

We'd love to explore more intelligent algorithms for runners and relayers to collaborate and more effectively find the treasure. One way to do this would be to employ Dijkstra's algorithm and place weights depending on a guiding location sent by the relayer, animal locations, and dangerous terrain. Our current logic just takes the shortest path to the location sent by the relayer but picks a random square close to it if that closest path is dangerous.

Another area we wanted to explore as we drafted our proposal but quickly realized wouldn't work out was generating "evil" players. These players could turn evil as the game progressed or start evil. As evil players, they'd either send incorrect info to relayers as runners, or direct runners to dangerous positions as relayers. There could be mechanisms for relayers and runners to detect evil players if their direct view of the game conflicted with info received, or if someone's info differed from the majority of other people's input. However, the biggest problem with this is that there wasn't a good way for players to recover a state after realizing that another player was malignant. Given we weren't saving intermediate states and extracting which states a certain player contributed to would be a nightmare, this wasn't something we considered shortly after trying to implement our project but remains a very interesting topic.

We'd also love to explore analyses on different setups of the game and understanding tendencies of where Adelphon goes. What we've created is a very interesting self contained system whose parameters can be easily altered and played around with. There could be very interesting insights if we ran analysis across large numbers of trials of different setups of the game, different seeding, etc. . We could potentially test different relayer strategies to see how to make the game more optimal and connect this system and learnings to modeling analogies real life environments.