Algorithm Project

Angry Birds

Semester: 403-404

Professor: Dr. Mir Hossein Dezfoulian

Fatemeh Saeedi

40212358020

Sahba Mirshahi

40212358040

GitHub link:

https://github.com/Farinaz-Saeedi/AngryBird

Table of Contents

Introduction

In this project, we implemented a program to guide birds from their home cities to enemy cities using the A* search algorithm . The project focuses on creating an efficient pathfinding solution while considering obstacles and enemy spies . The main goal is to ensure that the birds not only hit their targets successfully but also cause maximum damage . This project is implemented in C++ and utilizes concepts such as object-oriented programming , dynamic programming , and greedy algorithms . In the following sections , we provide details about the project structure , implementation , and workflow . We hope you enjoy reading this report!

Project Structure

Class Organization :

The project is designed using object-oriented programming principles and consists of several classes , explained as follows :

City : Represents each city on the map and serves as the parent class . Two classes Enemy (for pig cities) and Home (for bird cities) , inherit from it .

Bird : Represents each bird and stores its attributes , including damage , name , distance , and others .

Scenario : Represents a scenario , and seven scenario classes inherit from it . Each scenario has its own class and is executed based on user requests .

Controller : The most important class , where the main functions for managing the program are implemented .

Input Format :

In the following , the input files and their formats are explained .

Birds.txt : This file stores information about the birds . The user can refer to this file to become familiar with the birds and ultimately select the desired ones . The data format is as follows: Bird Name - Total Distance - Uncontrolled Distance - Spy Tolerance - Damage - Type

```
Venomtail 2500 500 3 100 A1
Skyrage 2500 500 4 25 A3
Ironbeak 2500 500 2 130 A2
Stonewing 5000 500 2 90 B1
Fury 5000 500 0 300 B2
Blaze 3000 700 2 110 C1
Viper 2900 900 1 10 C
Titan 10000 500 2 1488 D1
```

- The information of the birds

Cities.txt : In this file , the user must enter the cities along with the details .
The input format is as follows :
First , the number of cities , followed by each line containing :
- City Name
- Longitude - Latitude
- City Type (Normal/Enemy/Home)
- Presence of Spy (0/1)
- Defense Level (for enemy cities)
- Capacity (for home cities)

```
11
city0 600 0 Normal 0
city1 0 800 Home 0 3
city2 200 600 Home 1 4
city3 500 400 Home 1 3
city4 500 -200 Normal 0
city5 400 -100 Normal 1
city6 200 -50 Normal 0
city7 100 200 Normal 0
city8 -400 100 Enemy 1 4
city9 -400 -300 Enemy 1 2
city10 -200 0 Normal 0
```

- An example to write cities.txt

Scenario.txt : The user must enter the scenario number (from 1 to 7) . The program then accesses the corresponding scenario file where the scenario-specific information is entered . The input formats for each scenario are :

Scenario 1: Number of birds - Bird name (Titan)

Scenario 2: Number of birds , followed by each line :
Bird Name - Slingshot (Home) Name

Scenario 3: Number of birds type , followed by each line :
Bird Name - number of birds

Scenario 4: Number of birds , followed by each line : Bird Name - Base Name

Scenario 5: Number of nights (5) , number of birds , then for each bird :
Bird Name- Slingshot Name

Scenario 6: Number of birds , then for each bird :
Bird Name - Slingshot Name

Scenario 7: Number of nights (7) - Desired Damage - Number of birds , then for each bird : Bird Name - Cost - Slingshot Name

SpiesInScen5.txt : This file belongs to Scenario 5 , where new spies are discovered each night . The user must enter the spy information for each night in this file . The input format is as follows :

Each line contains the Night Number- Name of the Cities that have a spy

Key Functions of Each Class

Bird Class :

This class stores all the attributes of a bird , such as name , damage , distance , path , and type (A/B/C/D) . Setter and getter functions are implemented for managing these attributes . The type of birds is handled using an enum .

City Class :

This class represents a city , whether it is Normal , Enemy , or Home . The type of the city is managed using an enum . It stores the common information shared between all cities , such as the city name , coordinates (longitude and latitude) , and the city type .

Enemy Class :

This class inherits from the City class . It represents an enemy city . In addition to the common city information , it has a defense level attribute that it manages . The class also provides a pushReachBird function to add birds that have reached their destination to a particular vector , and a setBirdPath function to set the paths assigned to those birds .

Home Class :

This class also inherits from the City class and stores the capacity of each slingshot . Since this class represents a slingshot , a vector is defined to keep track of the birds in each slingshot place . push , delete , and getter functions are also implemented for it . The reduceCapacity function in this class decreases the slingshot capacity by one unit for further implementation in the desired scenarios .

Scenario Class :

This class serves as the parent class , and Scenarios 1 to 7 inherit from it . Two virtual functions , readInputs and printOutput, are implemented in this class and overridden by each scenario . The readInputs function is used to read the information for each scenario , while the printOutput function is used to print the outputs of each scenario , including damage and the path of each bird .

The **readBird** function in this class takes the name of the birds selected by the user for the scenario and , if the bird exists in the bird list file , reads its information and stores it in the corresponding vector .

Controller Class :

The most important class is the Controller , which implements the main pathfinding algorithms and serves as the core of the project . Below , we describe some of the key functions of this class :

readCities : This function is responsible for reading city information from a file , creating the appropriate objects and storing them . After reading the number of cities , it retrieves the common information for each city and depending on the city type , reads additional information (capacity for Home cities and defense level for Enemy cities). Finally , it creates the corresponding city objects .

readScenario : This function is responsible for creating a specific scenario and reading its corresponding inputs . It takes the scenario number as input , and then makes a shared_ptr of type Scenario . After checking the scenario number , the appropriate object is created , and its inputs are read . Finally , the created object is returned .

shootDownBird : This function is responsible for killing birds . First , the target enemy is identified . Then , all birds that have been detected by this enemy are collected . The birds are sorted based on their destructive power , since the enemy prioritizes destroying the most dangerous ones first . Finally , depending on the enemy's defense level , an appropriate number of birds are killed (shot down) .

run : This function is responsible for executing the entire flow of the program . First , all cities are read . Then , the desired scenario number is read , and by calling readScenario , an object of the Scenario class is created . Finally , the printOutput function of the selected scenario is executed . This displays the outputs related to the chosen scenario .

findBestPair : This function finds the best possible path for a given bird from a starting city to one of the enemy cities . For each enemy city , the path is calculated using the A* algorithm , and then the path cost and the number of spies along it are evaluated . If the current path is better than the previous ones , the path information is updated . Finally , the function returns the name of the best destination along with the result of the A* algorithm .

A - star :

Function Overview : The aStar function is an implementation of the A* pathfinding algorithm , customized for this project . Its purpose is to find the optimal path between two given cities (start and goal) for a specific bird object , while considering both geometric distances and domain-specific penalties such as the presence of spies or enemy defense systems .

Unlike the classic A* that only minimizes geometric distance , this version incorporates penalty costs , making the route not only shorter but also safer and more efficient according to the bird's capabilities .

Function Signature :

```
bool Controler::aStar (
  std::string start , // starting city name
  std::string goal , // target city name
  Bird myBird , // the bird object
  std::vector<std::shared_ptr<City>> & path , // output: final path
  ld & totalDistance , // output: real geometric distance
  ld & cost , // output: final penalized path cost )
```

Algorithm Workflow :

The algorithm begins by mapping city names to indices and checking for the trivial case where the start and goal are the same . All travel costs are initialized to infinity except for the starting city , which is set to zero . A priority queue is then used to always select the city with the lowest estimated cost , defined as the real travel cost plus the heuristic distance to the goal .

During the main loop , the city with the lowest estimated cost is extracted . If it is the goal , the path is reconstructed , the total distance is calculated , and a check ensures that the bird has enough range to complete the journey . If successful , the algorithm terminates with a valid solution . Otherwise , the algorithm explores neighboring cities : it computes the Euclidean distance for each move , applies penalties for spies and enemy defenses , and updates the cost if a better path is found . The search continues until the goal is reached . Since the heuristic is based on Euclidean distance alone , it remains admissible , while penalties are considered in the actual travel cost . This ensures that the chosen path is not just the shortest geometrically , but also the most **feasible** and **optimal** under the given constraints .

Scenario 1 :

This class overrides two functions : readInputs and printOutput . The readInputs function reads the input file for Scenario 1 , creates birds according to their type and quantity , assigns them to their respective homes , and updates each bird's home information . The printOutput function finds the best path and target for each bird using findBestPairFor , prints the bird's path , handles birds that cannot reach their target , executes attacks on enemy cities , and finally calculates and displays the total demolition caused by the surviving birds .

Scenario 2 :

This class overrides two functions : readInputs and printOutput .

readInputs : This function reads the input data for Scenario 2 from a text file . It creates Bird objects with their respective names , assigns each bird to its home city , and updates the corresponding Home objects with these birds .

printOutput : This function simulates the activities of the birds in Scenario 2 . For each bird , the best path to an enemy city is determined , it is checked whether the bird can reach the target and destroy it , and its status is updated . Then enemy attacks are executed , the total damage caused by the surviving birds is calculated , and the results are displayed in the output .

Scenario 3 :

This class uses the Hungarian algorithm to find optimal assignments and ultimately execute the actions . Below , we explain the functions and their implementation details .

OptionScen3 Structure : This structure stores a possible flight option for a bird in Scenario 3 . It includes the bird's index , its home city , the target city , the path from home to targe t, and the total cost of taking this path .

The hungarianMin function implements the Hungarian (Kuhn-Munkres) algorithm to find an optimal assignment between two sets : in this context , birds and target cities . The function takes a profit matrix as input , where each row represents a bird and each column represents a target city , and the values indicate the benefit or cost of assigning a specific bird to a specific city . The algorithm works by transforming the profit matrix into a square cost matrix and iteratively finding augmenting paths to minimize the total assignment cost . At the end , it returns a vector that maps each bird (row) to the index of its assigned city (column) , ensuring that the total cost is minimized and the assignment is optimal .

buildProfitMatrix : This function constructs the profit matrix . Each row corresponds to a bird , and each column corresponds to a target option . The matrix stores the cost of assigning each bird to each target . High default values are used for impossible assignments , ensuring that only valid bird-target pairs are considered in later optimization algorithms like the Hungarian method .

assignOptions : This function generates all possible assignments of birds to targets in Scenario 3 . Birds are sorted based on their demolition power . The function loops through homes that have available capacity , and for each bird , the path and cost (g-cost) to every enemy city are calculated using the A* algorithm . Valid options , including the bird , home , target , path , and cost are stored and the capacities of the homes are updated accordingly .

Scenario 4 :

It follows a process similar to Scenario 2 .

Scenario 5 :

The readInputs function loads scenario data from a file . It reads the number of nights and iterates through all birds , assigning each one a home city corresponding to its slingshot . Each bird is then added to its home's vector , ensuring that all homes keep track of their assigned birds . This prepares the scenario for managing bird locations and launches efficiently .

Scenario Strategy : In this scenario , to maintain continuity over five nights while maximizing destruction , a specific strategy is applied . For each night , different options are generated and stored , where each option contains the bird index , the starting city , the target city , the chosen path , and the expected damage . If an option has a spy-tolerance level lower than the number of spies encountered along its path , it means the bird will definitely reach its target and cause damage . Such guaranteed options are stored in the vector **firstOptions** , and only one of them is dispatched to an enemy city each night .

After every night, new spies are discovered, which means that reaching the target is no longer guaranteed for all remaining options . Therefore , the other options are stored in an unordered_map , where the enemy city's name is used as the key , and the value is a vector containing all options directed toward that city .

To ensure guaranteed damage on a given night, the function getWeakEnemy from the Controller class is used to identify the weakest enemy . Then , the algorithm sends a number of birds equal to that enemy's defense level plus one , ensuring destruction . Finally , if any birds remain unused , all remaining options from **allOptions** are dispatched toward enemy cities .

Scenario 6 :

The algorithm used in this class is very similar to the strategy applied in Scenario 5 , with the main difference being the absence of the allOptions vector . In this class , the giveBirdsID function assigns a unique ID to each bird , which ensures that removing birds from the vector during any night or step does not disrupt the order . Each bird's index is retrieved using its ID through the getBirdIndex function , allowing safe deletion without errors .

As long as **firstOptions** contains entries , it is used to maintain the attack sequence . Once **firstOptions** is exhausted and its birds have been deployed , birds stored in the **enemyToSecondOptions** map are sent based on the enemy's defense level plus one , ensuring continued attacks . The **night** variable is incremented after each complete attack , and at the end , the total number of consecutive attack nights along with the total damage inflicted is printed .

Scenario 7 :

This class handles multi-night attacks with a focus on reaching a target total damage while minimizing the cost of bird deployment . Birds are first loaded from a file with associated costs and home cities . Each bird is assigned a unique ID to ensure safe removal during the nightly attack sequence .

Nightly Attack Execution : For each night , all feasible attack options are generated by computing paths from home cities to enemy cities using the A* algorithm . Each option records expected damage , cost , path distance , and detection risk . The algorithm then uses a knapsack-like strategy to select the combination of birds that maximizes damage while respecting the maximum birds allowed per night . After each attack , birds used are removed safely from the active list . The process continues until all nights are completed or the target total damage is reached .

KnapsackMinCost Function : This function sorts attack options by damage-to-cost ratio and selects options greedily until the target damage or maximum bird limit is met . This ensures cost-effective damage allocation across nights .

Output and Tracking : After each night , the damage and cost of that night are reported , and the cumulative totals are updated . At the end of all nights , the total damage inflicted and total cost are printed .

Conclusion

This project successfully modeled the problem of coordinating bird attacks on enemy cities under real-world constraints such as limited flight range , defensive penalties , and the presence of spies . By integrating algorithmic approaches like A* for pathfinding and assignment strategies for resource allocation , the system was able to determine effective routes and maximize potential damage .

The main strength of the project lies in its combination of physical constraints (distance and capacity) with strategic factors (defense levels and penalties) , resulting in decisions that are both realistic and analytically useful . The modular design also enables scalability , allowing the framework to be extended with new heuristics , optimization techniques , or larger scenarios .

Nevertheless , the approach is not without limitations . The accuracy of results depends on carefully tuned heuristics and cost parameters , and very large-scale scenarios may lead to higher computational overhead .

References

- Introduction to Algorithms CLRS (3rd ed)
- https://www.tutorialspoint.com/cplusplus-program-to-implement-a-heuristic-to-find-the-vertex-cover-of-a-graph
- https://www.geeksforgeeks.org/dsa/a-search-algorithm/
- https://www.geeksforgeeks.org/dsa/hungarian-algorithm-assignment-problem-set-1-introduction/
- https://chatgpt.com/