Silent Night
-
Algorithmic Text-Based Game Engine;

A Depth-First Search (DFS) Based Adventure Simulation

BLG 223E - Data Structures
Istanbul Technical University
2024

Algorithmic Text-Based Game Engine showcasing implementation of custom data structures like stacks and doubly linked lists. Features depth-first search (DFS) for decision tree exploration and dynamic game state management. A versatile project highlighting algorithm design, data handling, and game development skills.

Note: Personal and professor-specific details have been removed to protect privacy.

Introduction

In this report, I delve into my experience developing a no-input played game using the stack and Depth-First Search (DFS) algorithm. The game challenges players to navigate through obstacles, solve puzzles, and make critical decisions affecting the game's outcome. Given the expansive nature of text-based games, the DFS algorithm was the ideal choice for its depth-oriented exploration, enabling a comprehensive examination of potential pathways to success. This document outlines the game's structure, essential data structures like Stack and GameState, the application of the DFS algorithm, and the challenges I navigated during the project's execution.

Data Structures

I relied on two key data structures: Stack and GameState. Stack, essential for navigating through the game's choices, operates on a Last In, First Out basis, ideal for tracking each decision's outcome and facilitating backtracking as needed. GameState captures the game's current conditions, like player position and inventory, allowing for dynamic responses to player actions. These structures are crucial for managing the game's complexities and enabling a depth-first exploration strategy, ensuring a seamless player experience through the game's intricate decision paths.

1. Stack Data Structure:

- Defines a stack using the DoublyList to store game states.
- Provides push, pop, and top operations to manage the stack.

2. GameState Management:

- createEmptyGameState: A function that initializes a new game state.
- copyGameState: Copies one game state to another, including resetting and duplicating the inventory and rooms.

Main Loop and DFS Logic

I integrated the Depth-First Search (DFS) algorithm to navigate through the decision-making tree of a text-based adventure game. The main loop initiates at the game's start, assessing all possible actions from the player's current state until a winning condition emerges or all paths are explored. This strategy, vital for evaluating every potential game outcome, significantly improved my navigation through the game's complexities, enabling the discovery of optimal victory paths and enhancing our narrative exploration.

Main Function:

- Initializes the game with an activeState and pushes it onto the stack.
- Uses a while loop to continue the game until a win condition is met.
- Uses nested for loops to iterate over all possible actions (i) and objects (j).
- For each action-object pair, it creates a potential new game state and applies the action to see if it leads to any changes.
- If there is a positive change, the new state is pushed onto the stack.
- If there's a losing condition (isThereAnyChange < 0), the code does nothing, effectively discarding this path.
- When a win condition is detected, the loop breaks.
- After exploring all actions for the current state, it pops the last state off the stack and continues.

Stack Implementation:

The stack in a DFS algorithm is used to store the game states that the algorithm has traversed. These states represent the different configurations of the game world at any given point, including the player's location, items in the inventory, and the status of game flags (like win or lose conditions).

HOW THE STACK OPERATES:

Push Operation:
When the DFS algorithm explores a new action, it "pushes" the resulting game state onto the stack. This action represents a decision point where the game can proceed in a particular direction based on the player's choice.

Pop Operation:
If the algorithm reaches a dead end—a state where no further progress can be made or a losing condition is encountered—it "pops" the game state off the stack. This operation effectively brings the algorithm back to the last decision point.

Top Operation:
The "top" operation allows the DFS to examine the current game state without removing it from the stack. This is useful for determining the next action to take.

Backtracking with the Stack:

Backtracking is a process where the algorithm returns to the previous state after reaching a state that doesn't lead to a solution. The stack makes this easy by keeping a history of the states:

When backtracking is needed, the stack pops off the top state, discarding the current path.
The next state down the stack is then examined for alternative actions that have not yet been tried.
This process continues until a winning condition is found or all possible states have been examined.
I use stack to keep track of game states as I simulate each possible player action in a systematic way. It is a means to dive deeper into the game's decision tree and easily backtrack to try different decisions when necessary.

It is the mechanism that enables the game to "remember" what it has already tried and what it hasn't. It stores the path taken so far, and when a path doesn't lead to success, it allows the game to go back to a previous point and try a different path. This is a key component of the DFS algorithm's ability to thoroughly explore all possible paths in the game's decision tree without getting lost or confused.

State copying: State copying is essential for several key reasons:

Exploration Without Side Effects:
Game likely has a complex set of states determined by player actions, such as moving between rooms, picking up objects, or interacting with NPCs. When the DFS algorithm explores an action, it needs to see the consequences of that action without permanently altering the game's current state. Creating a duplicate state allows the algorithm to "try out" actions. If an action doesn't lead to a desirable outcome, the algorithm can discard the duplicate and revert to the original state to try a different action.

Systematic Search Through Game States:
Project's goal is to find a sequence of actions that achieves a certain outcome, like escaping a dungeon. To systematically search for this sequence, the DFS algorithm needs to explore all possible actions from each game state. By copying states, the algorithm can explore one branch of actions (say, picking up a toilet paper), then backtrack and explore a different branch (like talking to a guard) from the exact same initial condition, ensuring that no potential solution is overlooked.

Backtracking to Previous States:
In this game with multiple decision points, the DFS algorithm needs to backtrack—that is, return to previous states—when it encounters a dead end or wants to explore a different set of actions. State

copying simplifies backtracking by allowing the algorithm to keep a stack of previous states. When it needs to backtrack, it can simply pop the last state off the stack, ensuring it returns to an exact replica of the previous decision point.

For this project specifically, the copyGameState function is designed to handle this task. When the algorithm decides to explore a particular action from a state, it:

-Creates a new, empty game state (a duplicate).
-Copies the content of the current state into this new state, ensuring that all aspects of the game's condition are replicated.
-Applies the new action to this duplicate state, leaving the original untouched.

This approach is particularly crucial for this game, where each decision can lead to significantly different outcomes. State copying ensures that my DFS algorithm can freely explore the vast landscape of potential game states without fear of losing its place or accidentally merging different paths of exploration.

GENERAL PSEUDOCODE

```
// Initialize the game
Start
  Create GameState first_state
  Call first_state.create_init_state()

  // Main loop: continues until the game is won or lost
  While (game is not won AND game is not lost):
    Call first_state.print_situation() // Print the current situation

    // Ask the player to select an action and an object
    Get action from user (1-Open, 2-Look At, 3-Pick Up, 4-Misbehave, 5-Talk To)
    Get object from user

    // Perform the selected action on the selected object
    effectID = first_state.advance(action, object)
    // effectID can be used to check if the last action changed something...

// Definitions for key functions and structures based on the game logic

Function create_init_state()
  Initialize rooms and objects according to the game's starting scenario
  Set initial room to "Cell"
  Initialize lose and win flags to false

Function advance(action, object)
  Switch (action)
    Case 1: // Open
      Try to change rooms or interact with objects that can be opened
    Case 2: // Look At
      Provide description or reveal hidden items
    Case 3: // Pick Up
      Add object to inventory if it can be picked up
    Case 4: // Misbehave
      Perform game actions that are considered misbehaving
    Case 5: // Talk To
      Interact with NPCs or objects capable of conversation
  Return effectID indicating the outcome of the action

Function print_situation()
  Print current room, visible objects, and inventory items

DoublyList and Node Structures
  Implement add, remove, and get functions to manage dynamic lists of objects and rooms

// End of pseudocode
```

ADVANCE FUNCTION

Action-Object Pairs:
Each action-object pair represents a unique decision point that the player can encounter. An action-object pair combines a potential action (such as "Pick Up", "Talk To", or "Open") with an object or character in the game environment (such as “toilet paper”, "Guard", or "Door"). This pairing delineates the specific interactions available to the player at any given moment in the game.

My program utilizes the advance function to simulate the outcomes of these interactions. When an action-object pair is selected, the advance function is called with these parameters to update the game state according to the rules and logic defined for that action and object. This function effectively models the consequences of the player's choices, altering the game state to reflect new conditions, such as changes in the player's inventory, the unlocking of a door, or the response of a character to the player's actions.

By systematically exploring these action-object pairs, the program navigates through the game's decision space. Each pair is evaluated in turn, with the advance function determining the viability and outcome of each decision. This approach allows the program to simulate the pathways leading toward the game's objectives.

```
int isThereAnyChange = potentialState.advance(i, j); // Applies the action and checks for changes in the game state.
```

```
struct GameState
{
    DoublyList<Object*> inventory;
    DoublyList<Room*> rooms;
    bool lose = 0, win = 0;
    int room_id = 0;

    void create_init_state();
    void print_situation();
    int advance(int, int);
};

int GameState::advance(int action, int object)
{
    int result = 0;

    if(action == 1)
    {
        if (object < rooms.get(room_id)->room_objects.elemcount)
        {
            if (room_id == 1 && rooms.get(room_id)->room_objects.get(
```

WINNING AND LOSING CONDITIONS:

The code navigates through the game's landscape, distinguishing between three types of actions: neutral, winning, and losing. This distinction is crucial for guiding the depth-first search (DFS) algorithm through the game's decision tree and determining the path that leads to the game's objective.

Neutral Actions: These actions result in a change in the game state that neither directly leads to a win nor results in an immediate loss. Neutral actions typically involve picking up useless items, talking to object, or engaging in non-critical interactions. When a neutral action is taken, the game state is updated to reflect the new circumstances. But the game continues without reaching a definitive conclusion. The DFS algorithm acknowledges these changes and continues to explore further actions from this new state.

```
int isThereAnyChange = potentialState.advance(i, j); // Applies the action and checks for changes in the game state.
if(isThereAnyChange == 0) {
    // No significant change occurred, continue exploring other actions.
```

Winning Actions: Winning actions are those that successfully achieve the game's final objective, such as talking to the guard for a toilet permit. When the code identifies a winning action, it triggers the game's win condition, effectively signaling the successful completion of the game. Upon recognizing a win, the DFS algorithm halts its exploration, as the primary goal has been attained. The sequence of actions leading to this winning state can then be recorded or displayed as a successful path through the game.

```
    // No significant change occurred, continue exploring other actions.
} else if(isThereAnyChange > 0) {
    cout << "Action: " << i << " Object: " << j << " has led to a change." << endl;

    // A positive change occurred, push this new state onto the stack for further exploration.
    states.push(potentialState);
    if(states.top().win) {
        // A winning state is achieved, break the loop to end the game exploration.
        break;
    }
```

```
Action: 2 Object: 1
Book: However you also have some rights! Yo
right to go to toilet! (Obtained: Toilet pe
Action: 2 Object: 1 has led to a change.
```

Losing Actions: Conversely, losing actions result in a game state that is deemed a failure, such as being killed by a guard. These actions trigger the game's lose condition. In such cases, the DFS algorithm must recognize this outcome and respond appropriately—typically by backtracking. The algorithm discards the losing state and returns to a previous state to explore alternative actions. This process involves "popping" the losing state off the stack and continuing the search from the last known good state.

```
        // If the current exploration path didn't lead to a win, backtrack by popping the last state.
        if (!states.top().win) {
            activeState = states.pop();
        } else {
            // A winning state has been found, end the exploration.
            break;
        }
    }
```

Challenges Encountered and Solutions

Challenge 1: Complexity of Game States Management
Managing the details of game states, including player location, inventory items, and interactions, was a significant challenge.

Solution: The GameState structure allowed for a detailed record-keeping of the game's current state, simplifies the handling of complex game scenarios but also ensures smooth transitions and updates to the game's state as players make different choices, thereby maintaining the integrity and continuity of the gameplay experience.

Challenge 2: Implementing an Effective Backtracking Mechanism
The nature of DFS required a reliable method for backtracking through the game's decision tree, especially when encountering dead ends or needing to explore alternate paths.

Solution: Utilizing a Stack data structure enabled the preservation and retrieval of previous game states. This approach facilitated a systematic exploration and testing of different game actions,

ensuring no potential solution was overlooked.

```
// If the current exploration path didn't lead to a win, backtrack by popping the last state.
if (!states.top().win) {
    activeState = states.pop();
} else {
    // A winning state has been found, end the exploration.
    break;
}
```

Challenge 3: Performance Optimization
The extensive game tree exploration demanded by the DFS algorithm posed significant performance challenges, particularly in complex game scenarios.

Solution: Optimizations in the DoublyList data structure enhanced overall performance, making the exploration process more efficient.

```
// I Defined a generic Stack data structure template using a doubly linked list for data storage.
template <typename T>
struct Stack {
    private:
        DoublyList<T> data; // I used Doubly linked list to store stack elements.
    public:
        void push(T& e); // Pushes an element to the top of the stack.
        T pop(); // Removes and returns the top element from the stack.
        T& top(); // Returns a reference to the top element without removing it.
        int elemcount = 0; // Keeps track of the number of elements in the stack.
};
```

Observations and Results

In this project, the implementation of the Depth-First Search (DFS) algorithm has been instrumental in navigating the complex decision trees of a text-based adventure game. Throughout the development and testing phases, several key observations were made**:**

Performance Efficiency: The DFS algorithm efficiently managed the game's expansive decision space, although scenarios with numerous branching paths It result win state within a second after a run. The algorithm's ability to prioritize depth over breadth ensured that all possible outcomes were explored systematically.

Decision Impact: Certain actions within the game significantly influenced the exploration path. For instance, selecting specific items or engaging in conversations with NPCs often led to branching paths that could either advance the game state towards victory or result in a dead-end.

Challenges Overcome: One of the main challenges encountered was managing the game's state complexity, especially when backtracking. The stack data structure proved invaluable in this regard, allowing for seamless transitions between game states without losing context.

Improvement Opportunities: While the current implementation is effective, there is potential for optimization, particularly in state management and action evaluation. Future iterations could explore heuristic-based approaches to prioritize actions more likely to lead to success, reducing the exploration of less promising paths.