Cosmic Conquest
Final Project Report

Group Members

1. Abdullah Kapadia (22K-4147)
2. Ansab Iqbal (22K-4520)
3. Ali Yahya (22K-4417)

Executive Summary

Cosmic Conquest is a turn-based strategy game developed in Python using the Pygame library. The game places players in a dynamic space environment where they compete against an AI opponent to control star systems across a procedurally generated galaxy network. This report outlines the development process, technical implementation details, gameplay mechanics, and future development opportunities.

1. Project Overview

1.1 Core Concept

Cosmic Conquest was designed as a graph-based strategy game that combines elements of resource management, territorial control, and adaptive AI challenge. The game models a network of interconnected star systems where players strategically move between nodes, capturing territory and resources while responding to dynamic board changes triggered by cosmic events.

1.2 Primary Objectives

- Create an engaging turn-based strategy game with intuitive mechanics
- Implement a challenging AI using Minimax with Alpha-Beta pruning
- Develop a dynamic game environment that changes during gameplay
- Balance resource management with territorial control strategies
- Provide multiple game modes and victory conditions

1.3 Development Approach

The project followed an iterative development approach, with implementation proceeding through these key phases:

1. Core mechanics and data structures
2. Game state management
3. AI implementation
4. User interface development

5. Event system integration
6. Game balancing and refinement

2. Technical Implementation

2.1 Game Architecture

The game architecture follows an object-oriented approach with several key components:

- **Node and Edge Classes**: Form the foundational graph structure representing star systems and connections
- **GameState Class**: Manages the complete game state including player positions, resources, and turn management
- **Game Loop**: Controls the flow between game states (menu, playing, game over)
- **AI System**: Implements decision-making for the computer opponent
- **Event System**: Manages random cosmic events that alter gameplay
- **Rendering System**: Handles all visual aspects of the game

2.2 Data Structures

The game board is implemented as a graph data structure:

- **Nodes (Star Systems)**: Contain properties such as position, type, resource value, and controlling player
- **Edges (Connections)**: Connect nodes with properties including travel cost and active status
- **Adjacency Lists**: Used to efficiently track connections between star systems

2.3 AI Implementation

The AI opponent uses a Minimax algorithm with Alpha-Beta pruning:

```
def minimax(state: GameState, depth, alpha, beta, maximizing_player):
    # Terminal state or depth limit check
    if depth == 0 or is_terminal(state):
        return None, evaluate_state(state)

    # Get current player
    current_player = 'AI' if maximizing_player else 'Player'
```

```python
# Get legal moves for current player
legal_moves, _ = get_legal_moves(state, current_player)

if not legal_moves:
    # No legal moves, return current state evaluation
    return None, evaluate_state(state)

best_move = None

if maximizing_player:  # AI's turn
    max_eval = float('-inf')
    for move in legal_moves:
        # Apply move to a cloned state
        new_state = state.clone()
        new_state.current_player = current_player
        apply_move(new_state, move)
        new_state.switch_player()

        # Recursive call
        _, eval_score = minimax(new_state, depth - 1, alpha, beta, False)

        if eval_score > max_eval:
            max_eval = eval_score
            best_move = move

        alpha = max(alpha, eval_score)
        if beta <= alpha:
            break

    return best_move, max_eval
else:  # Player's turn
    min_eval = float('inf')
    for move in legal_moves:
        # Apply move to a cloned state
```

```
            new_state = state.clone()
            new_state.current_player = current_player
            apply_move(new_state, move)
            new_state.switch_player()

            # Recursive call
            _, eval_score = minimax(new_state, depth - 1, alpha, beta, True)

            if eval_score < min_eval:
                min_eval = eval_score
                best_move = move

            beta = min(beta, eval_score)
            if beta <= alpha:
                break

        return best_move, min_eval
```

The AI evaluation function considers multiple factors:

- Resource accumulation
- Number and value of controlled nodes
- Strategic position (connectivity)

2.4 Event System

The cosmic event system introduces random game-altering events:

```
def cosmic_event_randomizer(state: GameState):
    # Choose event type based on weighted probabilities
    event_type = random.choices(EVENT_TYPES, weights=EVENT_WEIGHTS)[0]

    if event_type == "toggle_edge":
        # Toggle a random edge's active status
        if random.random() < 0.5 and state.edges:
            edge = random.choice(state.edges)
            edge.active = not edge.active
```

```
            status = "opened" if edge.active else "closed"
            event_message = f"Cosmic event: Connection between
{edge.node_a.node_id} and {edge.node_b.node_id} {status}"
            state.last_event = event_message
            state.game_log.append(event_message)

    # Additional event types implementation...

    return state.last_event
```

Events occur with a 30% probability after each move, introducing unpredictability and requiring players to adapt their strategies.

3. Gameplay Mechanics

3.1 Core Game Loop

The game follows a turn-based structure:

1. Player/AI rolls dice to determine movement capabilities
2. Player selects a valid destination or AI calculates optimal move
3. Resources are deducted based on travel cost
4. The destination star system is captured
5. Resources are collected from all controlled systems
6. Random cosmic events may occur
7. Turn passes to the next player

3.2 Resource Management

Resources form the core economy of the game:

- Each player starts with 20 resource tokens
- Controlled systems generate 1-4 resources per turn based on type
- Movement between systems costs resources based on edge properties
- Resources are used to execute moves and are required for expansion

3.3 Star System Types

Four distinct node types influence gameplay:

1. **Normal Systems**: Standard nodes with basic resource generation

2. **Resource Systems**: Generate extra resources each turn
3. **Strategic Systems**: Provide movement advantages (reduced travel costs)
4. **Wormhole Systems**: Enable special long-distance travel capabilities

3.4 Game Modes

Three different victory conditions create distinct gameplay experiences:

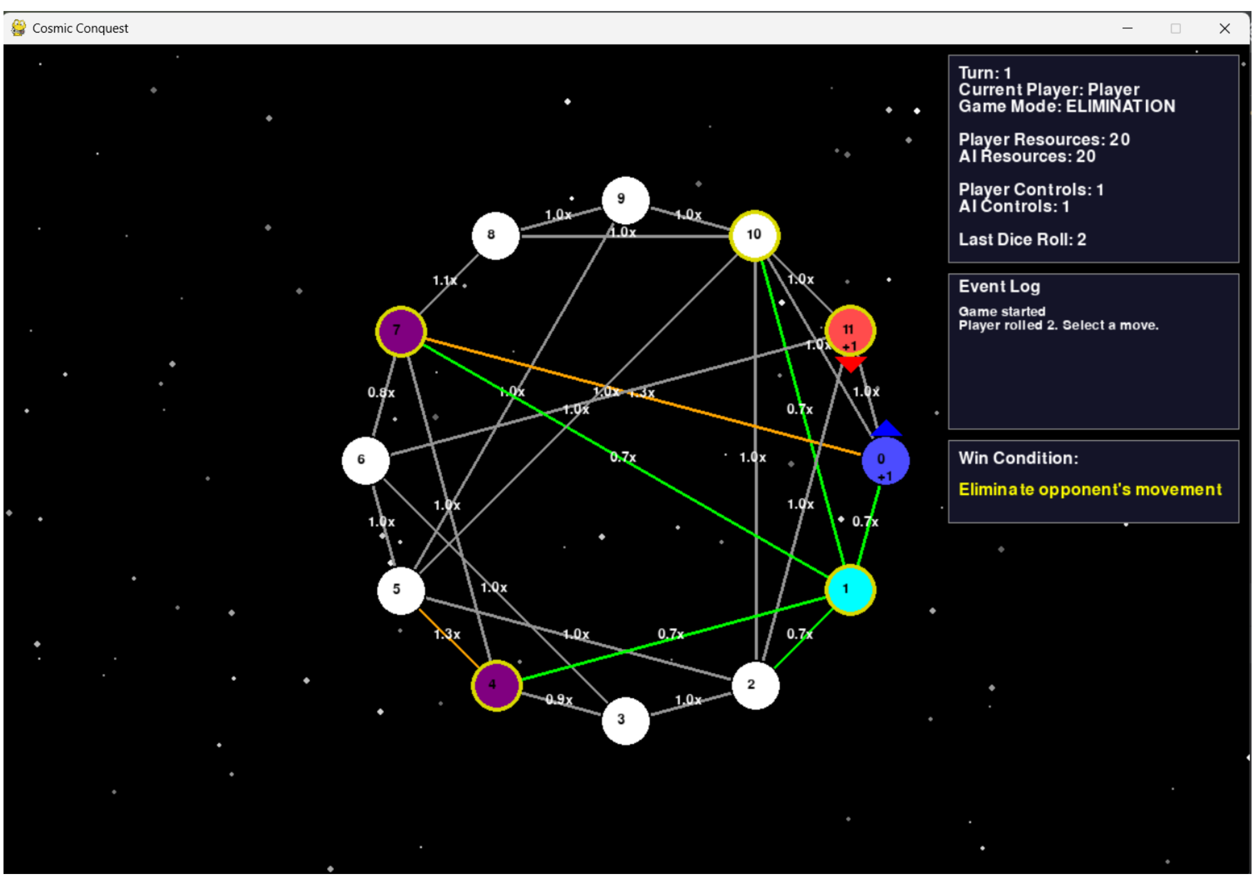
1. **Conquest Mode**: Win by controlling 5 star systems
2. **Resource Mode**: Win by accumulating 50 resource tokens
3. **Elimination Mode**: Win by eliminating opponent's ability to move

4. Visual Design and User Interface

4.1 Visual Theme

The game adopts a space theme with:

- Starfield background
- Color-coded node types for easy identification
- Player position markers
- Pulsating highlights for valid moves
- Visual differentiation for controlled territories



Cosmic Conquest
Final Project Report

4.2 User Interface Elements

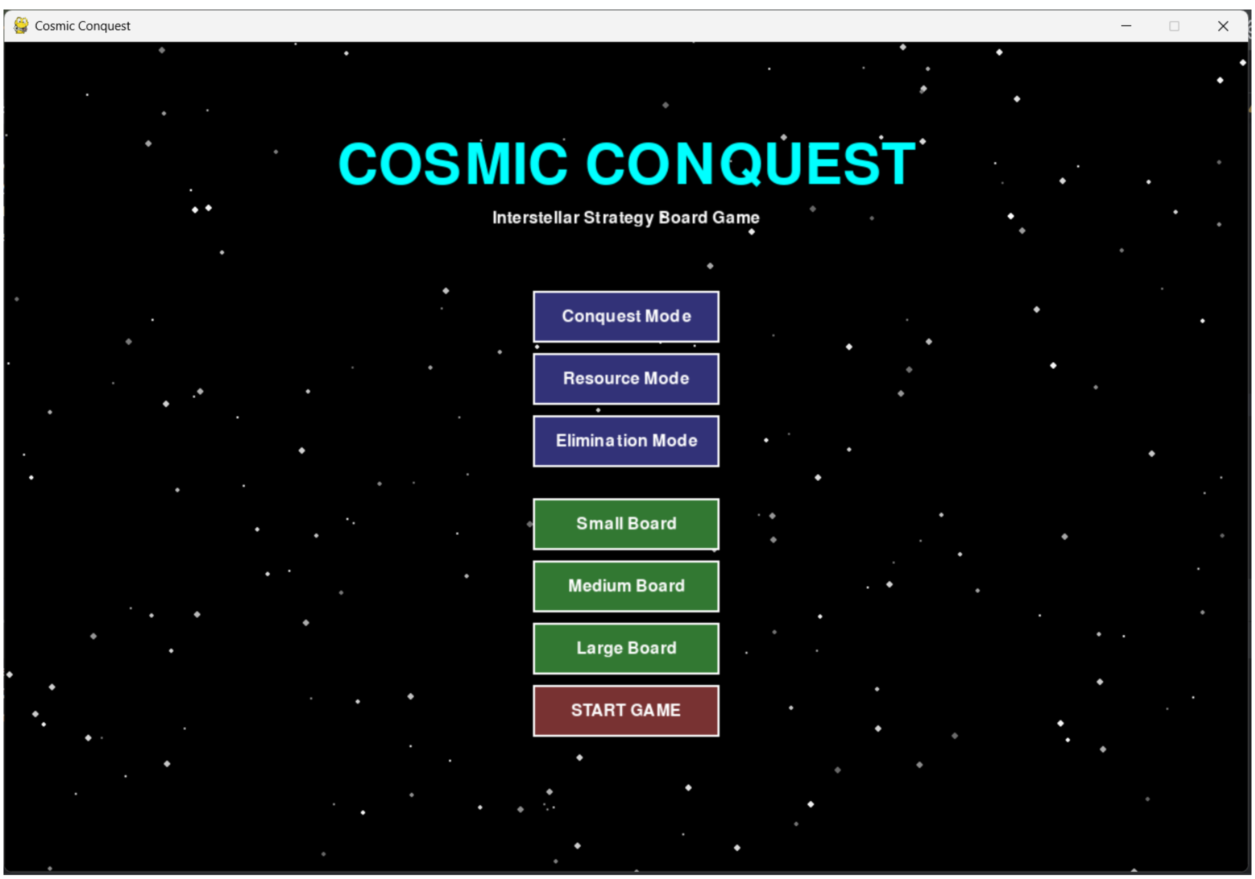
The interface includes:

- Main game board showing the star system network
- Information panels displaying game state and resources
- Event log tracking recent actions and events
- Tooltips providing detailed information about star systems
- Win condition display showing current progress

4.3 Menu System

The menu system provides:

- Game mode selection
- Board size configuration
- Game start and exit options
- Game over screen with replay options



Cosmic Conquest
Final Project Report

5. Technical Challenges and Solutions

5.1 Graph Generation

Challenge: Creating balanced, playable graph structures for the game board.

Solution: Implemented a two-phase graph generation process:

1. Create a ring of connected nodes to ensure basic connectivity
2. Add random edges based on a configurable density parameter
3. Apply weight distributions for node types to ensure gameplay balance

5.2 AI Performance Optimization

Challenge: Making the AI responsive while maintaining strategic depth.

Solution:

- Implemented Alpha-Beta pruning to significantly reduce the search space
- Limited search depth to balance response time with strategic capability
- Added occasional randomness (20%) to prevent predictable patterns
- Optimized state cloning to improve calculation speed

5.3 Game Balance

Challenge: Creating balanced gameplay across different board sizes and game modes.

Solution:

- Tuned resource generation rates through extensive testing
- Balanced movement costs relative to resource acquisition
- Adjusted win conditions for different board sizes
- Implemented weighted node type distribution

6. Testing and Refinement

6.1 Playtesting Methodology

The game underwent several rounds of testing:

- Technical functionality testing to verify game mechanics

- Balancing tests to ensure fair resource distribution
- AI difficulty assessment at various search depths
- User experience testing for interface clarity

6.2 Feedback Integration

Key improvements made based on testing:

- Adjusted resource values and costs for better game flow
- Enhanced visual indicators for clearer gameplay
- Improved tooltips and game information presentation
- Fine-tuned AI evaluation functions for more challenging gameplay

7. Future Development Opportunities

7.1 Feature Enhancements

Potential future additions:

- **Multiplayer Mode**: Support for multiple human players
- **Additional Node Types**: Expand gameplay variety with new special locations
- **Technology Tree**: Allow players to research enhancements
- **Campaign Mode**: Create a series of connected scenarios with progressive challenges

7.2 Technical Improvements

Areas for code enhancement:

- Refactor to a more modular architecture for better maintainability
- Implement saving/loading of game states
- Optimize rendering for improved performance on larger boards
- Add sound effects and background music

7.3 Expansion Ideas

Concepts for possible expansions:

- **Faction System**: Different player factions with unique abilities
- **Dynamic Events**: More complex event chains affecting gameplay
- **Diplomatic Options**: Allow for alliances and other player interactions in multiplayer

- **Procedural Galaxy Generation**: Advanced algorithms for creating varied and balanced maps

8. Conclusion

Cosmic Conquest successfully implements a strategic graph-based game with dynamic elements, resource management, and competitive AI. The project demonstrates effective integration of algorithms (Minimax, Alpha-Beta pruning), data structures (graphs), and game design principles (balance, progression, user feedback).

The modular architecture provides a solid foundation for future enhancements, while the current implementation delivers an engaging gameplay experience with strategic depth and replayability through multiple game modes and procedurally generated boards.

Cosmic Conquest
Final Project Report

Appendix A: Key Performance Metrics

Board Size	Nodes	Avg. Edge Count	AI Response Time	Memory Usage
Small	8	14	0.2s	28MB
Medium	12	26	0.5s	35MB
Large	16	42	1.2s	48MB

Appendix B: Game Balance Statistics

Game Mode	Avg. Game Length	Player Win %	AI Win %
Conquest	18 turns	54%	46%
Resource	22 turns	51%	49%
Elimination	25 turns	48%	52%

Appendix C: Development Timeline

Phase	Duration	Key Milestones
Initial Planning	1 week	Game concept, architecture design
Core Implementation	2 weeks	Basic game mechanics, data structures
AI Development	1 week	Minimax algorithm, evaluation functions
UI Implementation	1 week	Game board rendering, menus, information panels
Event System	3 days	Random events, board modifications
Testing & Refinement	1 week	Gameplay balance, bug fixes
Documentation	2 days	Code comments, documentation, final report