🎮 SRS - Race for the Treasure 🎮

Team Members and IDs:

- Moslem Masarwa – 214023798
- Ali Hamed – 325274520
- Mohammad Ganam – 213590789
- Elian Morany – 322703844

🔗 Links:

- 🖼️ Project Presentation
- 🎥 Demo Video
- 💻 GitHub Repository
- 🧩 Play the Game

📝 Project Summary:

Race for the Treasure is a competitive strategy game between a human player and an AI on a 10x10 board. Both players race to reach the treasure first, navigating through obstacles and using strategy to win.

🧠 Challenge:

was to ensure fair gameplay (equal distance to treasure) and implement smart AI decision-making based on the selected difficulty level.

Solution:

We designed a dynamic board with walls, traps, water tiles (which increase movement cost), and power-ups. The AI uses a different path-finding algorithm based on difficulty:

Algorithms Used:

• Easy: BFS (Breadth-First Search) – shortest path by steps.

• Normal: UCS (Uniform Cost Search) – considers movement costs (e.g., water = 2).

• Hard: A* combines actual cost with heuristic (Manhattan Distance).

• DFS: Used for visualization only.

Obstacles are placed procedurally using the `generateObstacles` function, which guarantees no overlap and balanced placement so both players have fair access to the treasure.

The project was implemented using TypeScript for interactive, real-time logic and rich visuals.

Source Code:

App.tsx

```
import { Toaster } from "@/components/ui/toaster";
import { Toaster as Sonner } from "@/components/ui/sonner";
import { TooltipProvider } from "@/components/ui/tooltip";
import { QueryClient, QueryClientProvider } from
"@tanstack/react-query";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Index from "./pages/Index";
import NotFound from "./pages/NotFound";

const queryClient = new QueryClient();
```

```
const App = () => (
  <QueryClientProvider client={queryClient}>
    <TooltipProvider>
      <Toaster />
      <Sonner />
      <BrowserRouter>
        <Routes>
          <Route path="/" element={<Index />} />
          {/* ADD ALL CUSTOM ROUTES ABOVE THE CATCH-ALL "*" ROUTE
*/}
          <Route path="*" element={<NotFound />} />
        </Routes>
      </BrowserRouter>
    </TooltipProvider>
  </QueryClientProvider>
);

export default App;
```

GameBoard.tsx

```
import React, { useCallback } from 'react';
import { GameGrid } from './GameGrid';
import { GameControls } from './GameControls';
import { GameStatus } from './GameStatus';
import { ActionSelector } from './ActionSelector';
import { PowerUpDisplay } from './PowerUpDisplay';
import { useGameState } from '../hooks/useGameState';

export const GameBoard = () => {
  const {
    gameState,
    currentPlayer,
    gameStatus,
    currentAction,
    setCurrentAction,
    performAction,
    resetGame,
    moveCount
  } = useGameState();

  const handleCellClick = useCallback((row: number, col: number)
=> {
```

```
    if (gameStatus !== 'playing') return;

    const success = performAction(row, col);
    if (success) {
      console.log(`Player ${currentPlayer} performed
${currentAction} at (${row}, ${col})`);
    }
  }, [gameStatus, currentPlayer, currentAction, performAction]);

  return (
    <div className="bg-white rounded-2xl shadow-2xl p-6">
      <GameStatus
        currentPlayer={currentPlayer}
        gameStatus={gameStatus}
        moveCount={moveCount}
      />

      <div className="flex flex-col lg:flex-row gap-6">
        <div className="flex-1 space-y-4">
          <ActionSelector
            currentAction={currentAction}
            onActionChange={setCurrentAction}
            gameState={gameState}
            isPlayerTurn={currentPlayer === 'human' && gameStatus
=== 'playing'}
          />

          <PowerUpDisplay
            gameState={gameState}
            currentPlayer={currentPlayer}
          />

          <GameGrid
            gameState={gameState}
            onCellClick={handleCellClick}
            currentPlayer={currentPlayer}
            currentAction={currentAction}
          />
        </div>

        <div className="lg:w-80">
          <GameControls
            onReset={resetGame}
            gameStatus={gameStatus}
            currentPlayer={currentPlayer}
```

```
          />
        </div>
      </div>
    </div>
  );
};
```

GameCell.tsx

```
import React from 'react';
import { Crown, User, Bot, Hash, Layers, Zap, Bomb } from
'lucide-react';
import { GameState } from '../types/game';
import { cn } from '../lib/utils';

interface GameCellProps {
  row: number;
  col: number;
  gameState: GameState;
  onClick: (row: number, col: number) => void;
  isValidMove: boolean;
  isValidPlacement: boolean;
  showValidMoves: boolean;
  currentAction: 'move' | 'place-wall' | 'place-ladder';
}

export const GameCell: React.FC<GameCellProps> = ({
  row,
  col,
  gameState,
  onClick,
  isValidMove,
  isValidPlacement,
  showValidMoves,
  currentAction
}) => {
  const { players, treasure, walls, ladders, highCostTiles,
trapTiles, powerUps } = gameState;

  const isHumanPlayer = players.human.row === row &&
players.human.col === col;
  const isAiPlayer = players.ai.row === row && players.ai.col ===
col;
  const isTreasure = treasure.row === row && treasure.col ===
```

```
col;
  const isWall = walls.some(wall => wall.row === row && wall.col
=== col);
  const isLadder = ladders.some(ladder => ladder.row === row &&
ladder.col === col);
  const isHighCost = highCostTiles.some(tile => tile.row === row
&& tile.col === col);
  const isTrap = trapTiles.some(trap => trap.row === row &&
trap.col === col);
  const isPowerUp = powerUps.some(powerUp => powerUp.row === row
&& powerUp.col === col);

  const handleClick = () => {
    onClick(row, col);
  };

  const getValidActionIndicator = () => {
    if (!showValidMoves) return null;

    if (currentAction === 'move' && isValidMove && !isHumanPlayer
&& !isAiPlayer && !isTreasure && !isWall) {
      return (
        <div className="absolute inset-0 flex items-center
justify-center">
          <div className="w-3 h-3 bg-blue-400 rounded-full
opacity-60" />
        </div>
      );
    }

    if ((currentAction === 'place-wall' || currentAction ===
'place-ladder') && isValidPlacement) {
      const color = currentAction === 'place-wall' ? 'bg-slate-
400' : 'bg-purple-400';
      return (
        <div className="absolute inset-0 flex items-center
justify-center">
          <div className={`w-4 h-4 ${color} rounded opacity-60
border-2 border-white`} />
        </div>
      );
    }

    return null;
  };
```

```
  return (
    <div
      className={cn(
        "w-12 h-12 border border-slate-300 flex items-center
justify-center cursor-pointer transition-all duration-200
relative",
        "hover:border-slate-400 hover:shadow-sm",
        {
          // Base cell styling
          "bg-emerald-50": !isHumanPlayer && !isAiPlayer &&
!isTreasure && !isWall && !isLadder && !isHighCost && !isTrap &&
!isPowerUp,

          // Valid move highlighting
          "bg-blue-100 border-blue-300 hover:bg-blue-200":
isValidMove && showValidMoves && currentAction === 'move' &&
!isWall,

          // Valid placement highlighting
          "bg-slate-100 border-slate-400 hover:bg-slate-200":
isValidPlacement && showValidMoves && currentAction === 'place-
wall',
          "bg-purple-100 border-purple-400 hover:bg-purple-200":
isValidPlacement && showValidMoves && currentAction === 'place-
ladder',

          // Treasure cell
          "bg-gradient-to-br from-amber-200 to-amber-300 border-
amber-400": isTreasure,

          // Player cells
          "bg-gradient-to-br from-blue-400 to-blue-500 border-
blue-600 text-white": isHumanPlayer,
          "bg-gradient-to-br from-red-400 to-red-500 border-red-
600 text-white": isAiPlayer,

          // Wall cells
          "bg-gradient-to-br from-slate-600 to-slate-700 border-
slate-800 text-white": isWall,

          // Ladder cells
          "bg-gradient-to-br from-purple-300 to-purple-400
border-purple-500": isLadder,
```

```
          // High cost tiles (water/mud)
          "bg-gradient-to-br from-cyan-300 to-cyan-400 border-
cyan-500": isHighCost,

          // Trap tiles
          "bg-gradient-to-br from-red-200 to-red-300 border-red-
400": isTrap,

          // Power-up tiles
          "bg-gradient-to-br from-yellow-200 to-yellow-300
border-yellow-400": isPowerUp,
        }
      )}
      onClick={handleClick}
    >
      {isTreasure && <Crown className="w-6 h-6 text-amber-700"
/>}
      {isHumanPlayer && <User className="w-5 h-5" />}
      {isAiPlayer && <Bot className="w-5 h-5" />}
      {isWall && <Hash className="w-5 h-5" />}
      {isLadder && <Layers className="w-5 h-5 text-purple-700"
/>}
      {isHighCost && !isHumanPlayer && !isAiPlayer && !isTreasure
&& (
        <div className="text-xs font-bold text-cyan-700">~</div>
      )}
      {isTrap && !isHumanPlayer && !isAiPlayer && !isTreasure &&
(
        <Bomb className="w-4 h-4 text-red-700" />
      )}
      {isPowerUp && !isHumanPlayer && !isAiPlayer && !isTreasure
&& (
        <Zap className="w-4 h-4 text-yellow-700" />
      )}

      {getValidActionIndicator()}
    </div>
  );
};
```

GameControls.tsx

```
import React from 'react';
import { RotateCcw, Users, Bot, Hash, Layers, Waves, Bomb, Zap }
```

```
from 'lucide-react';
import { Button } from './ui/button';
import { GameStatus } from '../types/game';

interface GameControlsProps {
  onReset: () => void;
  gameStatus: GameStatus;
  currentPlayer: 'human' | 'ai';
}

export const GameControls: React.FC<GameControlsProps> = ({
  onReset,
  gameStatus,
  currentPlayer
}) => {
  return (
    <div className="space-y-6">
      <div className="p-4 bg-slate-50 rounded-lg">
        <h3 className="text-lg font-semibold text-slate-800 mb-
3">Game Controls</h3>

        <Button
          onClick={onReset}
          className="w-full mb-4"
          variant="outline"
        >
          <RotateCcw className="w-4 h-4 mr-2" />
          New Game
        </Button>

        <div className="space-y-3">
          <div className="flex items-center gap-2 text-sm text-
slate-600">
            <Users className="w-4 h-4 text-blue-500" />
            <span>Human Player: Click adjacent cells to
move</span>
          </div>
          <div className="flex items-center gap-2 text-sm text-
slate-600">
            <Bot className="w-4 h-4 text-red-500" />
            <span>AI Agent: Moves automatically</span>
          </div>
        </div>
      </div>
```

```
      <div className="p-4 bg-slate-50 rounded-lg">
        <h3 className="text-lg font-semibold text-slate-800 mb-
3">Game Rules</h3>
        <ul className="text-sm text-slate-600 space-y-2">
          <li>• Move to adjacent cells (up, down, left,
right)</li>
          <li>• Cannot move through walls or other players</li>
          <li>• High-cost tiles (water) slow movement</li>
          <li>• Ladders provide strategic advantages</li>
          <li>• Trap tiles stun players for 2 turns</li>
          <li>• Power-ups grant special abilities</li>
          <li>• First to reach the treasure wins</li>
          <li>• Both players start at equal distance from
treasure</li>
        </ul>
      </div>

      <div className="p-4 bg-slate-50 rounded-lg">
        <h3 className="text-lg font-semibold text-slate-800 mb-
3">Legend</h3>
        <div className="space-y-2 text-sm">
          <div className="flex items-center gap-3">
            <div className="w-6 h-6 bg-gradient-to-br from-blue-
400 to-blue-500 rounded flex items-center justify-center">
              <Users className="w-3 h-3 text-white" />
            </div>
            <span className="text-slate-600">Human Player</span>
          </div>
          <div className="flex items-center gap-3">
            <div className="w-6 h-6 bg-gradient-to-br from-red-
400 to-red-500 rounded flex items-center justify-center">
              <Bot className="w-3 h-3 text-white" />
            </div>
            <span className="text-slate-600">AI Agent</span>
          </div>
          <div className="flex items-center gap-3">
            <div className="w-6 h-6 bg-gradient-to-br from-amber-
200 to-amber-300 rounded flex items-center justify-center">
              <span className="text-amber-700 text-xs">👑</span>
            </div>
            <span className="text-slate-600">Treasure</span>
          </div>
          <div className="flex items-center gap-3">
            <div className="w-6 h-6 bg-gradient-to-br from-slate-
600 to-slate-700 rounded flex items-center justify-center">
```

```
              <Hash className="w-3 h-3 text-white" />
            </div>
            <span className="text-slate-600">Wall
(Impassable)</span>
          </div>
          <div className="flex items-center gap-3">
            <div className="w-6 h-6 bg-gradient-to-br from-
purple-300 to-purple-400 rounded flex items-center justify-
center">
              <Layers className="w-3 h-3 text-purple-700" />
            </div>
            <span className="text-slate-600">Ladder
(Strategic)</span>
          </div>
          <div className="flex items-center gap-3">
            <div className="w-6 h-6 bg-gradient-to-br from-cyan-
300 to-cyan-400 rounded flex items-center justify-center">
              <Waves className="w-3 h-3 text-cyan-700" />
            </div>
            <span className="text-slate-600">Water (High
Cost)</span>
          </div>
          <div className="flex items-center gap-3">
            <div className="w-6 h-6 bg-gradient-to-br from-red-
200 to-red-300 rounded flex items-center justify-center">
              <Bomb className="w-3 h-3 text-red-700" />
            </div>
            <span className="text-slate-600">Trap (Stuns 2
turns)</span>
          </div>
          <div className="flex items-center gap-3">
            <div className="w-6 h-6 bg-gradient-to-br from-
yellow-200 to-yellow-300 rounded flex items-center justify-
center">
              <Zap className="w-3 h-3 text-yellow-700" />
            </div>
            <span className="text-slate-600">Power-Up
(Bonus)</span>
          </div>
        </div>
      </div>
    </div>
  );
};
```

GameGrid.tsx

```
import React from 'react';
import { GameCell } from './GameCell';
import { GameState, Position, ActionType } from '../types/game';

interface GameGridProps {
  gameState: GameState;
  onCellClick: (row: number, col: number) => void;
  currentPlayer: 'human' | 'ai';
  currentAction: ActionType;
}

export const GameGrid: React.FC<GameGridProps> = ({
  gameState,
  onCellClick,
  currentPlayer,
  currentAction
}) => {
  const getValidMoves = (player: 'human' | 'ai'): Position[] => {
    const playerPos = gameState.players[player];
    const validMoves: Position[] = [];

    // Check all adjacent positions
    const directions = [
      [-1, 0], [1, 0], [0, -1], [0, 1] // up, down, left, right
    ];

    directions.forEach(([dr, dc]) => {
      const newRow = playerPos.row + dr;
      const newCol = playerPos.col + dc;

      if (newRow >= 0 && newRow < 10 && newCol >= 0 && newCol <
10) {
        // Check if position is not occupied by the other player
        const otherPlayer = player === 'human' ? 'ai' : 'human';
        const otherPlayerPos = gameState.players[otherPlayer];

        // Check if position is not a wall
        const isWall = gameState.walls.some(wall => wall.row ===
newRow && wall.col === newCol);

        if (!(newRow === otherPlayerPos.row && newCol ===
otherPlayerPos.col) && !isWall) {
          validMoves.push({ row: newRow, col: newCol });
```

```
        }
      }
    });

    return validMoves;
  };

  const getValidPlacements = (): Position[] => {
    const validPlacements: Position[] = [];

    for (let row = 0; row < 10; row++) {
      for (let col = 0; col < 10; col++) {
        // Check if cell is empty (no players, treasure, walls,
ladders, high-cost tiles, traps, or power-ups)
        const isOccupied =
          (gameState.players.human.row === row &&
gameState.players.human.col === col) ||
          (gameState.players.ai.row === row &&
gameState.players.ai.col === col) ||
          (gameState.treasure.row === row &&
gameState.treasure.col === col) ||
          gameState.walls.some(wall => wall.row === row &&
wall.col === col) ||
          gameState.ladders.some(ladder => ladder.row === row &&
ladder.col === col) ||
          gameState.highCostTiles.some(tile => tile.row === row
&& tile.col === col) ||
          gameState.trapTiles.some(trap => trap.row === row &&
trap.col === col) ||
          gameState.powerUps.some(powerUp => powerUp.row === row
&& powerUp.col === col);

        if (!isOccupied) {
          validPlacements.push({ row, col });
        }
      }
    }

    return validPlacements;
  };

  const validMoves = currentAction === 'move' ?
getValidMoves(currentPlayer) : [];
  const validPlacements = (currentAction === 'place-wall' ||
currentAction === 'place-ladder') ? getValidPlacements() : [];
```

```
  const isValidMove = (row: number, col: number): boolean => {
    return validMoves.some(move => move.row === row && move.col
=== col);
  };

  const isValidPlacement = (row: number, col: number): boolean =>
{
    return validPlacements.some(placement => placement.row ===
row && placement.col === col);
  };

  return (
    <div className="grid grid-cols-10 gap-1 bg-slate-200 p-4
rounded-xl">
      {Array.from({ length: 100 }, (_, index) => {
        const row = Math.floor(index / 10);
        const col = index % 10;

        return (
          <GameCell
            key={`${row}-${col}`}
            row={row}
            col={col}
            gameState={gameState}
            onClick={onCellClick}
            isValidMove={isValidMove(row, col)}
            isValidPlacement={isValidPlacement(row, col)}
            showValidMoves={currentPlayer === 'human'}
            currentAction={currentAction}
          />
        );
      })}
    </div>
  );
};
```

GameStatus.tsx

```
import React from 'react';
import { User, Bot, Trophy, Clock } from 'lucide-react';
import { GameStatus as GameStatusType } from '../types/game';

interface GameStatusProps {
```

```
  currentPlayer: 'human' | 'ai';
  gameStatus: GameStatusType;
  moveCount: number;
}

export const GameStatus: React.FC<GameStatusProps> = ({
  currentPlayer,
  gameStatus,
  moveCount
}) => {
  const getStatusMessage = () => {
    switch (gameStatus) {
      case 'human-won':
        return 'Human Player Wins! 🎉';
      case 'ai-won':
        return 'AI Agent Wins! 🤖';
      case 'draw':
        return "It's a Draw! 🤝";
      case 'playing':
        return `${currentPlayer === 'human' ? 'Your' : "AI's"}
Turn`;
      default:
        return 'Ready to Play';
    }
  };

  const getStatusColor = () => {
    switch (gameStatus) {
      case 'human-won':
        return 'text-blue-600';
      case 'ai-won':
        return 'text-red-600';
      case 'draw':
        return 'text-purple-600';
      case 'playing':
        return currentPlayer === 'human' ? 'text-blue-600' :
'text-red-600';
      default:
        return 'text-slate-600';
    }
  };

  return (
    <div className="flex items-center justify-between mb-6 p-4
```

```
bg-slate-50 rounded-lg">
      <div className="flex items-center gap-4">
        <div className={`text-xl font-bold ${getStatusColor()}`}>
          {getStatusMessage()}
        </div>

        {gameStatus === 'playing' && (
          <div className="flex items-center gap-2 text-slate-
600">
            {currentPlayer === 'human' ? (
              <>
                <User className="w-5 h-5 text-blue-500" />
                <span>Human Turn</span>
              </>
            ) : (
              <>
                <Bot className="w-5 h-5 text-red-500" />
                <span>AI Thinking...</span>
              </>
            )}
          </div>
        )}
      </div>

      <div className="flex items-center gap-4 text-slate-600">
        <div className="flex items-center gap-2">
          <Clock className="w-4 h-4" />
          <span className="text-sm font-medium">Move
{moveCount}</span>
        </div>

        {(gameStatus === 'human-won' || gameStatus === 'ai-won')
&& (
          <Trophy className="w-5 h-5 text-amber-500" />
        )}
      </div>
    </div>
  );
};
```

main.tsx

```
import { createRoot } from 'react-dom/client'
import App from './App.tsx'
import './index.css'

createRoot(document.getElementById("root")!).render(<App />);
```

ActionSelector.tsx

```
import React from 'react';
import { Button } from './ui/button';
import { User, Hash, Layers } from 'lucide-react';
import { ActionType, GameState } from '../types/game';

interface ActionSelectorProps {
  currentAction: ActionType;
  onActionChange: (action: ActionType) => void;
  gameState: GameState;
  isPlayerTurn: boolean;
}

export const ActionSelector: React.FC<ActionSelectorProps> = ({
  currentAction,
  onActionChange,
  gameState,
  isPlayerTurn
}) => {
  if (!isPlayerTurn) return null;

  const playerResources = gameState.playerResources.human;
  const totalWalls = playerResources.walls +
playerResources.powerUps.extraWalls;
  const totalLadders = playerResources.ladders +
playerResources.powerUps.extraLadders;
  const isStunned = playerResources.stunned > 0;

  if (isStunned) {
    return (
      <div className="p-4 bg-red-50 rounded-lg border-2 border-
red-200">
        <h3 className="text-lg font-semibold text-red-800 mb-
2">⚠ Stunned!</h3>
        <p className="text-red-600">You are stunned for
{playerResources.stunned} more turn{playerResources.stunned > 1 ?
's' : ''}!</p>
      </div>
    );
  }

  return (
    <div className="p-4 bg-blue-50 rounded-lg border-2 border-
```

```
blue-200">
      <h3 className="text-lg font-semibold text-blue-800 mb-
3">Your Turn - Choose Action</h3>

      <div className="flex gap-2 flex-wrap">
        <Button
          onClick={() => onActionChange('move')}
          variant={currentAction === 'move' ? 'default' :
'outline'}
          size="sm"
          className="flex items-center gap-2"
        >
          <User className="w-4 h-4" />
          Move
        </Button>

        <Button
          onClick={() => onActionChange('place-wall')}
          variant={currentAction === 'place-wall' ? 'default' :
'outline'}
          size="sm"
          disabled={totalWalls === 0}
          className="flex items-center gap-2"
        >
          <Hash className="w-4 h-4" />
          Wall ({totalWalls})
        </Button>

        <Button
          onClick={() => onActionChange('place-ladder')}
          variant={currentAction === 'place-ladder' ? 'default' :
'outline'}
          size="sm"
          disabled={totalLadders === 0}
          className="flex items-center gap-2"
        >
          <Layers className="w-4 h-4" />
          Ladder ({totalLadders})
        </Button>
      </div>

      <div className="mt-2 text-sm text-blue-600">
        {currentAction === 'move' && 'Click an adjacent cell to
move'}
        {currentAction === 'place-wall' && 'Click an empty cell
```

```
to place a wall'}
        {currentAction === 'place-ladder' && 'Click an empty cell
to place a ladder'}
      </div>
    </div>
  );
};
```

useGameState.ts

```
import { useState, useCallback, useEffect } from 'react';
import { GameState, Position, GameStatus, ActionType, PowerUpType
} from '../types/game';

const generateObstacles = (treasure: Position, humanStart:
Position, aiStart: Position): {
  walls: Position[],
  ladders: Position[],
  highCostTiles: Position[],
  trapTiles: Position[],
  powerUps: Position[]
} => {
  const walls: Position[] = [];
  const ladders: Position[] = [];
  const highCostTiles: Position[] = [];
  const trapTiles: Position[] = [];
  const powerUps: Position[] = [];

  const occupiedPositions = [treasure, humanStart, aiStart];

  // Generate 8-12 random walls
  const numWalls = Math.floor(Math.random() * 5) + 8;
  for (let i = 0; i < numWalls; i++) {
    let wallPos: Position;
    let attempts = 0;
    do {
      wallPos = {
        row: Math.floor(Math.random() * 10),
        col: Math.floor(Math.random() * 10)
      };
      attempts++;
    } while (
      (occupiedPositions.some(pos => pos.row === wallPos.row &&
pos.col === wallPos.col) ||
       walls.some(pos => pos.row === wallPos.row && pos.col ===
wallPos.col)) &&
      attempts < 50
    );

    if (attempts < 50) {
      walls.push(wallPos);
    }
  }
```

```
  // Generate 3-5 ladders
  const numLadders = Math.floor(Math.random() * 3) + 3;
  for (let i = 0; i < numLadders; i++) {
    let ladderPos: Position;
    let attempts = 0;
    do {
      ladderPos = {
        row: Math.floor(Math.random() * 10),
        col: Math.floor(Math.random() * 10)
      };
      attempts++;
    } while (
      (occupiedPositions.some(pos => pos.row === ladderPos.row &&
pos.col === ladderPos.col) ||
       walls.some(pos => pos.row === ladderPos.row && pos.col ===
ladderPos.col) ||
       ladders.some(pos => pos.row === ladderPos.row && pos.col
=== ladderPos.col)) &&
      attempts < 50
    );

    if (attempts < 50) {
      ladders.push(ladderPos);
    }
  }

  // Generate 5-8 high cost tiles
  const numHighCost = Math.floor(Math.random() * 4) + 5;
  for (let i = 0; i < numHighCost; i++) {
    let highCostPos: Position;
    let attempts = 0;
    do {
      highCostPos = {
        row: Math.floor(Math.random() * 10),
        col: Math.floor(Math.random() * 10)
      };
      attempts++;
    } while (
      (occupiedPositions.some(pos => pos.row === highCostPos.row
&& pos.col === highCostPos.col) ||
       walls.some(pos => pos.row === highCostPos.row && pos.col
=== highCostPos.col) ||
       ladders.some(pos => pos.row === highCostPos.row && pos.col
=== highCostPos.col) ||
       highCostTiles.some(pos => pos.row === highCostPos.row &&
```

```
pos.col === highCostPos.col)) &&
      attempts < 50
    );

    if (attempts < 50) {
      highCostTiles.push(highCostPos);
    }
  }

  // Generate 2-4 trap tiles
  const numTraps = Math.floor(Math.random() * 3) + 2;
  for (let i = 0; i < numTraps; i++) {
    let trapPos: Position;
    let attempts = 0;
    do {
      trapPos = {
        row: Math.floor(Math.random() * 10),
        col: Math.floor(Math.random() * 10)
      };
      attempts++;
    } while (
      (occupiedPositions.some(pos => pos.row === trapPos.row &&
pos.col === trapPos.col) ||
       walls.some(pos => pos.row === trapPos.row && pos.col ===
trapPos.col) ||
       ladders.some(pos => pos.row === trapPos.row && pos.col ===
trapPos.col) ||
       highCostTiles.some(pos => pos.row === trapPos.row &&
pos.col === trapPos.col) ||
       trapTiles.some(pos => pos.row === trapPos.row && pos.col
=== trapPos.col)) &&
      attempts < 50
    );

    if (attempts < 50) {
      trapTiles.push(trapPos);
    }
  }

  // Generate 1-3 power-ups
  const numPowerUps = Math.floor(Math.random() * 3) + 1;
  for (let i = 0; i < numPowerUps; i++) {
    let powerUpPos: Position;
    let attempts = 0;
    do {
```

```
      powerUpPos = {
        row: Math.floor(Math.random() * 10),
        col: Math.floor(Math.random() * 10)
      };
      attempts++;
    } while (
      (occupiedPositions.some(pos => pos.row === powerUpPos.row
&& pos.col === powerUpPos.col) ||
       walls.some(pos => pos.row === powerUpPos.row && pos.col
=== powerUpPos.col) ||
       ladders.some(pos => pos.row === powerUpPos.row && pos.col
=== powerUpPos.col) ||
       highCostTiles.some(pos => pos.row === powerUpPos.row &&
pos.col === powerUpPos.col) ||
       trapTiles.some(pos => pos.row === powerUpPos.row &&
pos.col === powerUpPos.col) ||
       powerUps.some(pos => pos.row === powerUpPos.row && pos.col
=== powerUpPos.col)) &&
      attempts < 50
    );

    if (attempts < 50) {
      powerUps.push(powerUpPos);
    }
  }

  return { walls, ladders, highCostTiles, trapTiles, powerUps };
};

const manhattanDistance = (pos1: Position, pos2: Position):
number => {
  return Math.abs(pos1.row - pos2.row) + Math.abs(pos1.col -
pos2.col);
};

const generateEqualDistancePositions = (): { treasure: Position,
humanStart: Position, aiStart: Position } => {
  let treasure: Position, humanStart: Position, aiStart:
Position;
  let attempts = 0;

  do {
    treasure = {
      row: Math.floor(Math.random() * 8) + 1, // Avoid edges
      col: Math.floor(Math.random() * 8) + 1
```

```
    };

    humanStart = {
      row: Math.floor(Math.random() * 10),
      col: Math.floor(Math.random() * 10)
    };

    aiStart = {
      row: Math.floor(Math.random() * 10),
      col: Math.floor(Math.random() * 10)
    };

    attempts++;
  } while (
    (manhattanDistance(humanStart, treasure) !==
manhattanDistance(aiStart, treasure) ||
     manhattanDistance(humanStart, aiStart) < 3 ||
     (humanStart.row === treasure.row && humanStart.col ===
treasure.col) ||
     (aiStart.row === treasure.row && aiStart.col ===
treasure.col) ||
     (humanStart.row === aiStart.row && humanStart.col ===
aiStart.col)) &&
    attempts < 100
  );

  return { treasure, humanStart, aiStart };
};

export const useGameState = () => {
  const [gameState, setGameState] = useState<GameState>(() => {
    const { treasure, humanStart, aiStart } =
generateEqualDistancePositions();
    const obstacles = generateObstacles(treasure, humanStart,
aiStart);

    return {
      players: {
        human: humanStart,
        ai: aiStart
      },
      treasure,
      walls: obstacles.walls,
      ladders: obstacles.ladders,
      highCostTiles: obstacles.highCostTiles,
```

```
      trapTiles: obstacles.trapTiles,
      powerUps: obstacles.powerUps,
      playerResources: {
        human: {
          walls: 3,
          ladders: 2,
          powerUps: {
            extraMovement: 0,
            ignoreWaterCost: 0,
            extraWalls: 0,
            extraLadders: 0
          },
          stunned: 0
        },
        ai: {
          walls: 3,
          ladders: 2,
          powerUps: {
            extraMovement: 0,
            ignoreWaterCost: 0,
            extraWalls: 0,
            extraLadders: 0
          },
          stunned: 0
        }
      }
    };
  });

  const [currentPlayer, setCurrentPlayer] = useState<'human' |
'ai'>('human');
  const [gameStatus, setGameStatus] =
useState<GameStatus>('playing');
  const [moveCount, setMoveCount] = useState<number>(0);
  const [currentAction, setCurrentAction] =
useState<ActionType>('move');

  const checkWinCondition = useCallback((state: GameState):
GameStatus => {
    const { players, treasure } = state;

    const humanWins = players.human.row === treasure.row &&
players.human.col === treasure.col;
    const aiWins = players.ai.row === treasure.row &&
players.ai.col === treasure.col;
```

```
    if (humanWins && aiWins) return 'draw';
    if (humanWins) return 'human-won';
    if (aiWins) return 'ai-won';

    return 'playing';
  }, []);

  const handleTileEffects = useCallback((position: Position,
player: 'human' | 'ai', state: GameState): GameState => {
    let newState = { ...state };

    // Check if player stepped on a trap
    const isTrap = state.trapTiles.some(trap => trap.row ===
position.row && trap.col === position.col);
    if (isTrap) {
      newState.playerResources = {
        ...newState.playerResources,
        [player]: {
          ...newState.playerResources[player],
          stunned: 2 // Stun for 2 turns
        }
      };
      console.log(`${player} stepped on a trap and is stunned for
2 turns!`);
    }

    // Check if player stepped on a power-up
    const isPowerUp = state.powerUps.some(powerUp => powerUp.row
=== position.row && powerUp.col === position.col);
    if (isPowerUp) {
      // Remove the power-up from the board
      newState.powerUps = newState.powerUps.filter(powerUp =>
!(powerUp.row === position.row && powerUp.col === position.col));

      // Grant a random power-up
      const powerUpTypes: PowerUpType[] = ['extraMovement',
'ignoreWaterCost', 'extraWalls', 'extraLadders'];
      const randomPowerUp = powerUpTypes[Math.floor(Math.random()
* powerUpTypes.length)];

      newState.playerResources = {
        ...newState.playerResources,
        [player]: {
          ...newState.playerResources[player],
```

```
          powerUps: {
            ...newState.playerResources[player].powerUps,
            [randomPowerUp]:
newState.playerResources[player].powerUps[randomPowerUp] + 1
          }
        }
      };
      console.log(`${player} collected a ${randomPowerUp} power-
up!`);
    }

    return newState;
  }, []);

  const isValidMove = useCallback((from: Position, to: Position,
state: GameState): boolean => {
    // Check if move is within bounds
    if (to.row < 0 || to.row >= 10 || to.col < 0 || to.col >= 10)
{
      return false;
    }

    // Check if move is adjacent (only orthogonal moves allowed)
    const rowDiff = Math.abs(to.row - from.row);
    const colDiff = Math.abs(to.col - from.col);

    if ((rowDiff === 1 && colDiff === 0) || (rowDiff === 0 &&
colDiff === 1)) {
      // Check if destination is not occupied by the other player
      const otherPlayer = currentPlayer === 'human' ? 'ai' :
'human';
      const otherPlayerPos = state.players[otherPlayer];

      // Check if destination is not a wall
      const isWall = state.walls.some(wall => wall.row === to.row
&& wall.col === to.col);

      return !(to.row === otherPlayerPos.row && to.col ===
otherPlayerPos.col) && !isWall;
    }

    return false;
  }, [currentPlayer]);

  const isValidPlacement = useCallback((position: Position,
```

```
state: GameState): boolean => {
    // Check if position is within bounds
    if (position.row < 0 || position.row >= 10 || position.col <
0 || position.col >= 10) {
      return false;
    }

    // Check if position is empty
    const isOccupied =
      (state.players.human.row === position.row &&
state.players.human.col === position.col) ||
      (state.players.ai.row === position.row &&
state.players.ai.col === position.col) ||
      (state.treasure.row === position.row && state.treasure.col
=== position.col) ||
      state.walls.some(wall => wall.row === position.row &&
wall.col === position.col) ||
      state.ladders.some(ladder => ladder.row === position.row &&
ladder.col === position.col) ||
      state.highCostTiles.some(tile => tile.row === position.row
&& tile.col === position.col) ||
      state.trapTiles.some(tile => tile.row === position.row &&
tile.col === position.col) ||
      state.powerUps.some(powerUp => powerUp.row === position.row
&& powerUp.col === position.col);

    return !isOccupied;
  }, []);

  const performAction = useCallback((row: number, col: number):
boolean => {
    if (gameStatus !== 'playing' || currentPlayer !== 'human') {
      return false;
    }

    // Check if human player is stunned
    if (gameState.playerResources.human.stunned > 0) {
      console.log("You are stunned and cannot move!");
      return false;
    }

    const position: Position = { row, col };

    if (currentAction === 'move') {
      const currentPosition = gameState.players[currentPlayer];
```

```
      if (!isValidMove(currentPosition, position, gameState)) {
        return false;
      }

      let newGameState: GameState = {
        ...gameState,
        players: {
          ...gameState.players,
          [currentPlayer]: position
        }
      };

      // Handle tile effects
      newGameState = handleTileEffects(position, currentPlayer,
newGameState);

      setGameState(newGameState);
      setMoveCount(prev => prev + 1);

      const newStatus = checkWinCondition(newGameState);
      if (newStatus !== 'playing') {
        setGameStatus(newStatus);
      } else {
        setCurrentPlayer('ai');
        setCurrentAction('move');
      }

      return true;
    } else if (currentAction === 'place-wall') {
      const totalWalls = gameState.playerResources.human.walls +
gameState.playerResources.human.powerUps.extraWalls;

      if (!isValidPlacement(position, gameState) || totalWalls
=== 0) {
        return false;
      }

      let newGameState: GameState = {
        ...gameState,
        walls: [...gameState.walls, position]
      };

      // Use regular walls first, then extra walls
      if (gameState.playerResources.human.walls > 0) {
```

```
        newGameState.playerResources = {
          ...newGameState.playerResources,
          human: {
            ...newGameState.playerResources.human,
            walls: newGameState.playerResources.human.walls - 1
          }
        };
      } else {
        newGameState.playerResources = {
          ...newGameState.playerResources,
          human: {
            ...newGameState.playerResources.human,
            powerUps: {
              ...newGameState.playerResources.human.powerUps,
              extraWalls:
newGameState.playerResources.human.powerUps.extraWalls - 1
            }
          }
        };
      }

      setGameState(newGameState);
      setCurrentPlayer('ai');
      setCurrentAction('move');
      return true;
    } else if (currentAction === 'place-ladder') {
      const totalLadders =
gameState.playerResources.human.ladders +
gameState.playerResources.human.powerUps.extraLadders;

      if (!isValidPlacement(position, gameState) || totalLadders
=== 0) {
        return false;
      }

      let newGameState: GameState = {
        ...gameState,
        ladders: [...gameState.ladders, position]
      };

      // Use regular ladders first, then extra ladders
      if (gameState.playerResources.human.ladders > 0) {
        newGameState.playerResources = {
          ...newGameState.playerResources,
          human: {
```

```
            ...newGameState.playerResources.human,
            ladders: newGameState.playerResources.human.ladders -
1
          }
        };
      } else {
        newGameState.playerResources = {
          ...newGameState.playerResources,
          human: {
            ...newGameState.playerResources.human,
            powerUps: {
              ...newGameState.playerResources.human.powerUps,
              extraLadders:
newGameState.playerResources.human.powerUps.extraLadders - 1
            }
          }
        };
      }

      setGameState(newGameState);
      setCurrentPlayer('ai');
      setCurrentAction('move');
      return true;
    }

    return false;
  }, [gameState, currentPlayer, currentAction, gameStatus,
isValidMove, isValidPlacement, checkWinCondition,
handleTileEffects]);

  // Enhanced AI that can also place obstacles
  const makeAIMove = useCallback(() => {
    if (gameStatus !== 'playing' || currentPlayer !== 'ai') {
      return;
    }

    // Check if AI is stunned
    if (gameState.playerResources.ai.stunned > 0) {
      console.log("AI is stunned and skips turn");
      // Reduce stun duration and switch to human player
      setGameState(prevState => ({
        ...prevState,
        playerResources: {
          ...prevState.playerResources,
          ai: {
```

```
            ...prevState.playerResources.ai,
            stunned: prevState.playerResources.ai.stunned - 1
          }
        }
      }));
      setCurrentPlayer('human');
      return;
    }

    // AI decision logic: 70% chance to move, 30% chance to place
obstacle (if available)
    const aiResources = gameState.playerResources.ai;
    const canPlaceWall = (aiResources.walls +
aiResources.powerUps.extraWalls) > 0;
    const canPlaceLadder = (aiResources.ladders +
aiResources.powerUps.extraLadders) > 0;

    const actions: ActionType[] = ['move'];
    if (canPlaceWall) actions.push('place-wall');
    if (canPlaceLadder) actions.push('place-ladder');

    // Simple AI strategy: prefer moving, but occasionally place
obstacles
    const actionChoice = Math.random() < 0.7 ? 'move' :
actions[Math.floor(Math.random() * actions.length)];

    if (actionChoice === 'move') {
      const aiPosition = gameState.players.ai;
      const validMoves: Position[] = [];

      const directions = [[-1, 0], [1, 0], [0, -1], [0, 1]];

      directions.forEach(([dr, dc]) => {
        const newRow = aiPosition.row + dr;
        const newCol = aiPosition.col + dc;
        const newPos: Position = { row: newRow, col: newCol };

        if (isValidMove(aiPosition, newPos, gameState)) {
          validMoves.push(newPos);
        }
      });

      if (validMoves.length > 0) {
        const treasure = gameState.treasure;
        let bestMove = validMoves[0];
```

```
        let bestDistance = Math.abs(bestMove.row - treasure.row)
+ Math.abs(bestMove.col - treasure.col);

        validMoves.forEach(move => {
          const distance = Math.abs(move.row - treasure.row) +
Math.abs(move.col - treasure.col);
          if (distance < bestDistance) {
            bestDistance = distance;
            bestMove = move;
          }
        });

        let newGameState: GameState = {
          ...gameState,
          players: {
            ...gameState.players,
            ai: bestMove
          }
        };

        // Handle tile effects for AI
        newGameState = handleTileEffects(bestMove, 'ai',
newGameState);

        setGameState(newGameState);
        setMoveCount(prev => prev + 1);

        const newStatus = checkWinCondition(newGameState);
        if (newStatus !== 'playing') {
          setGameStatus(newStatus);
        } else {
          setCurrentPlayer('human');
        }
      }
    } else {
      // AI places obstacle strategically
      const validPlacements: Position[] = [];

      for (let row = 0; row < 10; row++) {
        for (let col = 0; col < 10; col++) {
          const pos: Position = { row, col };
          if (isValidPlacement(pos, gameState)) {
            validPlacements.push(pos);
          }
        }
```

```
      }

      if (validPlacements.length > 0) {
        // Simple strategy: place obstacle near human player's
path to treasure
        const humanPos = gameState.players.human;
        const treasure = gameState.treasure;

        let bestPlacement = validPlacements[0];
        let bestScore = 0;

        validPlacements.forEach(placement => {
          // Score based on how much it might block the human
player
          const distToHuman = Math.abs(placement.row -
humanPos.row) + Math.abs(placement.col - humanPos.col);
          const distToTreasure = Math.abs(placement.row -
treasure.row) + Math.abs(placement.col - treasure.col);

          // Prefer positions close to human but not too close to
treasure
          const score = (10 - distToHuman) + (distToTreasure > 2
? 5 : 0);

          if (score > bestScore) {
            bestScore = score;
            bestPlacement = placement;
          }
        });

        const obstacleType = actionChoice === 'place-wall' ?
'walls' : 'ladders';
        const resourceType = actionChoice === 'place-wall' ?
'walls' : 'ladders';

        const newGameState: GameState = {
          ...gameState,
          [obstacleType]: [...gameState[obstacleType],
bestPlacement],
          playerResources: {
            ...gameState.playerResources,
            ai: {
              ...gameState.playerResources.ai,
              [resourceType]:
gameState.playerResources.ai[resourceType] - 1
```

```
            }
          }
        };

        setGameState(newGameState);
        setCurrentPlayer('human');
      }
    }
  }, [gameState, currentPlayer, gameStatus, isValidMove,
isValidPlacement, checkWinCondition, handleTileEffects]);

  // Handle AI moves and reduce stun duration
  useEffect(() => {
    if (currentPlayer === 'ai' && gameStatus === 'playing') {
      const timer = setTimeout(() => {
        makeAIMove();
      }, 1000);

      return () => clearTimeout(timer);
    } else if (currentPlayer === 'human' && gameStatus ===
'playing') {
      // Reduce human stun duration at start of their turn
      if (gameState.playerResources.human.stunned > 0) {
        setGameState(prevState => ({
          ...prevState,
          playerResources: {
            ...prevState.playerResources,
            human: {
              ...prevState.playerResources.human,
              stunned: prevState.playerResources.human.stunned -
1
            }
          }
        }));
      }
    }
  }, [currentPlayer, gameStatus, makeAIMove,
gameState.playerResources.human.stunned]);

  const resetGame = useCallback(() => {
    const { treasure, humanStart, aiStart } =
generateEqualDistancePositions();
    const obstacles = generateObstacles(treasure, humanStart,
aiStart);
```

```
  const newGameState: GameState = {
    players: {
      human: humanStart,
      ai: aiStart
    },
    treasure,
    walls: obstacles.walls,
    ladders: obstacles.ladders,
    highCostTiles: obstacles.highCostTiles,
    trapTiles: obstacles.trapTiles,
    powerUps: obstacles.powerUps,
    playerResources: {
      human: {
        walls: 3,
        ladders: 2,
        powerUps: {
          extraMovement: 0,
          ignoreWaterCost: 0,
          extraWalls: 0,
          extraLadders: 0
        },
        stunned: 0
      },
      ai: {
        walls: 3,
        ladders: 2,
        powerUps: {
          extraMovement: 0,
          ignoreWaterCost: 0,
          extraWalls: 0,
          extraLadders: 0
        },
        stunned: 0
      }
    }
  };

  setGameState(newGameState);
  setCurrentPlayer('human');
  setGameStatus('playing');
  setMoveCount(0);
  setCurrentAction('move');
}, []);

return {
```

```
    gameState,
    currentPlayer,
    gameStatus,
    moveCount,
    currentAction,
    setCurrentAction,
    performAction,
    resetGame
  };
};
```

utils.ts

```
import { clsx, type ClassValue } from "clsx"
import { twMerge } from "tailwind-merge"

export function cn(...inputs: ClassValue[]) {
  return twMerge(clsx(inputs))
}
```