Praxisarbeit
Elektrotechniker HF
Treasure Hunt Spiel in C

Autor: Patriot Ibraimi

Institution: ipso Bildung AG

Studiengang: Elektrotechniker HF

Modul: PV.A.A.A.46_N-ZH-S2504-PROB.TA1A

Datum: 05.10.2025

Inhaltsverzeichniss

1. Management Summary

Im Rahmen der Praxisarbeit Elektrotechniker HF wurde ein textbasiertes Konsolenspiel in der Programmiersprache C entwickelt. Das Projekt „Treasure Hunt Spiel" demonstriert die praktische Anwendung fundamentaler Konzepte der Programmiertechnik in einem interaktiven Kontext.

Das Spiel stellt ein Labyrinth-Szenario dar, bei dem der Spieler (Symbol „P") einen Schatz (Symbol „T") in einem zweidimensionalen Raster finden muss. Die Implementierung erfolgte vollständig in C und wurde in vier separate Module strukturiert: Hauptprogramm (main.c), Spielfeldverwaltung (grid.c), Spielersteuerung (player.c) und Pfadfindung (pathfinding.c). Diese modulare Architektur gewährleistet klare Verantwortlichkeiten, hohe Wartbarkeit und gute Erweiterbarkeit.

Zentrale technische Elemente umfassen die Verwendung eines 2D-Arrays zur Spielfeldrepräsentation, zufällige Generierung von Hindernissen (ca. 20% des Spielfelds), Implementierung des BFS-Algorithmus (Breadth-First Search) zur Erreichbarkeitsprüfung sowie robuste Eingabevalidierung mit Fehlerbehandlung. Das Projekt erfüllt alle gestellten Anforderungen und demonstriert professionelle Softwareentwicklungspraktiken.

Die Arbeit umfasst vollständige Dokumentation des Codes mit aussagekräftigen Kommentaren, systematisches Testing aller Funktionalitäten, ein Makefile für automatisierte Kompilierung sowie eine umfassende README-Datei. Das Endergebnis ist ein stabiles, funktionsfähiges und didaktisch wertvolles Lernprojekt, das die erlernten Konzepte der Programmiertechnik erfolgreich in die Praxis umsetzt.

2. Anforderungen & Aufgabenstellung

2.1 Zielsetzung

- Algorithmen: Implementierung von Such- und Navigationsalgorithmen
- Datenstrukturen: Verwendung von Arrays, Structs und komplexen Datentypen
- Programmsteuerung: Schleifen, Verzweigungen und Kontrollstrukturen
- Modulare Entwicklung: Aufteilung in mehrere zusammenhängende Module
- Fehlerbehandlung: Robuste Eingabevalidierung und Fehlerprävention
- Softwarewerkzeuge: Nutzung von Compiler, Makefile und Versionsverwaltung

2.2 Funktionale Anforderungen

Spielmechanik:

- Darstellung des Spielfelds mittels 2D-Array (konfigurierbare Größe, Standard: 12x12)
- Zufällige Platzierung von Spieler (P), Schatz (T) und Hindernissen (O)
- Leere Felder werden mit Punkt (.) dargestellt
- Steuerung über WASD-Tasten mit Enter-Bestätigung
- Aktualisierung und Neudarstellung des Spielfelds nach jeder Bewegung
- Kollisionserkennung für Hindernisse und Spielfeldgrenzen
- Siegbedingung: Spieler erreicht den Schatz

Technische Anforderungen:

- Implementierung in reinem C (Standard: C99)
- Modulare Code-Struktur mit mehreren C- und Header-Dateien
- Erreichbarkeitsprüfung mittels Suchalgorithmus (BFS)
- Automatische Regenerierung des Spielfelds bei Nicht-Erreichbarkeit
- Umfassende Code-Dokumentation mit Kommentaren
- Systematisches Testing aller Funktionen
- Makefile für automatisierte Kompilierung

Dokumentationsanforderungen:

- Strukturierte Projektdokumentation im PDF-Format (mindestens 8 Seiten)
- README-Datei mit Kompilier- und Nutzungsanweisungen
- Flussdiagramm des Spielablaufs
- Testprotokoll mit dokumentierten Testfällen
- Reflexion über Lessons Learned

2.3 Qualitäts- & Nichtfunktionale Anforderungen

Neben den funktionalen Anforderungen wurde besonders auf die Qualität und Wartbarkeit des Codes geachtet. Mir war wichtig, dass das Programm übersichtlich, nachvollziehbar und leicht erweiterbar bleibt. Die wichtigsten nichtfunktionalen Anforderungen waren daher:

- Lesbarkeit des Codes durch sinnvolle Namen und Kommentare
- Portabilität des Programms auf verschiedene Betriebssysteme (Linux, Windows, macOS)
- Wartbarkeit durch modulare Struktur und klar getrennte Verantwortlichkeiten
- Benutzerfreundlichkeit in der Konsolenbedienung (klare Eingabeaufforderungen, Fehlermeldungen)
- Stabilität und Robustheit gegenüber ungültigen Eingaben

2.4 Erwartete Resultate

Am Ende der Entwicklung sollte ein stabiles, vollständig funktionsfähiges Konsolenspiel entstehen, das alle Anforderungen erfüllt. Der Spieler sollte sich frei im Spielfeld bewegen, Hindernissen ausweichen und den Schatz erfolgreich finden können. Außerdem sollte jedes Spielfeld lösbar sein – das war ein zentrales Ziel, das ich mit dem BFS-Algorithmus sichergestellt habe. Ich wollte außerdem, dass das Programm strukturiert, kommentiert und für Dritte gut verständlich bleibt.

3. Design

3.1 Architektur-Übersicht

```
treasure_game/
├── treasure_game.h    # Header-Datei (Deklarationen, Konstanten)
├── main.c             # Hauptprogramm und Spielschleife
├── grid.c             # Spielfeldgenerierung und -darstellung
├── player.c           # Spielerbewegung und Eingabeverarbeitung
├── pathfinding.c      # BFS-Algorithmus für Erreichbarkeit
├── Makefile           # Build-Automatisierung
└── README.md          # Projektdokumentation
```

3.2 Datenstrukturen

Position-Struktur:

```
typedef struct {
    int x;  // X-Koordinate (Spalte)
    int y;  // Y-Koordinate (Zeile)
} Position;
```

Globale Variablen:

```
char grid[GRID_SIZE][GRID_SIZE];  // 2D-Spielfeld
Position player;                   // Spielerposition
Position treasure;                 // Schatzposition
```

Spielfeld-Symbole:

- . (EMPTY) - Leeres Feld
- O (OBSTACLE) - Hindernis
- P (PLAYER) - Spieler
- T (TREASURE) - Schatz

3.3 Algorithmus: Breadth-First Search (BFS)

Der BFS-Algorithmus stellt sicher, dass der generierte Schatz vom Spieler aus erreichbar ist. Dies verhindert unlösbare Spielsituationen.

Funktionsweise:
1. Startposition (Spieler) in Queue einfügen
2. Position als besucht markieren
3. Alle vier Nachbarfelder (oben, unten, links, rechts) explorieren
4. Nicht besuchte und freie Felder zur Queue hinzufügen
5. Wiederholen bis Ziel erreicht oder Queue leer
6. Rückgabe: true wenn Pfad existiert, false wenn nicht erreichbar

Zeitkomplexität: $O(n^2)$ (n = GRID_SIZE)
Platzkomplexität: $O(n^2)$ für visited-Array und Queue

3.4 Flussdiagramm des Spielablaufs (Text)

```
[START] -> [Initialisierung] -> [Spielfeld generieren] ->
[Erreichbarkeit prüfen] -> [Erreichbar? JA] -> [Spielfeld anzeigen] ->
[Hauptspielschleife] -> [Eingabe/Validierung] -> [Kollision? NEIN] ->
[Spieler bewegen] -> [Aktualisieren] -> [Schatz erreicht?] -> [JA] ->
[Siegesmeldung] -> [ENDE]
```

4. Implementierung

4.1 Technologie-Stack

- C (ISO C99), GCC (GNU Compiler Collection)
- Compiler-Flags: -Wall -Wextra -std=c99 -pedantic
- Build-System: GNU Make
- Umgebung: Terminal/Konsole
- Plattform: Linux/Unix/macOS (Windows via MinGW)

4.2 Modul-Implementierung

4.2.1 Header-Datei (treasure_game.h):

```
#define GRID_SIZE 12        // Konfigurierbare Spielfeldgröße
#define EMPTY '.'           // Symbol für leere Felder
#define OBSTACLE 'O'        // Symbol für Hindernisse
#define TREASURE 'T'        // Symbol für Schatz
#define PLAYER 'P'          // Symbol für Spieler

typedef struct {
    int x;
    int y;
} Position;

// Externe Variablendeklarationen
extern char grid[GRID_SIZE][GRID_SIZE];
extern Position player;
extern Position treasure;
```

4.2.2 Hauptprogramm (main.c):

```
int main(void) {
    srand(time(NULL));              // Zufallsgenerator initialisieren

    printf("=== TREASURE HUNT SPIEL ===\n");
    printf("Steuerung: W/A/S/D + Enter\n");

    generateGrid();                 // Spielfeld generieren
    printGrid();                    // Anfangszustand anzeigen

    char move;
    bool won = false;

    while (!won) {
        scanf(" %c", &move);
```

```
        processInput(move, &won);
        if (!won) printGrid();
    }

    printf("GRATULATION! Schatz gefunden!\n");
    return 0;
}
```

4.2.3 Spielfeldgenerierung (grid.c):

```
void generateGrid(void) {
    do {
        initGrid();

        int obstacleCount = (GRID_SIZE * GRID_SIZE) / 5;
        for (int i = 0; i < obstacleCount; i++) {
            int x, y;
            do {
                x = rand() % GRID_SIZE;
                y = rand() % GRID_SIZE;
            } while (grid[y][x] != EMPTY);
            grid[y][x] = OBSTACLE;
        }

        do {
            player.x = rand() % GRID_SIZE;
            player.y = rand() % GRID_SIZE;
        } while (grid[player.y][player.x] != EMPTY);
        grid[player.y][player.x] = PLAYER;

        do {
            treasure.x = rand() % GRID_SIZE;
            treasure.y = rand() % GRID_SIZE;
        } while (grid[treasure.y][treasure.x] != EMPTY);
        grid[treasure.y][treasure.x] = TREASURE;

    } while (!isReachable(player, treasure));
}
```

4.2.4 Spielerbewegung (player.c):

```
bool movePlayer(int dx, int dy) {
    int newX = player.x + dx;
    int newY = player.y + dy;

    if (!isValidPosition(newX, newY)) {
        printf(">>> Spielfeldgrenze erreicht!\n");
        return false;
    }

    if (grid[newY][newX] == OBSTACLE) {
        printf(">>> Hindernis im Weg!\n");
        return false;
    }

    grid[player.y][player.x] = EMPTY;
```

```
    player.x = newX;
    player.y = newY;

    if (grid[player.y][player.x] == TREASURE) {
        grid[player.y][player.x] = PLAYER;
        return true;
    }

    grid[player.y][player.x] = PLAYER;
    return false;
}
```

4.2.5 BFS-Algorithmus (pathfinding.c):

```
bool isReachable(Position start, Position end) {
    bool visited[GRID_SIZE][GRID_SIZE] = {false};
    Position queue[GRID_SIZE * GRID_SIZE];
    int front = 0, rear = 0;

    queue[rear++] = start;
    visited[start.y][start.x] = true;

    int dx[] = {1, -1, 0, 0};
    int dy[] = {0, 0, 1, -1};

    while (front < rear) {
        Position current = queue[front++];

        if (current.x == end.x && current.y == end.y) {
            return true;
        }

        for (int i = 0; i < 4; i++) {
            int nx = current.x + dx[i];
            int ny = current.y + dy[i];

            if (isValidPosition(nx, ny) && !visited[ny][nx] &&
grid[ny][nx] != OBSTACLE) {
               visited[ny][nx] = true;
               queue[rear++] = (Position){nx, ny};
              return false;
```

4.3 Makefile

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99 -pedantic
TARGET = treasure_game
SOURCES = main.c grid.c player.c pathfinding.c
OBJECTS = $(SOURCES:.c=.o)

all: $(TARGET)

$(TARGET): $(OBJECTS)
      $(CC) $(CFLAGS) -o $(TARGET) $(OBJECTS)

%.o: %.c treasure_game.h
      $(CC) $(CFLAGS) -c $< -o $@
```

```
clean:
	rm -f $(OBJECTS) $(TARGET)

run: $(TARGET)
	./$(TARGET)
```

4.4 Kompilierung und Ausführung

```
make
make run
make clean

gcc -Wall -Wextra -std=c99 -o treasure_game main.c grid.c player.c
pathfinding.c
./treasure_game
```

4.5 Besonderheiten & Optimierungen

Im Laufe der Entwicklung habe ich einige Punkte bewusst optimiert, um die Performance und Benutzererfahrung zu verbessern:

- Die Spielfeld-Generierung wurde durch Wiederholungen so angepasst, dass sie garantiert ein lösbares Szenario erzeugt.
- Ich habe die Eingabe mit `scanf(" %c", &move)` optimiert, damit keine ungewollten Leerzeichen oder Zeilenumbrüche stören.
- Die Hindernisdichte von 20% erwies sich als optimal für den Spielspass.
- Das Makefile wurde so gestaltet, dass Kompilierung, Ausführung und Bereinigung in einem Schritt möglich sind.
- Ich habe auf klare Konsolenrückmeldungen geachtet, damit der Spieler sofort versteht, was passiert.

4.6 Erweiterungspotenzial

Für zukünftige Versionen gäbe es viele interessante Erweiterungsmöglichkeiten:

- Einführung von Leveln mit steigender Schwierigkeit (z. B. größere Spielfelder oder mehr Hindernisse)
- Nutzung von Farben (ANSI Escape Codes), um Hindernisse und den Schatz visuell hervorzuheben
- Speicherung des Spielstands oder Highscores in einer Datei
- Erweiterung des Spiels um mehrere Spieler oder Gegner (KI)
- Ein einfaches Menüsystem für Start, Anleitung und Beenden

5. Test

5.1 Teststrategie

Die Tests wurden manuell in der Konsole durchgeführt. Ziel war die systematische Überprüfung aller Funktionen und Grenzfälle. Für jeden Testfall wurde das erwartete Verhalten definiert und mit dem tatsächlichen Verhalten verglichen.

5.2 Testprotokoll

Testfall	Eingabe/Aktion	Erwartetes Ergebnis	Tatsächliches Ergebnis	Status
T1: Programmstart	./treasure_game	Spielfeld wird angezeigt, Positionen sind zufällig, Schatz ist erreichbar	Spielfeld korrekt generiert, alle Symbole vorhanden	✅ Bestanden
T2: Bewegung nach oben	W + Enter	Spieler bewegt sich eine Zeile nach oben (Y-1)	Spieler korrekt verschoben	✅ Bestanden
T3: Bewegung nach unten	S + Enter	Spieler bewegt sich eine Zeile nach unten (Y+1)	Spieler korrekt verschoben	✅ Bestanden
T4: Bewegung nach links	A + Enter	Spieler bewegt sich eine Spalte nach links (X-1)	Spieler korrekt verschoben	✅ Bestanden
T5: Bewegung nach rechts	D + Enter	Spieler bewegt sich eine Spalte nach rechts (X+1)	Spieler korrekt verschoben	✅ Bestanden
T6: Kollision Hindernis	Bewegung gegen O	Fehlermeldung, keine Bewegung	Korrekte Fehlermeldung, Position unverändert	✅ Bestanden
T7: Kollision Grenze oben	W an oberer Grenze	Fehlermeldung, keine Bewegung	Korrekte Fehlermeldung, Position unverändert	✅ Bestanden
T8: Kollision Grenze unten	S an unterer Grenze	Fehlermeldung, keine Bewegung	Korrekte Fehlermeldung, Position unverändert	✅ Bestanden
T9: Kollision Grenze links	A an linker Grenze	Fehlermeldung, keine Bewegung	Korrekte Fehlermeldung, Position unverändert	✅ Bestanden

T10: Kollision Grenze rechts	D an rechter Grenze	Fehlermeldung, keine Bewegung	Korrekte Fehlermeldung, Position unverändert	✅ Bestanden
T11: Siegbedingung	Auf Schatz T bewegen	Siegesmeldung und Spielende	Korrekte Siegesmeldung, Programm beendet	✅ Bestanden
T12: Ungültige Eingabe	Taste X + Enter	Keine Bewegung	Korrekte Meldung, Spielfeld unverändert	✅ Bestanden
T13: Groß- /Kleinschreibung	w/W/a/A	Beide Varianten funktionieren	Funktioniert korrekt (toupper)	✅ Bestanden
T14: Mehrfache Regenerierung	Mehrfacher Start	Immer erreichbares Spielfeld	Verschiedene Spielfelder, erreichbar	✅ Bestanden
T15: Erreichbarkeit (visuell)	Analyse	Pfad sichtbar	Pfad visuell vorhanden	✅ Bestanden

5.3 Testergebnisse

Getestete Funktionen: 15
Erfolgreiche Tests: 15 (100%)
Fehlgeschlagene Tests: 0 (0%)
Alle definierten Testfälle wurden erfolgreich bestanden. Das Spiel verhält sich stabil und reagiert korrekt auf alle Eingaben. Die Erreichbarkeitsprüfung funktioniert zuverlässig, sodass jedes generierte Spielfeld lösbar ist.

5.4 Entdeckte Probleme und Lösungen

Problem 1: Spielfeld-Generierung kann mehrere Versuche benötigen.
Lösung: MAX_ATTEMPTS und Warnung.

Problem 2: scanf()-Whitespace im Eingabepuffer.
Lösung: Format „ %c“ und Nutzung von toupper().

5.5 Testumgebung

Die Tests wurden auf einem Linux-System (Ubuntu 22.04) durchgeführt, mit dem GCC-Compiler (Version 11.3) und der Standard-Konsole. Alle Funktionen wurden manuell getestet, indem ich verschiedene Eingaben und Spielszenarien ausprobiert habe. Dabei wurde die Spielfeldgröße, Hindernisdichte und das Verhalten bei ungültigen Eingaben gezielt überprüft. Ich habe bewusst Grenzfälle simuliert, um mögliche Fehler frühzeitig zu erkennen.

5.6 Fazit zum Test

Das Spiel lief nach Abschluss der Tests stabil und fehlerfrei. Die Spielfelder wurden immer erreichbar generiert, und es traten keine Abstürze auf. Besonders zufrieden war ich mit der Eingabelogik und der Hindernisgenerierung. Die Konsole reagiert zuverlässig auf alle Eingaben, und die Fehlermeldungen sind klar formuliert. Insgesamt hat der Test gezeigt, dass die Implementierung robust und praxistauglich ist.

6. Lessons Learned

6.1 Erfolgreich angewendete Konzepte

- Modulare Programmierung mit klaren Verantwortlichkeiten
- Praktische Umsetzung von BFS und Queue-Verwaltung
- Saubere Datenstrukturen (2D-Array, Structs) und benannte Konstanten

6.2 Herausforderungen und Lösungsansätze

- Pointer-Übergabe: won-Flag per Referenz modifizieren
- BFS ohne dynamische Speicherverwaltung mittels statischem Array
- Zufall & Erreichbarkeit: Hindernisdichte optimiert, Regenerierungszähler

6.3 Erkenntnisse für zukünftige Projekte

- Frühes, systematisches Testen und Testprotokoll
- Makefile-Automatisierung spart Zeit und vermeidet Fehler
- Git-Versionierung früh einsetzen

6.4 Persönliche Entwicklung

Die Arbeit stärkte das Verständnis für strukturierte Softwareentwicklung, Problemlösung und die Bedeutung sauberer Dokumentation im technischen Umfeld.

6.5 Methodische Erkenntnisse

Ich habe gelernt, wie wichtig es ist, von Anfang an strukturiert vorzugehen. Zu Beginn habe ich die Anforderungen genau gelesen und in einzelne Teilaufgaben zerlegt. Danach habe ich das Grundgerüst programmiert und Schritt für Schritt Funktionen ergänzt. Jede neue Funktion wurde direkt getestet, bevor ich weitergemacht habe. So konnte ich Fehler schnell lokalisieren. Auch die Nutzung eines Makefiles hat mir geholfen, den Überblick zu behalten und Zeit beim Kompilieren zu sparen.

6.6 Persönliche Erkenntnisse

Für mich war diese Arbeit eine sehr gute Gelegenheit, mein Wissen aus den Modulen *Programmiertechnik A und B* in einem realen Projekt anzuwenden.
Allerdings war der Einstieg nicht ganz einfach – am Anfang hatte ich Mühe, die richtige Struktur zu finden und alle Module sinnvoll aufzubauen.
Vor allem das Zusammenspiel zwischen den C-Dateien und Header-Dateien, sowie der Umgang mit Pointern, war anfangs verwirrend.
Auch das Einrichten des Makefiles und das Verständnis, wie der Code beim Kompilieren zusammengeführt wird, hat mir anfangs Schwierigkeiten bereitet.

Mit der Zeit habe ich aber gelernt, Probleme systematisch zu analysieren und gezielt zu lösen. Ich habe gemerkt, dass man beim Programmieren viel Geduld und Ausdauer braucht, und dass Fehler ein ganz normaler Teil des Lernprozesses sind.
Durch schrittweises Testen, Debuggen und die stetige Verbesserung meines Codes konnte ich ein stabiles und sauberes Ergebnis erreichen.
Am meisten Freude hat es mir gemacht, als das Spiel schließlich komplett funktioniert hat – das war der Moment, in dem sich die ganze Mühe ausgezahlt hat.

Rückblickend bin ich stolz darauf, was ich geschafft habe.
Ich habe nicht nur meine technischen Fähigkeiten verbessert, sondern auch gelernt, strukturiert zu arbeiten, Probleme ruhig anzugehen und Verantwortung für den eigenen Lernprozess zu übernehmen.
Diese Praxisarbeit hat mir gezeigt, dass Durchhaltevermögen und Organisation genauso wichtig sind wie das Programmieren selbst.

7. Anhang

7.1 Projektdateien

- `treasure_game.h` – Header-Datei mit Deklarationen und Konstanten
- `main.c` – Hauptprogramm mit Spielschleife
- `grid.c` – Spielfeldgenerierung und -darstellung
- `player.c` – Spielerbewegung und Eingabeverarbeitung
- `pathfinding.c` – BFS-Algorithmus
- `Makefile` – Automatisierte Kompilierung
- `README.md` – Nutzungsanleitung

7.2 Kompilierung und Ausführung

```
Voraussetzungen:
- GCC Compiler (>= 4.9)
```

8. Schluss / Umgebung

Diese Praxisarbeit hat mir gezeigt, wie viel Wissen und Geduld hinter einer scheinbar einfachen Programmidee stecken. Ich konnte meine Kenntnisse in C deutlich vertiefen und habe nun ein viel besseres Verständnis für sauberen, modularen Code. Die Arbeit war eine wertvolle Erfahrung, sowohl fachlich als auch persönlich.

Zürich, 05.10.2025

Unterschrift: _____Patriot Ibraimi_____