Humboldt-Universität zu Berlin
Kultur-, Sozial- und Bildungswissenschaftliche Fakultät
Institut für Musik- und Medienwissenschaften
Seminar: JMP 6502 – Assemblerprogrammieren für Medienwissenschaftler
Wintersemester 2020/21
Dozent: Dr. Dr. Stefan Höltgen

Labyrinth-Rätsel

Die technische Implementierung eines Labyrinth-Rätsels in Assembler 6502

Vorgelegt von:
Angie Ehlert
Mühlenstraße 14
14943 Luckenwalde
Angie.ehlert@hotmail.de

Studiengang:
Medienwissenschaften
Matrikelnummer: 609865

Inhaltsverzeichnis

Inhaltsverzeichnis 1
Abbildungsverzeichnis 2
Einleitung 3
Vorbetrachtung 4
Lösungsmöglichkeiten eines Labyrinthes 6
Technische Implementierung 7
Funktionsweise des Labyrinth Rätsels 9
Auswertung/Ausblicke 17
Literaturverzeichnis 18

Abbildungsverzeichnis

Abbildung 1: Überblick Labyrinth-Rätsel 10
Abbildung 2: Bildschirm Zeichnen 11
Abbildung 3: Initialisierung 11
Abbildung 4: Hauptschleife 12
Abbildung 5: Tastaturabfrage 12
Abbildung 6: Bewegung nach links und rechts 13
Abbildung 7: Bewegung nach oben und unten 14
Abbildung 8: Zielpunkt 14
Abbildung 9: Wand und Kollision 15
Abbildung 10: Aktualisierung der Position 15
Abbildung 11: neue Position zeichnen 15
Abbildung 12: Ziel 16

Einleitung

Jeder Mensch hat zum Begriff des Labyrinths eine genaue Vorstellung. Meistens wird das Labyrinth als ein Rätsel definiert, das in die Irre führt. Viele bringen mit dem Leben eine Art Labyrinth in Verbindung, da man nie genau weiß, wo es hingeht. In unserer Gesellschaft sind Labyrinthe stark etabliert und werden sowohl als Symbole als auch im metaphorischen Sinn gebraucht. Meistens dienen die Lösungen von Labyrinth-Rätseln als ein Spiel, weil dabei die Entspannung im Vordergrund steht. Beispielsweise gibt es in den Gärten der Welt lebensgroße Labyrinthe, die sowohl aus Stein gepflastert als auch als Hecke gepflanzt sind, die die Besucher selbstständig lösen können.

Auch in der virtuellen Welt wie in Computerspielen finden Labyrinthe häufig Anwendung. Dabei sind sie meist Bestandteil der Quests, die der Gamer lösen muss. Durch die immer besser werdende Bildqualität sind die Labyrinthe in Spielen meist dreidimensional dargestellt und es gibt zusätzlich eine Karte, auf der es als Ganzes überblickt werden kann. Was ist nötig, um ein Labyrinth in einem Spiel zu implementieren? Um ebenfalls einen Einblick in die immer komplexer werdenden Technologien zu gewinnen, wird ein Labyrinth-Code in Assembler 6502 geschrieben. Diese Programmiersprache bietet sich an, um einen Einblick in die Funktion der Maschinensprache zu erhalten und somit einen minimalen Teil der Blackbox zu öffnen.

Um einen Labyrinth-Code zu schreiben, ist es wichtig, zuerst zu klären, um was es sich bei einem Labyrinth handelt und welche Arten es gibt. Mit den in der Vorbetrachtung herausgefilterten Eigenschaften von Labyrinthen sowie deren Arten ist es möglich, Merkmale die für das zu entwickelnde Rätsel zu bestimmen. Danach werden Lösungsmöglichkeiten für verschiedene Labyrinth-Arten beschrieben. Folglich werden maschinelle Experimente dargestellt, die sich mit dem Lösen von Labyrinthen beschäftigten und kurz dargestellt, welche Bedeutung diese für die technische Entwicklung hatten. Danach wird das in Assembler implementierte Rätsel genauestens erklärt und Bezüge zu dem theoretischen Teil der Vorbetrachtung gezogen. Dadurch wird versucht deutlich zu machen, welchen Hintergrund der implementierte Code hat. Hier wird auch kurz auf die technischen Möglichkeiten und Eigenschaften von Assembler 6502 eingegangen.

Zum Schluss wird in der Auswertung reflektiert, welche Erkenntnisse aus der Implementierung des Labyrinths in Assembler 6502 gewonnen werden konnten und Ausblicke gegeben, welche weiterführenden Implementierungs- sowie Forschungsmöglichkeiten sich anbieten.

Vorbetrachtung

Um selbst ein Labyrinth anfertigen zu können, ist es nötig zuerst herauszufinden, welche signifikanten Merkmale diese enthalten. Daher wird in diesem Kapitel der Fokus daraufgelegt, welche Merkmale und Eigenschaften ein Labyrinth aufweist.

Ein Labyrinth ist ein Rätsel, dessen Lösung allgemein darin besteht, durch Nachdenken in einem Irrgarten von einem Startpunkt zu einem Zielpunkt zu gelangen. Dabei kann sowohl die Lage des Startpunktes als auch die Lage des Zielpunktes variieren. Start und Ziel können dementsprechend sowohl außen als auch innen liegen. Meist liegt der Startpunkt außen und der Zielpunkt innen. Es gibt aber auch Formen, wo der Startpunkt (A) und das Ziel (B) außen liegen und das Labyrinth ein Hindernis darstellt, dass gelöst werden muss, um von A nach B zu gelangen.

„Oft wird das Labyrinthproblem folgendermaßen gestellt. Man steht zunächst außen, möchte zwar hinein, aber auch garantiert wieder herauskommen.“[1] In einigen Labyrinth-Varianten muss, wenn das Ziel erreicht wurde, wieder der Weg zurück zum Startpunkt gefunden werden. Bei wieder anderen liegt der Fokus darauf den kürzesten Weg zu finden und somit das Labyrinth schnell zu lösen.

Wird nun der Aufbau des Labyrinths betrachtet, so kann es als ein System aus Wegen und Kreuzungen in einem eingegrenzten Raum beschrieben werden. Dabei können diese Systeme in einfach zusammenhängende und mehrfach zusammenhängende Irrgärten unterschieden werden. Beim einfach zusammenhängenden Labyrinth gibt es keine abgesonderten Trennwände und das Labyrinth wirkt geradlinig. Wohingegen mehrfach zusammenhängende durch die Verschachtelung der Wände einen Eindruck der Unübersichtlichkeit vermitteln. [2] Je nach Schwierigkeitsgrad kann durch die Verschachtelung der Wege der Nutzer des Labyrinths in die Irre geführt werden.

> Briefly, a labyrinth consists of passages leading to meeting points with other passages, and at these junctions the wanderer must make a judicious choice of new directions. Such a situation corresponds to a geometric figure called a network or a graph. A graph consists of vertices, corresponding to the junctions of the passageways, and these vertices are connected by the lines or edges of the graph, corresponding to the passages themselves.[3]

Hier wird deutlich, dass es sich bei einem Labyrinth um ein mathematisches System

---

[1] Beutelspracher, Albrecht*: Luftschlösser und Hirngespinste. Bekannte und unbekannte Schätze der Mathematik, ans Licht befördern und mit neuem Glanz versehen*. Braunschweig/Wiesbaden: Friedr. Vieweg und Sohn, 1986. S. 2.

[2] Vgl.: Gardener, Martin: *Mathematische Rätsel und Probleme*. Braunschweig: Friedr. Vieweg & Sohn, Verlag, 1961. S. 121.

[3] Ore, Oystein: „An excursion into labyrinths“, *The Mathematics Teacher* 52/ 5 (1959): S. 367-370. Hier: S. 368.

handelt, in dem versucht wird, sich zu orientieren. Durch die geometrische Betrachtung eines Labyrinthes als Graphen, Linien und Kanten kann die Lösung berechnet werden. Schließlich ergibt sich daraus die Möglichkeit, die gefundene Lösungsstrategie mit einem Algorithmus zu beschreiben.

Labyrinthe können in Form, Größe und dem Material, aus dem sie bestehen, variieren. Dementsprechend ist es möglich, sie als zweidimensionale Figuren auf Papier oder auf dem Boden zu Zeichnen oder sie als dreidimensionale Objekte beispielsweise in einem Feld aus Mais zu mähen. „Other types of mazes, some stone laid, some used as decorative patterns and magic symbols, may be found among many peoples.“[4] Labyrinthe finden sich in verschiedensten Epochen in unterschiedlichen Kulturen wieder. So sehr sie sich auch in Form und Art unterscheiden mögen, ist in ihnen immer die gleiche Aufgabe implementiert: von einem Startpunkt aus ist das Ziel zu finden oder aber aus dem Irrgarten herauszufinden. Betrachtet man den geschichtlichen Kontext der Labyrinthe, lassen sich unterschiedliche Funktionen finden, für deren Zweck sie erschaffen wurden. Einige dienten der Dekoration, andere einem religiösen Ritus oder aber zum Schutz eines Schatzes.[5] Beispielsweise wurden die Gänge im inneren von Pyramiden als Labyrinth errichtet, um den Schatz vor Eindringlingen zu schützen.

Das Labyrinth findet ebenfalls in der griechischen Mythologie Erwähnung. Hier wurde die Lösung des kretischen Labyrinths von Knossos die zentrale Aufgabe des jungen Theseus. Er musste durch das Labyrinth hindurch und den Minotaurus (stierartiges Menschenwesen) töten und wieder hinausfinden. Hilfe erhielt Theseus von Ariadne - die Tochter des Königs Minos, der das Labyrinth errichten ließ – die sich in ihn verliebte und ihm einen Knäuel Faden gab. Mithilfe des Fadens gelang es Theseus, den richtigen Weg durch das Labyrinth zu finden. Der Faden von Ariadne diente ihm dabei als Orientierung, um herauszufinden, in welchen Ecken des Labyrinths er bereits war. So zeigte der Weg ohne Faden, dass er dort noch nicht war, der mit einem Faden, dass er dort schon einmal war, zwei Fäden, dass er sich in einer Sackgasse befindet und 3 Fäden, dass er im Kreis gelaufen ist.[6] Inspiriert von der Lösung mittels eines Ariadnefadens, wurde der Trémaux-Algorithmus abgeleitet, welcher im folgenden Kapitel erläutert wird.

[4] Ore: *An excursion on labyrinths*, S. 367.
[5] Vgl. Gardner: *Mathematische Rätsel und Probleme*, S. 119 f.
[6] Vgl. Zemanek, Heinz: „50 Jahre Kybernetik in Österreich“, *Elektrotechnik und Informationstechnik* 121/5 (2004): S. 171-179. Hier: S. 173.

Lösungsmöglichkeiten eines Labyrinthes

Die Möglichkeiten, um ein Labyrinth-Rätsel lösen zu können, variieren durch deren Beschaffenheit. Dabei gibt es verschiedene Strategien. „Man kann einen Irrgarten schnell lösen, indem man auf dem Papier alle Sackgassen schattiert, bis nur noch ein durchgehender Weg weiß bleibt.“[7] Dieses Verfahren funktioniert allerdings am besten bei einem auf Papier gezeichneten Irrgarten. Im Gegensatz dazu macht sich in dreidimensionalen Labyrinthen die Rechte-Hand-Regel am besten. Hier wird darauf geachtet, die rechte Hand immer an der Wand zu lassen und dem Weg zu folgen. So findet man sicher nicht den kürzesten Weg, aber immer das Ziel und den Weg heraus.[8] Diese Strategie kann auch bei Labyrinthen, die als Ganzes überblickt werden, angewendet werden. Dennoch besteht bei dieser Strategie die Gefahr, im Kreis herumzuirren, wenn der Irrgarten nicht einfach zusammenhängend ist.

Um dem Problem zu entgehen, in eine Endlosschleife zu geraten, bietet der Trémauxs Algorithmus eine weitaus bessere Möglichkeit zur Lösung des Labyrinths. Dieser funktioniert nach der einfachen Regel, an den Kreuzungen bestimmte Markierungen zu setzen, die eine Orientierungsmöglichkeit schaffen. Wird eine Kreuzung zum ersten Mal betreten, so kann man sich den Weg aussuchen und beispielsweise einen Strich als Markierung setzen. Kommt man hingegen an eine bereits bekannte Stelle und der Weg, den man gelaufen ist, war auch bekannt, so kehrt man um.[9] Dieses Verfahren sorgt dafür, dass der Lösende nicht in eine Schleife gerät. Denn sobald man den Weg zurückgegangen ist, wird dieser mit zwei Strichen markiert und verschließt damit den Weg. Dadurch gelingt es, die Schleife zu durchbrechen. Diese Methode zeigt sich als äußerst effektiv, um die Lösung zu finden, jedoch ist sie nicht dafür geeignet, den schnellsten Weg zu finden. Durch die Markierungen behält man schließlich den Überblick im Labyrinth. Das Besondere an diesem Algorithmus ist, dass dieser bei allen Arten von Labyrinthen (zwei- oder dreidimensional) anwendbar ist und immer zum Zielpunkt und wieder zurück zum Ausgangspunkt führt.[10] Auch bei der Lösung von virtuellen Labyrinthen ist es möglich diesen Algorithmus anzuwenden. Dabei kann der bereits gegangene Weg eingezeichnet bleiben, was einer Gedächtnisfunktion entsprechen würde.

Der Pledge-Algorihtmus funktioniert ähnlich wie die Rechte-Hand-Regel und ist nur dahingehend modifiziert, dass bei diesem Algorithmus die Gefahr geringer ist, in eine

[7] Gardener: *Mathematische Rätsel und Probleme*, S. 120.
[8] Vgl. Gardner, *Mathematische Rätsel und Probleme*, S. 121 f.
[9] Vgl. Snapp, Robert R. und Maureen D. Neumann: „An Amazing Algorithm“, *Mathematics Teaching in the Middle School* 20/ 9 (2015): S. 540-547. Hier: S. 543.
[10] Vgl. ebenda.

Schleife zu geraten. Das Besondere an diesem Algorithmus ist, dass es nicht möglich ist, von außen ein Ziel zu finden. Deren Erfolg beruht darauf, dass der Weg von innen nach außen gesucht wird. Bei diesem Algorithmus geht man so lange gerade aus, bis eine Wand erreicht wird, danach wendet man die Eine-Hand-Regel (entweder links oder rechts) an und umrundet das Hindernis so lange, bis die Umdrehungszahl bei 0 liegt. „Wir zählen die Vierteldrehungen, aber mit Richtungen: für eine Linksdrehung zählen wir um 1 hoch, für eine Rechtsdrehung um 1 herunter (auch bei der ersten Rechtsdrehung, die nach dem Auftreffen auf eine Wand ausgeführt wird).“[11] Durch das Zählen der Umdrehungen wird die Orientierung im Irrgarten beibehalten. In anderen Varianten wird der Winkel bestimmt, um das Hindernis zu umrunden.

Einige der hier gezeigten Lösungsstrategien wurden in technischen Geräten zur Lösung von Labyrinthen implementiert. Im folgenden Kapitel werden nun einige Beispiele für deren technische Implementierungen erläutert.

Technische Implementierung

Wissenschaftliche Betrachtungen von Labyrinthen fanden besonders in der Kybernetik und Psychologie Anwendung. Dabei wurde in der Psychologie mithilfe eines Labyrinthes das Lernen von Lebewesen analysiert.[12] In der Kybernetik wurden die gewonnenen Rückschlüsse des Lernens auf Maschinen angewendet. Durch die Analyse der Selbststeuerung von Maschinen durch ein Labyrinth wurden Grundsteine für das Machine Learning gelegt. Dabei bietet sich ein Labyrinth für ein solches Experiment des Lernens hervorragend an, um eine Maschine so zu konzipieren, dass es ihr möglich ist, sich selbstständig durch ein Labyrinth zum Ziel und wieder zurückzubewegen. Daher wurde sich im kybernetischen Zeitalter mit der Lösung des Labyrinth-Problems durch Maschinen beschäftigt. Experimente wie beispielsweise die von Claude E. Shannon gaben den Beweis, dass es möglich ist, ein Labyrinth durch eine Maschine lösen zu lassen. Diese Lösung des Labyrinth-Problems stellte er bereits 1951 auf der Macy Konferenz vor. „Der amerikanische Ingenieur und Mathematiker Claude E. Shannon, der Begründer der statistischen Informationstheorie und Pionier der Schaltalgebra, baute das erste Modell für die automatische Orientierung.“ [13] In diesem Experiment ließ Shannon eine

[11] Klein, Rolf und Tom Kamphans: „Der Pledge-Algorithmus: Wie man im Dunkeln aus einem Labyrinth entkommt“. In: *Taschenbuch der Algorithmen,* hrsg. Von Berthold Vöcking et al. Berlin/Heidelberg: Springer-Verlag, 2008. S.78.
[12] Vgl. Gardner: *Mathematische Rätsel und Probleme*, S. 122.
[13] Zemanek: *50 Jahre Kybernetik in Österreich,* S. 173.

mechanische Maus namens Theseus durch ein Labyrinth laufen, die in der Mitte jeden Feldes stehen blieb und den voraus liegenden Weg abtastete. Ein Sensor, der sich vorne an der Maschine befand, signalisierte, ob es sich bei dem Abgetasteten um einen Weg oder eine Trennwand handelt. Wurde eine Wand festgestellt, so dreht sich die Maschine in die nächste Windrichtung und probierte dies so lange aus, bis ein Weg gefunden wurde.[14] Claude E. Shannon baute mit seiner mechanischen Maus eine erste Maschine, die in der Lage war, sich selbst durch ein Labyrinth zu steuern. Zwar ist diese mechanische Maus nicht in der Lage, beim ersten Durchgang den kürzesten Weg zu finden, aber dennoch kann sie sich den Weg durch ein Relais Speicher merken und findet bei einem weiteren Durchlauf des gleich angeordneten Labyrinthes das Ziel schneller.

> Hat die ‚Maus' einmal ihren Weg zum Ziel gefunden, so ermöglichen Gedächtnis-Elemente es ihr, ein zweites Mal den Irrgarten ohne Fehler zu durchqueren. In Bezug auf *Trémauxs* Methode besagt dies, daß die Maus alle doppelt durchquerten Wege vermeidet und nur Wege benutzt, die sie einmal durchquert hat.[15]

Hier zeigt sich die Signifikanz der Gedächtnisfunktion, die für eine schnelle Lösung des Labyrinths unabdingbar ist. Dennoch war es mit dieser Methode nicht möglich, vom Ziel auch zurück zum Start zu gelangen, da sonst wieder der Suchalgorithmus greifen würde. Ein weiteres Problem ergab sich daraus, dass die Maus in eine Endlosschleife geraten konnte. Dieses Problem wurde dahingehend gelöst, dass ein Schrittzähler eingebaut wurde, durch den die Maus bei einer zu hohen Schrittzahl aus der Schleife ausbrach.[16]

Mit dem Phänomen der Orientierung im Labyrinth befassten sich ebenfalls der Computerpionier Heinz Zemanek und der Ingenieur Richard Eier, die eng zusammenarbeiteten. Richard Eier entwickelte ein Modell der Maus im Labyrinth, das dem von Shannon sehr ähnlich war. Dennoch arbeitete Eier das Konzept weiter aus und implementierte in seinem Modell eine Zusatzfunktion, die dem Konzept des Ariadne-Faden ähnelte.

> Und die Zusatzidee war der Ariadnefaden, über Shannons Modell hinausgehend, in der abstrakten Form eines weiteren Zweibit-Speichers zu jedem Feld (die vier Windrichtungen des Verlassens sind die zwei Bits, die Shannon vorgesehen hatte). Dieser Speicher hält eine der folgenden vier Möglichkeiten fest: (0) das Feld ist noch nicht betreten worden, (1) es liegt ein Faden, das Feld liegt am Weg vom Eingang zum Ziel; (2) es liegen zwei Faden, das Feld gehört daher zu einer Sackgasse; (3) es würden drei Fäden

[14]Vgl. Goldscheider, Peter und Heinz Zemanek: *Computer. Werkzeug der Information.* Berlin, Heidelberg: Springer-Verlag, 1971. S. 23 f.
[15]Gardner: *Mathematische Rätsel und Probleme*, S. 123.
[16]Vgl. Zemanek: *50 Jahre Kybernetik in Österreich*, S. 174.

> zusammenkommen, was eine Schleife zur Folge hatte - die Maus setzt daher eine gedachte Wand und sperrt damit den Weg in die Schleife. Mit der Ariadnefaden-Information gerät die Maus erstens niemals in eine Schleife und zweitens kann sie auch den Weg zum Eingang zurücklaufen - wie Theseus [...].[17]

Hier zeigt sich, dass die Gedächtnisfunktion von Eiers Maus nicht nur den richtigen Weg abspeicherte, sondern auch die Wege, die er einmal, zweimal oder beim Weitergehen sogar dreimal betreten würde. Diese Funktion ist gleichzusetzen mit dem Trémaux-Algorithmus, in dem durch Markierungen festgehalten wird, wie oft welcher Weg betreten wurde. Dadurch kann die Maus auch nicht mehr in eine Endlosschleife geraten, da sobald ein Weg zum dritten Mal betreten werden würde, dieser gemieden wird. Weiterhin ist es der Maus von Richard Eier, im Gegensatz zu Shannons Maus, möglich durch die verbesserte Speicherfunktion, den Weg vom Ziel auch wieder zurück zum Start zu finden.

Funktionsweise des Labyrinth Rätsels

Nachdem geklärt wurde, um was es sich bei einem Labyrinth handelt, wird in diesem Kapitel die Funktionsweise des Labyrinth-Codes erläutert. Dafür ist es notwendig, genauer darauf einzugehen, was charakteristisch für die Programmiersprache Assembler 6502 ist.

Zuerst einmal handelt es sich bei Assembler um eine niedere Programmiersprache, die dazu dient, die Maschinensprache besser zu verstehen.[18] Die Maschinensprache besteht aus 1 (Strom an) und 0 (Strom aus). Da es sich in vielerlei Hinsicht als äußerst schwierig erwies, ein Programm mittels Binärcode zu schreiben, wurde Assembler als Merkhilfe entwickelt. Es handelt sich dabei um die leserliche Form der Maschinensprache. Dementsprechend stellt Assembler eine Brücke zwischen Nutzer und Maschine dar und erlaubt dem Nutzer sogar, Kontakt zur Maschinensprache aufzunehmen und somit Einblicke in die Funktion der Hardware zu erhalten. Deshalb wird beim Umwandeln des Codes in Maschinensprache vom Assemblieren und nicht Kompilieren gesprochen, da Assembler den Code 1:1 in Maschinensprache übersetzt. Die Kürzel (Mnemonics), die die Handhabung der Bytecodes erleichtern, werden Opcodes (operationalcode) genannt und funktionieren als eine Art Lexikon für den Computer.

Die Zahl 6502 gibt Aufschluss über die Prozessorart. Da es sich bei dem 6502 um

[17]Zemanek: *50 Jahre Kybernetik in Österreich*, S. 174.

[18] Anders als die niederen Programmiersprachen befassen sich höhere Programmiersprachen mit spezifischen Problemen und benötigen einen Compiler, um den eingegebenen Code in Maschinensprache umzuwandeln. Beispiele für höhere Programmiersprachen wären JavaSkript und Phyton.

einen einfachen 8-Bit Mikroprozessor mit ca. 80 Befehlen handelt, bietet dieser sich für den ersten Versuch der Implementierung eines Labyrinth-Rätsels an. Das 6502-System besteht aus ca. 3510 Transistoren und hat einen Adressraum von 65536 Byte.

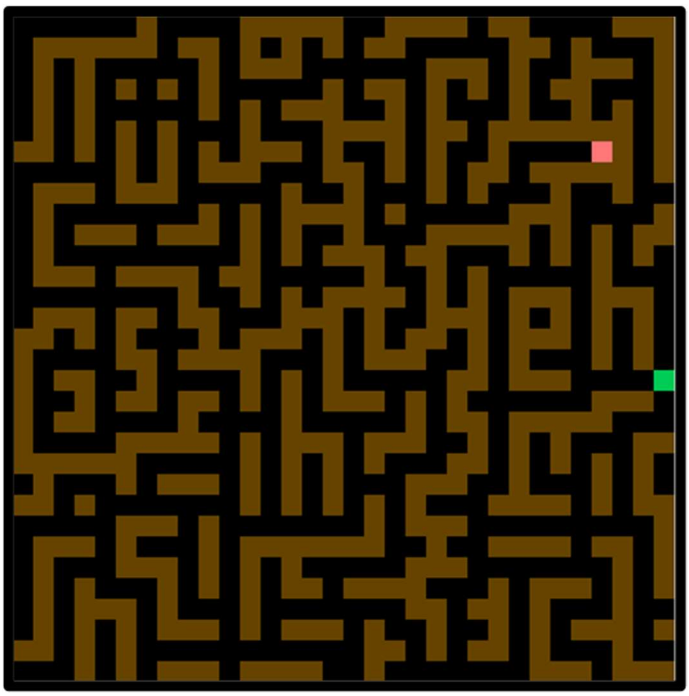

Abbildung 1: Überblick Labyrinth-Rätsel

Vor der Implementierung des Labyrinths war es nötig, einen genauen Blick auf den zur Verfügung stehenden Bildschirmspeicher zu werfen. Dieser besteht aus 4 Pages, in denen jeweils 256 Pixel zur Adressierung gegeben sind. Zum Programmieren stehen demnach 1024 Pixel[19] (das entspricht Pro Page 8x32 Pixel) zur Verfügung. Von der Adresse $0200 bis $05FF ist dementsprechend ein 1024 Byte großer Bildschirmspeicher nutzbar. Durch diesen begrenzen Bildschirmspeicher erfolgt die Programmierung des Labyrinths in zweidimensionaler Form. Um den vorhandenen Bildschirmspeicher effektiv nutzen zu können, bot es sich an, das Labyrinth zuerst auf Papier zu Zeichnen. Dabei konnte bereits darauf geachtet werden, welche Merkmale das Labyrinth beinhalten sollte.

Zu Beginn des Programms wird das Labyrinth auf dem Bildschirm gezeichnet. Die Wände des Labyrinths sind dabei braun und der Weg schwarz dargestellt. Mithilfe des Befehls DCB können Bytes direkt in den Bildschirmspeicher eingefügt werden (siehe Abbildung 2). Dementsprechend steht der Zahlenwert 9 für braun und der Zahlenwert 0 für schwarz. *=$0200 gibt an, ab welcher Bildschirmadresse die Zeichnung beginnt. Es gibt jeweil 32 DCB Zeilen, die 32 Zahlenwerte umfassen und den gesamten

[19]Hierzu im Vergleich, bewegt sich die Auflösung der digitalen Bilder im Megapixelbereich. Eine 4k Auflösung entspricht demnach eine Auflösung von 3840x2160 Pixel.

```
;Bildschirm Zeichnen 32x32

*=$0200
dcb 0,0,0,0,0,0,9,0,0,0,0,9,9,9,9,9,0,0,9,9,9,9,0,9,9,0,0,0,0,9,9,9
dcb 0,9,9,9,9,9,9,9,0,9,9,0,9,0,9,0,9,0,9,9,0,0,0,0,0,9,9,0,9,0,0,0,9
dcb 0,9,0,9,0,0,0,0,0,9,0,9,9,9,0,0,0,0,0,0,9,9,9,0,9,0,0,9,9,9,0,9
dcb 0,9,0,9,0,9,0,9,0,9,0,0,0,0,0,9,0,9,9,0,9,0,9,0,9,0,9,9,0,0,0,9
dcb 0,9,0,9,0,0,0,0,0,9,0,9,0,9,9,9,0,0,9,0,9,0,0,0,9,0,0,9,0,9,0,9
dcb 0,9,0,9,0,9,0,9,0,0,0,9,0,0,0,9,9,9,9,0,9,9,0,9,9,9,9,9,9,9,0,9
dcb 9,9,0,9,0,9,0,9,0,9,0,9,9,9,9,0,9,0,0,9,0,9,0,0,9,0,0,0,0,0,9,0,9
dcb 0,0,0,0,0,9,0,9,0,9,9,9,0,0,0,0,9,9,0,9,0,9,0,9,9,9,0,9,9,9,9,9,0,9
dcb 0,9,9,9,0,9,9,9,0,0,0,0,9,0,0,9,0,0,0,9,0,9,0,0,0,9,0,0,0,9,0,0,9,0
dcb 0,9,0,0,0,0,0,0,0,9,0,0,9,9,9,0,9,0,0,0,0,9,9,9,9,0,0,0,0,9,9,0
dcb 0,9,0,9,9,9,0,9,9,9,0,9,0,9,0,0,9,0,0,0,9,9,9,9,9,0,9,0,9,0,9,9
dcb 0,9,0,0,0,0,0,0,0,0,0,9,0,9,0,9,9,9,0,9,9,9,0,9,0,9,0,9,0,9,0,9,0
dcb 0,9,9,9,0,9,9,9,9,0,9,9,0,0,0,0,9,0,0,9,0,9,0,0,0,0,0,0,9,0,0,0
dcb 0,0,0,0,0,0,0,0,9,0,0,9,0,9,0,9,9,9,9,0,9,0,9,0,9,9,9,0,9,9,9,0
dcb 0,9,9,9,0,9,9,0,9,9,0,0,0,9,9,9,0,9,0,0,9,9,9,0,9,0,9,0,9,0,9,0
dcb 9,9,0,9,0,9,0,0,0,9,0,9,9,9,0,9,0,9,0,9,9,0,9,0,9,9,9,0,9,0,9,0
dcb 9,0,0,0,0,0,9,9,0,9,9,9,9,0,0,0,0,9,0,9,9,9,9,0,0,9,0,9,0,0,0,9,0,0
dcb 9,0,9,9,9,0,0,9,0,0,0,0,9,0,9,0,9,0,9,0,0,0,0,9,9,0,9,9,9,9,0,0,0,0
dcb 9,0,0,9,0,9,9,0,9,9,9,0,9,0,9,0,9,9,9,0,9,0,9,0,0,0,0,0,0,9,9,9,0
dcb 9,0,9,9,0,0,0,0,9,0,0,0,0,9,0,0,0,0,0,0,9,0,9,9,0,9,9,9,9,9,0,0,0
dcb 9,0,0,0,0,9,9,9,9,9,9,0,9,0,9,9,9,0,9,9,9,0,0,9,0,9,0,9,0,9,0,0,0,9,9
dcb 9,9,9,9,9,9,9,0,0,0,0,0,9,0,9,0,9,0,9,0,9,0,0,0,9,9,0,9,0,9,0,9,0,9,0
dcb 0,9,0,0,0,9,0,9,9,9,9,0,9,0,9,0,9,0,9,0,0,0,9,9,9,0,0,9,0,9,0,0,0,9,0,9,0
dcb 9,9,0,9,0,0,0,0,0,0,0,9,0,9,0,9,0,9,0,9,0,9,0,0,0,9,9,9,9,9,0,9,0,9,9
dcb 0,0,0,0,0,9,9,9,9,0,9,0,0,0,0,0,0,0,0,9,0,9,9,9,9,0,0,0,0,0,0,0,0,0,0,9
dcb 0,9,9,9,0,9,0,0,0,9,0,9,9,9,9,9,9,9,9,0,0,9,0,0,9,9,9,9,0,9,9,0,9
dcb 0,9,0,0,0,9,9,9,9,0,9,0,9,0,9,0,0,0,0,0,0,9,9,9,9,0,0,0,0,0,0,0,9,9,0,9
dcb 0,9,0,9,0,0,0,9,0,9,0,9,0,9,0,9,9,9,0,9,0,9,0,9,0,9,0,9,0,9,9,9,0,9,0,9
dcb 0,9,0,9,9,9,0,9,0,0,0,9,0,0,0,0,0,0,0,0,9,9,0,9,9,9,0,0,0,0,9,0,0,0,9,0,0
dcb 0,9,0,9,0,9,0,9,9,9,9,0,9,0,9,9,9,0,9,0,0,9,0,0,9,0,9,0,9,0,9,9,9,9,0,9
dcb 9,9,0,9,0,9,0,0,0,0,0,0,0,0,0,0,0,0,0,9,9,0,9,0,9,9,9,0,9,0,0,0,9,0,0
dcb 0,0,0,9,0,9,0,9,9,9,9,0,9,9,9,9,9,0,0,9,0,0,0,0,0,0,0,9,9,9,0,9,9,9
```

Abbildung 2: Bildschirm Zeichnen

Bildschirmspeicher darstellen. Das Labyrinth besteht aus 476 braunen Bildschirmpunkten. Bei der Erstellung des Labyrinthes wurde darauf geachtet, sowohl die Wände als auch die Wege nur 1 Pixel breit zu lassen. Dabei spielte bei diesem gestalterischen Design nur der ästhetische Faktor eine Rolle. Des Weiteren wurden zusätzlich zum eigentlichen Zielweg weitere Wege implementiert, die die Funktion von Sackgassen erfüllen und somit den Nutzer in die Irre führen können.

```
; init              ; Initialposition der Figur:
lda #$3f            ; rechter Rand oberste Zeile...
sta $10
lda #$04            ; ...auf der 4. Page
sta $11
```

Abbildung 3: Initialisierung

Nach der Zeichnung des Labyrinths erfolgt die Initialisierung des Anfangspunktes (siehe Abbildung 3). Dabei wird $11 für die Page 4 und $10 für die Position $3F auf dieser Page reserviert. Die Initialisierung hilft dabei, den Variablen einen definierten Anfangswert zuzuweisen. Das vereinfacht die Erstellung von weiteren Routinen. Nach der Initialisierung des Startpunktes wird mit dem Befehl JSR zur Subroutine *finishPoint* gesprungen. In dieser Subroutine wird der Zielpunkt auf den Bildschirm gezeichnet.

```
; main
loop:                          ; Hauptschleife
    jsr copyCurrentPosition    ; aktualisiere Position
    jsr checkKeys              ; prüfe Eingabe
    lda #$00                   ; Tastaturpuffer...
    sta $ff                    ; ....löschen
    jsr checkWallCollision     ; prüfe ob in Zielrichtung eine Wand ist
    jsr checkFinish            ; prüfe ob bereits am Ziel angekommen
    jsr drawNewPosition        ; neue Position malen
    jsr storePosition          ; speichere gemachten Schritt

jmp loop                       ; wiederhole Hauptschleife
```

Abbildung 4: Hauptschleife

Im Labyrinth-Rätsel ist eine Hauptroutine implementiert, die am Ende durch den Befehl JMP (jump to address) unendlich wiederholt werden kann (siehe Abbildung 4). Das Hauptprogramm gibt einen oberflächlichen Einblick zur Funktion des Programms. So wurde zuerst die Position aktualisiert und danach wurde die Tasteneingabe überprüft. Um zu gewährleisten, dass die alte Tasteneingabe nicht gespeichert wird, wurde der Tastaturpuffer mithilfe der Befehle LDA #$00 und STA $FF gelöscht. Weiterhin wird die Wandkollision sowie die Erreichung des Ziels überprüft. Danach wird die aktuelle Position aufgezeichnet und der gemachte bzw. der gehemmte Schritt gespeichert. Für die Sprünge aus dem Hauptprogramm in die einzelnen Unterprogramme wurde der Befehl JSR (jump to subroutine) genutzt. Diese Unterprogramme werden dann über den Befehl RTS (return from subroutine) beendet und es wird wieder zum Hauptprogramm gesprungen.

```
checkKeys:
   lda $ff          ; Tastatur abfragen
   cmp #$61         ; a gedrückt?
   beq left         ; Bewegung nach links
   cmp #$73         ; s gedrückt?
   beq down         ; Bewegung nach unten
   cmp #$64         ; d gedrückt?
   beq right        ; Bewegung nach rechts
   cmp #$77         ; w  gedrückt?
   beq up           ; Bewegung nach oben
   rts
```

Abbildung 5: Tastaturabfrage

Farblich ist der Anfangspunkt dunkelgrün durch $05 dargestellt, der sich auf dem Bildschirm an dem Punkt $043F befindet. Mithilfe des ACII-Kodes [20] konnte die Steuerung über die Tastenbelegung W-A-S-D in das Programm implementiert werden (siehe Abbildung 5). In der Subroutine *checkKeys* wird die Tastatur über den Befehl LDA $FF abgefragt. Danach wird mit dem Befehl CMP (compare to accumulator) verglichen, ob es sich bei der Eingabe um eine der Tasten handelt, die zur Steuerung notwendig sind. Mit dem Befehl BEQ (branch on equal to zero) wird bei gedrückter Taste in die Subroutine der jeweiligen Richtung gesprungen. So kann der dunkelgrüne Punkt mit den Tasten w = hoch, s = runter, a = links und d = rechts in Richtung Ziel gesteuert werden.

```
left:               ; Bewegung nach links
    dec $12         ; ein Feld nach links
    lda $12         ; lade Position auf Page
    and #$1f        ; Bit 7-5 maskieren (damit ggf. nur noch 31 übrig bleibt)
    cmp #$1f        ; 31? linker Rand?
    beq collision   ; wenn ja springe zu collision
    rts

right:              ; Bewegung nach rechts

    lda $12         ; lade Position auf Page
    and #$1f        ; Bit 7-5 maskieren (damit ggf. nur noch 31 übrig bleibt)
    cmp #$1f        ; 31? rechter Rand?
    beq collision   ; wenn ja springe zu collision
    inc $12         ; wenn nicht ein Feld nach rechts
    rts
```

Abbildung 6: Bewegung nach links und rechts

Bei der horizontalen Steuerung ist zu beachten, dass je nach Richtung unterschiedliche Befehle benötigt werden (siehe Abbildung 6). Bei einem Schritt nach rechts wird der Befehl INC (increment memory) angewendet und bei einem Schritt nach links der Befehl DEC (decrement memory). Weiterhin wird in beiden Richtungen überprüft, ob der Rand links bzw. rechts schon erreicht wurde. Ist das der Fall, wird mit dem BEQ Befehl in die Subroutine *collision* gesprungen und der Schritt gehemmt. Wenn dies jedoch nicht der Fall ist, wird der Punkt einen Schritt weiter gesetzt.

Um das Pixel eine Zeile höher zu setzen, wird der Befehl SBC #$20 (Subtract with Carry) verwendet, bei diesem werden genau 32 Stellen von der aktuellen Position subtrahiert (siehe Abbildung 7). Beim Befehl ADC #$20 (Add with Carry) werden 32 Stellen zur aktuellen Position addiert, wodurch das Pixel dann genau eine Zeile tiefer gesetzt wird. Ähnlich wie bei der vertikalen Bewegung, wird bei der horizontalen Bewegung eine Überprüfung des Randes vorgenommen. Dabei wird die Zielrichtung mit

[20] „ASCII steht für "American Standard Code for Information Interchange" (Amerikanischer Standardkode zum Informationsaustausch; international als IS0-7-Bit-Kode genormt).“ In: Zaks, Rodnay: Programmierung des 6502. übers. von Bernd Pol. Berkeley u.a.: SYBEX, 1980. S. 36. In dieser Tabelle sind die Codes für die Tastenbelegungen vorgeben.

```
up:                 ; Bewegung nach oben
    lda $12         ; lade Richtung
    sec             ; Carry setzen
    sbc #$20        ; eine Zeile höher
    sta $12
    bcc page_up     ; am obersten Rand?
    rts

page_up:
    dec $13
    lda #$1
    cmp $13         ; oberster Rand?
    beq collision   ; springe zu collison
    rts


down:               ; Bewegung nach unten
    lda $12         ; lade aktuelle Zeile (x)
    clc             ; Carry vorsichtshalber löschen
    adc #$20        ; eine Zeile tiefer
    sta $12
    bcs page_down   ; am unteren Page-Rand?
    rts

page_down:          ; unterster Rand
    inc $13
    lda #$6
    cmp $13         ; schon unterster Rand?
    beq collision   ; ....springe zu collision
    rts
```

Abbildung 7: Bewegung nach oben und unten

den Befehlen BCC (Branch if Carry Clear) oder BCS (Branch if Carry Set) verglichen. In der *up* Routine wurde diesbezüglich mit dem Befehl SEC das Carry Flag[21] gesetzt. Wird bei dem Befehl BCC festgestellt, dass das Carry-Flag gelöscht wurde, springt das Programm in eine weitere Subroutine *page_up*. Hier wird der oberste Rand überprüft und für den Fall, dass dieser erreicht wurde, wird der Schritt gehemmt, indem in die Subroutine *collision* gesprungen wird. Ähnlich verhält es sich mit der *down* Routine, nur das hier mit dem Befehl CLC (Clear Carry Bit), das Carr Flag zuerst gelöscht wird. Wenn mit dem Befehl BCS festgestellt wurde, dass das Carry Flag gesetzt wurde, wird wie zuvor erläutert, in eine Subroutine gesprungen.

```
; roter Punkt als Zielpunkt

finishPoint:
   lda #$0a      ;Lade die Farbe rot in den Akku
   sta $02dc     ;Speichere die Farbe an die Bildschirmadresse $0355
   rts
```

Abbildung 8: Zielpunkt

[21] Carry Flag ist der Übertragsbit, der einen Übertrag anzeigt oder Bits aufnehmen kann.

Das Ziel ist rot $0A markiert und befindet sich an der Bildschirmadresse $02DC (siehe Abbildung 8). Diese Farbe wurde erst mit dem Befehl LDA in den Akku geladen und danach mit dem Befehl STA an der gewünschten Bildschirmadresse gespeichert.

```
collision:
    jsr copyCurrentPosition      ;Kollision erkannt, mache keinen Schritt
    rts

checkWallCollision:
    ldy #$00         ;lade Fake-vektor
    lda ($12),Y      ;aktuelle Position
    cmp #$09         ;vergleiche ob es Wand ist
    beq collision    ;wenn Wand dann springe zu collision
    rts
```

Abbildung 9: Wand und Kollision

Wie in der Ausführung zur Hauptschleife erläutert, wird bei jedem Schritt überprüft, ob es sich bei dem auszuführenden Schritt um eine Wand handelt oder nicht. Dies wird in der Subroutine *checkWallCollision* durchgeführt (siehe Abbildung 9). Hier wird die aktuelle Position mithilfe des Befehls CMP #$09 mit der Farbe Braun verglichen. Handelt es sich um die Farbe braun, wird der Schritt gehemmt und nicht ausgeführt. Diesbezüglich wird dann in die Subroutine *collision* gesprungen.

```
copyCurrentPosition:    ;aktuelle Position in Richtung -> kein Schritt gemacht, Position aktualisiert
  lda $10
  sta $12
  lda $11
  sta $13
  rts
```

Abbildung 10: Aktualisierung der Position

Bei der Feststellung eines Bildschirmrandes wird zur Subroutine *collision* gesprungen. Von dieser Subroutine wird dann weiter in die Subroutine *copyCurrentPosition* gesprungen (siehe Abbildung 10). Hier wird die vorherige Position aktualisiert. Dadurch wird der Schritt gehemmt und das Pixel bleibt an der gleichen Stelle stehen.

```
drawNewPosition:
   lda #$0d         ; lade Farbe hellgrün...
   ldy #$00         ; als alte...
   sta ($10),Y      ; ...Position

   lda $00          ; lade neue Position
   ldy #$00
   sta ($12),Y

   lda #$05       ; lade dunkelgrün...
   sta $00        ; ...als neue Positionfarbe
   rts
```

Abbildung 11: neue Position zeichnen

Wenn jedoch ein Schritt ausgeführt wird, wird dieser in der Subroutine *drawNewPosition* gesetzt (siehe Abbildung 11). Dabei wird die alte Position hellgrün und die neue Position dunkelgrün abgespeichert. Der gegangene Weg wird somit in hellgrün $0D dargestellt, um einen besseren Überblick zu erhalten, in welcher Ecke des Rätsels der Spieler bereits war. Das soll die Orientierung im Labyrinth erleichtern. Werden an dieser Stelle weitere Farbveränderungen in das Labyrinth implementiert, so würde dies eine Anwendung des Témaux Algorithmus ermöglichen. Dafür müssten die Befehle so angegeben werden, dass der Nutzer anhand der Farbe erkennen kann, ob er das Feld ein-, zwei- oder dreimal betreten hat.

```
checkFinish:
   cmp #$0a          ; überprüfe ob der rote Punkt erreicht wurde...
   beq finish        ;...wenn ja gehe zu finish
   rts


finish:

      lda $fe       ; lade Zufallszahl...
      sta $00       ; speichere die Farbe Schwarz
      lda $fe       ; Lade weitere Zufallszahl...
      and #$3
      clc           ; Clear carry
      adc #$2
      sta $01
      lda $05
      ldy #$0
      sta ($00),y
  jmp finish
```

Abbildung 12: Ziel

Bei der Erreichung des roten Zielpunktes wird das Labyrinth durch eine Animation, in der sich das Labyrinth nach und nach auflöst, beendet (siehe Abbildung 12). Hierfür wird mit dem Vergleichsbefehl CMP der ausgeführte Schritt immer wieder mit der Farbe Rot verglichen. Mit dem BEQ (Branch if equal) Befehl wird, wenn das rote Pixel erreicht wurde, zur Unter-Routine „Finish“ gesprungen. Hier wird über eine Zufallszahl $FE ein Wert für die Bildschirmadresse festgelegt, an der ein schwarzer Punkt gesetzt wird und danach ein weiterer zufälliger Wert auf dem Bildschirm gesetzt wird. Die Schleife wiederholt sich und das Labyrinth löst sich nach und nach auf. Von dem Nutzer wird nicht erwartet, an den Startpunkt zurückzukehren. Das Rätsel ist mit der Erreichung des roten Punktes zu Ende.

Versucht man nun die vorher betrachteten Algorithmen auf das implementierte Labyrinth-Rätsel anzuwenden, dann fällt auf, dass einige für deren Lösung geeignet sind und andere nicht. So kann das Labyrinth beispielsweise nicht durch die Rechte-Hand-Regel gelöst werden, da durch dessen Aufbau die Wahrscheinlichkeit sehr groß ist, in eine Schleife zu geraten. Auch der Pledge-Algorihtmus kann in diesem Labyrinth nicht

angewendet werden, da der Startpunkt außen und das Ziel innen liegt. Wird der Code hinsichtlich der Farbveränderung der Felder noch weiter modifiziert, so bietet sich für die Lösung dieses Labyrinths der Trémaux-Algorithmus an.

Auswertung/Ausblicke

Bei der Erstellung des Programmcodes für das Labyrinth-Rätsel wurde deutlich, dass es eine Vielzahl an weiteren Gestaltungsmöglichkeiten gibt. Beispielsweise kann ein Rand um das Labyrinth gezogen werden sowie die Farbwahl verändert werden. Weiterhin könnten noch weitere Level ergänzt werden, die immer schwieriger werden. Zum Beispiel würde das Rätsel schwieriger werden, wenn bei einer Wandkollision die Position wieder an den Anfangspunkt zurückspringt. Oder aber durch die Implementierung eines Schrittzählers, der nur eine bestimmte Anzahl von Schritten zulässt. Weiterhin könnte die alte Position gelöscht werden, sodass keine Gedächtnisfunktion mehr enthalten ist.

Als weiterführende Forschung wäre es möglich, den Code des Labyrinth-Rätsels dahingehend zu verändern, dass die Maschine selbst dieses Rätsel löst. Hierzu bietet sich die Implementierung des Trémaux Algorithmus an. Andererseits könnte man das Labyrinth so verändern, dass ein anderer Lösungsalgorithmus anwendbar ist.

Beim Programmieren des Rätsels hat sich gezeigt, dass es viele verschieden Möglichkeiten der Codierung gibt und es sicherlich von Vorteil ist, vorher schon klare Gedanken zu haben, wie das Programm aufgebaut sein sollte. Weiterhin zeigte sich, dass die Grafik, die im 6502 vorhanden war, nur begrenzte Programmiermöglichkeiten zulässt und nicht mehr zeitgemäß ist. Dennoch ist es für das Lernen und Verstehen von den eigentlichen Prozessen, die in Maschinen vor sich gehen, sehr zu empfehlen. Dadurch, dass Assembler der Maschinensprache sehr nahekommt, konnten einige Prozesse, die im Innern der Maschine vor sich gehen, enthüllt werden.

Auch hat der Exkurs zu den Labyrinthen hat gezeigt, dass sie sich in Form und Gestalt unterschieden. Dennoch ist in ihnen immer die gleiche Aufgabe vorhanden: von einem Startpunkt zu einem Ziel zu gelangen. Dementsprechend gibt es viele Algorithmen und Möglichkeiten zur Lösung von Labyrinthen. Durch die Mathematisierung der Lösungsmöglichkeiten ist es möglich, diese in Maschinen zu implementieren. Beispiele wie die Experimente von Claude Shannon und Richard Eier zeigen, dass bereits in den 60er-Jahren Forschungen in dem Bereich des maschinellen Lernens mittels Labyrinthen unternommen wurden und somit wichtig für die Entwicklung von technologischen Fortschritten waren.

Literaturverzeichnis

Beutelspracher, Albrecht*: Luftschlösser und Hirngespinste. Bekannte und unbekannte Schätze der Mathematik, ans Licht befördern und mit neuem Glanz versehen*. Braunschweig/Wiesbaden: Friedr. Vieweg und Sohn, 1986.

Diaz, Paloma, Ignacio Aedo und Fivos Panetsos: „Labyrinth, an abstract model for hypermedia applications. Discribtion of its static components“, *Information Systems* 22/8 (1997): S. 447-464.

Gardener, Martin: *Mathematische Rätsel und Probleme*. Braunschweig: Friedr. Vieweg & Sohn, Verlag, 1961.

Goldscheider, Peter und Heinz Zemanek: *Computer. Werkzeug der Information.* Berlin/Heidelberg: Springer-Verlag, 1971.

Klein, Rolf und Tom Kamphans: „Der Pledge-Algorithmus: Wie man im Dunkeln aus einem Labyrinth entkommt“. In: *Taschenbuch der Algorithmen,* hrsg. Von Berthold Vöcking et al. Berlin/Heidelberg: Springer-Verlag, 2008.

Ore, Oystein: „An excursion into labyrinths“, *The Mathematics Teacher* 52/ 5 (1959): S. 367-370.

Snapp, Robert R. und Maureen D. Neumann: „An Amazing Algorithm“, *Mathematics Teaching in the Middle School* 20/ 9 (2015): S. 540-547.

Zaks, Rodnay: Programminerung des 6502. übers. von Bernd Pol. Berkeley u.a.: SYBEX, 1980.

Zemanek, Heinz: „Von der Fernmeldetechnik zur Informationstechnik- Richard Eier, die Digitalisierung und sein Weg“, *Elektrotechnik und Informationstechnik* 121/5 (2004): S. 162-164.

Zemanek, Heinz: „50 Jahre Kybernetik in Österreich“, *Elektrotechnik und Informationstechnik* 121/5 (2004): S. 171-179.