14. September 2000
Thomas Dübendorfer

Backtracking mit Heuristiken

(Folie 1)

Inhalt und Ablauf

(Folie 2) Der Vortrag steigt über ein einführendes Beispiel ins Thema **„Backtracking“** ein. Beim Ergründen der Konzepte werden wir auf dieses Beispiel zurückgreifen. Danach werden wir uns fragen, wie der Computer in einem Labyrinth den Weg finden kann und unser neues Wissen zur Lösung anwenden. Die anschliessenden Laufzeitbetrachtungen werden uns den Weg ebnen für ein weiteres Beispiel, die Springerwege. In diesem Zusammenhang werden wir erfahren, was **„Heuristiken“** sind. Als Abschluss werden die wichtigsten Erkenntnisse zusammengefasst.

1. Einführendes Beispiel

Die Begriffe „Backtracking“ und erst recht „Heuristiken“ mögen fremd tönen. Dennoch verwenden viele von uns diese Techniken regelmässig beim Surfen im Datenmeer des Internets. Um uns bewusst zu machen, wie wir beim Suchen nach Informationen auf Webseiten im Internet genau vorgehen, stellen wir uns folgende Aufgabe:

(Folie 3) **Wann wurde über die Initiative „Schweiz ohne Armee“ abgestimmt?**
Da wir die Antwort im Internet auf den Seiten der Bundesverwaltung vermuten, steigen wir direkt auf die Homepage von **www.admin.ch** ein anstatt eine allgemeine Suchmaschine zu verwenden.

(Folie 4) Die Einstiegsseite der schweizerischen Bundesverwaltung lässt uns aus fünf Sprachen auswählen. Wir nehmen der Einfachheit halber „deutsch“.

(Folie 5) Die Folgeseite sieht etwas weniger aufgeräumt aus und präsentiert uns eine Vielzahl von Links. Nachdem wir viele davon verworfen haben, da sie uns wohl nicht direkt zur gesuchten Antwort bringen, klicken wir auf „Volksabstimmungen“ unten rechts.

(Folie 6) Wieder erhalten wir eine Seite in komplett anderem Design. Doch die Lösung scheint nun nahe zu sein. Wir klicken auf „Daten der eidgenössischen Volksabstimmungen“, da wir hoffen, dort unser gesuchtes Abstimmungsdatum zu finden.

(Folie 7) Doch oh Schreck! Wir werden mit über 200 Abstimmungsdaten ohne Textangaben regelrecht eingedeckt. Wir müssten jedes Datum einzeln anklicken, um den zugehörigen Abstimmungstitel lesen zu können. Dies scheint uns bei weitem zu zeitaufwändig, sodass wir mit dem „Back“-Knopf diese Sackgasse fluchtartig verlassen.

(Folie 8) Auf der erscheinenden Seite waren wir bereits, doch seit unserem letzten Besuch hat der angeklickte Link seine Farbe verändert; er ist jetzt als „besucht“ markiert. Dies hilft uns, nicht zweimal in dieselbe Sackgasse zu laufen. Wir verwenden diesmal den Link mit der Beschriftung „Ergebnisse der eidgenössischen Volksabstimmungen“.

(Folie 9) Gefunden! Eine lange Liste mit dem Datum und Titel jeder eidgenössischen Volksabstimmung seit 1848 wird aufgebaut. Mit der Textsuche des Browsers innerhalb einer Webseite finden wir schnell die gesuchte Antwort: Die Volksinitiative „für eine Schweiz ohne Armee und für eine umfassende Friedenspolitik“ hat am **26. Nov. 1989** stattgefunden. Somit haben wir die Aufgabe mit wenigen Klicks und nur einer falschen Abzweigung gelöst.

„Alles längst bekannt“ werden jetzt wohl einige denken. Wir haben aber soeben unbewusst beim Springen durch die Webseiten die Techniken von „Backtracking“ und „Heuristik“ benutzt. Deshalb führt der nächste Abschnitt nun in die Konzepte zu „Backtracking“ ein.

2. Konzepte zu Backtracking

(Folie 10) Eine kompakte Definition zu Backtracking lautet:
„**Backtracking** ist *eine systematische Art der Suche* in einem *vorgegebenen Lösungsraum.*
Wenn eine Teillösung in eine *Sackgasse* führt, dann wird der *jeweils letzte Schritt rückgängig* gemacht."

Das Rückgängigmachen eines Schrittes nennt man „Back-tracking", daher der Name „Backtracking".

Und was hat diese etwas abstrakte Definition nun mit dem Internet Beispiel von vorhin zu tun? Sehr viel wie wir gleich sehen werden:
- Wir klicken *systematisch* die verschiedenen Hyperlinks nacheinander an ohne dabei Links mehrfach zu besuchen.
- Der *vorgegebene Lösungsraum* entspricht den durch Hyperlinks verknüpften Webseiten auf www.admin.ch.
- Der „Back"-Button im Browser bringt uns von einer „*Sackgasse*" zur jeweils vorherigen Seite zurück. Somit wird das Verzweigen auf eine falsche Seite *rückgängig* gemacht.

Lesen Sie nun nochmals die Definition von Backtracking durch. Plötzlich erscheint die Definition in einem neuen Licht und weniger abstrakt.

(Folie 11-1: Die Notation –1,-2 usw. weist auf die Animationsschritte in den Folien hin)
Erinnern wir uns nochmals genau, wie wir bei unserer Suche nach dem Abstimmungsdatum auf www.admin.ch vorgegangen sind:
1. Wir wählen „deutsch" aus den fünf möglichen Sprachen *(11-2)*.
2. Links wie „VBS" *(11-3)*, „Bundesrat" *(11-4)* oder „Kantone online" *(11-5)* haben wir von vornherein verworfen.
3. Wir haben schliesslich „Volksabstimmungen" angeklickt *(11-6)*.
4. Da wir nach dem Abstimmungs*datum* gesucht haben, war es naheliegend den Link „Daten zu den eidgenössischen Volksabstimmungen" anzuklicken *(11-7)*.
5. Leider sind wir dadurch in eine Sackgasse geraten mit einer Unzahl an Daten, aber ohne Abstimmungstitel. Hier haben wir einen „Back-tracking"-Schritt gemacht, indem wir den „Back"-Knopf des Browsers benutzt haben *(11-8)*.
6. Wieder zurück auf der vorherigen Seite zeigte uns der Browser den besuchten Link andersfarbig an, sodass wir systematisch den nächsten Link „Ergebnisse zu den eidgenössischen Volksabstimmungen" auswählten *(11-9)* und schliesslich unsere Antwort gefunden haben.

(Folie 12) Kommen wir nun zu einem weiteren wichtigen Konzept, den **Heuristiken**. Damit nicht alle möglichen Lösungswege ausprobiert werden müssen, verwenden wir Heuristiken. Das sind „Strategien, die *mit höherer Wahrscheinlichkeit* (jedoch *ohne Garantie*) das Auffinden einer *Lösung beschleunigen* sollen." (Quelle: Schüler-Duden „Die Informatik", S. 236, Bibliograph. Institut, Mannheim/Wien/Zürich, 1986)

Die Analogie im Internet Beispiel ist dabei folgende: Wir klicken zuerst diejenigen Hyperlinks an, welche gemäss unserer Einschätzung am direktesten zur Lösung führen. Deshalb haben wir Hyperlinks wie „VBS", „Bundesrat" oder „Kantone online" gar nicht erst angeklickt, sondern haben direkt „Volksabstimmungen" aufgerufen.

Das hat unsere Suche wohl wesentlich *beschleunigt*. Wir hatten jedoch *keine Garantie*, dass wir nicht noch schneller ans Ziel hätten gelangen können.

(Folie 13-1) Fragen wir uns nun, ob wir auf den Webseiten nicht auch anders hätten suchen können. Der Suchbaum enthält als Knoten die „Webseiten" und als Kanten die „Hyperlinks". Wir starten an der Wurzel und verzweigen dann weiter über Hyperlinks.

Dabei können wir nun *Ebene für Ebene* durchsuchen, was wir als **Breitensuche** bezeichnen. Dazu würden wir jeweils von der aktuellen Webseite alle Links in je einem neuen Fenster öffnen *(13-2,3)*. Dann im ersten neuen Fenster wieder alle Links in neuen Fenstern öffnen *(13-4,5)*, dann im zweiten neuen Fenster *(13-6)* usw. Dies führt zu einer Unzahl an offenen Fenstern *(13-7)*.

(13-8) Alternativ können wir auch *möglichst schnell in die Tiefe suchen*, was wir dann als **Tiefensuche** bezeichnen. Wir gehen somit dem ersten Link nach *(13-9)* und wählen auf der erscheinenden Webseite wieder einen Link *(13-10,11,12,13,14)* usw.

Backtracking ist demzufolge **Tiefensuche**.

(Folie 14) Schliesslich können wir unsere nun analysierte Vorgehensweise in einem kompakten, aber allgemeinen Algorithmus mit dem Namen **„FindeLoesung“** aufschreiben.

Ein **allgemeiner Backtracking Algorithmus:**
boolean **FindeLoesung**(int index, Lsg loesung, ...) { ... }

In index steht die aktuelle Schrittzahl, welche für jeden gültigen durchgeführten Schritt ums eins erhöht wird. Die Lösung wird rekursiv über Teillösungen berechnet und in loesung zwischengespeichert. Die Teillösungen loesung werden als Referenz übergeben.

Der Algorithmus arbeitet nach folgendem Muster:

1. Solange es noch neue Teil-Lösungsschritte gibt:
   a) Wähle einen neuen Teil-Lösungsschritt schritt;
      Für die Wahl des nächsten Teil-Lösungsschrittes kann unter Umständen eine **Heuristik** angewendet werden, wie wir noch sehen werden.
   b) Falls schritt gültig ist:
      I) Erweitere loesung um schritt;
      II) Falls loesung vollständig ist, return true, sonst:

```
if (FindeLoesung(index+1,loesung)) { // rekursiv
    return true;     // Lösung gefunden
} else {       // Wir sind in einer Sackgasse
    Mache schritt rückgängig; // Backtracking
}
```

2. Gibt es keinen neuen Teil-Lösungsschritt mehr, so: return false

In unserem Internetbeispiel können wir als schritt das Klicken auf einen Hyperlink nehmen. Die loesung wäre dann der durch Klicken zurückzulegende Weg von der Start-Webseite zur Ziel-Webseite, welche die gesuchte Information enthält.

Wir kennen nun die „Regeln“ wie Backtracking als Algorithmus funktioniert. Allerdings empfiehlt es sich, zuerst ein konkretes Beispiel damit durchzuspielen. Erst dann wird man den Algorithmus wirklich verstehen.

3. Eingesperrt im Labyrinth

a) Der Backtracking-Ansatz

(Folie 15-1) Um garantiert einen Weg vom Start zum Ziel zu finden, wählen wir den Backtracking-Ansatz als unsere Lösungsstrategie**:**

- *Systematisch* vom aktuellen Feld im Labyrinth nach 1. oben *(15-2)*, 2. rechts *(15-3)*, 3. unten *(15-4)*, und 4. links *(15-5)* abzweigen
- *Besuchte* Felder *markieren*
- In *Sackgassen* die Züge zurücknehmen (*Backtracking*)

(Folie 16-1) Wir sind die sportliche Person in der Mitte des Labyrinths. Um die Schritte besser beschreiben zu können, legen wir ein (x,y)-Koordinatensystem über das Labyrinth *(16-2)*. Zusätzlich markieren wir die besuchten Felder braun. Wir zweigen systematisch ab nach 1. oben, 2. rechts, 3. unten, 4. links. Wir starten somit nach rechts, da uns oben eine Wand den Weg versperrt *(16-3)*. Von Feld (2,1) zweigen wir nach oben *(16-4)* und von dort nach links ab *(16-5)*. Dort stellen wir fest, dass wir in einer Sackgasse sind *(16-6)*; wir können keine weiteren unbesuchten Felder mehr erreichen und sind auch noch nicht am Ziel. Wir nehmen deshalb den letzten und vorletzten Schritt zurück *(16-7,8)*. Wir stehen nun wieder auf Feld (2,1) und zweigen diesmal nach unten ab, da wir früher bereits nach oben abgezweigt sind und da wir nach rechts das Feld verlassen würden *(16-9)*. Nun geht alles ganz schnell *(16-10,11,12,13)* und schon sind wir am Ziel (0,0) angelangt. Rechts neben dem Labyrinth steht unser Lösungsbaum und wir stellen fest, dass wir tatsächlich eine Tiefensuche durchgeführt haben.

(Folie 17) Nachdem wir glücklich den Ausgang des Labyrinths erreicht haben, erinnern wir uns nochmals an den allgemeinen Backtracking Algorithmus. Diesmal ist er nicht mehr ausformuliert, sondern in Java- und Pseudocode verfasst. Dies erleichtert uns eine spätere Übernahme in ein Programm.

```
boolean FindeLoesung(int index, Lsg loesung, ...) {
        // Allgemeiner Backtracking Algorithmus
        // index = Schrittzahl, loesung = Referenz auf Teillösung
        while (es gibt noch neue Teil-Lösungsschritte) {
                Wähle einen neuen Teil-Lösungsschritt schritt; // Heuristik
                if (schritt ist gültig) {
                        Erweitere loesung um schritt;

                        if (loesung noch nicht vollständig) {
                                // rekursiver Aufruf von FindeLoesung
                                if (FindeLoesung(index+1,loesung,...)) {
                                        return true; // Lösung gefunden
                                } else { // wir sind in einer Sackgasse
                                        Mache schritt rückgängig; // Backtracking
                                }
                        } else return true; // Lösung gefunden -> fertig
                }
        } return false;
} // Bei true als Rückgabewert steht die Lösung in loesung
```

Im Labyrinthbeispiel entspricht ein gültiger schritt der Wahl einer möglichen Abzweigungsrichtung vom aktuellen Feld aus. In loesung steht zuletzt der gefundene Weg vom Start zum Ziel.

Wir haben nun eine gute Grundlage, um selbst Backtracking zur Lösung eines Problems verwenden zu können. Als nächstes benutzen wir den Computer dazu, um ein beliebiges Labyrinth vorlösen zu lassen.

b) Der Computer zeigt den Weg

(Folie 18) Wir schreiben in Java-Pseudocode eine Backtracking Prozedur

boolean **FindeLoesung**(int index, Lsg loesung, int aktX, int aktY)

um in einem Labyrinth mit KxL Feldern vom **Start** zum **Ziel** einen Weg zu finden.

Mit index bezeichnen wir die aktuelle Schrittzahl und in loesung übergeben wir eine Referenz auf die aktuelle Teillösung. Das Parameterpaar (aktX, aktY) bezeichnet die aktuelle Feldposition. Die Prozedur **FindeLoesung** ruft sich rekursiv auf. Man beachte, dass der Parameter loesung *by reference* übergeben wird und somit nur einmal Speicherplatz belegt. Wertänderungen von loesung sind deshalb auch in der aufrufenden Prozedur sichtbar sind.

Beim ersten Aufruf von FindeLoesung wird das Startfeld in (aktX, aktY) übergeben. Während der Suche wird bei den rekursiven Aufrufen das aktuelle Feld in (aktX, aktY) übergeben. Das Programm weiss erst dann, dass es am Ziel ist, wenn die Prozedur **ausgangGefunden**(aktX, aktY) „true" zurückgibt. In diesem Fall steht in loesung der gefundene Weg und FindeLoesung liefert „true" zurück. Wird hingegen kein Weg gefunden, so liefert FindeLoesung „false" zurück.

(Folie 19) Links sehen wir ein Beispiel eines 3x3 Labyrinthes. Wir starten im Zentrum und wollen zum Ziel gelangen. Die zugehörige Lösung soll von der Prozedur FindeLoesung als Pfad berechnet werden. Das Resultat soll in einem zweidimensionalen Array loesung.feld[][] vom Typ Integer stehen. Den Feldwert loesung.feld[x][y] an der Position (x,y) definieren wir als

-1, wenn das Feld als Sackgasse erkannt wurde
0, wenn das Feld nicht besucht wurde
>0, wenn das Feld zum Lösungsweg gehört

Die Feldwerte des Lösungsweges geben die Besuchsreihenfolge an.

Wie wir auf der Abbildung sehen, entspricht die Form des Lösungsweges einer Rechtsspirale.

(Folie 20) Damit es uns leichter fällt, den Backtracking Algorithmus zu unserem Labyrinth-Problem zu verstehen, wählen wir nun folgendes Vorgehen:
Zuerst gibt es eine kleine Demonstration des animierten Algorithmus. Danach betrachten wir den Algorithmus als Java-Pseudocode. Anschliessend schauen wir uns erneut die Demonstration des animierten Algorithmus an und werden uns dabei an den Pseudocode erinnern. Nach dem Vortrag erhalten Sie Gelegenheit, die vollständige Java-Implementierung von FindeLoesung zu studieren.

Demonstration des animierten Algorithmus:
Das Java Programm Labyrinth kann entweder direkt von der Webseite gestartet werden oder man kann es herunterladen und mit dem Befehl „java Labyrinth" starten. Für die spätere erneute Demonstration lassen wir das Programm offen, damit wir das gleiche Labyrinth erneut verwenden können.

(Folie 21) Sie sehen den ersten Teil des Backtracking Algorithmus zum Labyrinth, implementiert in Java-Pseudocode. Die Parameter wurden bereits früher besprochen.
Wir stehen beim Aufruf auf dem Feld (aktX, aktY) und wollen zur Zielposition gelangen, an welcher die Prozedur **ausgangGefunden**(aktX, aktY) „true" zurückgibt. Solange noch nicht alle Richtungen ausprobiert wurden wiederholen wir folgende Schritte:
Wir wählen als neuen Teil-Lösungsschritt eine neue Richtung, d.h. zuerst nach oben, dann nach rechts, dann nach unten und im letzten Versuch nach links und merken uns die aktuelle Richtung in der Variable schritt. Sofern wir eine Strategie kennen, um schneller ans Ziel zu kommen, könnten wir bei der Reihenfolge zur Richtungswahl eine Heuristik anwenden. Im allgemeinen Labyrinth sind uns jedoch keine guten Heuristiken bekannt. Nun stellen wir für den gewählten Schritt fest, ob er auch gültig ist und speichern dies als Wahrheitswert in der Variablen „ok".

Der Schritt ist gültig, wenn er
- innerhalb des Labyrinth-Spielfeldes bleibt,
- nicht durch eine Wand führt und
- nicht auf ein bereits besuchtes Feld führt.

(Folie 22) Sofern der Schritt nun also gültig ist, erweitern wir unsere Teillösung um den Schritt und berechnen die neue Position (neuX,neuY). Das Feld an der neuen Position wird mit der aktuellen Schrittzahl, welche in der Variable index steht, markiert.

Ist die Lösung noch nicht vollständig, d.h. die Prozedur **ausgangGefunden**(aktX, aktY) liefert „false" zurück, so rufen wir die Prozedur FindeLoesung mit den Parametern index+1, loesung und (neuX,neuY) rekursiv auf. Wir suchen damit eine Lösung von Feld (neuX,neuY) zum Ziel. Wird ein solcher Weg gefunden, geben wir „true" zurück. Ansonsten müssen wir den Schritt rückgängig machen und markieren das Feld (neuX,neuY) als „Sackgasse". Das Zurücknehmen des Schrittes wird als „Backtracking" bezeichnet, wie wir gesehen haben.

Im „else"-Zweig zum if-Test, ob die Lösung vollständig ist, geben wir „true" zurück. Dieses „true" wird dann durch alle rekursiv aufgerufenen Prozeduren hindurch weitergegeben, wenn diese sich beenden. Die aktuelle Rekursionstiefe entspricht dem aktuellen Wert der Schrittzahl index.

Zuletzt steht im Algorithmus „return false", da wir nach Verlassen der while-Schleife keine weiteren Lösungsschritte mehr tätigen können, weil wir bereits alle möglichen Schritte vom aktuellen Feld aus erfolglos durchprobiert haben.

Nachdem wir nun den Algorithmus formuliert haben, werden wir die erneute Demonstration zum animierten Algorithmus noch besser verstehen.

Erneute Demonstration des animierten Algorithmus:
Das Java Programm Labyrinth verfügt im Menü „Animation" über einen Befehl „Weg aus Labyrinth entfernen". Damit wird der bei der ersten Demonstration gefundene Lösungsweg gelöscht und kann erneut mit „Weg finden mit Animation" gesucht werden.

4. Laufzeit

(Folie 23) Bei jedem Algorithmus sollten wir uns fragen, wie schnell er ein gegebenenes Problem lösen kann. Liefert er keine Antwort innert nützlicher Zeit, so ist ein Algorithmus praktisch unbrauchbar.

Bei der Tiefensuche werden bei
- max. k möglichen Verzweigungen von jeder Teillösung aus und
- einem Lösungsbaum mit maximaler Tiefe von n

im schlechtesten Fall $1 + k + k^2 + k^3 + ... + k^n = (k^{n+1} - 1) / (k - 1) = \mathbf{O(k^n)}$ Knoten im Lösungsbaum erweitert.

Im Labyrinthbeispiel gab es pro Schritt maximal k=4 mögliche Verzweigungen (oben, rechts, unten oder links). Die maximale Tiefe n entsprach der Weglänge des Lösungsweges.

Im Internetbeispiel entsprachen die Anzahl Hyperlinks pro Webseite der Anzahl k maximal möglicher Verzweigungen von jeder Teillösung aus. Die maximale Tiefe n des Lösungsbaumes war gleich der Anzahl Mausklicks auf dem direkten Weg von der Start- zur Ziel-Webseite.

(Folie 24) Die Tiefensuche und somit und auch Backtracking haben im schlechtesten Fall mit $\mathbf{O(k^n)}$ eine **exponentielle Laufzeit**. Bei grosser Suchtiefe n und Verzweigungsgrad k>1 dauert die Suche somit oft sehr lange.
In der Komplexitätstheorie sagt man, dass Algorithmen mit einer nicht polynomiellen Laufzeit zu langsam sind. Für Probleme mit grosser Suchtiefe n wird Backtracking deshalb zu lange brauchen wie das Beispiel eindrücklich zeigt: Bei einem Verzweigungsgrad k=10, einer Rechengeschwindigkeit von 1000 Knoten pro Sekunde und einer Suchtiefe von 12 brauchen wir im schlimmsten Fall 35 Jahre auf eine Lösung zu warten. Zum Glück können aber gute Heuristiken die Suche erheblich beschleunigen.

5. Springerwege

(Folie 25) Das Problem:
Ein **Springerweg** ist ein Pfad aus Springerzügen auf einem Schachbrett, der jedes Feld genau einmal besucht.

Wir betrachten die Aufgabenstellung etwas genauer:
Ein *Springer* kann max. *8 mögliche Orte* anspringen, wie wir aus der Abbildung entnehmen können. Ein *Zug* ist *gültig*, wenn das neue Feld innerhalb des Spielfeldes liegt und unbesucht ist. Der Springer soll systematisch in der angegebenen Reihenfolge springen.

Wir merken uns den bisherigen Pfad auf dem Schachbrett in einem zweidimensionalen Array of Integers, wobei wir in den Feldern die *„Besuchsreihenfolge"* speichern.

Sofort fällt uns auf, dass wir mit der systematischen Suche von Backtracking das Problem lösen können.

a) Lösungsansatz mit Backtracking

(Folie 26) Der **Backtracking Algorithmus** geht nach folgendem Muster vor:
- Systematisch gemäss vorgegebener Reihenfolge springen
- Bei „Sackgassen" Sprünge zurücknehmen (Backtracking)

Wir haben einen Springerweg, sobald die Weglänge der Anzahl Felder auf dem Schachbrett entspricht. Somit scheint das Problem schon fast gelöst zu sein. Wenn wir uns jedoch an die Laufzeitbetrachtungen erinnern und dabei den *Verzweigungsgrad* k=8 (d.h. Anzahl mögliche Springerzüge) und eine *Suchtiefe* von n=64 (d.h. Anzahl Felder des 8x8-Schachbrettes) annehmen so können wir erahnen, welche unerfreuliche Erkenntnis uns erwartet.

Die Schachbrett-Abbildung zeigt einen Springerweg auf einem Schachbrett der Grösse 6x6. Begonnen wurde unten links. Die einzelnen Felder sind in der Besuchsreihenfolge markiert. Wir sehen, dass jede der Nummern 1 bis 36 in der Abbildung genau einmal vorkommt. Weiter erkennen wir, dass die Nummern durch Springerzüge erzeugt wurden.

b) Heuristiken zur Optimierung

(Folie 27) Wir stellen fest: Bei dieser Methode kann das Finden einer Lösung schon für ein 8x8-Schachbrett *mehrere Tage* dauern!

Der Grund ist folgender: Wir suchen zwar systematisch, aber wenig intelligent! Wie die Abbildung zeigt, springt unser Algorithmus auf dem 8x8 Schachbrett von Schritt 17 aus zur roten Position 18. Dadurch erzwingen wir unbewusst, dass die Ecke mit der grünen 18 als letztes Feld auf dem Springerweg besucht werden muss.

Eine verbesserte **Strategie** lautet: Immer **in die Ecke springen**, wenn möglich! Dadurch verhindern wir ein langes nutzloses Suchen, da wir die schlecht erreichbaren Ecken möglichst früh besuchen und somit nicht erst kurz vor Schluss feststellen müssen, dass wir eine oder mehrere unbesuchte Ecken gar nicht mehr anspringen können. Dies macht den Algorithmus schon erheblich schneller. Anstatt in Tagen finden wir einen Springerwege für das 8x8-Schachbrett schon in wenigen Sekunden. Bei grösseren Schachbrettern beschleunigt diese Heuristik aber zu wenig.

(Folie 28) Das Problem der Springerwege ist schon recht alt.

Heuristik von Warnsdorf (1823):
Es muss immer auf das Feld gesprungen werden, welches am schlechtesten erreichbar ist.

Als *Mass* für die schlechte Erreichbarkeit eines Feldes dienen die Anzahl Felder, welche man von diesem in einem Sprung erreichen kann. Je weniger, um so schlechter erreichbar ist ein Feld.

Wir bauen diese Regel ein:
Anstatt in der vorgegebenen Reihenfolge zu springen, berechnen wir jeweils die Erreichbarkeit aller vom aktuellen Ort anspringbarer Felder und springen zuerst zum am schlechtest erreichbaren.

Diese Heuristik macht unser Programm „intelligenter" und dadurch schneller.

(Folie 29) Die *Auswirkungen* der Heuristik von Warnsdorf sind frappant:
- Springerweg für 8x8 in unter 1 Sekunde.
- Springerweg für 50x50 in wenigen Sekunden.
- Springerweg ab ca. 60x60 ziemlich langsam.

Der Grund ist folgender: Bis 56x56 Felder brauchen wir dank der Heuristik von Warnsdorf keinen einzigen Backtracking-Schritt zu machen, wie eine Analyse des Algorithmus zeigt. Der Aufwand zur Bestimmung der Erreichbarkeiten der Felder macht sich also mehr als bezahlt.

Bemerkung: Es gibt nicht nur einen möglichen Springerweg auf dem 8x8-Schachbrett, sondern 33,4 Billionen oder genauer 33‘439‘123‘484‘294. (Quelle: ECCC TR95-047, Electronic Colloquium on Computational Complexity, 1995, M. Löbbing, I. Wegener, http://www.eccc.uni-trier.de/eccc/)

6. Zusammenfassung

(Folie 30) Fassen wir nun als Erinnerungsstütze die wichtigsten Erkenntnisse zu Backtracking zusammen.

Backtracking

- ist eine **systematische** Suchstrategie und findet deshalb immer eine **optimale Lösung**, sofern vorhanden, und sucht höchstens einmal in der gleichen „Sackgasse“
- ist einfach zu implementieren mit **Rekursion**
- macht eine **Tiefensuche** im Lösungsbaum
- hat im schlechtesten Fall eine exponentielle **Laufzeit $O(k^n)$** und ist deswegen primär für kleine Probleme geeignet
- erlaubt Wissen über ein Problem in Form einer **Heuristik** zu nutzen, um den Suchraum einzuschränken und die Suche dadurch zu beschleunigen

Die Techniken von Backtracking und Heuristik finden nicht nur bei Spielen auf dem Schachbrett wie Springerwege, Schach oder beim „n Damen Problem“ ihre Anwendung, sondern auch in der Spieltheorie allgemein, bei einigen Routenplanern und bei allen Problemen, wo eine systematische Lösungssuche unter Zuhilfenahme von Heuristiken sinnvoll erscheint.

Ich danke für Ihre Aufmerksamkeit.

Im Anhang sind für die Anwendung und Vertiefung des Stoffes zwei Übungen zu Backtracking aufgeführt. Sowohl zum „Labyrinth“ wie auch zum „n Damen Problem“ ist ein Java Programm mit Animation des Algorithmus inklusive Java-Sourcecode vorhanden.