

Zusammenfassung

Artificial Intelligence

Simon Erni
Simon Neidhart

26. Januar 2016

0 Einführung

0.1 Definition Artificial Intelligence

Intelligenz ist die Fähigkeit, Ziele zu definieren und Verhalten zu entwickeln, diese auch zu erreichen. In AI gibt es zwei Teilbereiche; Einerseits versucht man intelligente 'Agenten' zu bauen, andererseits versucht man die menschliche Intelligenz zu verstehen.

AI besteht aus den 3 Layern; Wahrnehmung, Kognition und Aktion.

0.2 Symbolisch & Subsymbolische Repräsentationen

Symbolische Repräsentationen bedeuten, dass man z.B. für ein Objekt Stuhl Definitionen sucht, um diesen zu beschreiben. (z.B. er hat eine horizontale Oberfläche zum Draufsitzen sowie eine vertikale für den Rücken usw.).

Subsymbolische Repräsentationen bedeuten, dass man keine konkrete Definition erstellt, sondern dem System einfach Beispiele des Objekts zeigt und es selbst das Muster erlernt, wie denn jetzt ein Stuhl aussieht. So bringt man auch Kindern etwas bei.

0.3 Knowledge-Intensive vs Knowledge Sparse Methods

Knowledge Sparse beschreibt ein Gebiet, bei dem einige Axiome (Gesetze) bekannt sind, aber sonst nicht gerade viel. Beispielsweise benutzt der Google Such Algorithmus Indexe und den Page Rank Algorithmus.

Knowledge Intensive bezeichnet einen Ansatz, bei dem sehr vieles über das Problem bekannt ist, d.h. ein Schachspiel-Eröffnungs-Buch hat jede Kombination von Eröffnungsspielen analysiert.

1 Rationale Agenten

1.1 Eigenschaften Rationale Agenten

Ein rationaler Agent ist ein System mit folgenden 4 Eigenschaften, resp. Fähigkeiten:

1. Es kann die Umwelt wahrnehmen
2. Observationen werden eingesetzt um Entscheidungen zu fällen
3. Entscheidungen resultieren in einer Aktion
4. Alle Entscheidungen sind rational

Der letzte Punkt bedeutet, dass immer die Option gewählt wird, bei dem das System den meisten Nutzen hat. Das heisst also, dass ein rationaler Agent ein Belohnungssystem haben muss. Rationales Verhalten kann somit durch rationales Denken erreicht werden.

Wenn die Umwelt aber komplex ist, wird es schwierig, alle nötigen Informationen zu verarbeiten, weswegen man in diesen Situationen auch von 'eingeschränkter Rationalität' spricht.

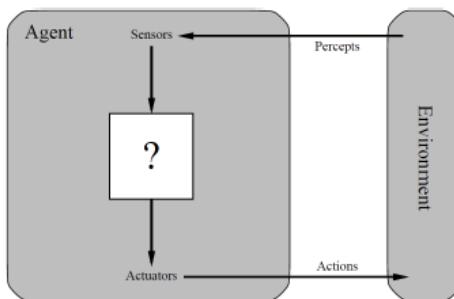


Abbildung 1.1: Übersicht Aufbau rationaler Agent

Wie in Abbildung 1.1 gezeigt, nimmt ein Agent Informationen über seine Umgebung mit Sensoren auf, verarbeitet diese irgendwie (dargestellt durch das Fragezeichen) und reagiert darauf über seine Aktoren.

1.2 Beispiele für rationale Agenten

Ein Mensch hätte also z.B. Augen, Tastsinn & Geruchssinn als Sensoren, und Arme / Beine oder Sprache als Aktoren. Das Gehirn wäre seine Kognition.

Ein Saugroboter wie in Bild 1.2 könnte z.B. wahrnehmen, ob er in Feld *A* oder *B* ist und könnte feststellen, ob das aktuelle Feld dreckig ist oder nicht. Er könnte dann als Aktionen entweder Saugen, das Feld wechseln oder auch nichts tun.

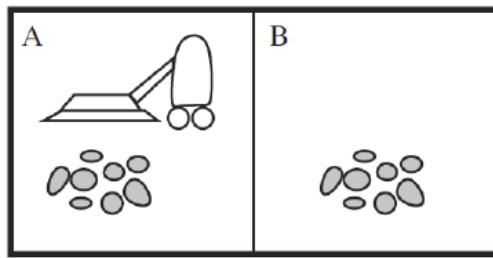


Abbildung 1.2: Rationaler Agent Saugroboter

1.3 Modellierung / Struktur

Alle Agenten, die man modelliert, haben folgende Eigenschaften: Sie haben eine **Sequenz von Informationen** über ihre Umgebung. Dabei muss man sich entscheiden, ob er auch irgendwann etwas vergisst. Das zweite ist die **Agenten Funktion**. Sie beschreibt das Mapping von einer Sequenz von Informationen über die Umgebung zu einer Aktion. Das **Agenten Programm** ist dann die Implementation von der Agenten Funktion und für jeden Agenten natürlich spezifisch. Die **Architektur** eines Agenten bietet Schnittstellen zur Umgebung (Aktionen / Sensoren).

Agenten haben zudem folgende Teilprogramme:

- **Erkundung** Der Agent nimmt Informationen über die Umgebung auf durch Erkundungsaktionen.
- **Lernen** Der Agent lernt über Beobachtungen dazu.
- **Autonomie** Der Agent kann mit falschen oder unvollständigen Informationen umgehen.

1.4 Performance

Die Leistung (Performance) eines rationalen Agenten misst man *von aussen* mittels Bewertungskriterien. Beispielsweise für den Staubsauger-Agenten - Quadratmeter pro Stunde, Energieverbrauch, ... Man misst sie von aussen, weil die gewählten Aktionen eines Agenten Einfluss auf die Umgebung haben kann, oder eben auch nicht. Das kann der rationale Agent ja auch gar nicht wissen.

Eine optimale Leistung ist oft unerreichbar - zu komplexes Problem oder alle benötigten Informationen sind nicht verfügbar. Daher kann ein rationaler Agent mittels seines Wissens und seiner Beobachtungen nur die *erwartete* Performance maximieren. Ein **allwissender Agent** hingegen kann auch die *reale* Performance verbessern - weil er ja genau weiß, was passieren wird wenn er Aktion X ausführt.

1.4.1 Definition

Für jede mögliche Sequenz von Beobachtungen soll ein rationaler Agent eine Aktion auswählen, welche basierend auf seinen Beobachtungen und Wissens die erwartete Performance maximiert.

1.5 Umgebungen

Für jeden Agenten sind natürlich folgende Eigenschaften definiert;

1. Sensoren
2. Aktoren

3. Umgebung

4. Bewertungskriterium

Die Umgebung kann nach folgenden Eigenschaften charakterisiert werden;

Zugänglich vs. Unzugänglich Ist die Umgebung für die Sensoren zugänglich? d.h. können alle relevanten Aspekte der Umgebung von Sensoren erfasst werden?

Deterministisch vs. Stochastisch Deterministisch meint, dass der nächste Zustand der Umgebung komplett von der Aktion des Agenten abhängig ist. Stochastisch bedeutet, dass sie der Realität entspricht.

Episodisch vs Sequentiell Episodisch bedeutet, dass die Qualität einer Aktion nur anhand der aktuellen Perzeption der Umgebung bestimmt ist - d.h. z.B. das Sortieren von Objekten. Sequentiell wäre das Spielen von Schach - dort ist die aktuelle Aktion abhängig von der vorherig gewählten.

Statisch vs. Dynamisch Verändert sich die Umgebung während der Agent überlegt?

Diskret vs. Kontinuierlich Ist die Umgebung diskret (Schachspiel) oder kontinuierlich (Roboter bewegt sich im Raum)?

Einzel-Agent vs. Multi-Agent Gibt es mehrere Agenten? Sind sie konkurrierend / kooperierend?

Wissend vs. Unwissend Wie viel weiss der Roboter zu Beginn?

1.6 Typen von Agenten

1.6.1 Simple Reflex

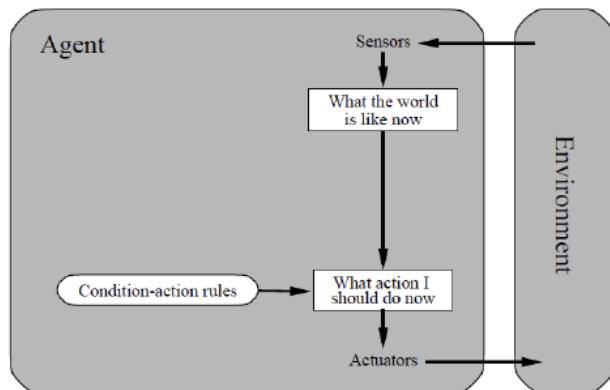


Abbildung 1.3: Reflex Agent

Reflex Agenten nehmen den Input aus der Welt, interpretieren ihn (z.B. bei Video-Bildern) und leiten daraus ihre nächste Aktion ab. z.B. Auto bremst - ich bremse. Heisse Herdplatte - Hand weg.

Es können so auch unendliche Wiederholungen entstehen, wenn die Umgebung nicht ganz-erfassbar ist. Das ist lösbar mittels Zufallsgeneratoren. So kann ein Staubsauger, der einfach immer nach links gehen würde, auch einmal nach rechts gehen. Siehe als Übersicht auch Abbildung 1.3;

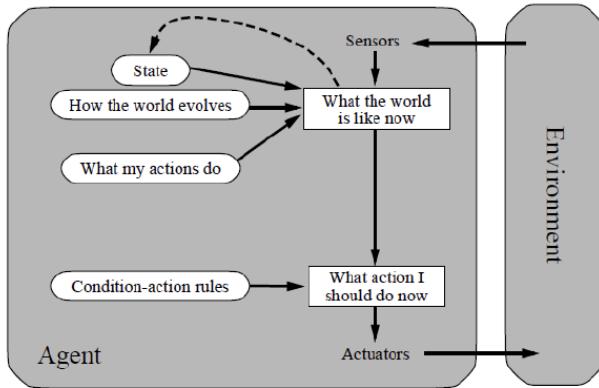


Abbildung 1.4: Model Agent

1.6.2 Model-based Reflex Agent

Der Model-based Reflex Agent speichert zusätzlich noch sein **internes Welt-Modell** ab (das ist natürlich abhängig von dem, was er bereits 'gesehen' hat), weiss in etwa, was der **Effekt seiner Aktionen** sind und wie sich die **Welt zirka verändern wird**.

1.6.3 Goal-Based

Wahnsinn, jetzt weiss ich, wie sich die Welt verändert. Was will ich aber genau? Was ist der Sinn des Lebens? Was sind meine Ziele?

Das hat der *Goal-based Agent* nämlich - Ziele. Es evaluiert jetzt einfach zusätzlich noch, wie sich seine Aktionen auf das Erreichen seines gewünschten Ziels auswirken wird.

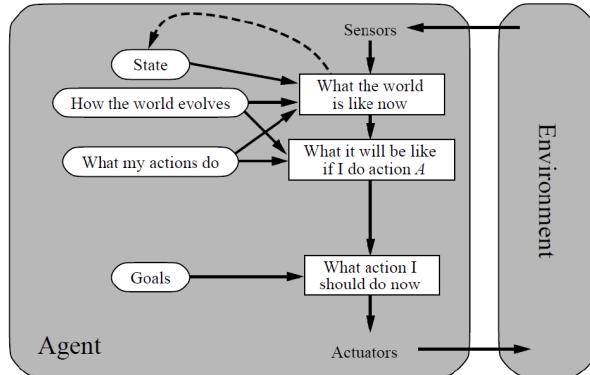


Abbildung 1.5: Goal Based

1.6.4 Utility-Based

Wenn jetzt mehrere Varianten zur Verfügung stehen, kann der Agent mit seiner *utility-Funktion* entscheiden, welche dieser Aktionen ihm am meisten nützt.

1.6.5 Learning

Lernende Agenten starten mit einer unbekannten Umgebung und ohne Wissen. Sie haben ein **learning element**, mit dem sie Verbesserungen am **performance element** ausführen, welches zuständig

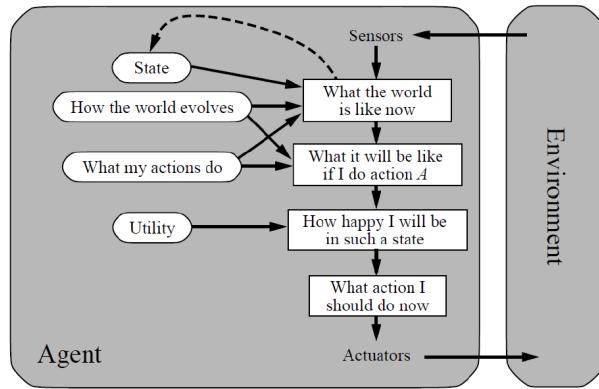


Abbildung 1.6: Utility Agent

ist für die Auswahl von Aktionen. Der **critic** ist etwas externes, dass die Performance des Agenten bestimmt. Der **problem generator** bestimmt auch Aktionen, mit denen der Agent etwas dazulernen kann, er stellt sich also selbst Aufgaben, damit er testen kann ob das gut war oder nicht.

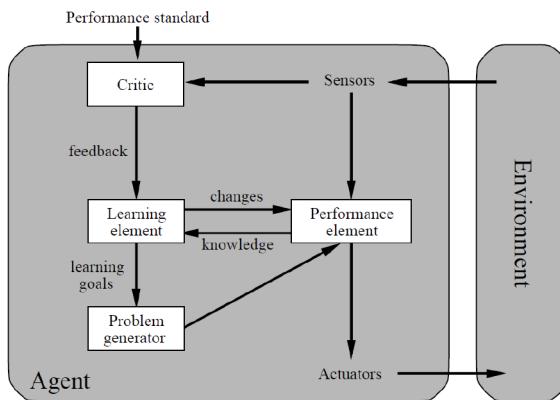


Abbildung 1.7: Learning Agent

1.6.6 Lernfragen

1. Was ist ein rationaler Agent?
2. Wie können wir Umgebungen charakterisieren?
3. Welche Typen von rationalen Agenten unterscheiden wir? Was können Sie?

2 Suchalgorithmen

2.1 Systematische Suche

Ein rationaler Agent sieht die Welt in verschiedenen Stati, z.B. wären da der aktuelle Zustand und der Zielzustand der Welt. Der rationale Agent muss jetzt eine Folge von Aktionen finden, welche ihn vom Initialzustand in den Zielzustand bringt.

2.1.1 Voraussetzungen für eine Suche

Ein rationaler Agent ist in den Ferien und muss so schnell wie möglich von Arad nach Bukarest , da dort sein Flieger geht. Wie schafft er das? Mit der Suche eines schnellstmöglichen Weges.

Generell kann man sagen, dass wenn die Umwelt folgende Eigenschaften erfüllt, der Agent eine **Sequenz von Aktionen** suchen kann, welche ihn zum Zielzustand führen werden:

1. **Observable** - Der Agent weiss, wo er ist.
2. **Static** - Die Umwelt verändert sich nicht plötzlich.
3. **Deterministic** - Jede Aktion hat den gewünschten Effekt.
4. **Discrete** - Nur eine finite Anzahl von Aktionen ist möglich in jedem Zustand.

Im Beispiel Arad nach Bukarest sind diese Eigenschaften gegeben. Der Agent weiss, wo er sich befindet, die Städte verändern nicht plötzlich den Ort, und wenn er von einer Stadt in die nächste fährt, dann fährt er auch dorthin. Und, gegeben die Abstraktion, er kann nicht unendlich viele Städte direkt erreichen. Ein Status wäre hier z.B. die aktuelle Stadt auf der Karte, die Aktionen die Reise zwischen den Städten - also die Kanten des Graphen, die Aktionskosten die Distanz-Informationen auf den Kanten.

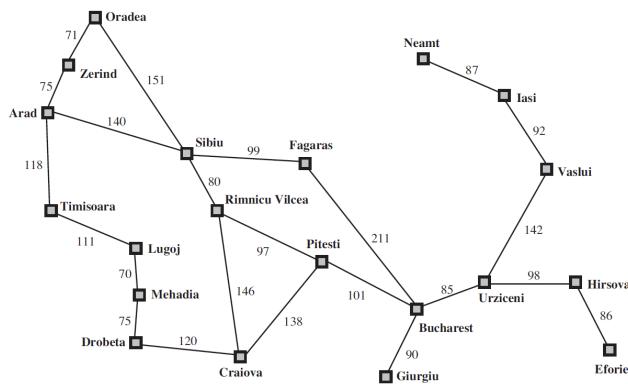


Abbildung 2.1: Beispiel für eine Umgebung wo eine Suche möglich ist

2.1.2 Begrifflichkeiten

Initial State Der erste Status des Agenten

State Space Alle möglichen Stati (z.B. der Graph aller Städte)

Actions Alle möglichen Aktionen

Transition Model Eine Funktion, welche als Input einen Status und eine Aktion hat, und daraus einen neuen Status generiert. z.B. $\text{succ}(\text{Arad}, (\text{Arad-Sibiu})) = \text{Sibiu}$.

Goal Test Sind wir schon da? Sind wir schon da?

Path Die Sequenz der Aktionen.

Path Costs Die Kostenfunktion über den Gesamtpfad. Meistens die Summe der Kosten aller Einzelaktionen, aber z.B. bei einer Aktion 3 für 2 wäre es wieder anders.

Solution Pfad vom Initialzustand zum Zielzustand.

Optimal Solution Pfad vom Initialzustand zum Zielzustand mit den niedrigsten Kosten.

Search Costs Die Kosten (Zeit & Speicherverbrauch) um eine Lösung zu finden.

2.1.3 Beispielmodelle

8 Puzzle

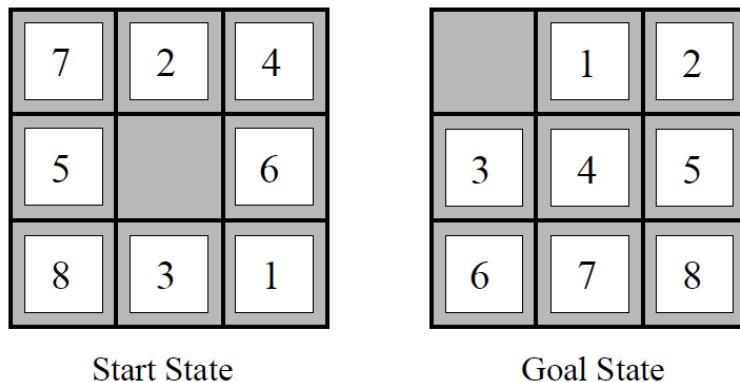


Abbildung 2.2: Das Spiel 8 Puzzle

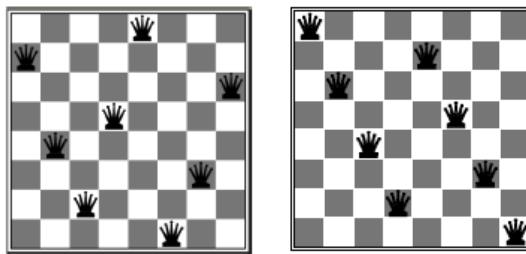
In diesem Spiel wäre der aktuelle Status die Position der Spielsteine. Die Aktionen wären *bewege das leere Feld* (simpler als die anderen zu bewegen) und der Goal Test wäre dann einfach *ist der Zielzustand = aktueller Zustand*. Die Pfadkosten, sind hier definiert als jede Aktion kostet 1.

8 Queens

In diesem Spiel geht es darum, 8 Damen auf einem Feld so zu platzieren, dass keine Dame eine andere Dame bedroht. Hier ist der Zielzustand im Vorfeld unbekannt.

Der aktuelle Status wäre wieder die Position der Damen auf dem Brett, der Initialstatus wäre ein leeres Brett, die *Successor function*, dass man eine Dame zum Brett hinzufügt, und der Goal Test ist auch klar - es sind 8 Damen auf dem Brett die sich nicht bedrohen. Die Pfadkosten sind hier eigentlich egal, wir sind ja nur an der Lösung interessiert.

Optimaler wäre die Successor Function so zu definieren, dass man die Damen einfach Spaltenweise hinzufügt.



(a) Zielzustand 1 (b) Zielzustand 2

Abbildung 2.3: Mögliche Zielzustände

2.1.4 Suchbaum

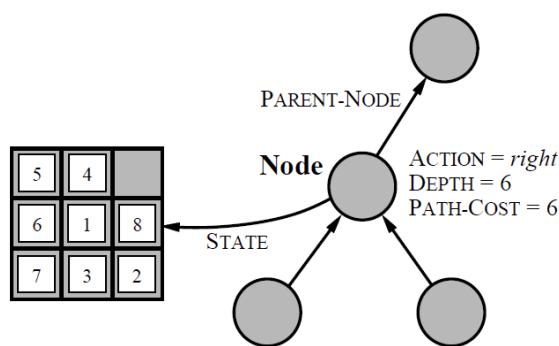


Abbildung 2.4: Suchbaum Aufbau

Ein Suchbaum beschreibt die Reihenfolge, in der Knoten im Suchraum besucht werden. Ein Knoten beschreibt dabei einen Status, die letzte Aktion, die zu ihm geführt hat (denn Status + Aktion = Status) und die aktuellen Pfadkosten. Er hat verschiedene Kinder, welche durch entsprechende Aktionen erreicht werden können.

Frontier Die Knoten, welche gerade von der Suchfunktion gefunden wurden und die noch nicht weiter untersucht werden.

Repeated State Stati, welche im aktuellen Suchbaum schonmal vorkamen.

Redundant Paths Wenn 2 Pfade gefunden werden, welche zum selben Status gelangen.

Search Space = Suchraum. Ein Graph, indem die Knoten alle Stati im *State Space* darstellen und die durch entsprechende Aktionen verbunden sind.

Node expansion Generiert alle Nachfolge-Knoten, gegeben die Aktionen.

Search Strategy Bestimmt, welcher Knoten als nächstes expandiert / exploriert wird.

Tree-based Search Suche bei der nur die Frontier gespeichert wird und die bereits besuchte Knoten immer wieder besuchen kann.

Graph-based Search Suche bei der zusätzlich die besuchten Knoten ebenfalls gespeichert werden.

Dead End Wenn bei einem Knoten keine unbekannten Kinder mehr vorhanden sind.

Explored Set Ein Set aus besuchten Knoten. Wird nicht benötigt, wenn der Suchraum keine Schleifen enthält.

2.1.5 Suchraum

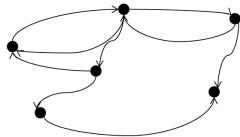


Abbildung 2.5: Beispiel für Suchraum 1

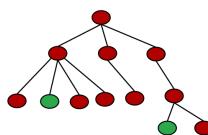


Abbildung 2.6: Beispiel für Suchraum 2

Dieser Suchraum in Abbildung 2.6 hat bestimmte Eigenschaften.

- b** Branching Factor - max. Anzahl Kinder eines Knotens - hier **4**
- d** Depth of shallowest goal - hier wäre die Tiefe des ersten grünen Knotens **2**.
- m** Maximum Length - Maximale Pfadlänge - hier **3**.

2.1.6 Breadth-First Search (BFS)

Hier wird zuerst in die Breite gesucht. Das heisst, dass wenn ein Knoten gefunden wird, wird er in eine FIFO Queue gesteckt, welche dann nacheinander abgearbeitet wird. BFS findet alle Lösungen

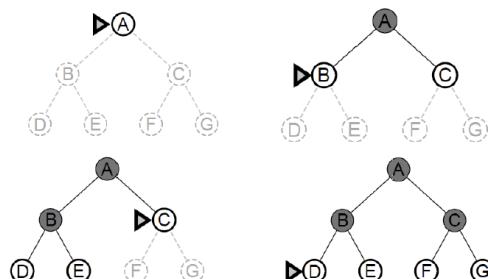


Abbildung 2.7: Breadth-First Search

und wenn die Aktionen alle dieselben positiven (oder null) Kosten haben, dann findet es auch die optimale Lösung. Es findet ja immer die Lösung, die am wenigsten tief im Baum vorhanden ist.

Zeitbedarf & Zeitkomplexität

Ein BFS Suchbaum muss ja bis zur Tiefe des Lösungsobjekt komplett aufgebaut werden. Wenn **b** der *branching factor* ist und **d** die Tiefe der Lösung heisst das, dass diese Anzahl an Knoten besucht werden müssen:

$$\sum_{i=1}^d b^i = b + b^2 + b^3 + \dots + b^d \approx O(b^d)$$

Speicherbedarf & Speicherkomplexität

BFS speichert offenbar jeden besuchten Knoten ab, da der Pfad zur gefundenen Lösung ja auch bekannt sein muss. Daher ist der Speicherbedarf, gleich wie der Zeitbedarf:

$$\sum_{i=1}^d b^i = b + b^2 + b^3 + \dots + b^d \approx O(b^d)$$

Die Frontier hat $O(b^d)$ und das explored set $O(b^{d-1})$.

2.1.7 Depth-First Search (DFS)

Die Depth First Search ist ähnlich wie die Breadth-First Search. Die gefundenen Knoten werden hier aber in eine LIFO Queue gesteckt. Das bewirkt, dass zuerst in die Tiefe gesucht wird und erst dann schrittweise in die Breite. DFS ist keine optimale Suche, es kann sein dass eine Lösung tief unten

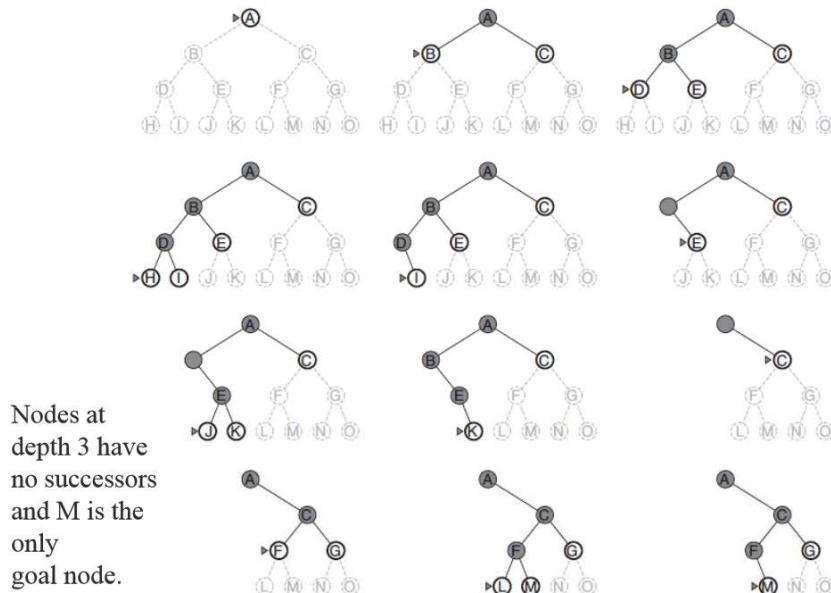


Abbildung 2.8: Depth-First Search

'vergraben' liegt, und wenn sie dann unendlich tief ist, geht sie dort natürlich auch suchen. Wenn die besuchten Knoten zusätzlich nicht gespeichert werden, dann kann sie auch für immer im Kreis herum suchen.

Zeitbedarf & Zeitkomplexität

Eine DFS Suche findet im schlechtesten Fall die Lösung erst wenn der ganze Baum durchsucht wurde. **m** heisst ja die maximale Tiefe und **b** wieder der *branching factor*.

$$b^m \approx O(b^m)$$

Speicherbedarf & Speicherkomplexität

DFS muss nicht der gesamte Baum speichern, sondern nur der Pfad der gerade zum Objekt führt.

$$bm \approx O(bm)$$

In Abbildung 2.8 sieht man das schön. Dort hat der Baum $b = 2, m = 3$. In keinem Fall müssen also mehr als $2 * 3 = 6$ Knoten gespeichert werden.

Wenn der Suchraum allerdings noch Schlaufen beinhaltet, so muss noch die besuchten Knoten gespeichert werden. Das kann bedeuten, dass im schlechtesten Fall der gesamte Baum abgespeichert werden muss.

$$b^m \approx O(b^m)$$

2.1.8 Depth-Limited Search (DLS)

DLS ist DFS, einfach mit einem Limit wie tief die Suche gehen darf. Läuft jetzt nicht mehr unendlich, aber wenn die Lösung tiefer ist als das Limit wird sie natürlich nicht gefunden.

Zeitbedarf & Zeitkomplexität

Gleich wie DFS, einfach anstatt Tiefe des Baums die Tiefe des Suchlimits. Das heisst Zeitbedarf ist jetzt ($l = \text{Limit}$)

$$b^l \approx O(b^l)$$

Speicherbedarf & Speicherkomplexität

... und Speicherbedarf:

$$bl \approx O(bl)$$

2.1.9 Iterative Deepening Search (IDS)

Wie DLS, einfach wird das Limit schrittweise angehoben. Ist also irgendwie eine Mischung aus BFS und DFS. Die ersten Schritte der Suche werden immer wiederholt, was aber sich aber nicht auf die Gesamtkomplexität auswirkt. Es findet immer eine Lösung wenn eine da ist und ist sogar optimal, wenn die Kostenfunktion nicht abnimmt mit der Tiefe des Suchbaums.

Zeitbedarf & Zeitkomplexität

Komplexität gleich wie DLS / DFS, einfach anstatt Limit wieder der gesamte Baum. Aber der echte Zeitbedarf ist etwas grösser, da ja jeder Knoten besucht wird.

$$(d)b + (d - 1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + 1b^d \approx O(b^d)$$

Speicherbedarf & Speicherkomplexität

... und Speicherbedarf ebenfalls:

$$bd \approx O(bd)$$

2.1.10 Uniform Cost Search (UCS)

Hier werden wieder die Nodes analog DFS und BFS gefunden. Die Queue hier ist aber keine FI-FO oder LIFO, sondern eine **Priority Queue**. Das heisst, dass die gefundenen Nodes anhand ihrer bisherigen **Gesamt-Pfadkosten** sortiert werden und zuerst die Knoten expandiert werden, welche die geringsten Kosten haben. So findet UCS die optimale Lösung, sofern die Nachfolgeknoten immer grössere Gesamtkosten haben als der aktuelle Knoten.

Wichtig bei der Implementation wäre noch, dass nicht beim Finden, resp. expandieren der Frontier getestet wird, ob dies ein Zielknoten ist, sondern erst, wenn der nächste Knoten aus der sortierten Queue genommen wird. Ansonsten wäre das Resultat wieder nicht optimal. Zudem wird, wenn ein Knoten gefunden wird mit niedrigeren Kosten als derselbe Knoten hat, der in der Frontier vorkommt, dieser in der Frontier ersetzt.

Wenn ein Pfad 0 oder weniger kostet und unendliche tief runter geht, dann bleibt UCS dort stecken, weil es wirklich nur die Kosten anschaut.

Zeit- & Speicherkomplexität

$O(b^{d+1})$ wenn alle Aktionskosten identisch sind

$O(b^{1+\lfloor o/c \rfloor})$ wenn sie nicht identisch sind. o = Kosten optimale Lösung, c = minimale Kosten einer Aktion.

2.1.11 Fragen

By which means are search problems formulated?

1. Zielzustand formulieren
2. Zustandsraum definieren
3. Mögliche Aktionen festlegen
4. Suchkosten - suche ich unendlich lange oder mache ich mal
5. Aktionskosten - Pfadkosten

How can search problems differ in the characteristics of actions and states?

Search problems can have different sets of actions and states. The way the problem is modeled decides.

Which properties are used to characterize a search method?

Completeness, Time and Space Complexity and optimality.

Can you explain how BFS, DFS, DLS, IDS, UCS work?

See above.

Compare the 5 uninformed search methods wrt. time complexity, space complexity, optimality, completeness

See above.

2.2 Heuristic (informed) search

Die uninformierte Suche nimmt einfach irgendwelche Knoten und sucht bei diesen weiter. Bei der informierten, heuristischen, Suche werden den Knoten (Schätz-) Werte zugewiesen, durch eine evaluierungsfunktion $f(n)$.

$$f(n) = g(n) + h(n)$$

Wobei $g(n)$ die Kosten vom Erstzustand bis zum aktuellen Zustand und $h(n)$ die geschätzten Kosten bis zum Ziel sind. $h(n)$ ist hierbei die *Heuristik*.

2.2.1 Eigenschaften von $h(n)$

1. $h(n) = 0$ wenn n ein Zielzustand ist.
2. $h(n)$ ist **admissible**.
Es muss die realen Kosten bis zum Ziel unterschätzen.
3. $h(n)$ ist **consistent**
Vom Knoten n wären es ≈ 10 bis ins Ziel. Zum Nachfolgeknoten n' kostet es 1. Dann darf der Nachfolgeknoten n' nicht plötzlich weniger als ≈ 9 haben.

2.2.2 Greedy Search

Greedy Search verwendet ausschliesslich die Heuristik, um den Weg ins Ziel zu finden. Es garantiert keine optimale Lösung und auch nicht, dass es überhaupt eine Lösung findet. Im Beispiel mit dem Weg von Sibiu nach Bukarest würde es immer den Knoten wählen, der näher bei Bukarest ist als die anderen.

2.2.3 A* Search

A* benutzt die vergangenen Kosten $g(n)$ und die geschätzten nachfolgenden Kosten $h(n)$ und verwendet das als Sortierung für den als nächstes zu evaluierenden Knoten, verwendet also $f(n)$.

$$f(n) = g(n) + h(n)$$

Es ist garantiert, dass wenn es eine Lösung gibt, dass A* diese findet, sofern die Heuristik die Eigenschaften von 2.2.1 erfüllt. Zudem ist die gefundene Lösung optimal. Der Nachteil ist aber, dass die Zeit- und Speicherkomplexität exponentiell wachsen. Das heisst, dass alles von der gewählten Heuristik abhängt.

2.2.4 Beam Search

Beam Search baut einen Breadt-First Suchbaum auf, aber speichert in jedem Schritt davon nur die n -besten und evaluiert danach auch nur diese. n ist dabei die *Beamwidth*. Ist offensichtlich nicht komplett und auch nicht optimal, aber in der Praxis trotzdem manchmal nützlich, wenn der Suchraum riesig ist.

2.2.5 IDA*

IDA* sucht zuerst in die Tiefe und bricht aber ab, wenn die geschätzten Kosten einen gewissen Schwellenwert überschreiten.

2.2.6 Branch and Bound

Sind mehrere Lösungen gesucht, so kann wenn eine Lösung gefunden wurde, können alle Knoten die höhere Kosten haben, weggeschnitten werden. Denn die Heuristik unterschätzt die Kosten ja immer, dann kann das gar keine gute Lösung mehr werden.

2.2.7 Heuristiken

I don't know! - Make an educated guess! — Oliver Töngi, always.

Das ist eine Heuristik. Eine gute Heuristik, ist informiert (und daher spezifisch) über das Problem.

Manhattan Distance

Für das 8 Puzzle, siehe Abbildung 2.2, könnten z.B. die Distanzen der falsch platzierten Steine zu ihrer richtigen Position eine Heuristik darstellen. Das wäre die Manhattan Distance (weil Manhattan ja so schachbrettartig aufgebaut ist, dass man sagen kann *zwei Blocks entfernt*).

Linear Conflict Heuristic

Wenn sich zwei Steine für die Lösung im Weg sind, also mehr Moves gemacht werden müssen als nötig, wird noch extra was dazuberechnet.

Pattern Database

Man kann z.B. Teillösungen schon vorausberechnen und wiederverwenden. Für ein grösseres 8 Puzzle z.B. den Rand des Puzzles anhand einer Datenbank bilden und dann der innere Teil erst lösen. Im Schach macht man das auch so, dass man am Ende des Spiels eine vorberechnete Strategie fährt, da dort ja die Zustandsräume nicht mehr so gross sind.

Gaschnigs Heuristic

Man nehme ein zwei Constraints des Problems weg und suche da eine einfach berechenbare Lösung. z.B. 8 Puzzle als String formatieren und dann schauen, wie aufwändig es ist, das der Reihe nach zu sortieren. Wird häufig gemacht zum noch eine bessere Heuristik herauszufinden.

2.2.8 Summary Questions

Explain the underlying ideas of best-first search, greedy search and beam search.

Best first search wählt einfach anhand einer Funktion den nächstbesten Knoten zum expandieren aus. Greedy Search ist eine Art von Best First Search und verwendet einfach nur die geschätzten Kosten bis zum Zielpunkt als Evaluation. Beam Search baut einen Breadt-First Suchbaum auf, aber speichert in jedem Schritt davon nur die n -besten und evaluierter danach auch nur diese. n ist dabei die *Beamwidth*.

How do the A* and IDA* algorithms work?

A* evaluiert die realen, bekannten Kosten bis zu einem Knoten und die geschätzten Kosten bis zum Ziel und wählt immer die besten aus. IDA* macht Tiefensuche, evaluiert dabei

What are the principal properties of A*?

A* needs an admissible and consistent heuristic to work. It requires exponential complexity to solve the problem.

Why is IDA* more efficient in practice?

It doesn't need exponential memory to work.

What is the principal idea behind branch and bound?

Die Knoten wegschneiden, welche garantiert nicht zur Lösung führen.

What are heuristics, which role do they play in informed search, and how can we develop good heuristics?

Heuristics are educated guesses, they help us to direct the search in the search space towards the desired goal state in an efficient manner. Heuristics are often developed with problem relaxation, whereas one or more constraints from the problem are dropped and a solution is found for the easier problem, from which process the distance to the solution with all restrictions can be guessed.

What is an admissible heuristics?

A heuristic which underestimates the real costs to the goal state.

What can happen to A* if it is used with a non-admissible heuristic?

It's guaranteed to find the optimal solution, because the too highly estimated nodes are not further evaluated.

2.3 Local Search

Wenn wir nicht daran interessiert sind, wie wir zur Lösung gelangen, sondern nur die Lösung wichtig ist, können wir auch Local Search benutzen, das heisst wir starten irgendwo im Suchraum, evaluieren den aktuellen Knoten und bewegen uns irgendwie dorthin wo wir es besser vermuten. *Complete* bedeutet im Kontext der lokalen Suche, dass es eine Lösung findet und *Optimal* meint, dass die Lösung dann auch das globale Maximum / Minimum ist.

2.3.1 Hillclimbing

Hier nimmt man irgend einen Knoten und generiert die Nachfolge Knoten daraus und jagt diese durch die Evaluierungsfunktion. Man nimmt dann einfach den besten Knoten und fangt wieder von vorne an. Also man geht einfach so weit rauf wies gerade geht, findet aber nicht immer eine Lösung und diese ist auch nicht immer optimal. Hier im Beispiel ist wieder das 8 Damen Problem. Die Damen sind wieder jeweils in einer Spalte, dann wirds einfacher. Für jede Spalte wählt man jetzt das Feld für die Dame aus, an dem sich am wenigsten Damen attackieren.

Limitationen von Hillclimbing

Hillclimbing ist aber nicht wirklich gut für dieses Problem, nur in 14% aller Fälle, führt dieses Vorgehen auch zur Lösung. Sonst bleibt es irgendwo eben in einem lokalen Minima oder einem Plateau - rundherum ist es nur flach - stecken. Was macht man in so einem Fall?

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	14	15	13	16	13
15	14	17	15	15	14	16	16
17	15	16	18	15	15	15	15
18	14	15	15	15	14	15	16
14	14	13	17	12	14	12	18

Abbildung 2.9: Hillclimbing mit den 8 Queens

2.3.2 Random Restart

Entweder man fängt wieder von vorne an und wählt zufällig einen anderen Ausgangspunkt und steigt da hoch. Theoretisch findet man so immer die Lösung, da der Anfangsstatus irgendwann zufällig als der initial Status generiert wird. Funktioniert super wenn es nicht so viele Lokale Maximas oder Plateaus gibt. Geht weniger gut wenn es exponentiell viele lokale Maximas gibt - siehe NP Probleme.

2.3.3 Random Walk

Oder man geht ab und zu auch mal wieder irgendwie gerade aus oder wieder runter.

2.3.4 Tabu Search

Man merkt sich, wo man schon war und geht da nicht mehr hin bei der Suche, weil dort wars ja nicht optimal.

2.3.5 Simulated Annealing

Zuerst wird häufig herumgesprungen, dann mit der Zeit immer weniger. Die Abnahme des Herum-springens bezeichnet man als *cooling*.

2.3.6 Genetische Algorithmen

Jetzt wird die Suche umgedreht - aus zwei Knoten generiert man einen neuen Knoten. Die Basis Idee stammt aus der Evolution und man geht nach dem folgenden Schema vor:

1. Kreuzen von Genen
2. Zufälliges Mutieren
3. Selektieren der Besten

Man könnte z.B. die Stati als String abspeichern und diese aufsplitten und zufällig rekombinieren und danach wieder durch einen Evaluations Algorithmus laufen lassen.

Genetische Algorithmen sind aber nicht gut zur Optimierung - ansonsten wären wir ja alle gleich.

Man findet aber eine Anzahl an guten Lösungen - was auch der Sinn der Evolution ist, das Überleben unter sich verändernden Bedingungen.

2.3.7 Ant Colony Optimization

Warum sucht eigentlich immer nur einer? Nehmen wir mehrere, die Suchen! Analog der Optimierung von Pfaden von Ameisenkolonien versucht man das zu lösen.

2.3.8 Summary Questions

How do local and systematic search methods differ?

Systematische Suchen besuchen jeden Zustand von einem Anfangszustand aus, bis sie etwas gefunden haben. Lokale Suchen fangen irgendwo einmal an.

How do hillclimbing and simulated annealing work?

Hillclimbing sucht das nächste lokale Maximum und Simulated annealing macht dasselbe, springt noch etwas herum, mit der Zeit springt es aber immer weniger herum (cooldown) und sollte dann bei einem möglichst guten Maxima landen.

What can we say about the theoretical properties of local search methods?

- Nicht vollständig
- Nicht optimal
- Wenig Speicherverbrauch
- Laufzeit selbst wählbar - z.B. wie viele Generationen man haben möchte.

What techniques exist to escape from local minima and plateaus?

Random Restart, Random Walk

Why does local search often work well in practice?

Weil es muss nicht unbedingt die optimale Lösung sein, einfach eine gute Lösung.

What other ideas for local search techniques exist?

Biologische Ansätze wie Schwarmintelligenz oder genetische Algorithmen.

3 Game Theory - Gleichzeitige Spiele

3.1 Rationale Entscheidungen

Eine Entscheidung ist rational, wenn sie für den Spieler zumindest gleich gut ist, wie jede andere mögliche Option.

3.2 Definition gleichzeitiges Spiel

Ein Spiel besteht aus:

1. Einem Set aus Spielern ($2 - \infty$)
2. Ein Set von möglichen Aktionen für jeden Spieler
3. Eine Präferenz jedes Spielers für ein *action profile*

3.2.1 Action Profile

Die Spiele laufen ja gleichzeitig ab, d.h. beide Spieler entscheiden gleichzeitig und unabhängig. Dann ist ein Action Profile die Entscheidung beider Spieler. Also wenn es zwei Aktionen gibt, a und b und zwei Spieler, P_1 und P_2 , dann gibt es z.B. folgende Action Profiles:

$$(a, a), (a, b), (b, a), (b, b)$$

Diesen Actionprofiles wird dann noch eine Präferenzf für die Action Profiles in Form einer einfachen Reihenfolge zugeordnet. Diese ist für jeden Spieler verschieden und könnte z.B sein für den Spieler 1:

$$u(a, a) > u(a, b) > u(b, b) > u(b, a)$$

Anhand dieser Reihenfolge kann dann noch die Payoffs festlegen. Es ist egal, wie gross / klein der Payoff ist, wichtig wäre nur, dass eben diese Reihenfolge stimmt. Also z.B.

$$u(a, a) = 4, u(a, b) = 3 \dots$$

Diese Payoffs und entsprechenden Aktionsprofile können auch in einer Matrix dargestellt werden, also irgendwie so (Beispiel aus dem Gefangenendilemma):

		Clyde (column player)	
		Quiet	Fink
Bonnie (row player)	Quiet	2,2	0,3
	Fink	3,0	1,1

Abbildung 3.1: Payoffmatrix

3.2.2 Gefangenendilemma

Bonnie and Clyde are suspects in a major crime and held in separate cells. There is enough evidence to convict each of them of a minor offense, but not enough evidence to convict either of them of the major crime unless one of them acts as an informer against the other. If they both stay quiet, each will be convicted of the minor offense and spend 1 year in prison. If one and only one of them finks, (s)he will be freed and used as a witness against the other, who will spend 20 years in prison. If both fink, each will spend 5 years in prison.

Das Gefangenendilemma kommt auch in der realen Welt häufig vor, zB. beim Überfischen der Meere, Dopen, ...

3.3 Best Response Function

Wenn man die gewählten Aktionen der anderen Spieler kennt, nimmt man natürlich die Option, welche einem selbst am meisten nützt, alles andere wäre ja nicht rational. Die Position mit dem Pfeil

Analysis for Bonny (row player):

- Clyde chooses quiet → best response is to fink
- Clyde chooses fink → best response is to fink

	Quiet	Fink
Quiet	2,2	0,3
Fink	3,0	1,1

Analysis for Clyde (column player):

- Bonnie chooses quiet → best response is to fink
- Bonnie chooses fink → best response is to fink

	Quiet	Fink
Quiet	2,2	0,3
Fink	3,0	1,1

Combined Analysis:

	Quiet	Fink
Quiet	2,2	0,3
Fink	3,0	1,1



Abbildung 3.2: Best Responses für das Prisoners Dilemma

ist ein Gleichgewicht. Keine Partei hat von sich aus dem Willen, von ihrer Position abzuweichen, weil das für sie direkt eine negative Konsequenz hätte.

3.4 Dominante & Dominierte Strategien

3.4.1 Dominante Strategie

Eine Strategie, welche egal was der andere macht, **immer** die Beste ist, ist dominant. Spielen Sie *immer* eine dominante Strategie.

3.4.2 Dominierte Strategie

Eine Dominierte Strategie ist **nie** die beste Option, egal was der andere macht. Spielen Sie *niemals* eine dominierte Strategie.

3.5 Pure Strategy Nash Equilibrium

Ein Aktionsprofil (d.h. die gewählten Aktionen aller Spieler) ist ein *pure strategy Nash equilibrium*, wenn es die beste Antwort zu jeder anderen Aktion der anderen Spieler ist. Siehe Abbildung 3.2 -

dort zeigt der Pfeil auf das Aktionsprofil, welches ein solches Nash equilibrium ist. Kein Spieler hat einen Anreiz, davon abzuweichen, auch wenn keine Polizei oder so da ist.

3.5.1 Beispiel für mehrere Nash Equilibriias

Ein Spiel kann mehrere Nash Equilibriias haben.

Suppose that two cars are driving at each other from perpendicular directions. The stoplight is red for one of them and green for the other. If the police could not ticket the driver, would they want to break the law?

		Player B	
		Go	Stop
Player A	Go	-5, -5	1, 0
	Stop	0, 1	-1, -1

Abbildung 3.3: Ein Spiel mit zwei Nash Equilibriias

3.5.2 Beispiel für kein Nash Equilibrium

- If kicker shoots left and goalie jumps left, kicker has 40% chance of scoring
- If kicker shoots left and goalie jumps right, kicker has 90% chance of scoring
- If kicker shoots to the middle, kicker has 60% chance of scoring

		Goalie	
		Left	Right
Penalty Kicker	Left	4, -4	9, -9
	Middle	6, -6	6, -6
	Right	9, -9	4, -4

Abbildung 3.4: Beispiel einer Payoffmatrix ohne ein Nash Equilibrium

Hier ist interessanterweise noch die Aktion *middle* vom Torschützen eine dominierte Option - in jedem Fall, was der Goalie macht, gibt es eine bessere Option.

3.5.3 Nash Equilibrium vs Pareto Optimal

Es heisst nicht, dass ein Nash Equilibrium die beste Option für beide Spieler ist. Eigentlich wäre es ja besser, wenn beide stillschweigen würden. Die beste Option - global gesehen - wäre also ein *Pareto optimales* Aktionsprofil.

3.5.4 Mixed Strategy Nash Equilibrium

Jedes Pure Strategy Nash Equilibrium ist auch ein Mixed Strategy Nash Equilibrium. Macht Wahrscheinlichkeitsverteilung über mögliche Aktion.

		Tax payer	
		Honest	Cheat
		Audit	2,0
		Not Audit	0,4

Abbildung 3.5: Beispiel Mixed Strategy Nash Equilibrium

$$\begin{aligned}
 E_{Audit} &= 2q + 4(1 - q) = 4 - 2q \\
 E_{Not\,Audit} &= 4q + 0(1 - q) = 4q \\
 4 - 2q &= 4q \Rightarrow q = 2/3 \\
 E_{Honest} &= 0p + 0(1 - p) = 0 \\
 E_{Cheat} &= -10p + 4(1 - p) = 4 + -14p \\
 0 &= -14p + 4 \Rightarrow p = 2/7
 \end{aligned} \tag{3.1}$$

3.5.5 Sequential Game

Sequential move after another -> Müssen Endlich in Anzahl Spielzügen und Entscheidungsmöglichkeiten pro Zug sein Nash Equilibrium wird bei Sequentiellen Games mit Backward Induction gefunden!

Theorem of Zermelo

Jedes endliche Spiel mit 2-Spieler und perfekter Information hat:

- First-mover Advantage: 4 Gewinnt
- Second-mover Advantage: Schrere Stein Papier nacheinander gespielt

Minimax

- Totaler Gewinn eines Spielers ist Verlust des anderen - SP1 + 1 & SP2 -1
- Game-Tree vorausberechnen: Eigene Entscheidungen Maximieren, Gegnerische Minimieren
- DFS Suche da ohne Tiefe meisten Bäume nicht zeitig berechnet werden können.
- Alpha-Beta Pruning: Teilbäum die keine bessere Lösung geben können müssen nicht weiter durchsucht werden
- Alpha: höchster Wert von allen MAX Vorgänger eines MIN Knoten
- Beta: niedrigster Wert von allen MIN Vorgänger eines MAX Knoten

4 Problemkomplexität

4.1 Einleitung

Exponentielle Komplexität ist schlecht, polynomiale in Ordnung, Lineare perfekt. Schnellere Computer helfen bei solchen Problemen leider nicht. Das gute ist, man kann mit nicht-perfekten Lösungen meistens leben, das heisst, wenn sie nicht beliebig schlecht sind.

4.1.1 Entscheidungsprobleme

Aus Bequemheitsgründen sind alle Probleme so formuliert, dass sie am Schluss entweder ein *Ja* oder *Nein* zurückliefern. z.B. *Gegeben dieses Schachbrett - gewinnt Weiss?*

4.2 Turing Maschine

Turing Maschine operiert auf einem (endlosen) Band und kann lesen / schreiben und das Band nach links oder rechts bewegen. Es ist die Simplest-Mögliche Rechenmaschine und trotzdem kann sie, wenn man genügend Zeit dafür hat, jedes mathematische Problem lösen, dass auch ein anderer Computer lösen kann - sprich, der 1'000 CHF PC von heute 'kann' nicht mehr als der vor 10 Jahren, nur er kann es schneller.

4.2.1 Deterministic Turing Maschine (DTM)

Ein Regelbuch, welches immer maximal eine Aktion beschreibt für jede gegebene Situation. Kann P Probleme in polynomialer Zeit lösen, NP nur in exponentieller Zeit.

4.2.2 Non-Deterministic Turing Maschine (NTM)

Ein Regelbuch, welches auch mehrere Aktionen für eine gegebene Situation ausführen kann. Eine NTM nimmt für jede Variante 'durch Magie & Glück' die korrekte, und kann dadurch NP Probleme in polynomialer Zeit lösen. Jede solche Maschine kann durch eine deterministische Maschine simuliert werden, aber die muss halt jede mögliche Auswahl dabei simulieren und braucht dadurch exponentiell viel Zeit.

4.3 NP Probleme

4.3.1 Beispiel

Wir haben eine Menge, bestehend aus diesen Zahlen:

$$S = \{-10, -3, -2, 7, 14, 15\}$$

Gibt es eine Menge bestehend aus diesen Zahlen, deren Summe 0 ergibt?

4.3.2 Definition

Entscheidungsprobleme, welche in polynomialem Zeit lösbar sind mit einem Algorithmus, welcher irgendwie immer die richtige Entscheidung wählt. Es bedient sich dabei bei dem nicht-deterministischen Modell der Berechnung. Siehe Abschnitt 4.2.2. NP Probleme haben eines gemeinsam: Die Verifikation der Lösung ist möglich in polynomialem Zeit. Das Finden aber mit jeder zusätzlichen Zahl exponentiell schwierig.

4.4 P = NP

P = NP besagt, dass alle Probleme in NP, das heisst die Probleme, welche nur in exponentieller Zeit gelöst werden können, irgendwann auch in polynomialem Zeit von Computern gelöst werden können. Das würde bedeuten, dass für Probleme, bei denen wir die Lösung einfach verifizieren können, die Lösung auch einfach finden können. Ist bis heute (Januar 2016) nicht bewiesen.

4.4.1 NP Schwierig

Ein Problem, welches mindestens gleich schwierig wie alle anderen Probleme in NP ist. Tetris ist z.B. ein solches NP schweres Problem, d.h. es ist das schwierigste Problem in NP - wenn es NP wirklich gibt.

4.4.2 NP Komplett

NP Komplett ist eine Untermenge der NP Probleme und bedeutet, dass jedes Problem in dieser Menge dasselbe zu Grunde liegende Problem besitzt - z.B. das *satisfiability* Problem. Das heisst, dass eine Lösung zu diesem Problem in polynomialem Zeit, alle anderen NP Probleme gleichzeitig auch lösen würde. Ein NP komplettes Problem, ist daher auch NP schwierig - weil das zu Grunde liegende Problem muss ja min. gleich schwer sein wie alle anderen Probleme in NP, um alle anderen Probleme auch lösen zu können.

4.4.3 Problemreduzierung

z.B. Dijkstra funktioniert nur mit Graphen, welche ein Gewicht haben. Das heisst, dass Probleme mit Graphen ohne Gewicht auch mit Dijkstra gelöst werden können, wenn man einfach ein Gewicht von 1 nimmt. Man wandelt das Ausgangsproblem also irgendwie um in ein anderes Problem. Wir haben also ein Problem A und wandelt das in ein Problem B um. B ist also mindestens so schwierig sein wie A. Dies geschieht bei NP Kompletten Problemen. Diese sind alle auf einander reduzierbar. Das heisst, wenn ein Problem, welches NP Komplett ist, in P gelöst wird, dann sind alle anderen NP kompletten Problem (und NP Probleme) in P gelöst, weil diese ja auf dieses Problem reduzierbar sind. Das heisst auch, dass wenn wir zeigen können, dass wenn ein bekanntes NP komplettes Problem auf ein neues Problem reduzierbar ist, dieses Problem ebenfalls NP komplett ist. Das erste Problem, welches NP komplett ist, ist das *satisfiability* Problem.

4.5 Satisfiability Problem

Gibt es eine Kombination von Boolean - Variablen, welche machen dass folgender Ausdruck wahr ist:

$$(X \vee Y) \wedge \neg X$$

Wäre eine Instanz des Satisfiability Problems.

4.6 Andere Problem-Mengen

1. EXP

Probleme, die in exponentieller Zeit lösbar sind, z.B. Schach, oder Tetris.

2. R

Probleme, die in endlicher Zeit lösbar sind.

3. Co-NP

...

5 Constraint Programming

5.1 Constraint vs optimization problems

5.1.1 Constraint Problem (CP)

- 3 Zutaten: Variablen, Frames und Constraints (Bedingungen)
- Jede Variable nimmt Werte aus einem bestimmten Set an (genannt Frame)
- Eine Constraint zeigt an, dass bestimmte Zuweisungen verboten sind.
- Eine Lösung ist eine gültige Zuweisung von Werten zu Variablen
- Gültig meint in dem Fall dass alle Zuweisungen die Constraints erfüllen
- Es kann keine, eine oder hunderttausendmillionen Lösungen geben.

5.1.2 Constraint optimization problem (OP)

- Ein Constraint-Problem mit einer Zielfunktion
- Eine Lösung erfüllt alle Constraints und *optimiert* die Zielfunktion.

5.1.3 Beispiel Grocery Store

Ein Junge geht in den Laden und kauft sich vier Sachen. Der Kassierer verrechnet 7.11\$. Als das Kind gehen will, ruft ihm der Kassierer nach: "Warte kurz, ich habe die Items multipliziert anstatt sie zu addieren. Aber, jetzt sehe ich gerade, dass ich auf das gleiche Resultat komme, wenn ich sie addiere!"

Was waren die Preise der vier Sachen?

Grocery Store Model

Ein Modell ist nichts anderes als eine mathematische Beschreibung des Problems:

- Eine Variable pro Produkt → p_1, p_2, p_3, p_4
- Da ich lieber mit Integern als mit Kommawerten rechne ergibt sich
- Dass Preise nicht weniger als 0 und nicht mehr als 711 sind.
- Deswegen haben alle Variablen Integerwerte zwischen 0 und 711 (Frame)
- Constraint 1: $p_1 + p_2 + p_3 + p_4 = 711$
- Constraint 2: $p_1 * p_2 * p_3 * p_4 = 711 * 100^3$ Dieses $*100^3$ ist, um den Kommafehler auszugleichen, der sich sonst ergeben würde.

Model implementation with OR-Tools

```
1 Solver solver = new Solver("Grocery");
2
3 // One variable for each product:
4 IntVar p1 = solver.MakeIntVar(0, 711);
5 IntVar p2 = solver.MakeIntVar(0, 711);
6 IntVar p3 = solver.MakeIntVar(0, 711);
7 IntVar p4 = solver.MakeIntVar(0, 711);
8
9 // Prices add up to 711:
10 solver.Add(p1 + p2 + p3 + p4 == 711);
11
12 // Product of prices is 711:
13 solver.Add(p1 * p2 * p3 * p4 == 711 *
14     100 * 100 * 100);
```

Standardrezept dazu:

1. Neuen Solver erstellen
2. Anfragen der Entscheidungsvariablen mit definierten Rahmen vom Solver
3. Constraints zum Solver hinzufügen

Solving the model with OR-Tools

```
1 DecisionBuilder db = solver.MakePhase(
2     new IntVar[] {p1, p2, p3, p4}, //
3         Decision variables to resolve
4     Solver.INT_VAR_SIMPLE, // Variable
5         selection policy for search
6     Solver.INT_VALUE_SIMPLE); // Value
7         selection policy for search
8
9 solver.NewSearch(db);
10
11 while (solver.NextSolution()) {
12     Console.WriteLine("Product 1: " + p1.
13         Value());
14     Console.WriteLine("Product 2: " + p2.
15         Value());
16     Console.WriteLine("Product 3: " + p3.
17         Value());
18     Console.WriteLine("Product 4: " + p4.
19         Value());
20     Console.WriteLine();
21 }
22
23 solver.EndSearch();
```

Standardrezept dazu:

1. DecisionBuilder mit Entscheidungsvariablen, die man lösen will erstellen
2. Starte den Suchprozess und gib die Lösungen aus

Symmetry breaking

Momentan erhalten wir 24 Lösungen, die jedoch alle identisch sind. Deswegen brauchen wir noch die Symmetry-breaking constraint:

```
1 // Symmetry breaking constraints:
2
3     solver.Add(p1 <= p2);
4     solver.Add(p2 <= p3);
5     solver.Add(p3 <= p4);
```

5.1.4 Beispiel SEND + MORE = MONEY

- The characters S, E, N ,D, M, O, R, Y stand for digits between 0 and 9
- Numbers are built from digits in the usual, positional notation
- Repeated occurrence of the same character denote the same digit
- Different characters must take different digits
- Numbers must not start with a zero
- The following equation must hold: $SEND + MORE = MONEY$

Cryptogram Model with something missing

```
1 Solver solver = new Solver("Cryptogram");
2
3 // One decision variable for each character:
4 IntVar S = solver.MakeIntVar(0, 9);
5 IntVar E = solver.MakeIntVar(0, 9);
6 IntVar N = solver.MakeIntVar(0, 9);
7 IntVar D = solver.MakeIntVar(0, 9);
8 IntVar M = solver.MakeIntVar(0, 9);
9 IntVar O = solver.MakeIntVar(0, 9);
10 IntVar R = solver.MakeIntVar(0, 9);
11 IntVar Y = solver.MakeIntVar(0, 9);
12 IntVar[] vars = new IntVar[] {S, E, N, D, M, O, R, Y};
13
14 // SEND + MORE = MONEY:
15 IntVar send =(S*1000+E*100+N*10+D).Var();
16 IntVar more =(M*1000+O*100+R*10+E).Var();
17 IntVar money = (M * 10000 + O * 1000 + N * 100 + E * 10 + Y).Var();
18
19 solver.Add(send + more == money);
20
21 // Leading characters must not be zero:
22 solver.Add(S != 0);
23 solver.Add(M != 0);
```

Cryptogram Solutions

```
1 DecisionBuilder db = solver.MakePhase( vars,
2 Solver.INT_VAR_SIMPLE,
3 Solver.INT_VALUE_SIMPLE);
4
5 solver.NewSearch(db);
6
7 while (solver.NextSolution()) {
8 Console.WriteLine(send.Value() + "+" + more.Value() + "=" + money.Value());
9 }
10 // Display the number of solutions found:
11 Console.WriteLine("Solutions: " + solver.Solutions());
12
13 solver.EndSearch();
```

Ergibt 155 verschiedene, nicht-symmetrische Lösungen. Aber wir haben etwas vergessen!

AllDifferent Constraint

- Verschiedene Variablen müssen verschiedene Werte annehmen.
- Für acht Variablen wären das $(8 * 7) / 2 = 28$ constraints
- OR-Tools hat aber einen speziellen constraint für das:

```
1 IntVar[] vars = new IntVar[] {S, E, N, D, M, O, R, Y};  
2 // All characters must take different values:  
3 solver.Add(vars.AllDifferent());
```

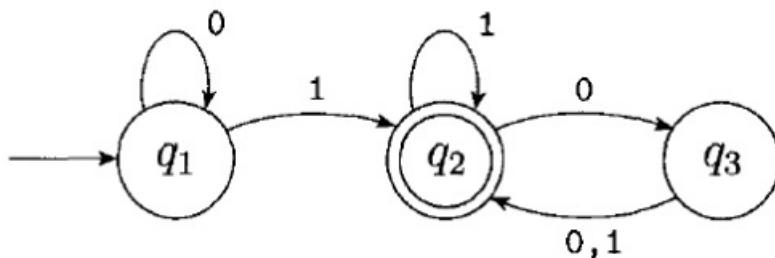
- Mit dieser Constraint gesetzt bleibt eine Lösung:

$$9567 + 1085 = 10652$$

5.1.5 Global Constraints

AllDifferent ist zum Beispiel eine global Constraint. Wann immer möglich diese Global Constraints verwenden.

5.1.6 Deterministic Finite Automaton - DFA



Besteht aus 5 Zutaten:

- Eine Sammlung von input Symbolen (alphabet, 0&1)
- Eine Sammlung von Stati q_1, q_2, q_3
- Nur ein Start Status
- Transitions: $(q_1, 0, q_1), (q_1, 1, q_2), (q_2, 0, q_3), (q_2, 1, q_2), (q_3, 0, q_2), (q_3, 1, q_2)$

5.2 Algorithmen

5.2.1 Implementation einer AllDifferent Constraint für ein Sudoku

```
1 Solver solver = new Solver("Sudoku");  
2 // 9x9 Matrix of Decision Variables in {1..9}:  
3 IntVar[,] board = solver.MakeIntVarMatrix(9, 9, 1, 9);  
4 // Pre-Assignments (only one displayed):
```

```

7 solver.Add(board[0, 0] == 4);
8 ...
9
10 IEnumerable<int> CELL = Enumerable.Range(0, 3);
11 IEnumerable<int> RANGE = Enumerable.Range(0, 9);
12
13 foreach (int i in RANGE) {
14     solver.Add((from j in RANGE select board[i,j]).ToArray().AllDifferent());
15     solver.Add((from j in RANGE select board[j,i]).ToArray().AllDifferent());
16 }
17 foreach (int i in CELL) {
18     foreach (int j in CELL) {
19         solver.Add((from di in CELL from dj in CELL select
20         board[i * 3 + di, j * 3 + dj]).ToArray().AllDifferent());
21     }
22 }

```

5.2.2 The human approach vs Backtrack search

Ein Mensch schaut zuerst, ob es rein logisch gesehn möglich ist, eine Zahl in ein Sudoku einzufüllen. Der Backtrack search hingegen geht nach folgenden Schritten vor:

1. Besuche alle Variablen (Felder) in einer vordefinierten Reihenfolge, beispielsweise von oben nach unten und von links nach rechts.
2. Versuche, eine Variable in die nichtausgefüllten Felder hineinzufüllen aus dem Set der noch nicht ausprobierten Variablen.
3. Überprüfe, ob die Constraints ok sind.
 - a) Wenn mindestens einer verletzt ist, versuche eine andere Variable. Wenn kein Wert mehr im Set der noch nicht ausprobierten Variablen ist, mache einen backtrack zur letzten ausgefüllten Variablen (im Baum eine Stufe höher) und versuche dort einen anderen Wert.
 - b) Wenn kein Constraint verletzt wurde, fahre mit der nächstmöglichen Variablen fort.

Ein Suchbaum sieht z.B. wie in Abbildung 5.1 dargestellt aus.

5.2.3 Advantages & Disadvantages of Backtrack Search

- Backtrack search ist ein Complete-Solution-Algorithmus, das heisst, er wird immer eine Lösung finden, wenn sie existiert
- Der Worst-Case (also wenn keine Lösung existiert) ist, dass Backtrack Search alle möglichen Zuweisungen von Werten zu Variablen vornehmen muss, der Aufwand ist also exponentiell.
- Eine clevere Implementation der Suche muss nur den aktuellen Suchbaumast in das Memory legen. Das bedeutet, dass die Speicherkomplexität linear zur Anzahl Variablen ist.
- Suchen sind gut für die Parallelisation, man kann zum Beispiel verschiedene Äste auf verschiedene Kerne verteilen.

5.2.4 Forward Checking - What humans do

Hier werden systematisch im Voraus alle Variablen, die überhaupt gar nicht möglich sind, aus dem Set der nicht-ausprobierten Variablen gestrichen (Siehe Abbildung 5.2).

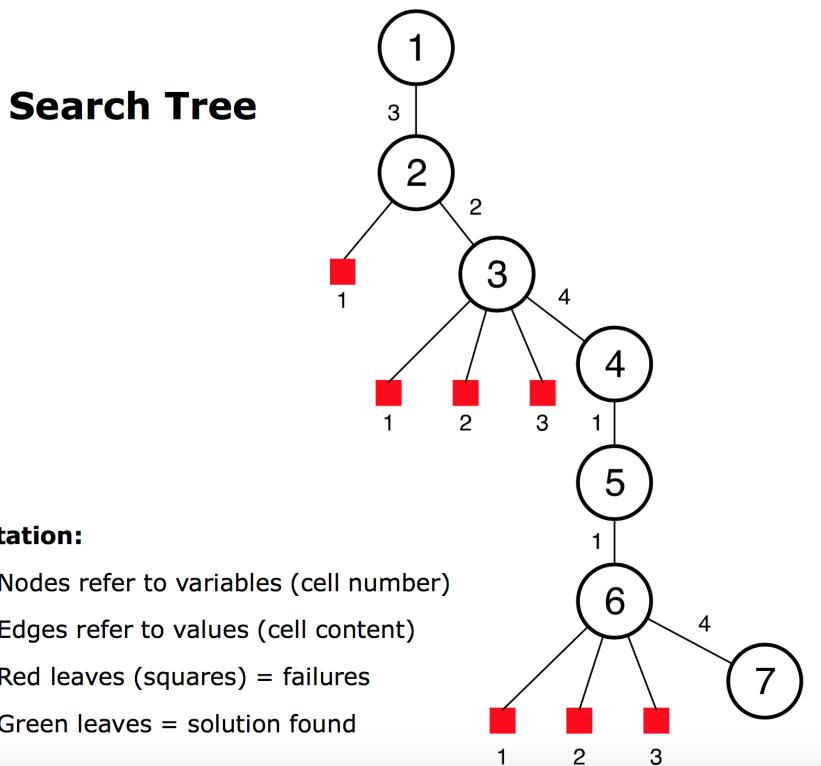


Abbildung 5.1: Backtrack Search Tree

Abbildung 5.2: Backtrack Search Tree

Eigenschaften

- Forward checking führt zu keinem Resultat, sondern macht das Problem simpler. Ist also kein Complete Solution Algorithmus.
 - Jetzt könnte man den Backtrack search auf das vereinfachte Puzzle anwenden.
 - Dann würden wir der ersten nicht ausgefüllten Zelle die erste mögliche Zahl aus dem Set zuweisen (1) und dann könnte man nochmals Forward Checking machen, um das Set der anderen Variablen zu verkleinern.

5.2.5 Bounds consistency

Kann man noch mehr als Forward Checking machen? NA KLAR!

Value graphs

Dazu eine kurze Geschichte:

Alice, Bob und Cecile sind Securitys in einem Nachtclub

Am nächsten Wochenende werden von Freitag bis Sonntag Partys sein. Ein Security muss pro Nacht mindestens anwesend sein. Alice und Bob können beide freitags und samstags arbeiten, aber am Sonntag nicht. Cecile hingegen hat keine Freunde und könnte deswegen das ganze Wochenende Arbeiten. Erstelle einen Schichtplan!

- Variablen: Alice, Bob und Cecile
- Werte: Freitag, Samstag und Sonntag
- Frames: A und B können jeweils die Werte {1, 2} annehmen, C kann {1, 2, 3} annehmen.
- Constraint: AllDifferent(A,B,C).

Das kann man nun als Value Graph darstellen (Abbildung 5.3).

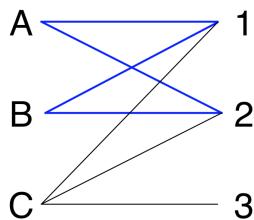


Abbildung 5.3: Value Graph Bounds Consistency

Strongly connected component

Wenn wir nun den Value Graph anschauen, sehen wir direkt, dass C = 3 ist.

Somit gibt es zwei Lösungen: (A = 1, B = 2, C = 3) und (A = 2, B = 1, C = 3)

Grundregel: keine Variable ausserhalb der Strongly connected component (Blau dargestellt), darf einen Wert innerhalb derselben annehmen.

C darf also NICHT den Wert 1 oder 2 annehmen, da diese in der Strongly Connected Component liegen.

5.2.6 Domain Consistency

Kann man noch mehr machen? Jup, weil Bounds consistency schaut nur Wertintervalle an, ignoriert aber Intervalle mit Löchern. Eine kleine Änderung an der Geschichte macht Bounds consistency nutzlos, aber für einen menschlichen Leser hat es dennoch eine einfache Lösung.

Alice, Bob und Cecile sind Securitys in einem Nachtclub

Am nächsten Wochenende werden von Freitag bis Sonntag Partys sein. Ein Security muss pro Nacht mindestens anwesend sein. Alice und Bob können beide freitags und sonntags arbeiten, aber am Samstag haben sie ein Picknick. Cecile hingegen hat keine Freunde und könnte deswegen das ganze

Wochenende Arbeiten. Erstelle einen Schichtplan!

Modifikationen:

- Frames: A und B können nun Werte von $\{1, 3\}$ annehmen, und C $\{1, 2, 3\}$.

Lösungen:

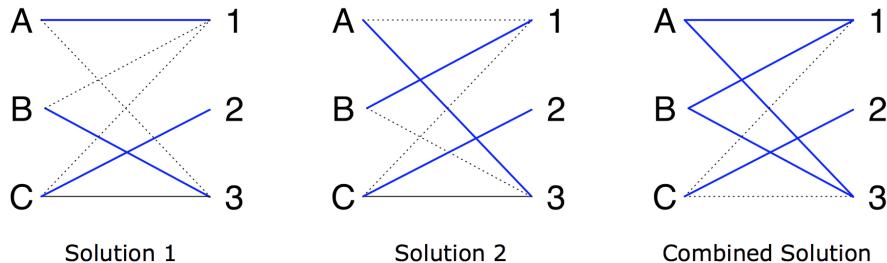


Abbildung 5.4: Value Graph Domain Consistency

Wieso findet Bounds consistency keine Lösung

Bounds consistency schaut nur Intervalle an, die zusammenhängend sind ($\{1, 3\}$ ist kein zusammenhängendes Intervall).

Vergleich von Initial Propagation

4	1 2	8	2 3	3	2 3	1	1	6	5 6
3	2	5 6	5	3	6	6	1	7	9 7 8
6	5	6	5	1	7	6	4	5	6 5 6
9	8	9	8	9	8	9	8	9	8 9
1	6	1	5	4 5	8	6	4	5	3 2
7	9	8	9	8	9	8	2	5	3
1	4	6	3	3	8	2	5	7	3
5	9	2	4	4	7	7	8	7	6
8	3	7	6	2	5	9	4	1	3
2	7	4	3	3	1	3	1	3	3
3	6	5	6	3	2	3	1	2	3
6	5	6	5	1	4	5	2	3	5
9	8	9	7	8	9	7	8	9	8
3	1	5	2	3	3	2	3	1	2
9	8	9	7	8	9	7	8	9	4

Forward Checking

4	1 2	8	5 6	2 3	1	1			
3	2	5	1	7	2	3	4 5	6	5
9	8	9	8	9	8	9	8	9	8 9
7	6	1	5	4 8	9	1	5	3	2
1	4	6	3	3	8	2	5	7	3
5	9	2	7	4	1	7	8	6	3
8	3	7	6	2	5	9	4	1	3
2	7	4	3	8	5	6	1	3	3
6	5	8	3	2	1	4	5	2	3
9	8	9	7	8	7	8	7	8	9
3	1	5	2	3	3	2	3	1	2
9	8	9	7	8	9	7	8	9	4

Bounds Consistency

4	2	8	5	6	3	1	7	9	7
3	5	9	1	7	2	4	6	8	3
7	6	1	4	8	9	5	3	2	7
1	4	6	7	9	8	2	5	7	3
5	9	2	7	4	1	7	8	6	3
8	3	7	6	2	5	9	4	1	3
2	7	4	3	8	5	6	1	3	3
6	8	2	1	4	7	9	5	7	9
3	1	5	8	9	7	6	2	4	6
9	8	9	7	8	9	7	8	9	5

Domain Consistency

Abbildung 5.5: Vergleich von Initial Propagation

5.2.7 Optimization Problems

Es muss nicht nur eine Lösung gefunden werden sondern die beste Lösung.

Beispiel Graph Colouring

Es muss versucht werden einen Karte so zu zeichnen das kein Land welches sich berürt die gleiche Farbe hat, es sollen möglichst wenig Farben verwendet werden(Optimale Lösung).

Es werden zwei Arten von Entscheidungsvariablen genutzt:

- Nodes: Speichert Farbe jedes Landes
- Colors: Zählt wie häufig jede Farbe gebraucht wird

Ein globaler Constraint zählt die Verwendung jeder Farbe:

- `makeCount(IntVar[] nodes, int color, IntVar count)` ensures that $|i : nodes[i] == color| == count$

Optimization Problems Beispiel

```

1 Solver solver = new Solver("Coloring");
2 // One Decision Variable per Node:
3 IntVar[] nodes = solver.MakeIntVarArray(nbNodes, 0, nbNodes-1);
4 foreach (var edge in edges) {
5     solver.Add(nodes[edge.Item1] != nodes[edge.Item2]);
6 }
7
8 // Some limited Symmetry Breaking:
9 solver.Add(nodes[0] == 0);
10 // The number of times each Color is used:
11 IntVar[] colors = solver.MakeIntVarArray(nbNodes, 0, nbNodes-1);
12 for (int i = 0; i < colors.Length; i++) {
13     // MakeCount fügt neue Globale Konstante hinzu!
14     solver.Add(solver.MakeCount(nodes, i, colors[i]));
15 }
16 // Objective function: the Number of Colors used:
17 IntVar obj = solver.MakeSum(
18     (from j in colors select (j > 0).Var()).ToArray().Var();
19     ...
20 DecisionBuilder db = solver.MakePhase
21 (
22     nodes, Solver.INT_VAR_SIMPLE, Solver.INT_VALUE_SIMPLE);
23 // Remember only the best solution found:
24 // Wenn MakeBestValueSolutionCollector = false => minimieren;; true maximieren
25 SolutionCollector collector = solver.MakeBestValueSolutionCollector(false);
26 collector.AddObjective(obj);
27 // What to remember in addition to the objective function value:
28 collector.Add(nodes);
29 if (solver.Solve(db, collector)) {
30     // Extract best solution found:
31     Assignment sol = collector.Solution(0);
32     Console.WriteLine("Solution found with " + sol.ObjectiveValue() + " colors.");
33     for (int i = 0; i < nodes.Length; i++)
34     {
35         Console.WriteLine("Node " + i + " obtains color " + sol.Value(nodes[i]));
36     }
37 }
```

Standart Rezept:

- 1 Erstelle DecisionBuilder mit decision variable welche gelöst werden soll
- 2 Erstelle SolutionCollector und füge objective function hinzu
- 3 Suchprozess starten und alle Lösungen ausgeben.

6 Bayessche Netze - Reasoning under Uncertainty

6.1 Einführung

Viele Probleme sind oft zu komplex, um mit ihnen intuitiv umzugehen. Deswegen muss man diese aufspalten. Bayessche Netze sind ein guter Weg, um Abhängigkeiten zwischen Wahrscheinlichkeiten zu vereinfachen.

6.2 Begriffe

- Sensitivität: Wieviel Prozent des Resultats sind true-positive?
- Spezifität: Wieviel Prozent des Resultats sind true-negative?
- true-positive: Richtig erkannte positive Ausgänge eines Tests (Beispiel: $p(\text{positive}|\text{HIV})$).
- true-negative: Richtig erkannte negative Ausgänge eines Tests (Beispiel: $p(\text{negative}|\neg\text{HIV})$).
- false-positive: Falsch erkannte positive Ausgänge eines Tests (Beispiel: $p(\text{positive}|\neg\text{HIV})$).
- false-negative: Falsch erkannte negative Ausgänge eines Tests (Beispiel: $p(\text{negative}|\text{HIV})$).

6.3 From Real World to Random Variables

Folgende Geschichte:

In Los Angeles sind Einbrüche und Erdbeben häufig. Beide können einen Alarm auslösen. Wenn der Alarm losgeht, kann es sein, dass mich meine Nachbarn, John oder Mary, anrufen.

1. Wo sind Unsicherheiten in der Geschichte?
2. Wenn diese gefunden sind, muss man diesen sogenannte Zufallsvariablen ("Random variables") zuweisen:
 - $B \Rightarrow$ Einbruch
 - $E \Rightarrow$ Erdbeben
 - $A \Rightarrow$ Alarm
 - $J \Rightarrow$ John ruft an
 - $M \Rightarrow$ Mary ruft an
3. Danach muss man das Set der möglichen Werte jeder Variablen herausfinden (Hier nur true oder false, die Zufallsvariablen sind also binär).

6.4 Global- und Marginalverteilung

Die multivariate Verteilung (Joint-Distribution) $P(A, B, E, J, M)$ über alle Variablen ist die **Globale Wahrscheinlichkeitsverteilung**. Wenn diese bekannt ist, kann man daraus - durch den Vorgang der **Marginalisation** - die anderen multivariaten Verteilungen ableiten. Diese werden Marginalverteilungen genannt.

6.5 Bayes Theorem

$$P(B_j|A) = \frac{P(B_j)P(A|B_j)}{P(A)}$$

Sprich: Die Wahrscheinlichkeit, dass B_j gegeben A eintritt, ist die Wahrscheinlichkeit, dass B_j eintritt mal die Wahrscheinlichkeit, dass A gegeben B_j eintritt, durch die Wahrscheinlichkeit, dass A eintritt (Denke an das AIDS-Beispiel, die Wahrscheinlichkeit, dass jemand tatsächlich AIDS, gegeben Test ist positiv, hat, ist gleich der Wahrscheinlichkeit, überhaupt AIDS zu haben (ist ja länderspezifisch) mal der Wahrscheinlichkeit, bei AIDS überhaupt positiv getestet zu werden, durch die Wahrscheinlichkeit, positiv zu sein).

6.6 Causal Relations and Causal Networks

Hier fragt man sich eigentlich, welche Variablen von welchen abhängen. Zum obigen Beispiel lässt sich sagen, dass:

- Über die Abhängigkeit von Einbruch oder Erdbeben nichts genaueres bekannt ist.
- Der Alarm von Einbruch oder Erdbeben abhängt.
- Der Anruf von Mary oder John vom Alarm abhängt.

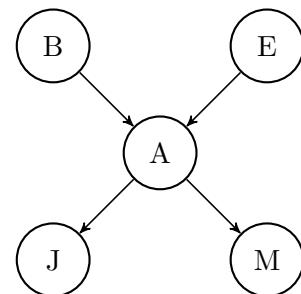


Abbildung 6.1: Causal relations

6.6.1 Conditional Dependency Relations

Bedingte Abhängigkeitsbeziehungen entstehen durch die in Abbildung 6.1 dargestellten Pfeile. Somit lässt sich sagen, dass die Wahrscheinlichkeit, dass A eintrifft direkt von der Wahrscheinlichkeit abhängt, ob E und B eingetroffen sind. Aus der Wahrscheinlichkeit von A lässt sich wiederum **direkt** die Wahrscheinlichkeit für J oder M berechnen. Dies wird so geschrieben:

$$\begin{aligned} P(A|E, B) &\Rightarrow \text{Wahrscheinlichkeit des Alarms, gegeben Einbruch und Erdbeben} \\ P(J|A) &\Rightarrow \text{Wahrscheinlichkeit, dass Mary anruft, gegeben Alarm} \\ &\text{usw...} \end{aligned} \tag{6.1}$$

J ist also **BEDINGT UNABHÄNGIG** (**conditional independent**) von B , wenn der Wert von A bekannt ist.

6.6.2 Factorization

Für die Faktorisierung ist es wichtig, dass die Knoten aufgezeichnet werden. Nur direkt eintreffende Pfeile haben einen Einfluss auf den Knoten (E hat zum Beispiel keinen Einfluss auf B und umgekehrt).

Dann lässt sich die folgende Faktorisierung rein optisch herleiten:

$$P(A, B, E, J, M) = P(J|A) * P(M|A) * P(A|B, E) * P(B) * P(E)$$

Für die Werte würde man jetzt gegebene Wahrscheinlichkeiten einsetzen.

6.7 Fusion algorithm

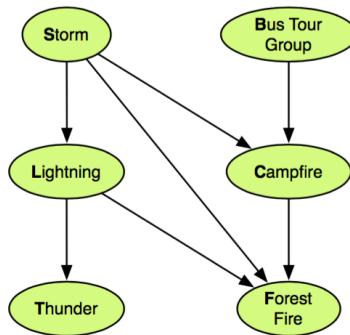
Der Fusion Algorithmus ist ein Algorithmus, um bedingte Wahrscheinlichkeiten ressourceneffizienter berechnen zu können. Als Input wird benötigt:

- eine sogenannte Wissensbasis (knowledgebase), die am Anfang aus den faktorisierten 6.6.2 Marginalverteilungen besteht.
- eine Eliminationssequenz (hier empfiehlt es sich meiner Meinung nach, eine Sequenz zu wählen, die am Anfang eine Variable hat, die wenig und nicht in allzu grossen Faktoren vorkommt)

Für dieses Beispiel verwenden wir das Beispiel aus den Übungen:

4 Fire Detection (1 Point)

The following network is part of an early detection and alarm system for forest fires.



1. Give the factorization of the global probability distribution represented by this Bayesian networks.
2. Show the formula that computes the probability of a forest fire given a campfire and thunder. Identify the marginals to be computed.
3. Carry out the fusion algorithm for $p(F, T)$ with the elimination sequence $L \rightarrow S \rightarrow B \rightarrow C$ and determine the complexity of these computations.
4. Can you find a better elimination sequence that shows lower complexity?

6.7.1 Marginalization

Für Aufgabe 3 mit der Eliminationssequenz $L \rightarrow S \rightarrow B \rightarrow C$ machen wir Folgendes:

1. Die Wissensbasis ist: $P(S, L, T, F, C, B) = P(F|C, S, L) * P(L|S) * P(T|L) * P(C|B, S) * P(S) * P(B)$
Alle Faktoren, die L enthalten summieren: $\psi_L = \sum_L P(F|C, S, L) * P(L|S) * P(T|L)$
2. Die Wissensbasis hat sich verändert, die Faktoren, die L enthalten verschwinden, stattdessen wird ψ_L in die Wissensbasis aufgenommen. ψ_L enthält die Variablen $\{F, C, S, T\}$. Das ist wichtig, da man ja wissen muss, wann man ψ_L wieder verwenden muss zur Elimination. Die Wissensbasis ist nun: $P(C|B, S) * P(S) * P(B) * \psi_L$
Nun summiert man alle Faktoren, die S enthalten: $\psi_S = \sum_S P(C|B, S) * P(S) * \psi_L$

3. Neue Wissensbasis: $P(B) * \psi_S$ (ψ_S enthält die Variablen: $\{F, C, T, B\}$)
B entfernen: $\sum_B P(B) * \psi_S$
4. Neue Wissensbasis: ψ_B (ψ_B enthält die Variablen: $\{F, C, T\}$)
C entfernen: $\psi_C = \sum_C \psi_B$
$$P(F, T) = \sum_{F,T} \psi_C$$

6.7.2 Back substitution

Für die Rücksubstitution (also um die richtige Berechnungssequenz für diese Eliminationssequenz zu erhalten) geht man obige Schritte eigentlich rückwärts und setzt ein:

1. Ausgangslage:
$$P(F, T) = \sum_{F,T} \psi_C$$
2. ψ_C rücksubstituieren:
$$P(F, T) = \sum_{F,T} (\sum_C \psi_B)$$
3. ψ_B rücksubstituieren:
$$P(F, T) = \sum_{F,T} (\sum_C (\sum_B P(B) * \psi_S))$$
4. ψ_S rücksubstituieren:
$$P(F, T) = \sum_{F,T} (\sum_C (\sum_B P(B) * (\sum_S P(C|B, S) * P(S) * \psi_L)))$$
5. ψ_L rücksubstituieren:
$$P(F, T) = \sum_{F,T} (\sum_C (\sum_B P(B) * (\sum_S P(C|B, S) * P(S) * (\sum_L (P(F|C, S, L) * P(L|S) * P(T|L))))))$$

7 Markovketten - Reasoning under uncertainty

part 4

7.1 Unterschied Markovketten und Bayes'sche Netze

Der Faktor Zeit wird bei Bayes'schen Netzen nicht explizit beachtet, sie beachten lediglich Gegebenheiten, wie z.B.:

- Staus sind häufiger in der Rush Hour
- Die Meinungen der Leute ändern sich mit der Zeit
- Das Wetter von morgen hängt von heute ab.

Markovketten bieten nun die Möglichkeit, diese Gegebenheiten genauer zu analysieren.

7.2 Erstes Beispiel: Marktanalyse

Der Kantinenchef der Mensa hat 70% der HSLU-Studenten als regelmässige Kunden. Er will eine Firma anheuern, um den Effekt einer aggressiven Marketingkampagne abschätzen zu können. Die Resultate sagen aus, dass nach einer Woche der Kampagne ein Student, der vorher immer in der Mensa gegessen hat, mit 90% Wahrscheinlichkeit immer noch in der Mensa essen wird. Ein Student, der nicht in der Mensa gegessen hat, wird mit 40% Wahrscheinlichkeit sein Verhalten ändern.

7.2.1 Vorgehen

1. Wo ist Unsicherheit und Zeit in dieser Geschichte?
 - Unsicherheit: 90% bleiben, 40% ändern Verhalten
 - Zeit: Periode von einer Woche
2. Identifikation aller relevanten Komponenten und Zuweisen der Zufallsvariablen dafür.
 - M → Mensa
 - E → Elsewhere
3. Identifizieren des Sets von möglichen Werten für die Zufallsvariablen (Dies wird *Frame* genannt):
 - M → M = 90%
 - M → E = 10%
 - E → M = 40%
 - E → E = 60%
4. Zeichnen des Transitionsdiagrammes: Grundregel für ein Transitionsdiagramm: Die Pfeile, die von einer Variablen weg zeigen, summieren sich zu 1.

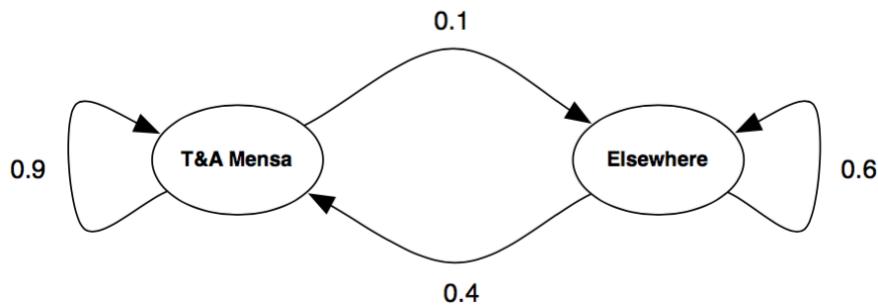


Abbildung 7.1: Transitiondiagramm Mensa

7.2.2 Transitionsmatrix

Abbildung 7.1 ist für Menschen sehr einfach zu verstehen. Computer brauchen allerdings eine andere Darstellung, die **Transitionsmatrix**:

$$M_{transition} = \begin{bmatrix} 0.9 & 0.1 \\ 0.4 & 0.6 \end{bmatrix}$$

Die Zeilen dieser Matrizen summieren sich zu 1, es ist also eine stochastische Matrix. Benutzt wird sie folgendermassen:

1. Man hat gegebene Inputwerte. In diesem Fall essen 70% der Leute in der Mensa. Das heisst, dass 30% ausserhalb essen.
2. Man erstellt nun einen Vektor, der die Ausgangslage beschreibt: $\begin{bmatrix} 0.7 & 0.3 \end{bmatrix}$
3. Man rechnet den Vektor mal die Matrix hoch die Anzahl der Zeitperioden (Wochen in diesem Fall): $\begin{bmatrix} 0.7 & 0.3 \end{bmatrix} M_{transition}^n$, wobei n die Anzahl Wochen ist.
4. Resultate auslesen.

7.3 Stationary Distribution

Setzt man für die obige Aufgabe den Anfangszustand auf 80% der Studenten essen in der Mensa und der Rest auswärts, erhält man immer wieder denselben Vektor als Resultat:

$$\begin{bmatrix} 0.8 & 0.2 \end{bmatrix} \begin{bmatrix} 0.9 & 0.1 \\ 0.4 & 0.6 \end{bmatrix} = \begin{bmatrix} 0.8 & 0.2 \end{bmatrix}$$

Das ist wichtig, denn das ist gleich auch die stationäre Verteilung der Aufgabe.

7.3.1 Berechnung

Das ist ja schön und gut, aber wie berechnet man jetzt die stationäre Verteilung? Das ist einfach und zwar stellt man ein Gleichungssystem auf, welches man aus der Transitionsmatrix erstellt:

$$\begin{aligned}
 \begin{bmatrix} p_1 & p_2 \end{bmatrix} \begin{bmatrix} 0.9 & 0.1 \\ 0.4 & 0.6 \end{bmatrix} &= \begin{bmatrix} p_1 & p_2 \end{bmatrix} \\
 \Rightarrow 0.9p_1 + 0.4p_2 &= p_1 \\
 0.1p_1 + 0.6p_2 &= p_2
 \end{aligned} \tag{7.1}$$

Dieses Gleichungssystem ist linear abhängig, deswegen **unlösbar**!

Die Wahrscheinlichkeit, dass jemand überhaupt irgendwo isst, ist allerdings bei 100% (Für dieses Beispiel). Diese Bedingung lässt sich in das Gleichungssystem einfügen. Man darf jedoch zusätzlich eine der beiden anderen Gleichungen streichen (die die einem am wenigsten gefällt):

$$\begin{aligned} 0.9p_1 + 0.4p_2 &= p_1 \\ p_1 + p_2 &= 1 \end{aligned}$$

ergibt aufgelöst:

$$\begin{aligned} p_1 &= 0.8 \\ p_2 &= 0.2 \end{aligned} \tag{7.2}$$

TADAAAAA!!!! #super #GleichigssytemSindEasy #BestDayEva<3 #StationaryDistributionAt80%/20%

Gut, nachdem wir uns beruhigt haben, gehen wir zum nächsten Thema über.

7.4 Absorbing and Transient states

Ein absorbierender Status ist, wenn die Wahrscheinlichkeit, ihm zu entkommen bei 0 liegt. Anderst gesagt: Wenn kein Pfeil von ihm wegzeigt (ausgenommen ein Schleifenpfeil mit 100%), kommt man nicht mehr raus.

Beispiel einer Geschichte mit absorbierenden Zuständen:

Hey Bob, ich möchte ein Spiel spielen!

Alice nimmt eine faire Münze und Bob nimmt einen fairen sechsseitigen Würfel. Wenn Alice dran ist, wirft sie die Münze. Wenn die Münze Kopf zeigt, gewinnt Alice. Wenn die Münze Zahl zeigt, ist Bob an der Reihe.

Wenn Bob an der Reihe ist, wirft er den Würfel. Wenn 1 gezeigt wird, gewinnt er, wenn der Würfel 6 zeigt, ist Alice dran. Ansonsten darf Bob nochmal würfeln.

Das Transitionsdiagramm ist in 7.2 dargestellt.

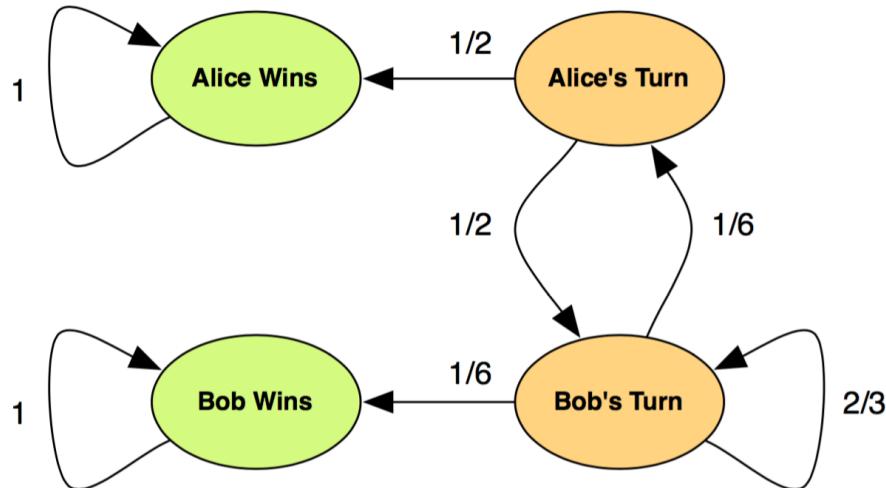


Abbildung 7.2: Transitionsdiagramm Alice und Bob

Die Transitionsmatrix dazu sieht folgendermassen aus:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1/2 & 0 & 0 & 1/2 \\ 0 & 1/6 & 1/6 & 2/3 \end{bmatrix}$$

Dabei ist folgende Grundregel zu betrachten: *Wenn absorbierende Zustände vorhanden sind, schreibe sie immer zuerst!* (das haben wir bereits beachtet). Dann lässt sich die Matrix aufteilen in folgende Abschnitte:

$$\begin{array}{c|c} I & 0 \\ \hline R & Q \end{array}$$

Wobei die Matrix

- I für die Wahrscheinlichkeiten steht, in denen zwischen absorbierenden Zuständen gewechselt wird.
- 0 für die Wahrscheinlichkeiten steht, einen absorbierenden Zustand zu verlassen.
- R für die Wahrscheinlichkeit steht, in einen absorbierenden Zustand zu gelangen aus einem transienten.
- Q für die Wahrscheinlichkeit steht, zwischen transienten Zuständen zu wechseln.

$$\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \hline 1/2 & 0 & 0 & 1/2 \\ 0 & 1/6 & 1/6 & 2/3 \end{array} = \begin{array}{c|c} I & 0 \\ \hline R & Q \end{array}$$

7.4.1 In the long run

Die Formel, um die Gewinnwahrscheinlichkeit pro Spieler zu bestimmen, ist folgende:

$$\begin{bmatrix} I & 0 \\ (I - Q)^{-1}R & 0 \end{bmatrix}$$

In unserem Beispiel wäre $(I - Q)^{-1}R$

$$(I - Q) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 1/2 \\ 1/6 & 2/3 \end{bmatrix} = \begin{bmatrix} 1 & 1/2 \\ 5/6 & 1/3 \end{bmatrix}$$

weil:

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \quad (7.3)$$

$$(I - Q)^{-1} = 4 * \begin{bmatrix} 1/3 & 1/2 \\ 1/6 & 1 \end{bmatrix} = \begin{bmatrix} 4/3 & 2 \\ 2/3 & 4 \end{bmatrix}$$

$$(I - Q)^{-1} * R = \begin{bmatrix} 4/3 & 2 \\ 2/3 & 4 \end{bmatrix} * \begin{bmatrix} 1/2 & 0 \\ 0 & 1/6 \end{bmatrix} = \begin{bmatrix} 2/3 & 1/3 \\ 1/3 & 2/3 \end{bmatrix}$$

Das bedeutet, dass die Gewinnchance für Bob bei $1/3$ liegt und für Alice bei $2/3$, wenn Alice zuerst spielt und umgekehrt, wenn Bob zuerst spielt.

7.4.2 Expected number of transitions

Aus der obigen Matrix $(I - Q)^{-1}$ können wir noch mehr sagen. Die Zeilen dieser Matrix addiert ergeben die Anzahl der Transitionen (Runden), die es dauert, bis der erwartete Wert herauskommt. Wenn nun also Alice als erstes zieht, dauert es durchschnittlich $2 + 4/3 = 3.33$ Runden, bis sie gewinnt.

Wenn Bob zuerst zieht, dauert es durchschnittlich $2/3 + 4 = 4.66$ Runden, bis er gewinnt.

8 Statistische Evaluation

8.1 Lage- u. Streumasse, Häufigkeitsverteilung

8.1.1 arithm. Mittelwert

$$\bar{x} = \frac{1}{n}(x_1 + x_2 + \dots + x_n) = \frac{1}{n} \sum_{i=1}^n x_i$$

8.1.2 Median

1. Ordnen der Werte in aufsteigender Reihenfolge
2. Anzahl Werte * 0.5 (oder α - Quantil) ergibt aufgerundet die Stelle in der Liste mit dem Median Wert.

Beispiel:

$$\overleftarrow{x} = [1, 2, 4, 5, 7] \Rightarrow n = 5 \Rightarrow 5 * 0.5 = 2.5$$

Aufgerundet ergibt sich 3 also ist der 3te Wert (4) der Wert des Median.

8.1.3 Varianz

$$var(x) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

8.1.4 Standardabweichung

$$s_x = \sqrt{var(x)}$$

8.2 Diskrete Verteilung

8.2.1 Poisson Verteilung

$$f(x) = P(X = x) = \frac{\lambda^x}{x!} \exp(-\lambda)$$

Mittelwert: $\mu = \lambda$ Varianz: $\sigma^2 = \lambda$

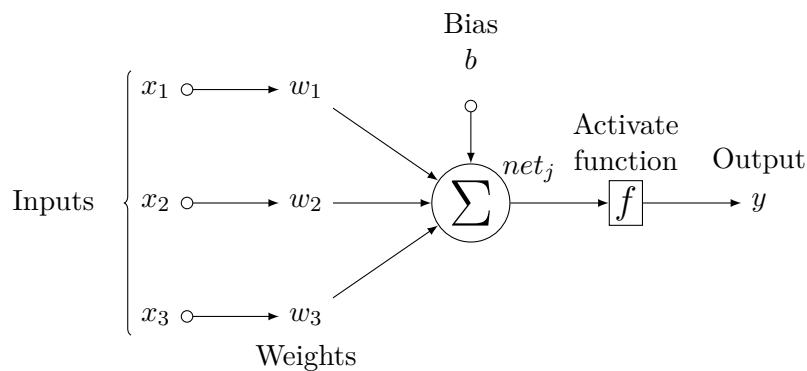
9 Neuronale Netze

9.1 Motivation

- Künstliche Neuronale Netzwerke (engl. Artificial Neural Networks [ANN]) imitieren biologische Neuronale Systeme.
- Grosse Anzahl Verarbeitungseinheiten, genannt Neuronen
- Verbundene Neuronen tauschen Signale aus
- Neuron kombiniert die periodisch ankommenden Signale (anderer, benachbarter Neuronen) und berechnet daraus ein Signal, welches an weitere, benachbarte Neuronen geschickt wird
- Eine grosse Anzahl zusammen arbeitender Neuronen kann komplexe Aufgaben lösen
- Heute arbeitende ANN haben ein Grösse von weniger als 1% des menschlichen Gehirns

9.2 Abstraktion eines Neurons

- Ein Neuron hat mehrere Eingänge mit gewichteten Inputs
- Neuron **feuert**, wenn eine Funktion des Inputs einen bestimmten Schwellwert überschreitet
- Typische Dauer eines Prozesses (Gesichtserkennung) $\approx 1/10$ s dabei sind ca. 100 Ebenen/-Schritte involviert
- Output eines Neurons ist wieder Input mehrerer weiterer Neuronen.



9.2.1 Net-Input

Der net-Input eines Knotens j ergibt sich aus:

$$net_j = \sum_{i=0}^m x_i w_{ij}$$

Wobei:

- x_i : **Input** des Knotens i ($x_0 = 1$ das ist fix!)
- w_{ij} : **Gewicht** (also aus der Gewichtsmatrix stammender Wert) des Signals der Verbindung von Knoten i zu Knoten j ; manchmal auch als w_{ji} geschrieben.
- w_{0j} : **Bias**

9.2.2 Aktivierungsfunktion

Der net-Input des Knotens muss nun noch durch eine Aktivierungsfunktion geschickt werden, um zu bestimmen, ob das Neuron schlussendlich feuert oder nicht.

$$y_j = \phi(\text{net}_j) = \phi\left(\sum_{i=0}^m x_i w_{ij}\right)$$

Heaviside als Aktivierungsfunktion

Wenn für net_j nach obiger Berechnung ein Wert (z.B. 0.2) in die Heavisidefunktion eingesetzt wird, kommt es darauf an, wie diese definiert ist. Grundsätzlich kann man aber sagen, dass net_j in der Abbildung 9.1 der x-Achsen-Wert ist.

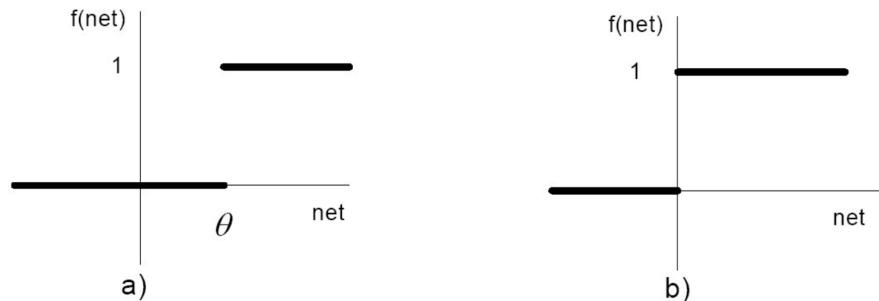


Abbildung 9.1: Heaviside Funktion

Für a) wäre der Wert der Funktion 0 (wenn man davon ausgeht, dass θ bei 0.5 den Sprung macht). Für b) wäre er 1.

Stückweise lineare und sigmoide Funktionen als Aktivierungsfunktion

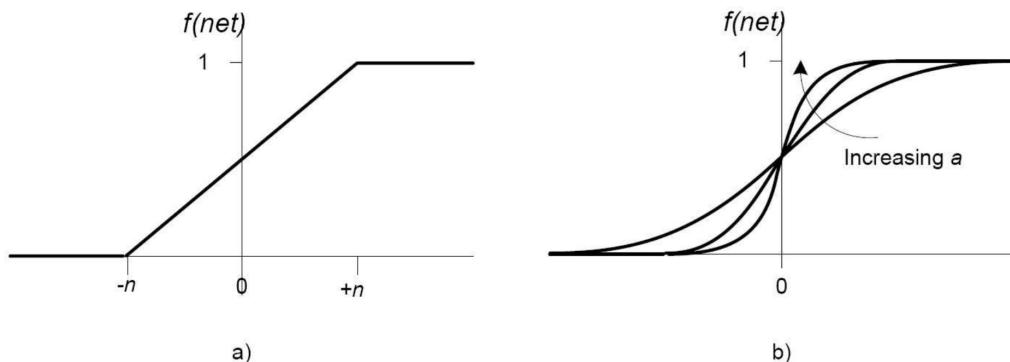


Abbildung 9.2: Lineare und sigmoide Funktion

$$f(\text{net}) = \underbrace{\begin{cases} 0 & \text{für } \text{net}_j \leq -n \\ \frac{\text{net}_j}{2n} + \frac{1}{2} & \text{für } -n \leq \text{net}_j \leq n \\ 1 & \text{für } \text{net}_j \geq n \end{cases}}_{a)} \quad b) \quad f(\text{net}) = \frac{1}{1 + e^{-a\text{net}}}$$

Geht $a \rightarrow \infty$, ist die Sigmoidfunktion (b) gleich der Schwellenfunktion mit dem Unterschied, dass sie an jeder beliebigen Stelle ableitbar ist.

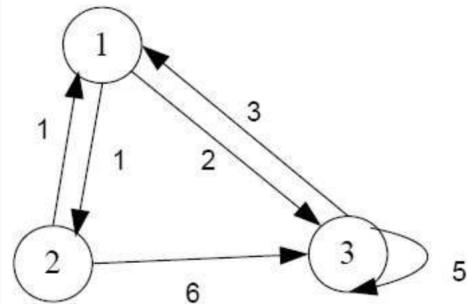
Weitere Form der Sigmoidfunktion:

$$\begin{aligned} f(a) &= \tanh a = \frac{e^a - e^{-a}}{e^a + e^{-a}} \\ f'(a) &= \frac{df}{da} = 1 - [\tanh(a)]^2 \end{aligned} \quad (9.1)$$

9.3 Architektur neuronaler Netze

9.3.1 Beschreibung durch Matrix

Das Matrixelement w_{ij} enthält das Gewicht der Verbindung von Neuron i zum Neuron j (in der Literatur findet man auch die umgekehrte Definition!)



$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 6 \\ 3 & 0 & 5 \end{bmatrix}$$

Die Matrix beschreibt somit nicht nur die Topologie, sondern enthält zudem auch die Information über die Gewichte. Die Matrix kommt auch oft transponiert daher!

9.4 Single Layer Feedforward Netze

9.4.1 Wie bestimmt man die Gewichte

Neuronenverbindungen, die oft genutzt werden, verstärken sich! Oder,
Übung macht den Meister!

Korrektur des Gewichtes w_{ij} vom Input x_i zum Output y_j :

$$\Delta w_{ij} = \eta x_i y_j, \quad \eta > 0 \text{ eine Konstante, typisch } 0 < \eta < 1/2$$

Verwendet wird meist die auf Widrow und Hoff (1960) zurückgehende (Delta-) Δ -Regel:

$$\Delta w_{ij} = \eta x_i e_j, \quad \text{neues Gewicht ist } w'_{ij} = w_{ij} + \Delta w_{ij}$$

$e_j = t_j - y_j$: Differenz zwischen Zielwert t_j und tatsächlichem Output y_j .
 $e_j > 0 \rightarrow w_{ij}$ wird erhöht; $e_j < 0 \rightarrow w_{ij}$ wird reduziert.

9.5 Multi Layer Feedforward Netze

9.5.1 Mehrstufige Perzeptrons (Backpropagation Netze)

- Eingabeschicht liefert am Knoten i den Wert x_i (aus Trainingsmuster)
- (Jede) versteckte Schicht produziert

$$h_j = \sigma(\text{net}_j) = \sigma\left(\sum_{i=0}^{n_i} w_{ij} x_i\right)$$

- Ausgabeschicht liefert im Knoten k

$$o_k = \sigma(\text{net}_k) = \sigma\left(\sum_{j=0}^{n_h} \tilde{w}_{jk} h_j\right) = \sigma\left(\sum_{j=0}^{n_h} \tilde{w}_{jk} \sigma\left(\sum_{i=0}^{n_i} w_{ij} x_i\right)\right)$$

- Fehler $e_k = t_k - o_k$ (mit Zielwert t_k)
- Minimiere Fehler abhängig von allen Gewichten
- Verwende Gradientenabstiegsverfahren (oder ähnlich)

Backpropagation-Regel

Verallgemeinerte Delta-Regel (gilt für w_{ik} und \tilde{w}_{ik})

$$\Delta w_{ik}(n+1) = \eta h_i \delta_k + \alpha \Delta w_{ik}(n)$$

für mehrstufige Netze (η : Lernfaktor). Der Trägesfaktor α berücksichtigt Änderung aus dem letzten Schritt (Momentum). Dabei ist δ_k gegeben durch

- im **Output-Layer**:

$$\delta_k = \sigma'(\text{net}_k) (t_k - o_k)$$

- im **Hidden-Layer** (Summation über alle n_j Nachfolger k von Neuron j):

$$\delta_j = \sigma'(\text{net}_j) \sum_{k=1}^{n_j} \tilde{w}_{jk} \delta_k$$

- Für logistische Funktion $\sigma(x) = (1 + e^{-x})^{-1}$ gilt: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$ und damit wegen $\sigma(\text{net}_k) = o_k$ sofort $\sigma'(\text{net}_k) = o_k(1 - o_k)$
- Für die tanh-Funktion gilt: $\sigma'(\text{net}_k) = (1 - o_k^2)$

Backpropagation-Algorithmus

- ① Lese Inputmuster/Zielwert-Paar (x, t)
- ② Für jedes Neuron j der versteckten Schicht berechne
$$h_j = \sigma\left(\sum_{i=0}^{n_i} w_{ij} x_i\right)$$
- ③ Für jedes Neuron k der Ausgabeschicht berechne
$$o_k = \sigma\left(\sum_{j=0}^{n_h} \tilde{w}_{jk} h_j\right)$$
, Fehler $e_k = t_k - o_k$ und das daraus resultierende Delta $\delta_k = o_k(1 - o_k)e_k$
- ④ Korrigiere die Gewichte \tilde{w}_{ik} der Ausgabeschicht entsprechend
$$\Delta \tilde{w}_{jk}(n+1) = \eta h_j \delta_k + \alpha \Delta \tilde{w}_{jk}(n)$$
- ⑤ Für jedes Neuron j in der versteckten Schicht berechne das Delta
$$\delta_j = h_j(1 - h_j) \sum_{k=1}^{n_o} \delta_k \tilde{w}_{jk}$$
. Summiert wird über alle Nachfolger k des Neurons j
- ⑥ Korrigiere die Gewichte w_{ik} der versteckten Schicht entsprechend
$$\Delta w_{ij}(n+1) = \eta x_i \delta_j + \alpha \Delta w_{ij}(n)$$
- ⑦ Falls noch keine Konvergenz oder noch nicht alle Inputmuster/Zielwert-Paare gehe zu Schritt 1

10 Linear Regression

10.1 Geradengleichung

Gesucht ist eine Gerade in der Form $y = m * x + b$ bei der die Abstände zu den Punkten minimal werden.

$$m = \frac{s_{xy}}{s_x^2}, \quad b = \bar{y} - m\bar{x}$$
$$s_{xy} = \frac{1}{n-1} \sum_{j=1}^n (x^{(j)} - \bar{x})(y^{(j)} - \bar{y})$$
$$s_x^2 = \frac{1}{n-1} \sum_{j=1}^n (x^{(j)} - \bar{x})^2$$

10.2 Confidence Interval

1. Confidence level (γ) wählen (z.B. 0.95 oder 0.99)
2. Lösen der Gleichung

$$F(c) = \frac{1}{2}(1 + \gamma), \quad \text{z.B. } F(x) = \frac{1}{2}(1 + 0.95) \Rightarrow F(x) = 0.975$$

Um c zu erhalten muss dann in der t-Distributions Tabelle mit $n - 2$ Freiheitsgraden der Wert abgelesen werden. z.B. Ergibt bei 5 sample $5 - 2 = 3$ Freiheitsgraden. Aus der Tabelle kann nun der Wert für 0.975 und 3 abgelesen werden. (3.182)

3. Berechnen von

$$(n-1)s_y^2 = \sum_{j=1}^n (y^{(j)} - \bar{y})^2$$

4. Berechnen von

$$q_0 = (n-1)(s_y^2 - m^2 s_x^2)$$

5. Berechne

$$K = c \sqrt{\frac{q_0}{(n-2)(n-1)s_x^2}}$$

- 6.

$$\mathbf{CONF}_\gamma\{m - K \leq \tilde{m} \leq m + K\}$$

10.3 Correlation Coefficient

Der Correlation Coefficient gibt den linearen Zusammenhang der Testsample an.

$$r = \frac{s_{xy}}{s_x s_y}$$

- r=1 Die Sample liegen auf einer Geraden (linear zusammenhängend)
- r=0 Die Sample sind nicht linear zusammenhängen und ergeben eine Wolke

11 Multiple Regression

Multiple Regression wird verwendet wenn ein Resultat von mehreren Variablen abhängig ist.

11.1 Geradengleichung

$$x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 \Rightarrow y = X\theta + \epsilon$$

In Matrix Schreibweise

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_1^{(0)} & x_1^{(1)} & \vdots & x_1^{(m)} \\ x_2^{(0)} & x_2^{(1)} & \vdots & x_2^{(m)} \\ \vdots & \vdots & \ddots & \vdots \\ x_n^{(0)} & x_n^{(1)} & \vdots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_m \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$
$$\theta = (X X^T)^{-1} X y$$