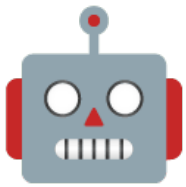


Artificial Intelligence - Search & Optimization

`I.BA_AISO` — Zusammenfassung

Author(s) Dominic, Elias, Laura
Date 30. May. 2026
Pages 89

Inhaltsverzeichnis

1. Agents and Environments

1.1. Agent

DEFINITION: Ein Agent ist ein Computersystem, das in einer Umgebung eingebettet ist und dort eigenständig handelt, um seine Ziele zu erreichen.

- **Wahrnehmung**: Nutz **sensors** z.B. Kamera
- **Entscheidung**: Wählt die Handlung aus, die den grössten Erfolg verspricht
- **Handlung**: Führt die Aktion über **actuators** (Aktoren) aus und verändert so die Umgebung

1.1.1. Rational Agents

DEFINITION: Ein **rational agent** wählt für jede mögliche **percept sequence** (Wahrnehmungssequenz) die Aktion aus, die sein **performance measure** (Erfolgsmass) maximiert.

- **Rationalität hängt ab von**:
  - **Performance Measure**: Der Massstab, der den Erfolg definiert
  - **Prior Knowledge**: Das Vorwissen des Agenten über seine Umgebung
  - **Actions**: Die Handlungen, die der Agent überhaupt ausführen kann
  - **Percept Sequence**: Die bisherige Historie aller Wahrnehmungen
- **Rationalität vs. Perfektion**:
  - **Rational**: Maximiert den **erwarteten Erfolg** basierend auf Informationen
  - **Perfekt**: Maximiert den **tatsächlichen Erfolg** (erfordert oft Allwissenheit)
- **Beispiel Meteorit**: Prüfung vor Strassenüberquerung ist rational (erwarteter Erfolg), mangels Allwissenheit bei Einschlag eines Meteoriten jedoch nicht perfekt (tatsächlicher Erfolg).

1.1.2. Modeling Agents

- **Percept sequence (Perzeptsequenz)**: Historie aller bisherigen Wahrnehmungen des Agenten
- **Agent's function**: Theoretische Tabelle, die jeder **percept sequence** eine **action** zuordnet
- **Agent program**: Implementierung der **agent function**

percept sequence	Action
<A, clean>	Right
<A, dirty>	Suck
<B, clean>	Left
<B, dirty>	Suck
<A, clean>, <A, clean>	right
<B, clean>, <B, clean>	left
...	...

1.1.3. Performance Measure

DEFINITION: Objektive Bewertung der Umweltzustände durch einen **externen Beobachter**.

- **Fokus**: Bewertet das **Resultat** in der Welt, nicht den Aufwand oder Glauben des Agenten
- **Zweck**: Verhindert Selbsttäuschung (Agent deklariert Erfolg, ohne die Welt zu verbessern)
- **Risiko**: Falsche Metriken führen zu falschem Verhalten
  - z. B. Dreck wiederholt aufsaugen und auswerfen, um Punkte zu sammeln.

BEISPIEL:

Sinnvolle Metrik:
- **Percepts**: location, cleanness
- **Actions**: left, right, suck, wait
- **Metrik**: +100 (clean cell), −10 (suck), −1 (move)

1.1.4. Characteristics

English	Synonym	Erklärung
persistence	Dauerbetrieb	Der Agent läuft kontinuierlich und versucht eigenständig, seine Ziele zu erreichen und Stabilität zu wahren.
autonomy	Selbstbestimmung	Fähigkeit, ohne menschliches Eingreifen Entscheidungen zu treffen und Aufgaben eigenständig zu priorisieren.
social ability	Kooperationsfähigkeit	Fähigkeit zur Kommunikation und Abstimmung mit anderen Komponenten, um gemeinsam Aufgaben zu lösen.
reactivity	Reaktionsvermögen	Der Agent nimmt seine Umwelt wahr und kann auf dortige Veränderungen unmittelbar und passend antworten.
proactivity	Vorausschauendes Handeln	Aktives Prüfen verschiedener Szenarien und Ergreifen der Initiative in einer zielorientierten Art und Weise.
adaptability	Wandlungsfähigkeit	Überdenken der Optionen und Wechsel der Strategie, falls ein ursprünglicher Plan das Ziel nicht mehr erreicht.

1.2. PEAS

Begriff (English)	Definition	Beispiele (Shop-Agent)
Performance Measure	Der Massstab, an dem der Erfolg des Agenten in seiner Umgebung gemessen wird	Anzahl korrekt beantworteter Fragen
Environment	Die spezifische Umgebung oder Welt, in der sich der Agent bewegt und agiert	Bahnhofsladen Rigi Kaltbad
Actuators	Die Werkzeuge oder Komponenten, mit denen der Agent Handlungen in der Welt ausführt	Lautsprecher und Motoren
Sensors	Die Geräte, mit denen der Agent Informationen und Reize aus seiner Umwelt aufnimmt	Kamera und Mikrofone

1.3. Properties of Environments

DEFINITION: Das **Environment** bestimmt den Charakter und den Schwierigkeitsgrad eines KI-Problems.

- Unterschiedliche **Environments** brauchen unterschiedliche Algorithmen
- **Mensch als Agent?**: Ein Objekt wird als Agent modelliert, wenn es ein **performance measure** maximiert, das vom Handeln anderer Agenten abhängt
  - **Sprach-App**: Mensch gilt nicht als Agent
  - **Schach-App**: Mensch gilt als Agent
- **Schlüssel zum Erfolg**: Umgebungen für den Agenten einfacher gestalten

Name	Arten	Definition	Beispiele
Observability	**Fully** vs. **Partially**	Alle relevanten Aspekte der Umgebung für die Sensoren zugänglich?	**Fully**: Go, Zauberwürfel **Partially**: Poker, Strassenverkehr
Predictability	**Deterministic** vs. **Stochastic**	Trifft der erwartete Effekt der Handlung sicher wie geplant ein?	**Deterministic**: Go, Bremsen auf trockener Strasse **Stochastic**: Poker, Fahren auf verschneiter Fahrbahn
Time dependency	**Episodic** vs. **Sequential**	Nur aktueller Zustand relevant oder auch Langzeiteffekte?	**Episodic**: Eier-Sortiermaschine (kein Langzeiteffekt) **Sequential**: Schach, Poker, Go (Einfluss auf das gesamte Spiel)
Dynamics	**Static** vs. **Dynamic**	Ändert sich Umwelt während der Agent überlegt?	**Static**: Sudoku, Poker, Go **Dynamic**: Basketball, autonomes Fahren
Discrete vs. Continuous	**Discrete** vs. **Continuous**	Begrenzte Anzahl Möglichkeiten oder stetige Variablen (unendliche viele Möglichkeiten)?	**Discrete**: Sudoku, Schach, Poker **Continuous**: Autofahren, Greif-Roboter (stetige Variablen)

Name	Arten	Definition	Beispiele
Number of agents	**Single** vs. **Multiagent**	Ein einzelner Agent oder mehrere interagierende Agenten?	**Single**: Sudoku, Greif-Roboter **Multi-agent**: Basketball, Go, Poker, autonomes Fahren
Knowledge	**Known** vs. **Unknown**	Regeln der Umwelt bekannt oder müssen diese erst gelernt werden?	**Known**: Fahrplan, Go, Poker **Unknown**: Navigation in einer unbekannten Stadt

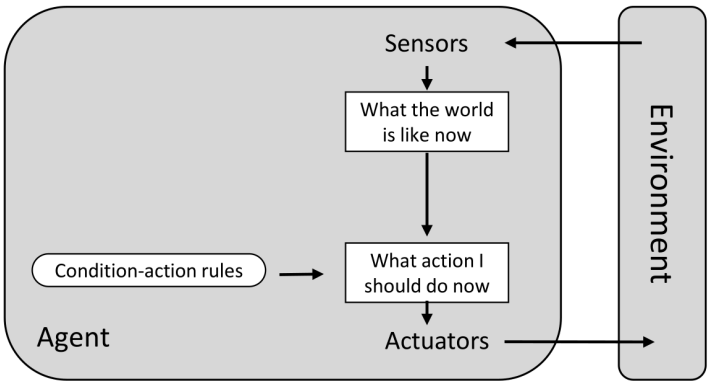
1.3.1. Realwelt-Problem

- **Herausforderung**: Höchste Komplexität entsteht durch Kombination schwieriger Eigenschaften
- **Partially observable**: Der Agent sieht nicht alles
- **Nondeterministic / Stochastic**: Aktionen haben keine garantierten Folgen
- **Dynamic**: Die Welt verändert sich ständig
- **Continuous**: Es gibt unendlich viele Zustände und Zeitpunkte
- **Multi-agent**: Es agieren mehrere Akteure gleichzeitig
- **Fazit**: Diese Kombination beschreibt Probleme der **echten Welt**

1.4. Type of agents

1.4.1. Simple Reflex Agent

- Reagiert direkt auf Wahrnehmungen per Condition-Action-Regel
- Kein Speicher kein Weltmodell
- Braucht vollständig beobachtbare Umgebung sonst drohen Endlosschleifen



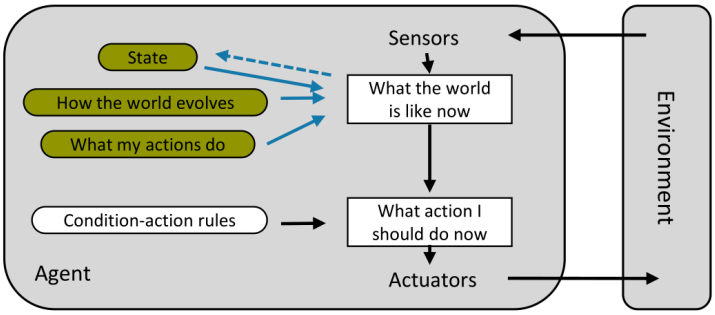
Vor- und Nachteile

- **Vorteil**: Sehr einfach und schnell
- **Nachteil**: Sehr begrenzter Anwendungsbereich da bei unvollständiger Beobachtbarkeit Endlosschleifen drohen

BEISPIEL: Staubsauger-Agent, wenn schmutzig dann saugen

1.4.2. Model-based Reflex Agent

- Speichert internen Zustand und nutzt ein Weltmodell
- Berücksichtigt die Historie der Wahrnehmungen
- Weiss wie Aktionen die Welt verändern



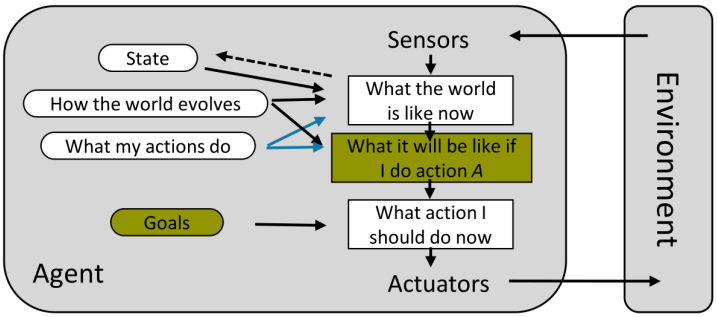
Vor- und Nachteile

- **Vorteil**: Kann auch in teilweise beobachtbaren Umgebungen erfolgreich agieren
- **Nachteil**: Muss die Welt und ihre Entwicklungen intern nachbilden

BEISPIEL: Foto-Roboter, erkennt Person fragt nach Erlaubnis wendet Stil an

1.4.3. Goal-based Agent

- Nutzt explizite Ziele
- Simuliert Auswirkungen von Aktionen vor der Ausführung
- Nutzt Suche und Planung zur Zielerreichung



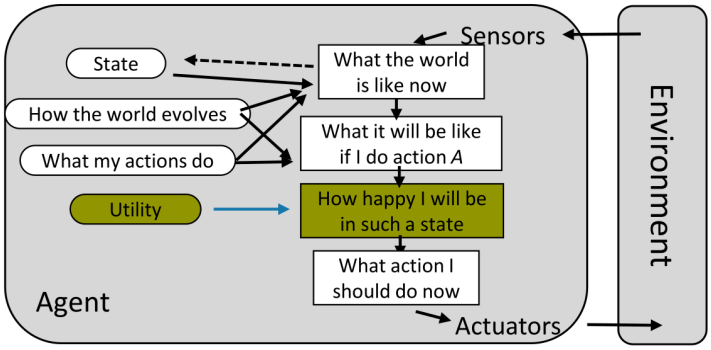
Vor- und Nachteile

- **Vorteil**: Handelt vorausschauend und zielgerichtet
- **Nachteil**: Benötigt aufwändige Planung und Suche um die richtige Aktionsfolge zu finden

BEISPIEL: Systematische Planung von Aktionsabfolgen um ein definiertes Ziel zu erreichen

1.4.4. Utility-based Agent

- Maximiert den erwarteten Nutzen
- Wägt bei mehreren Optionen den besten Weg ab
- Bewertet Zustände oft mit reellen Zahlen



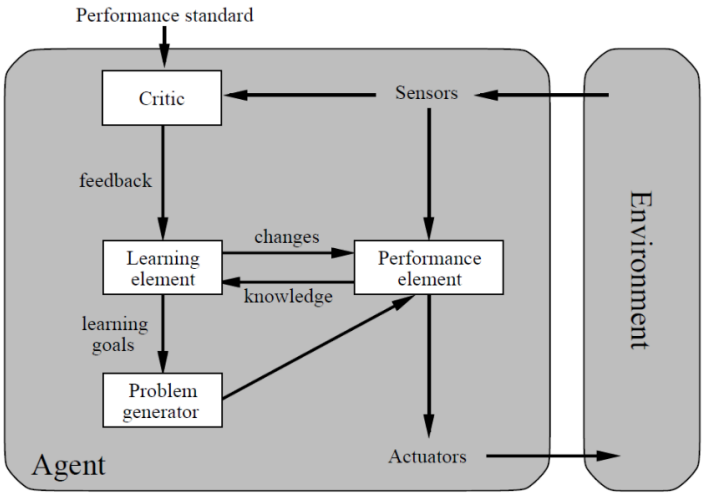
Vor- und Nachteile

- **Vorteil**: Kann zwischen konkurrierenden Zielen abwägen und bewusste Entscheidungen treffen
- **Nachteil**: Komplexe Abbildung von Zuständen auf eine mathematische Nutzenfunktion erforderlich

BEISPIEL: Agent der zwischen konkurrierenden Zielen abwägt und den nützlichsten Weg wählt

1.4.5. Learning Agent

- Lernt aus Erfahrung und verbessert sich stetig
- Kann ohne Vorwissen in unbekannten Umgebungen starten



Vor- und Nachteile

- **Vorteil**: Steigert Kompetenz mit der Zeit und bewältigt anfangs unbekannte Umgebungen
- **Nachteil**: Benötigt kontinuierliches Feedback und Zeit zum Sammeln von Erfahrungen

BEISPIEL: Ein selbstfahrendes Auto das aus menschlichen Eingriffen lernt und die Fahrweise anpasst

1.5. Summary Questions

- What is a rational agent?
  - Wählt für jede Wahrnehmung die Aktion zur Maximierung seines Erfolgsmasses (Performance Measure).
- How to we model an agent?
  - Als Kombination aus Architektur (Sensoren und Aktuatoren) und Programm, das Wahrnehmungen auf Aktionen abbildet.
- What is the performance measure?
  - Ein unabhängiger Bewertungsmassstab, der den Zustand der Umgebung aus externer Sicht bewertet, um die Aufgabenerfüllung zu messen.
- How can we characterize environments?
  - fully vs partially vs not observable, deterministic vs stochastic, episodic vs sequential, static vs dynamic, discrete vs continuous, single-agent vs multi-agent und known vs unknown

1.6. lookup table

DEFINITION: Gedächtnis eines einfachen KI Agenten.

Kennzahlen:

- 1: Anzahl Einträge in der lookup table zu einem bestimmen Zeitpunkt (z.b. at time-step 3)
- 2: Total Einträge für Speicherung aller möglichen Sequenzen nach einer Anzahl time-steps (after)

Gegebene Angaben:

- P: set of possible percepts (hubs, Knoten)

- T: lifetime, Anzahl Bewegungen

Berechnung:

1: P^T

2: $\sum_{t=1}^{T} |P|^T$

1.6.1. Beispiel

- P= 4
- T = 3

```python
number_of_entries_at_t3 = 64  # 4^3, da t3
total_number_of_entries_for_three_steps = 84  #4^1 + 4^2 + 4^3 = 4 + 16 + 64 = 84

def calculate_total_number_of_entries(percepts, lifetime):
    total = 0
    for t in range(1, lifetime + 1):
        total += (percepts ** t)
    return total
```

1.7. performance measure vs. utility function

- performance measure ist eine externe Evaluation des Agents
- utility function ist eine interne Funktion des agents um mögliche Aktionen zu bestimmen

2. Modeling Search Problems

2.1. Problem Solving Agent

- Problemformulierung abstrahiert von der realen Welt
- konzentriert sich auf die Eigenschaften von Zuständen
- möglichen Handlungen.

—

- Initial state: Ort des Agents
- Goal: einen State erreichen (Bucharest) durch eine Sequenz von Aktionen

-> **Search**: Sucht eine geeignete Abfolge von Aktionen und führt diese aus

2.1.1. Solving the Problem with Search

Wenn die Umgebung (Environment)

- **observable**
- **static**
- **deterministic**
- **discrete**

kann der Agent nach einer geeigneten Sequenz von Aktionen suchen

2.1.2. Concepts to Describe a Search Problem

Initial State: Zustand von wo aus die Suche gestartet wird

- S_initial = In(Arad)

Actions: Beschreibung der möglichen Aktionen

- In(Arad): {Go(Sibiu), Go(Timisoara), Go(Zerind)}

Transition Model: Beschreibung des Ergebnis einer Aktion (Nachfolgende Aktionen)

- Result(In(Arad), Go(Zerind)) = In(Zerind)

State Space: Alle möglichen Zustände

- stateSpace = {In(Arad), In(Sibiu), In(Timisoara)…}

Goal Test: Testet den aktuellen Zustand auf End-Zustand

- In(Bucharest)

2.1.3. Concepts to Describe a Search Space

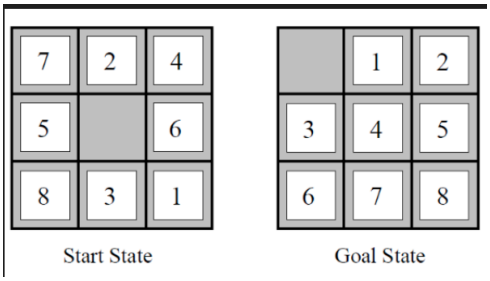
Path: eine Sequenz von Aktionen die von einem Zustanden in den anderen führen

Path Cost: Kostenfunktion g über Pfaden. In der Regel die Summe der Kosten der Aktionen entlang des Pfades

- -> Leistungsmass (performance measure)

Solution: Pfad vom initial state zum goal state

2.1.4. Example: 8 Puzzle



States: Beschreibung der Position jeder Kachel und des leeren Feldes.

Initial State: Ausgangsposition des Puzzles. `<7, 2, 4, 5, 9, 6, 8, 3, 1>`

Actions: Verschieben des leeren Feldes nach links, rechts, oben oder unten.

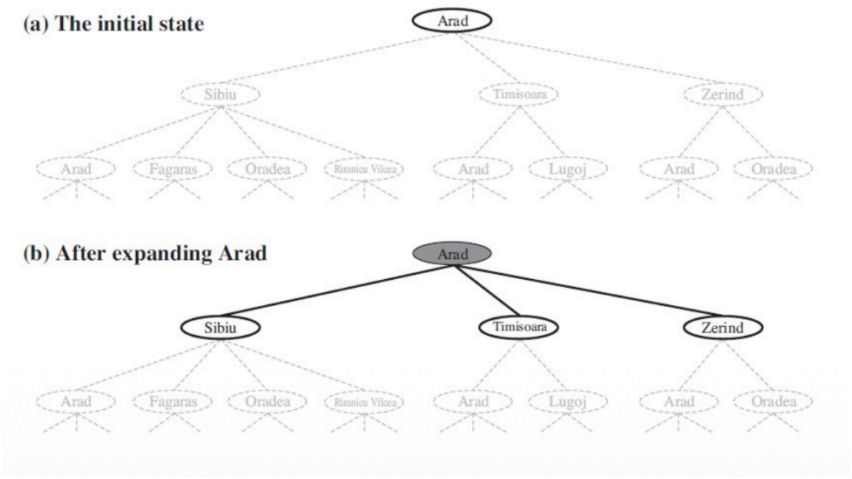
Goal Test: Entspricht die Konfiguration dem Zielzustand? `<9, 1, 2, 3, 4, 5, 6, 7, 8>`

Path Cost: Summe der Schritte, die jeweils 1 Einheit kosten.

2.2. Searching Solutions

2.2.1. Search Tree

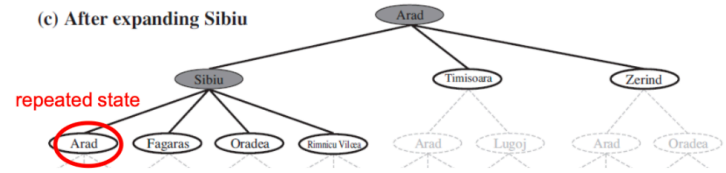
root = initial state **branches** = actions **node** = states



2.2.1.1. General Search Algorithm

- startet mit einen **initial state**
- erweitern den Zustand wiederholend, in dem Nachfolger generiert werden
- stoppt wenn ein Zielstatus expandiert ist
- oder alle erreichbaren Zustände berücksichtigt wurden

2.2.1.2. Repeated States



Loopy paths: erreicht *nie* eine bessere Lösung, da die Kosten addiert werden (es wird etwas doppelt ausgeführt)

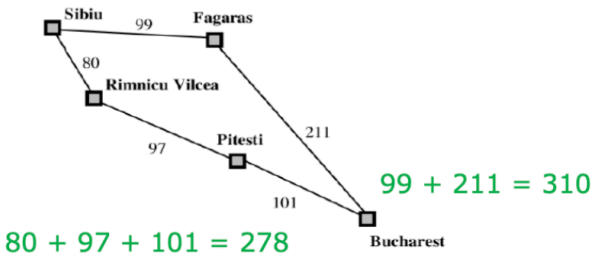
2.2.1.3. Dead End

dead ends = wenn der Node, keine unerkundete Kinder mehr hat

2.2.1.4. The Search Tree

- beschreibt in welcher Reihenfolge die Nodes abgearbeitet werden
  - eine sequenzielle Beschreibung der Reihenfolge
  - manchmal kehrt man zu einem übergeordneten Knoten zurück, um die verbleibenden untergeordneten Knoten zu untersuchen

2.2.1.5. Redundant Paths



Behalte nur **einen** Pfad mit den geringsten Kosten.

2.2.1.6. Implementing Search Tree

Datenstruktur für jeden node `n`

- `n.state` : mit diesem Knoten verbundener Zustand
- `n.parent` : search node, welcher diesen code generiert (keinen für den root)
- `n.action` : Aktion die von `n.parent` zu `n` führt (keinen für den root)
- `n.path_cost` : die Kosten `g(n)` für den Pfad vom initial state zum nood (folgt dem parent Referenz)

  und manchmal ergänzende Attribute (z.B. `n.depth`)

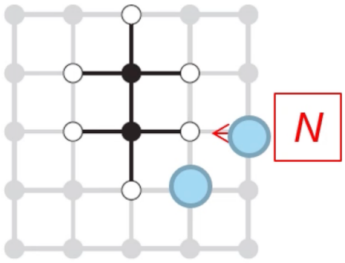
2.2.2. Concepts to Describe a Search Algorithm

- **Node expansion / Knotenausdehnung**: Generierung aller Nachfolgeknoten unter Berücksichtigung der verfügbaren Aktionen
- **Frontier / Grenze**: Menge aller für die Erweiterung verfügbaren Knoten
- **Search strategy / Suchstrategie**: legt fest, welcher Knoten als nächstes erweitert wird
- **Set of visited states**: (die erkundete Knotengruppe) alle Knoten, die zu einem bestimmten Zeitpunkt durch die Suchstrategie erweitert wurden

2.2.3. Separation of States by the Frontier

Die Grenze (weiss) trennt den erforschten Bereich des Zustandsraums (schwarz) von unerforschten Bereich (grau).

zur Grenze hinzugefügt:



2.2.4. Tree based search algorithm

- behält nur die Grenze bei, nicht jedoch die besuchten Zustände
- wenn der Suchraum ein Graph ist, kann er einen Zustand wiederholt betreten, was möglicherweise zu Endlosschleifen führt
- funktioniert gut für baumbasierte Suchräume und benötigt weniger Speicherplatz

```txt
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

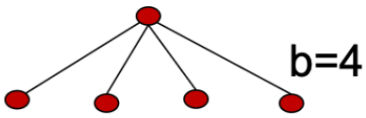
2.2.5. Graph based search algorithm

- speichert die Menge der besuchten Zustände
- die Art von Algorithmus, die wir normalerweise verwenden müssen, wenn wir graphbasierte Suchräume erkunden
  - remember: Die graphbasierte Suche generiert auch einen Suchbaum

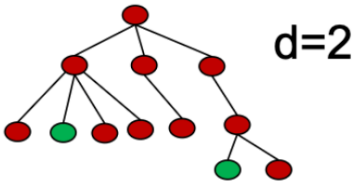
```txt
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored set
```

2.2.6. Quantitative Characterization of a Search Space

b: branching factor / Verzweigungsfaktor



d: Tiefe des flachsten Zielknotens



- grün = Zielknoten

m: maximale Länge eines beliebigen Pfades im Zustandsraum (Bild oben m = 3)

2.2.7. Evaluating Search Algorithms

Completeness / Vollständigkeit

- Ist die Strategie garantiert, eine Lösung zu finden, wenn es eine gibt?

Optimality / Optimalität

- Findet die Strategie die beste Lösung (mit den niedrigsten Pfadkosten)?

Time Complexity / Zeitkomplexität

- Wie viel Zeit braucht man, um eine Lösung zu finden?
  - In der Regel Worst-Case-Analyse
  - In der Regel gemessen in generierten Knoten
- Abhängig vom Verzweigungsfaktor / der Suchtiefe

Space Complexity / Raumkomplexität

- Wie viel Speicher benötigt die Suche?
  - In der Regel Worst-Case-Analyse
  - In der Regel gemessen in (gleichzeitig) gespeicherten Knoten
- Abhängig vom Verzweigungsfaktor / der Suchtiefe

2.3. Summary Questions

1. Which concepts are used to describe search problems?
2. What is the difference between tree-based and graph-based search?
3. What is the set of explored nodes used for?
4. Why don't we need a set of explored nodes when the search space is a tree?

3. Systematic Search

3.1. Uniformed (blind) Search Strategies

- keine Informationen über die Länge oder die Kosten des Pfades

Beispiele:

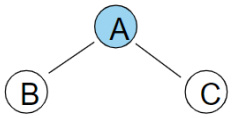
- Breadth-first search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Uniform-cost search

3.2. Breadth-Frist Search (BFS, Breitensuche)

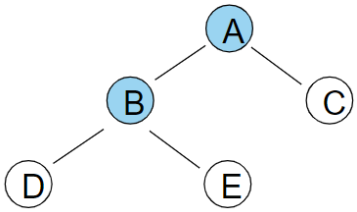
- **FIFO** queue
  - Nodes werden in der Reihenfolge ihrer Erzeugung erweitert
- Durchsucht den Zustandsraum **Schicht für Schicht**
- findet immer zuerst den oberflächlichsten Zielzustand
- ist **vollständig**
- wenn jede Aktion identisch, nicht negative Kosten hat -> die Lösung ist optimal
  - Bsp. Puzzle
- Wenn Aktionen nicht identische Kosten haben -> die Lösung ist suboptimal
  - Bsp. Zugnetz

3.2.1. Vorgehen

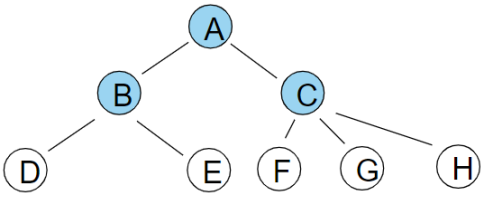
frontier (queue): A



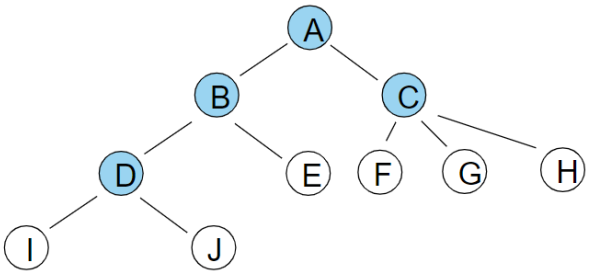
frontier: **B**, C



frontier: **C**, D, E



frontier: **D**, E, F, G, H



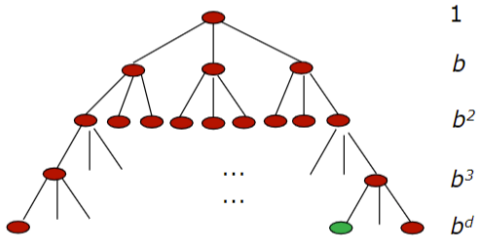
frontier **E**, F, G, H, I, J

3.2.2. Time Complexity of BFS

`b` = max. branching factor

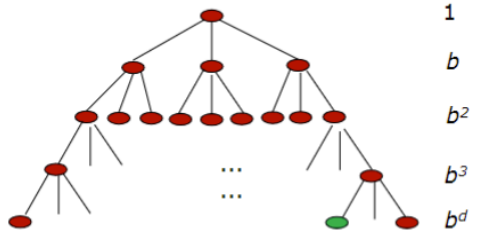
`d` = Tiefe der Lösung

maximal Anzahl Nodes die expandiert werden: $1 + b + b^2 + b^3 + ... + b^d = \sum_{n=0}^{d} b^n$



3.2.3. Space Complexity of BFS

- jeder generierte Node wird im memory gespeichert
- Speicher für die Warteschlange: $O(b^d)$
- Speicher für Menge der bereits besuchten Knoten: $O(b^{d-1})$



3.2.4. Code BFS Algorithmus

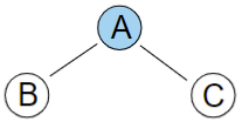
```txt
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node <- a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier <- a FIFO queue with node as the only element
    explored <- an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node <- POP(frontier) /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child <- CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier <- INSERT(child, frontier)
```

3.3. Depth-First Search (DFS, Tiefensuche)

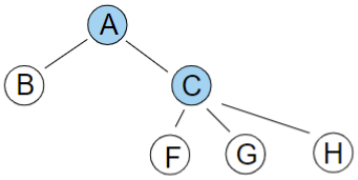
- nimmt immer zuerst den längsten / tiefsten Pfad
- **LIFO** queue (frontier)
- wenn es eine Sackgasse (keine Kinder mehr) ist -> **backtracking** (geht zum nächsten Node, welcher noch unentdeckte Kinder hat)
- Allgemein: gefundene Lösung ist nicht optimal
  - nur für tree-bases search komplett garantiert
  - graph-based search: die besuchten Nodes müssen gemerkt werden -> Loops

3.3.1. Vorgehen

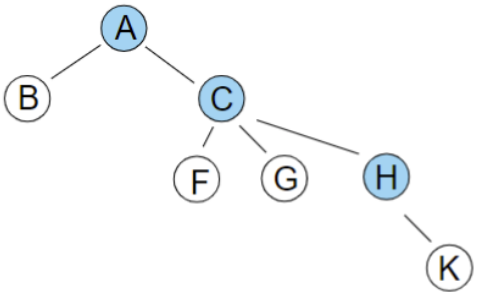
Frontier: A



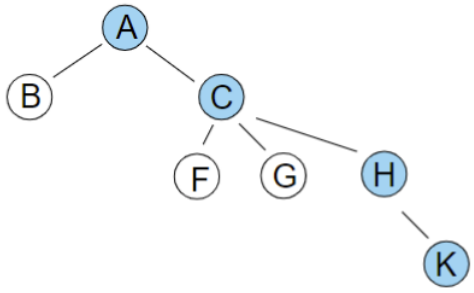
Frontier: B, **C**



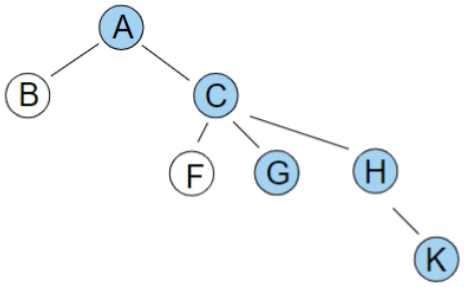
Frontier: B, F, G, **H**



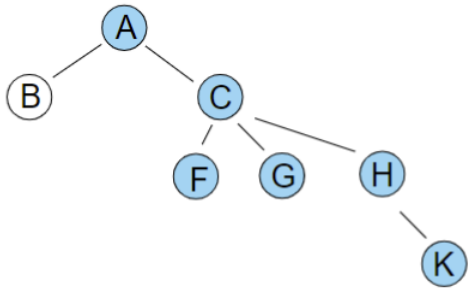
Frontier: B, F, G, **K**



Frontier: B, F, **G**(backtrack)



Frontier: B, **F**



Frontier: **B** (backtrack)

3.3.2. Time Complexity of DFS

$O(b^m)$

- wort case: alle Nodes müssen besucht werden
- dies kann grösser werden, als der Ursprung, wenn die bereits besuchten Nodes nicht gemerkt werden

3.3.3. Space Complexity of DFS

$O(\mathrm{bm})$

- man muss nur die nodes vom root zu den Blättern speichern (frontier) -> low memory complexity
- sobald ein nodes besucht wurde, kann er aus dem memory gelöscht werden
- aber Speicher für die besuchten nodes

3.4. Depth-Limited Search (DLS, tiefenbeschränkte Suche)

- DFS aber mit depth limit
- incomplete wenn das depth limit kürzer ist als die Länge der kürzesten Lösung
- die erste gefunden Lösung ist evtl. nicht die optimalste

3.4.1. Time complexity of DLS

wie DFS, aber $m = l$ $O(b^l)$

3.4.2. Space complexity of DLS

wie DFS, aber $m = l$

$O(\text{bl})$

3.4.3. Code

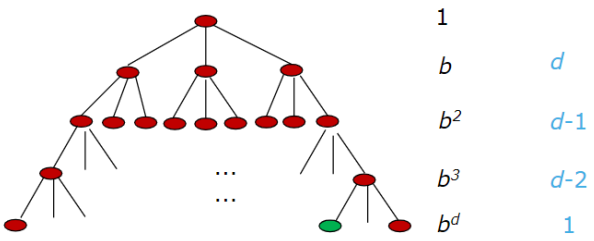
```txt
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    else if limit = 0 then return cutoff
    else
        cutoff_occurred? <- false
        for each action in problem.ACTIONS(node.STATE) do
            child <- CHILD-NODE(problem, node, action)
            result <- RECURSIVE-DLS(child, problem, limit - 1)
            if result = cutoff then cutoff_occurred? <- true
            else if result != failure then return result
        if cutoff_occurred? then return cutoff else return failure
```

3.5. Iterative Deepening Depth-First Search (iterative Tiefensuche)

- kombiniert die Vorteile von breadth-first (BFS) und depth-first (DFS) search
  - BFS: komplett
  - BFS: optimal, wenn alle Aktionen die gleichen Kosten haben
  - DFS: es müssen nur die nodes entlang eines Pfades gespeichert werden
  - time complexity nur ein bisschen höher als DFS
- ist anzuwenden wenn
  - tree search ausreichend ist
  - alle Kosten identisch sind
  - die Tiefe der Lösung unbekannt ist

3.5.1. Time complexity of Iterative Deepening DFS

$\mathrm{db} + (d-1)b^2 + (d-2)b^3 + \ldots + 1b^d \in O(b^d)$



3.5.2. Space complexity of Iterative Deepening DFS

$O(\mathrm{bd})$

3.5.3. DFS vs. BFS

b = branching factor d = minimal solution length

b=10, d=5

Time complexity

BFS: `1 + 10 + 100 + 1000 + 10’000 + 100’000 = 111’111`

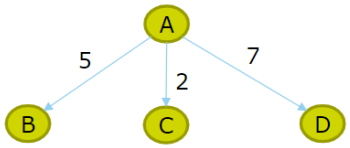
Iter. Deep. DFS: `6 + 50 + 400 + 3000 + 20’000 + 100’000 = 123’456`

Nur 11% mehr für b=10

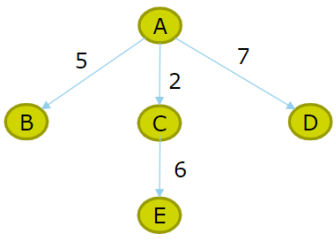
3.6. Uniform-Cost Search (UCS)

- Pfadkosten für jeden Knoten werden berücksichtigt $g(n)$
- **priority queue**
  - nodes mit den geringsten Kosten werden als erstes abgearbeitet
- Bedingungen, damit immer der günstigste Pfad gewählt wird:
  - es wird erst geprüft, ob es der Zielknoten ist, wenn der Node aus der Warteschlange entnommen wird
    - wenn dies vorher geprüft wird, könnte es sein, dass ein längerer Weg gewählt wurde
  - Knoten werden in der Warteschlange ersetzt, wenn ein günstigeren Weg gefunden wurde
- optimal (für nicht negative Pfadkosten)
  - immer wenn ein node aus der Warteschlange genommen wird, ist das der kürzeste Wege
  - berücksichtigt nur die Pfadkosten
- komplett (wenn Kosten > 0, sonst bleibt der Algorithmus stecken)

3.6.1. Vorgehen



frontier: B(5), C(2), D(7)



frontier: B(5), D(7), E(8)

3.6.2. Time and space complexity

$O\left(b^{1+\left\lfloor \frac{C^*}{\epsilon} \right\rfloor}\right)$

- C^*: Kosten der optimalen Lösung, Pfadkosten > ϵ
- b^{d+1}, wenn alle Pfadkosten gleich sind

3.6.3. Code

```txt
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node <- a node with STATE = problem.INITIAL-STATE, PATH-COST = 0

    # Frontier ist eine Priority Queue,
    # sortiert nach den niedrigsten Pfadkosten (PATH-COST).
    frontier <- a priority queue ordered by PATH-COST, with node as the only element

    explored <- an empty set

    loop do
        # Wenn die Warteschlange leer ist, gibt es keine Lösung
        if EMPTY?(frontier) then return failure

        # Wählt den Knoten mit den geringsten Pfadkosten aus der Frontier
        node <- POP(frontier)

        # (2) Der Goal-Test erfolgt erst beim Entnehmen (Expansion) des Knotens,
        # nicht bereits beim Erzeugen. Das garantiert die Optimalität.
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)

        # Den aktuellen Zustand als besucht markieren
        add node.STATE to explored

        for each action in problem.ACTIONS(node.STATE) do
            child <- CHILD-NODE(problem, node, action)

            # Falls der Zustand neu ist: ab in die Frontier
            if child.STATE is not in explored or frontier then
                frontier <- INSERT(child, frontier)

            # (3) Falls der Zustand bereits in der Frontier ist, aber der neue Weg
            # dorthin günstiger ist, wird der alte Knoten durch den besseren ersetzt.
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

3.7. Summary

- Iterative deepening -> bevorzug bei einem grossen Suchraum, wenn die Tiefe der Lösung nicht bekannt ist
- DFS -> bei kleinem Speicherbedarf
- BFS -> selten in der Praxis verwendet

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]
Time	$O(b^d)$	$O\left(b^{1+\left\lfloor \frac{C^*}{\varepsilon} \right\rfloor}\right)$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^d)$	$O\left(b^{1+\left\lfloor \frac{C^*}{\varepsilon} \right\rfloor}\right)$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes[c]	Yes	No	No	Yes[c]

- b: branching factor
- d: Lösungstiefe
- m: Maximale Tiefe des Suchbaums
- l: depth limit

Hochgestellt Zeichen:

- a: b ist endlich
- b: wenn die Schrittkosten nicht weniger sind als ε
- c: wenn die Schrittkosten gleich sind
- d: wenn beide Richtungen Breitensuche verwenden

4. Heuristic Search

4.1. Node Expansion

4.1.1. Uninformed Search

- Starrer Prozess
- Kein Wissen über die Kosten vom Node bis zum Ziel
  - Wir wissen nicht ob wir uns näheren oder entfernen
- Bsp. FIFO, LIFO queues

4.1.2. Informed Search

- Wissen nutzen
- Heuristic Algorithmen $h(n)$ schätzen die Kosten bis zum Ziel (Entfernung)

4.2. Evaluation Function

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

- $g(n)$ = Kosten vom Initial-State bis zum aktuellen State
  1. präziser Wert
  2. blau
- $h(n)$ = Kosten vom aktuellen State zum Ziel-Status
  1. geschätzter Wert
  2. rot

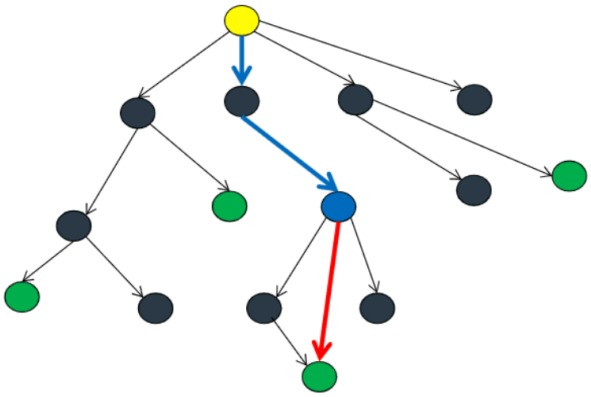


Abbildung 1:

4.3. Heuristic Function h(n)

- Verwendet Wissen um die Suche zu lenken
- Geschätzte Kosten von node n bis zum Ziel
- Je kleiner $h(n)$ desto näher ist n beim Zielknoten

Beim Beispiel Bucharest ist das die Luftdistanz

- geschätzter Wert und nicht die tatsächlichen Wegkosten

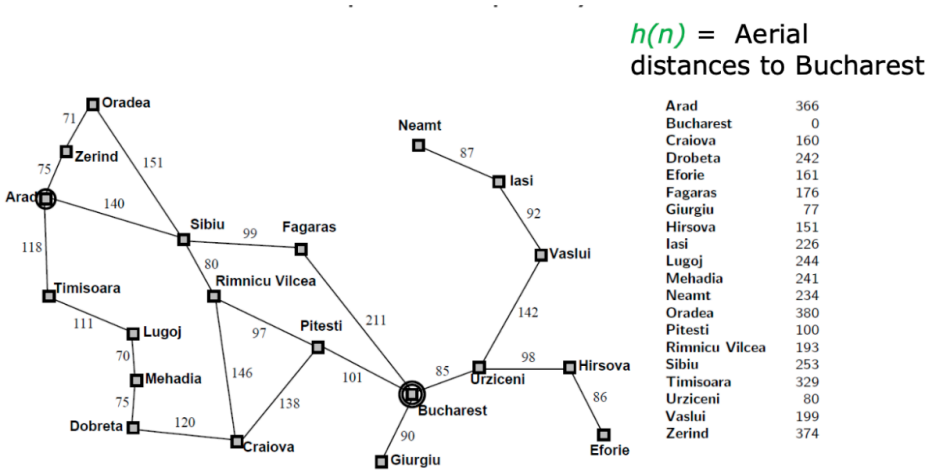


Abbildung 2:

4.3.1. Properties

1. Grundlegende Eigenschaften von $h(n)$:

$$h(n) = \begin{cases} 0 \\ \text{if } \ n \ \text{ is a goal state} \\ > 0 \\ \text{otherwise} \end{cases}$$

   - Immer positiv
   - 0 beim Zielknoten
   - Wird kleiner beim Nähern
2. h ist zulässig (admissible)
3. h ist monoton (consistent)

Eine guter heuristischer Algorithmus muss mind. 1 und 2 erfüllen

4.3.2. Admissible - Zulässig

- $h^*(n)$ = Kosten vom optimalen Weg vom Knoten n zum Zielknoten
- h ist zulässig (oder optimistisch), wenn für alle n garantiert ist:

$$h(n) \leq h^*(n)$$

- h ist eine optimistische Schätzung der tatsächlich anfallenden Kosten
  - Es unterschätzt die echten Kosten

4.3.2.1. Beispiel

- $h^*(n)$ = Tatsächliche Kosten vom optimalen Weg von n nach Bucharest
- $h(n)$ = Luftdistanz von n nach Bucharest

4.3.3. Consistent - Konsistent

Dreicksungleichheit: jede Seite des Dreiecks kann nicht länger als die Summe der anderen zwei Seiten sein

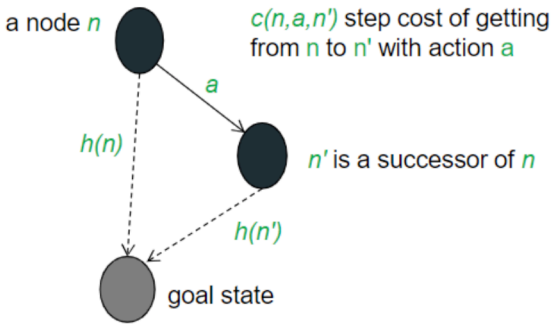


Abbildung 3:

$$h(n) \leq c(n, a, n') + h(n')$$

4.4. Greedy

$$f(n) = h(n)$$

- Berücksichtig nur die heuristische Funktion
  - Kürzester Weg zum Ziel
- Berücksichtig **keine** Pfadkosten
- Gegenstück zu Uniform-Cost Search

4.4.1. Beispiel

Beispiel Bucharest:

- Verwendet **nur** Luftdistanz
- **keine** Pfadkosten

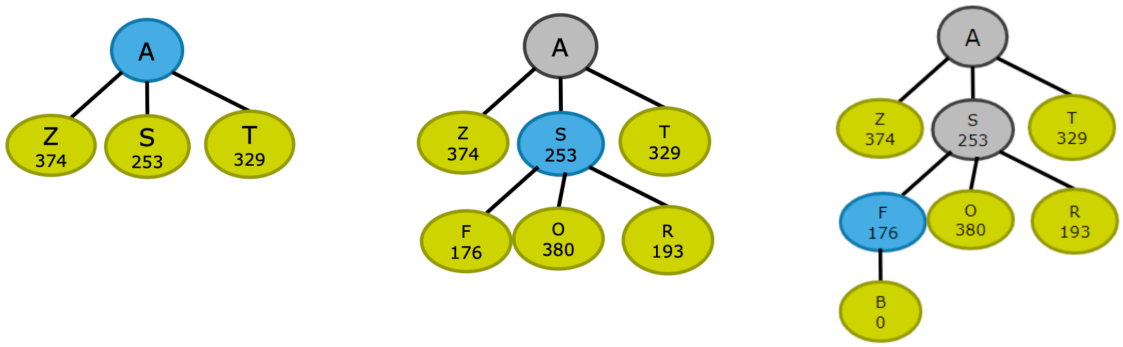


Abbildung 4:

4.4.2. Properties

- Minimale Suchkosten
  - Expandiert nur einen einzigen Weg
- Nicht optimal
- Inkomplett
  - Kann in dead-ends oder loops steckenbleiben
- Zeit und Speicherkomplexität: $O(b^m)$
  - $m =$ max Suchtiefe

4.4.3. Incompleteness

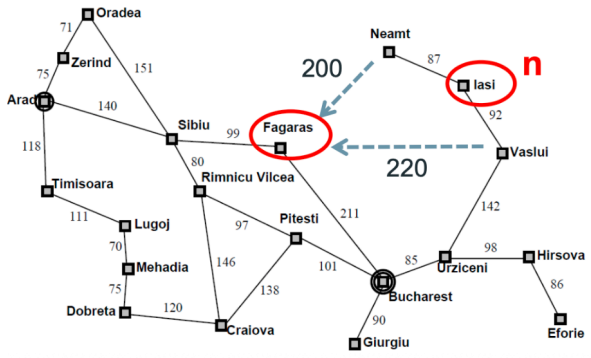


Abbildung 5:

- Kürzester heuristischer Weg ist „Neamt“
- Jedoch führt dies nicht zu einer globalen korrekten Lösung
  - Dead-End

4.5. `A*` Algorithmus

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

- Berücksichtig **Heuristik** $h(n)$ **sowie Pfadkosten** $g(n)$

- Kombiniert Greedy-Search mit Uniform-Cost Search
- Nimmt kürzesten/günstigsten Weg von allen bekannten Knoten
  - Macht also auch Backtracking

4.5.1. Beispiel

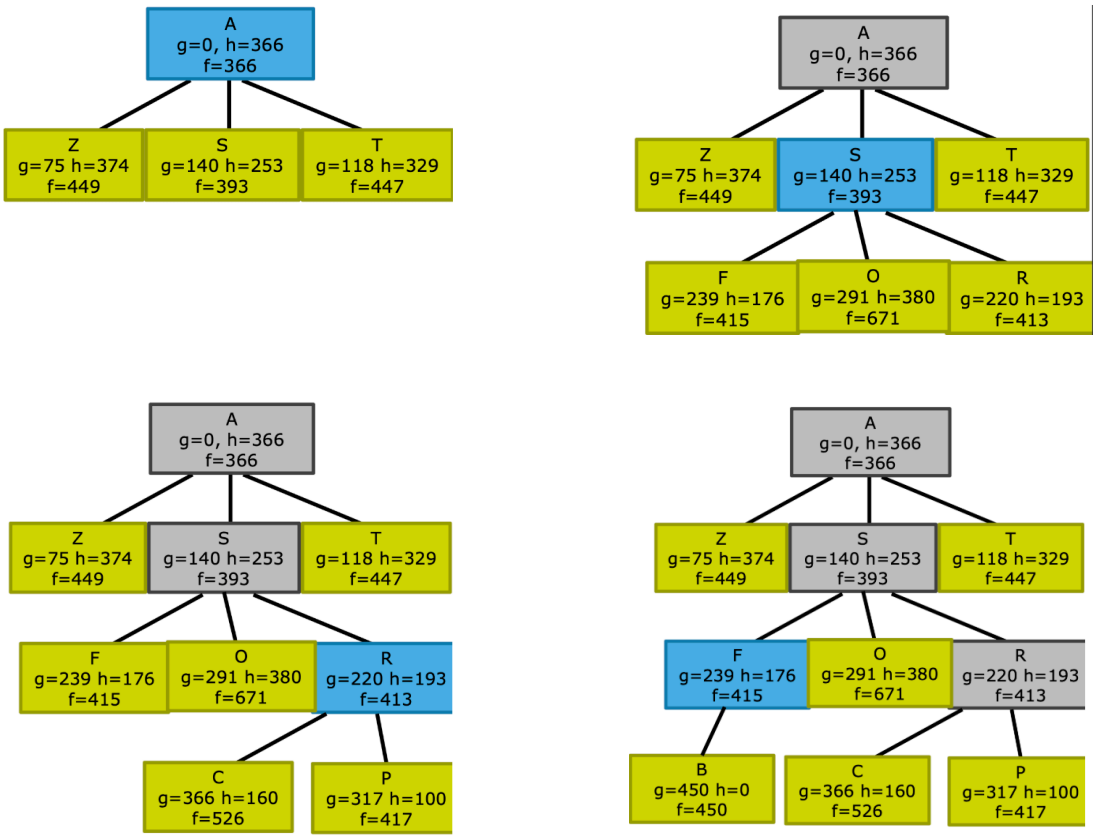


Abbildung 6:

4.5.2. Properties

Komplett

- Wenn eine Lösung existiert, ist gewährleistet, dass:
  1. Jeder Node hat eine endliche Nummer an Nachfolger und
  2. Jede Aktion hat positive und endliche Kosten

Optimal

- Erste gefundene Lösung → minimal Kosten
  - Wenn h admissible (für Bäume) oder consistent (für Graphen)
- Wenn es admissible heuristisch für Graphen ist, müssen alle nodes $f(n) \leq C^*$ (Kosten der optimalen Lösung)

Zeit & Speicherkomplexität

- Exponentiell
- Eignet sich für kleine Graphen und Bäume

4.5.3. f(n)-basierten Konturen (Höhenlinien)

- **Suchmuster:** `A*` breitet sich vom Startknoten in konzentrischen Bändern (Konturen) mit steigenden $\boldsymbol{f(n)}$-**Kosten** aus.
- **Effizienz:** Gute Heuristiken strecken diese Bänder gezielt in Richtung des Ziels und fokussieren sich eng auf den optimalen Pfad.
- **Pruning (Beschneidung):** Knoten mit $f(n) > C^*$ (Kosten des optimalen Pfads) werden nicht expandiert, da sie keine bessere Lösung liefern können.

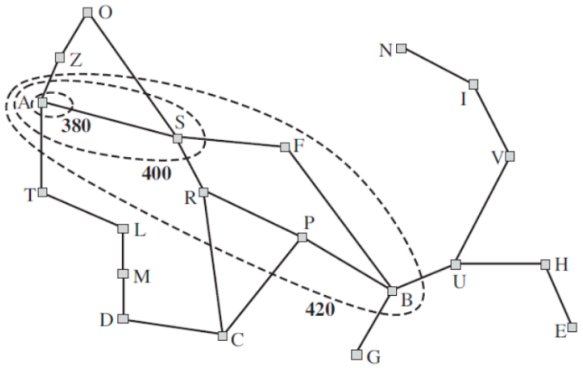


Abbildung 7:

4.5.4. Branch and Bound unter zulässigen Heuristiken

- **Pruning-Prinzip:** Sobald eine erste Lösung mit Kosten C gefunden wurde, können alle Knoten mit $f(n) > C$ ignoriert werden.
- **Konsistente Heuristik:** Die erste gefundene Lösung ist garantiert die **optimale Lösung**
  - der Algorithmus kann sofort stoppen.
- **Admissible Heuristik:** `A*` muss alle Knoten mit Kosten $f(n) \leq C$ expandieren, um sicherzustellen, dass die gefundene Lösung wirklich optimal ist.

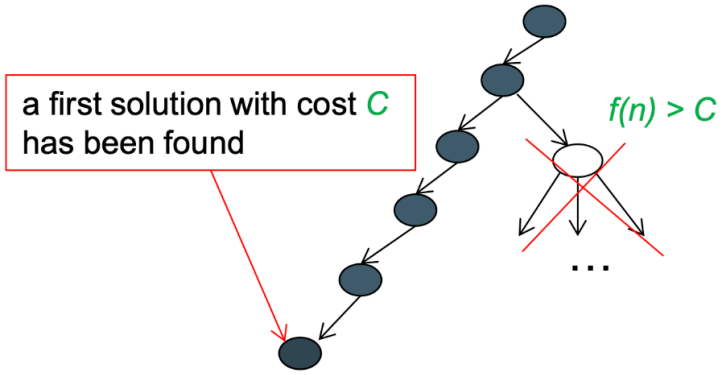


Abbildung 8:

4.6. Iterative Deepening `A*` - `IDA*`

- **Problem:** `A*` benötigt im schlimmsten Fall (Worst-Case) exponentiell viel Speicherplatz.
- **Grundidee:** Nutzung einer iterativen Tiefensuche (ähnlich wie Iterative-Deepening DFS).
- **Unterschied:** Anstatt die Suchtiefe (Pfadlänge) zu begrenzen, wird die Suche durch einen Schwellenwert für die **f-Kosten** begrenzt.

4.6.1. Ablauf

- **Suchstrategie:** In jeder Iteration wird eine normale **Tiefensuche (Depth-First Search)** durchgeführt.
- **Abbruch (Cut-off):** Ein Zweig wird sofort abgeschnitten, wenn seine Gesamtkosten den aktuellen Schwellenwert (threshold) überschreiten: $f(n) = g(n) + h(n) > \text{threshold}$.
- **Dynamischer Schwellenwert:**
  1. **Start:** Beginnt mit den geschätzten Kosten des Startknotens (also $h(\text{Start})$).
  2. **Update:** Für die nächste Iteration wird der Schwellenwert auf das **Minimum aller f-Werte** gesetzt, die in der aktuellen Iteration den Schwellenwert überschritten haben und abgeschnitten wurden.

WICHTIG: Die Suche ist erst fertig, wenn der Zielknoten expandiert wird (nicht schon beim Generieren).

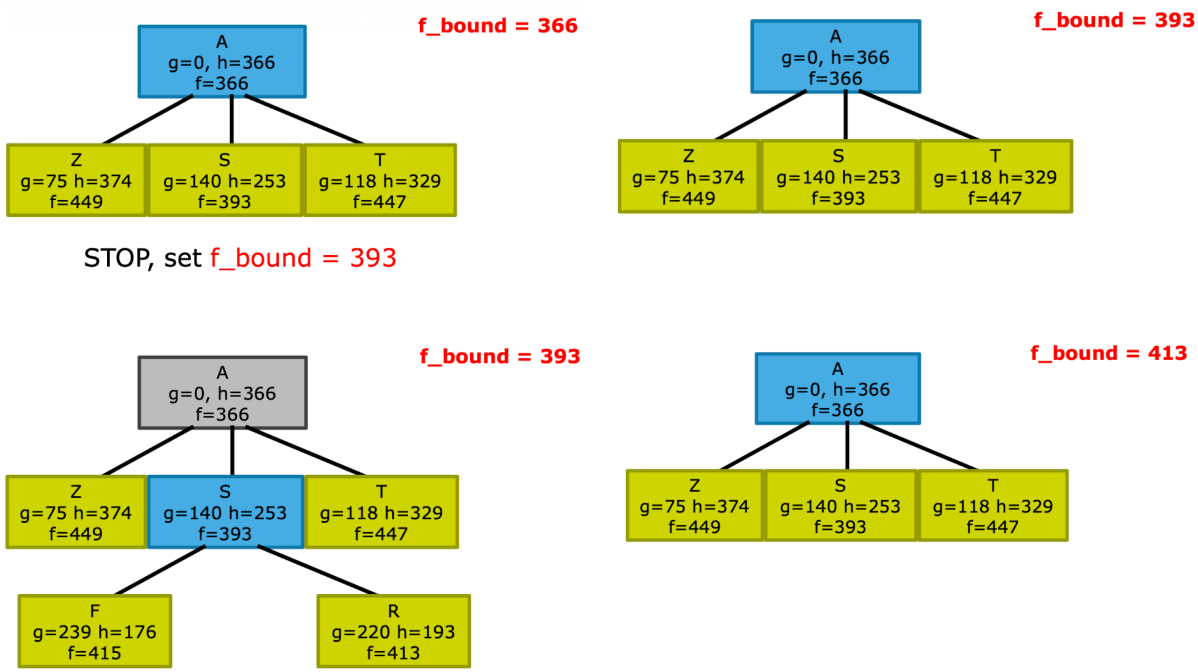


Abbildung 9: Bild 3 (unten links): Startet wieder „von vorne“, da es evtl. einen Weg geben könnte mit den Kosten 413, welche *nicht* durch S geht

4.6.2. Properties

- Optimal wenn h admissible / zulässig ist
- Speicher-Komplexität: $O(\text{lb})$
  - b = Branching Factor
  - l = Länge des längsten generierten Pfades
  - Kein exponentielles Wachstum

Problem der Duplikate:

- Erheblicher Overhead (Zusatzaufwand) im Vergleich zu `A*`
- Standardversion erkennt mehrfach besuchte Knoten nicht
- *Gegenmassnahme:* Partielle Duplikat-Eliminierung in der Praxis anwenden

Iterativer Overhead:

- Zusatzaufwand durch wiederholte Suchen (steigendes f-Limit)
- Meistens gering, aber nicht immer vernachlässigbar

4.7. Heuristics

- Abschätzung, ob ein Knoten „gut“ oder „schlecht“ ist (basierend auf der Distanz zum Ziel)
- Heuristiken als „Faustregeln“, intuitive Urteile oder gesunder Menschenverstand
- Entscheidend für die Performance der informierten Suche
- Informelle Methoden zur Problemlösung

4.7.1. Design

- **Informiertheit** als Erfolgskriterium (Steuerung zu vielversprechenden Bereichen, frühes Erkennen von Sackgassen)
- **Ziel**: Nahezu optimale Lösungen unter praktischen Bedingungen
- Trennung von Heuristik und Suchalgorithmus
- Notwendigkeit von Domänenverständnis und empirischen Tests verschiedener Varianten

4.7.2. Effektiver Verzweigungsfaktor `b*`

- Massstab für die Qualität einer Heuristik
- Mathematische Definition: $N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$
  - N = Gesamtanzahl der von `A*` generierten Knoten

  - d = Lösungstiefe
- Experimentelle Messungen von b^* als Orientierungshilfe

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	-	539	113	-	1.44	1.23
16	-	1301	211	-	1.45	1.25
18	-	3056	363	-	1.46	1.26
20	-	7276	676	-	1.47	1.47
22	-	18094	1219	-	1.48	1.28
24	-	39135	1641	-	1.48	1.26

4.7.3. Spezifische Heuristiken (z.B. 8-Puzzle)

Linear Conflict Heuristic

- Basiert auf der Manhattan-Distanz
- **Zusatzstrafe (+2):** Wird addiert, wenn zwei Plättchen in der gleichen Zeile/Spalte liegen, ihr Ziel ebenfalls dort ist, sie aber in der falschen Reihenfolge stehen
- **Grund**: Die Plättchen blockieren sich und müssen umeinander herum geschoben werden

Gaschnig's Heuristic

- **Relaxed Problem:** Jedes Plättchen kann direkt mit dem leeren Feld (Blank / 9) getauscht werden
- **Heuristischer Wert:** Die genaue Anzahl der benötigten Tauschvorgänge (Swaps) bis zum Ziel
- **Ablauf:**
  1. Ist das Blank am falschen Platz, tausche es mit dem Plättchen, das eigentlich dorthin gehört
  2. Ist das Blank am richtigen Zielplatz (aber das Puzzle nicht gelöst), tausche es mit einem beliebigen, falsch platzierten Plättchen

4.7.4. Lernen / Entwickeln von Heuristik-Funktionen

Relaxierte Probleme (Problem Relaxation)

- Ein Problem mit weniger Einschränkungen (reduzierte Preconditions bei Aktionen) wird als relaxiertes Problem bezeichnet
- Die Kosten der Optimallösung dieses relaxierten Problems ergeben stets eine zulässige (admissible) Heuristik für das Originalproblem

Beispiele (Bedingungen beim N-Puzzle entfernen)

- `CLEAR` & `ADJ` entfernt → Misplaced Tile Heuristik
- Nur `CLEAR` entfernt → Manhattan Distance Heuristik
- Nur `ADJ` entfernt → Gaschnig's Heuristik

Moderne Suchalgorithmen

- Analysieren die Domäne und die spezifische Problemstellung
- Entwickeln eine massgeschneiderte Heuristik für das genaue Problem, noch bevor die eigentliche Suche losgeht

5. Local Search

DEFINITION: Local Search ist ein Verfahren zur Lösungssuche, bei dem man von einem aktuellen Zustand ausgeht, Nachbarn betrachtet und zu besser bewerteten Zuständen wechselt.

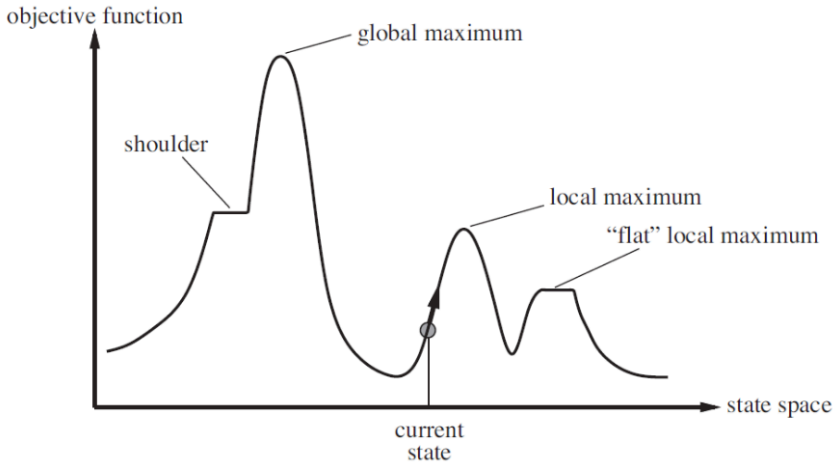
5.1. Grundidee der lokalen Suche

1. Start mit einem Zustand im Suchraum
2. Betrachte und bewerte Nachbarzustände
3. Wechsle zu besseren Zuständen

- Fokus auf der Lösung statt auf dem Pfad
- Benötigt wenig Speicher
- Liefert oft gute Lösungen in grossen Zustandsräumen
- **Beispiele**:
  - 8-Queens: Korrekte Platzierung der Damen
  - Traveling Salesperson: Verbindung aller Städte

5.2. Zustandsraum-Landschaft

- Visualisiert die Bewertungsfunktion im Zustandsraum
- Höhere Punkte repräsentieren eher gute Lösungen
- **Eigenschaften lokaler Suchalgorithmen:**
  - **Complete**: Findet eine Lösung, sofern eine existiert
  - **Optimal**: Findet das globale Maximum oder Minimum
- **Topologische Merkmale (1D-Landschaft):**
  - **Global maximum**: Der absolut höchste Punkt im Suchraum
  - **Local maximum**: Ein lokaler Hochpunkt (suboptimale Lösung)
  - **Flat local maximum**: Ein flaches Plateau auf einem Hochpunkt
  - **Shoulder**: Eine flache Ebene auf einem ansteigenden Ast
  - **Current state**: Der aktuelle Punkt, der sich aufwärts bewegt



5.3. Hill Climbing

5.3.1. Funktionsweise

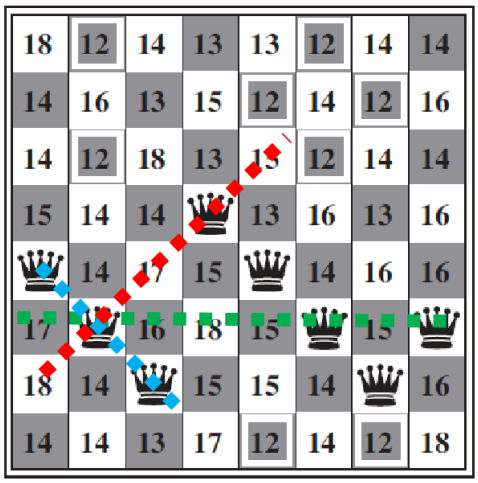
- auch Steepest Ascent Search / Greedy Local Search

Prinzip:

- Geht immer in die Richtung mit der höchsten Steigung (bester Nachbar)
- Stoppt, wenn kein Nachbar besser ist als der aktuelle Zustand
- Bei mehreren gleich guten Nachbarn entscheidet der Zufall

5.3.2. Beispiel: 8-Queens

- **Suchraum:** Reduziert auf $16 * 10^6$ Zustände (exakt eine Dame pro Spalte)
- **Zustand & Nachfolger:** Verschieben einer Dame in derselben Spalte ergibt 56 mögliche Nachbarn (Branching Factor)
- **Bewertung (h):** Zählt die Konflikte (sich angreifende Damenpaare)
- **Lokale Minima:** Algorithmus bleibt oft stecken (z.B. bei $h = 1$), Erfolgsquote liegt bei ca. 14%



5.3.3. Vor- und Nachteile

- **Vorteile:** Findet sehr schnell gute Lösungen
- **Nachteile:** Weder vollständig noch optimal
- **Probleme:** Bleibt oft in lokalen Maxima, Ridges oder Plateaus stecken

5.4. Probleme lokaler Suche

5.4.1. Lokale Minima und lokale Maxima

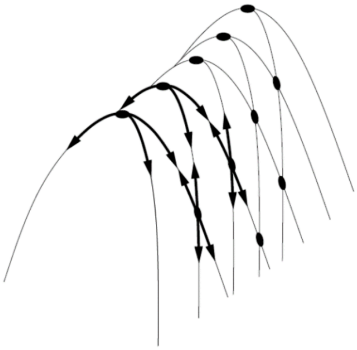
- Alle Nachbarzustände weisen schlechtere Bewertungen auf
- Der Algorithmus bleibt in suboptimalen Zuständen gefangen und terminiert zu früh

5.4.2. Plateaus

- Alle benachbarten Knoten haben gleich gute Bewertungen
- Der Algorithmus irrt ziellos und lange ohne Lösung umher, da es keine klare Steigung gibt

5.4.3. Ridges (Kämme)

- Eine Abfolge von lokalen Maxima, die nicht direkt miteinander verbunden sind
- Der Algorithmus muss sich zwingend erst verschlechtern (runtergehen), um zum nächsten Maximum aufsteigen zu können



5.5. Strategien zum Entkommen

5.5.1. Tabu Search

- **Prinzip:** Führt eine Tabu-Liste der bereits besuchten Zustände
- **Nutzen:** Verhindert Rückkehr in bereits erkundete Regionen
- **Nachteil:** Speicherbedarf widerspricht der Grundidee der lokalen Suche

5.5.2. Random Restarts

- **Prinzip:** Kompletter Neustart mit zufälligem Startzustand bei Stagnation
- **Eigenschaft:** Theoretisch vollständig, in der Praxis jedoch nicht alle Starts testbar
- **8-Queens Beispiel**:
  - 14% Erfolgsquote
  - $\sim$ 4 Schritte bei Erfolg, 3 Schritte bei Fehlschlag
  - **Aufwand**: $\sim$ 7 Neustarts und 22 Schritte bis zur Lösung

5.5.3. Random Walk / Noise

- **Prinzip:** Ungerichtete Schritte (Noise) zulassen
- **Nutzen:** Hilft auf Plateaus durch definierte Anzahl an Schritten auf gleicher Ebene
- **8-Queens Beispiel**:
  - Max. 100 gleichwertige Züge steigern die Erfolgsquote auf 94%
  - **Aufwand**: Im Schnitt 21 Schritte für eine Lösung, 64 Schritte bei einem Fehlschlag

5.5.4. Erfolgsfaktoren

- **Abhängigkeit:** Wirksamkeit hängt stark von Problemklasse und Suchraum-Struktur ab
- **Einfache Räume:** Random Restarts finden schnell gute Lösungen
- **NP-harte Probleme:** Besitzen exponentiell viele lokale Maxima, Escape extrem schwer

5.6. Genetische Algorithmen

5.6.1. Grundidee

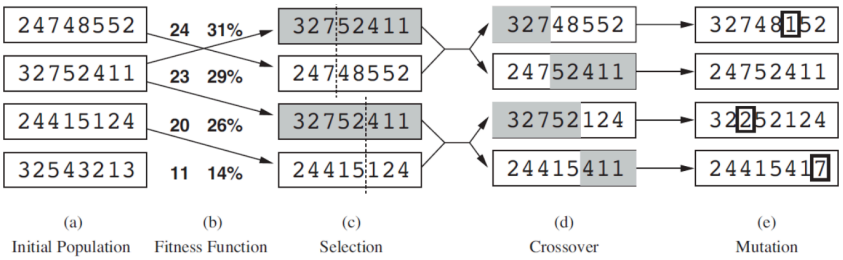
- **Ansatz:** Statt einen einzelnen Zustand zu betrachten, werden zwei Eltern-Zustände zu einem neuen Nachfolger kombiniert
- **Inspiration:** Nachahmung der natürlichen Evolution (sexuelle Reproduktion)
- **Zutaten:**
  - Kodierung des Zustands als String (Gen)
  - Fitness-Funktion zur Bewertung der Zustände
  - Eine Population von Zuständen

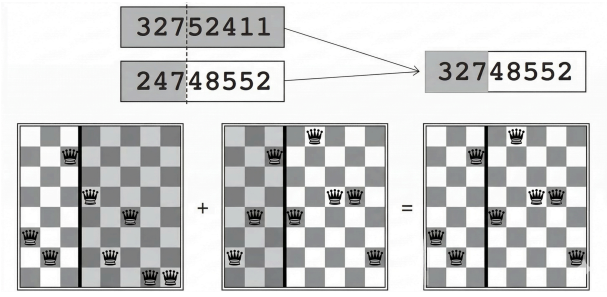
5.6.2. Selektion, Mutation und Crossover

- **Ablauf in Schritten:**
  1. **Bewertung:** Fitness der gesamten Population bestimmen
  2. **Selektion:** Auswahl der Eltern (höhere Fitness = höhere Wahrscheinlichkeit)
  3. **Crossover:** Kreuzung/Kombination der Eltern-Gene zu neuen Nachkommen
  4. **Mutation:** Zufällige Veränderung von Genen mit einer gewissen Wahrscheinlichkeit
- **Vorteile:** Generiert eine gute Vielfalt an passablen Lösungen (Mixability)
- **Nachteile:** Sehr rechenintensiv, Parameter oft schwer optimal einzustellen

5.6.3. Beispiel: 8-Queens als Gene

- **Kodierung:** Eine Zahlenkette gibt die Position der Damen in den jeweiligen Spalten an
- **Fitness:** Anzahl der sich **nicht** angreifenden Damenpaare (Lösung = Wert 28)
- **Selektion:** Bessere Konfigurationen werden eher für die Kombination ausgewählt
- **Crossover:** Zahlenketten zweier Zustände werden an einem Punkt geteilt und überkreuzt zusammengesetzt
- **Mutation:** Einzelne Zahlen in der Kette verändern sich zufällig und gehen zurück in die Population





5.6.4. Grenzen genetischer Algorithmen

- GAs sind nicht gut für reine Optimierungsprobleme
- Finden nicht zwingend die absolut beste Lösung
- Evolution sorgt für das Überleben der Fittesten unter wechselnden Bedingungen, zielt aber nicht auf die blinde Maximierung der reinen Fitness ab

5.7. Simulated Annealing

5.7.1. Grundidee

- Systematisches Hinzufügen von Rauschen („Noise“) zur lokalen Suche
- Zu Beginn sind grosse, zufällige Sprünge erlaubt

- Das Rauschen wird über die Zeit schrittweise reduziert
- Ermöglicht anfangs grosse Sprünge im Suchraum und später ein gezieltes Optimieren
- **Vorteile:** Kann gezielt lokale Maxima verlassen, findet bei optimalem Schedule sehr oft das globale Maximum
- **Nachteile:** Deutlich langsamer als einfaches Hill Climbing, das Finden eines guten Abkühlungsplans ist in der Praxis oft aufwendig

5.7.2. Temperatur und Abkühlungsplan

- Schedule steuert den Temperaturabfall über die Zeit
- **Hohe Temperatur**: Zufällige Bewegungen dominieren die Suche
- **Sinkende Temperatur**: Züge zu schlechteren Nachbarn werden zunehmend unwahrscheinlicher
- Erreicht die Temperatur den Wert 0, stoppt der Algorithmus und liefert den aktuellen Zustand

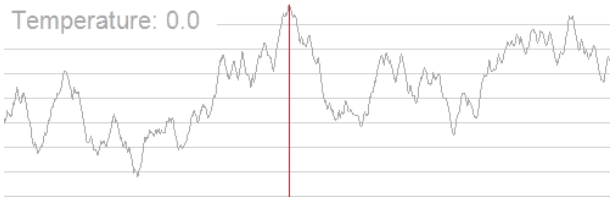


Abbildung 10: Die rote Linie springt nach links und rechts hin und her und stoppt beim Abkühlen schliesslich bei einem Maximum.

5.7.3. Unterschied zu Hill Climbing

- Hill Climbing akzeptiert ausschliesslich bessere Nachbarn
- Simulated Annealing akzeptiert auch schlechtere Nachbarn mit der Wahrscheinlichkeit $e^{\Delta \frac{E}{T}}$
- Dies ermöglicht gezielt das Entkommen aus lokalen Maxima
- Die Akzeptanzwahrscheinlichkeit sinkt bei grösseren Verschlechterungen (ΔE) und bei abnehmender Temperatur (T)

5.8. Zusammenfassung

- **Schwächen:** Lokale Suche ist anfällig für lokale Optima, Plateaus und Ridges
- **Lösungsansatz:** Hinzufügen von Zufall (Randomness) und Neustarts (Restarts)
- **Komplexere Verfahren:** Genetische Algorithmen und Simulated Annealing
  - Bauen auf den grundlegenden Ideen der lokalen Suche auf
  - Nutzen gezielt Randomisierung und das Behalten vielversprechender Kandidaten zur weiteren Optimierung

6. Game Theory I

6.1. Einleitung

Theory of Rational Choice

Zwei Unterscheidungen bei decision making agents:

- **Game Theory**: Wettbewerbsorientierte Entscheidungsfindung → braucht Rationalität
  - **Rationalität**: Ein Akteur wählt diejenige Aktion, die gemäss seinen Präferenzen mindestens so gut ist wie jede andere verfügbare Option
- **Constraint Optimization**: Nicht-wettbewerbsorientierte Entscheidungsfindung

Game Theory in der Praxis

- Börsenhandel
- Zugang zu kritischen Ressourcen (z.B. Auktionen)
- Kreditvergabe
- Gewährleistet Fairness ohne vertrauenswürdige Dritte
- Festlegung von Preisen (Starbucks, Buchungsplattformen, Fluggesellschaften)
- Platzierung von Radarkontrollen und Metalldetektoren
- Erstellung eines Sicherheitskonzeptes, z.B. Flughafen Los Angeles
- Bestimmung welche Flüge von Luftsicherheitsbeamten begleitet werden
- Analyse von Software und Sicherheitsprotokollen
- Anwendungen in Wirtschaft, Biologie, Sozialwissenschaften,...

6.2. Wichtige Definitionen

6.2.1. Dominante Strategie

DEFINITION: Strategie, die für einen Spieler immer der besten Ertrag ergibt

- **Merkmal**: Liefert in jeder Zeile / Spalte den besten Wert für den Spieler (unabhängig vom Verhalten des Gegners)
- Kann existieren, muss aber nicht
- **Sollte immer gespielt werden**

Auszahlungsmatrix (Payoff Matrix)

	Fink	**Quiet**
Fink	**5**,5	**0**, 20
Quiet	20, **0**	1, 1

Spieler 1 (Zeile) — Spieler 2 (Spalte)

Beispiel: „Fink“ ist hier die dominante Strategie. Egal was der Gegner wählt, „Fink“ bringt immer das beste Ergebnis.

6.2.2. Dominierte Strategie

DEFINITION: Strategie schneidet **immer** schlechter ab, es gibt **immer** eine bessere Strategie

- **Merkmal**: In der Zeile / Spalte gibt es einen höheren Wert (es gibt immer eine bessere Alternative)
- Kann existieren, muss aber nicht
- **Sollte nie gespielt werden**

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

Beispiel: „Middle" ist hier die dominierte Strategie für den Kicker. Egal wohin der Goalie springt, eine andere Richtung („Left" oder „Right") liefert immer einen höheren Wert.

6.2.3. Pure Strategy Nash Equilibrium

DEFINITION: Wenn die Aktion von jedem Spieler die beste Antwort für jeden anderen Spieler ist

- **Merkmal**: Stabiler Zustand, bei dem niemand einen Anreiz hat, seine Strategie einseitig zu ändern
- Überlappende Strategien
- Es kann null, ein oder mehrere nash equilibria geben
- **Dilemma**: Ein Nash Equilibrium ist oft nicht das beste Gesamtergebnis (nicht Pareto-optimal)

	Fink	**Quiet**
Fink	5,5	0,20
Quiet	20,0	1,1

Beispiel: (Fink, Fink) ist das Gleichgewicht

6.2.4. Spieldefinition

Ein strategisches Spiel hat:
1. Ein Set von Spieler
2. Für jeden Spieler ein Set von Aktionen (auch Strategien genannt)
3. Eine Präferenz für jedes Aktionsprofil jedes Spielers

Aktionsprofil: Jeder Spieler wählt eine Aktion

Präferenz eines Spielers: vollständige Ordnung von Aktionsprofilen
- $a > b$: der Spieler präferiert strikt a vor b

6.2.5. Payoff Matrix

- Repräsentiert Spiel mit zwei Spielern und deren Aktionen
- Um dominte-, dominierende Strategien und nash equilibria zu finden
- **Wichtig zu den Zahlenwerten (Utilities)**:
  - Die exakten Werte sind an sich egal
  - Sie sind frei wählbar, solange die Präferenzordnung konsistent bleibt
  - Ein Spiel wird durch Präferenzen definiert, nicht durch absolute Zahlen

Beispiel:

		Clyde (column player)	
		Fink	**Quiet**
Bonnie (row player)	**Fink**	5,5	0,20
	Quiet	20,0	1,1

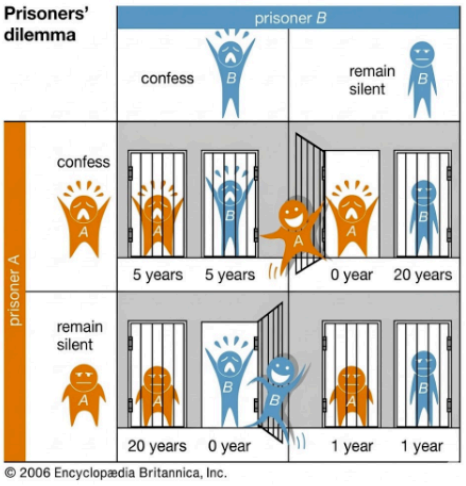
6.3. Allgemeines Vorgehen zur Spielanalyse

Um ein strategisches Spiel systematisch zu lösen, empfiehlt sich dieses schrittweise Vorgehen:

1. **Spiel formalisieren:**
   - Spieler (Players) bestimmen
   - Aktionen (Actions) festlegen
   - Alle möglichen Kombinationen als Aktionsprofile (Action Profiles) auflisten
2. **Präferenzen definieren:**
   - Präferenzordnung für jeden Spieler aufstellen (z.B. $a > b$)
   - Diese Ordnung in quantitative Nutzenwerte (Utilities, u) übersetzen
3. **Payoff Matrix erstellen:**
   - Die ermittelten Utilities in die Auszahlungsmatrix übertragen
4. **Spiel analysieren:**
   - Beste Antworten (Best Responses) für jeden Spieler ermitteln
   - Auf dominante und dominierte Strategien prüfen
   - Nash-Gleichgewichte (Nash Equilibria) identifizieren

6.4. Beispiele

6.4.1. Prisoner's Dilemma



		Clyde (column player)	
		Fink	**Quiet**
Bonnie (row player)	**Fink**	5,5	0,20
	Quiet	20,0	1,1

6.4.1.1. Analyse

Analyse für Bonny (Zeile):

- Clyde wählt Schweigen → beste Antwort ist Aussagen
- Clyde wählt Aussagen → beste Antwort ist Aussagen

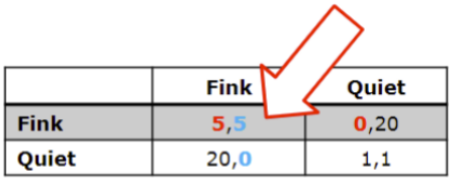
	Fink	**Quiet**
Fink	**5**,5	**0**,20
Quiet	20,0	1,1

Analyse für Clyde (Spalte):

- Bonnie wählt Schweigen → beste Antwort ist Aussagen
- Bonnie wählt Aussagen → beste Antwort ist Aussagen

	Fink	**Quiet**
Fink	5,**5**	0,20
Quiet	20,**0**	1,1

Kombinierte Analyse:



	Fink	**Quiet**
Fink	**5**,**5**	**0**,20
Quiet	20,**0**	1,1

6.4.1.2. Lösung

- Beide wählen „Aussagen“ und ändern ihre Strategie auch nicht → **Rationalität**
- Aussagen ist eine dominante Stratege für Bonnie und Clyde
- Aussagen ist ein nash equilibrium

6.4.2. Coke und Pepsi - Steady State

- Preise
- **Cola**: Spalte, rot
- **Pepsi**: Zeile, blau

	1$	2$	3$
1$	0,0	50,-10	40,-20
2$	-10,50	20,20	90,10
3$	-20,40	10,90	50,50

6.4.2.1. Lösung

	1$	2$	3$
1$	0,0	50,-10	40,-20
2$	-10,50	20,20	90,10
3$	-20,40	10,90	50,50

- Keine Dominante Strategie
- 3 Dollar ist eine dominierende Aktion für beide Spieler
- Nash Equilibrium: 1 Dollar
- **Wirtschaft**: Wettbewerb führt zu tieferen Preisen (braucht Durchsetzung)

6.4.3. Autos - law enforcement

- Zwei Autos fahren gegeneinander

	Go	Stop
Go	-5,-5	1,0
Stop	0,1	-1,-1

6.4.3.1. Lösung

	Go	Stop
Go	-5,-5	1,0
Stop	0,1	-1,-1

- Keine dominate Strategie
- Zwei nash Equilibrium (Go, Stop), (Stop, Go)

6.4.4. Penalty Game

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

6.4.4.1. Lösung

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

- Dominierende Strategie: Mitte (nie die beste Antwort)
- Kein Nash Equilibria

6.4.5. Doping in Sport

Action Profiles:

(dope, not), (dope, dope), (not, dope), (not, not)

Spieler eins:

(dope, not) ≻ (not, not) ≻ (dope, dope) ≻ (not, dope)

u(dope, not) = 4 ≻ u(not, not) = 3 ≻ u(dope, dope) = 2 ≻ u(not, dope) = 1

Spieler zwei:

(not, dope) ≻ (not, not) ≻ (dope, dope) ≻ (dope, not)

u(not, dope) = 4 ≻ u(not, not) = 3 ≻ u(dope, dope) = 2 ≻ u(dope, not) = 1

	Not	Dope
Not	3,3	1,4
Dope	4,1	2,2

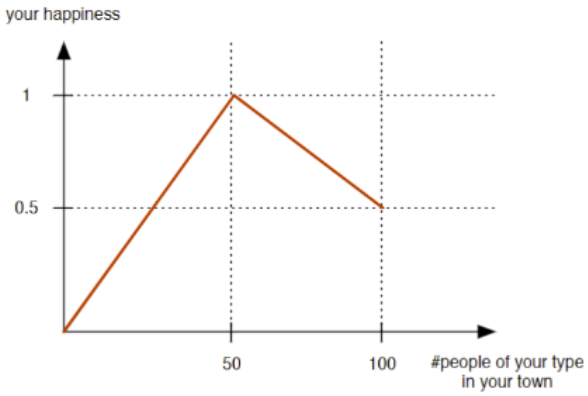
6.4.5.1. Lösung

- Dominante Strategie: Dope (immer die beste Antwort)
- Dominierte Strategie: Not (nie die beste Antwort)
- Nash Equilibrium: (Dope, Dope)

6.5. Location Game: More than two players

- Zwei Städte: East und West
- 100′000 Frauen, 100′000 Männer
- Alle müssen ihren Wohnort wählen
- Menschen bevorzugen gemischte Communities

Präferenzfunktion:



6.5.1. Strict Nash Equilibrium (stable equilibrium)

DEFINITION: Durch einen einseitigen Wechsel sinkt der Payoff rapide

Beispiel:

- Alle Menschen der gleichen Art leben in der gleichen Stadt
- Wenn eine Person wechselt, sinkt die happiness rapide (von 0.5 zu 0)
- **Fazit**: Keine Änderung, da es für den Einzelnen sofort viel schlechter wird (absolut stabil)

6.5.2. Weak Nash Equilibrium (instable equilibrium)

DEFINITION: Durch einen einseitigen Wechsel ändert sich der Payoff kaum

Beispiel:

- Jede Stadt ist 50/50 aufgeteilt
- **Fazit**: Instabil, da keine starke Verschlechterung bei Wechsel. Einseitige Abweichungen führen zum Lawineneffekt (Avalanche effect), der das Gleichgewicht zerstört.

7. Mixed Strategy

DEFINITION: man entscheidet sich nicht für eine feste Aktion sondern für eine Wahrscheinlichkeitsverteilung

7.1. Best Response

mathematisch **höchsten Durchschnittsgewinn (Expected Payoff)**

7.2. Beispiel

	Rock	Scissors	Paper
Rock	0,0	1,-1	-1,1
Scissors	-1,1	0,0	1,-1
Paper	1,-1	-1,1	0,0

Spieler 2 spielt folgende Wahrscheinlichkeit (0.25, 0.25, 0.5)

Spieler 1 findet die best Response, anhand des höchsten expected payoffs:

Spieler 1 spielt folgendes:

$$E_1(\text{Rock}) = 0.25 \cdot 0 + 0.25 \cdot 1 + 0.5 \cdot -1 = -0.25$$

$$E_1(\text{Scissors}) = 0.25 \cdot -1 + 0.25 \cdot 0 + 0.5 \cdot 1 = 0.25$$

$$E_1(\text{Paper}) = 0.25 \cdot 1 + 0.25 \cdot -1 + 0.5 \cdot 0 = 0$$

Daraus ergibt sich, dass $E_1(\text{Scissors})$ die **best Response** ist.
- hat höchsten Erwartungswert

7.3. Mixed Strategy Nash Equilibria

DEFINITION: Ein **gemischtes Nash-Gleichgewicht** beschreibt einen stabilen Zustand, in dem jeder Spieler seine Aktionen nach einer exakt berechneten **Wahrscheinlichkeitsverteilung** wählt.

Diese Verteilung ist so gewählt, dass die Gegner keinen Anreiz haben, von ihrer eigenen Strategie abzuweichen, da jede ihrer Optionen denselben erwarteten Nutzen bringt.

Jedes Spiel (mit einer begrenzten Anzahl an Spielzügen) hat **garantiert** mindestens ein Mixed Strategy Nash-Equilibria.

Jedes Pure Strategy Equilibria ist auch ein Mixed Strategy Nash Equilibria

7.4. Finding Mixed Strategy Equilibrium

- q und $q - 1$ für Spieler 1 definieren
- p und $p - 1$ für Spieler 2 definieren
- Gleichung für Spielzüge aufstellen
  - für Spieler 1
  - und für Spieler 2
- beide Gleichungen gleichstellen berechnen
  - q berechnen
  - p berechnen

7.4.1. Beispiel

- mixed strategy Djokovic: $p, 1 - p$
- mixed strategy Federer: $q, 1 - q$

Federer gegen Djokovic:

payoff für Federer gegen mix strategy von Djokovic
- Left $q \cdot 50 + (1 - q) \cdot 80 = 80 - 30q$
- Right: $q \cdot 90 + (1 - q) \cdot 20 = 20 + 70q$

		Djokovic (anticipates at the net)	
		q	1-q
		Left	**Right**
Federer (plays passing shot)	p	**Left**	50,50	80,20
	1-p	**Right**	90,10	20,80

In einem equilibrium, beide payoffs müssen gleich sein Left = Right: $80 - 30q = 20 + 70q$ $q = 0.6$ mix von Djokovic ist: (0.6, 0.4)

Djokovic gegen Federer: payoff für Djokovic gegen mix strategy von Federer
- Left $p \cdot 50 + (1 - p) \cdot 10 = 10 - 40p$
- Right: $p \cdot 20 + (1 - q) \cdot 80 = 80 + 60p$

In einem equilibrium, beide payoffs müssen gleich sein Left = Right: $10 - 40p = 80 + 60p$ $q = 0.7$

mix von Federer ist: (0.7, 0.3)

Fazit
- Federer: (0.7, 0.3)
- Djokovic: (0.6, 0.4)
- beide spielen eine best response der anderen mixed strategy
- (0.7, 0.3) und (0.6, 0.4) sind ein mixed strategy Nash equilibrium

8. Game Theory III - Google AdWords Case Study

- Google AdWords generiert den grössten Teil der Werbeeinnahmen von Google
- Der Umsatz stieg von 2001 bis 2024 enorm an und erreichte 264.59 Milliarden US-Dollar im Jahr 2024
- Wenn Nutzer eine Suchanfrage stellen, findet im Hintergrund eine Auktion für die Werbeplätze statt
- Bietende definieren dafür ein Tagesbudget sowie ein maximales Gebot pro Suchanfrage

8.1. Auktionsarten

8.1.1. First Price Auction

DEFINITION: Auktion, bei der das höchste Gebot gewinnt und der Gewinner sein eigenes Gebot zahlt

- **Payoff**: Wenn das Gebot b_i gewinnt, ist der Nutzen für Spieler i gleich $v_i - b_i$. Wenn das Gebot verliert, ist der Nutzen 0
- **Verhalten**: Bei dieser Auktionsform ist „Under-bidding“ (unter dem wahren Wert bieten) eine dominante Strategie

8.1.1.1. Beispiel

- Wert des Objekts $v = 3\$$

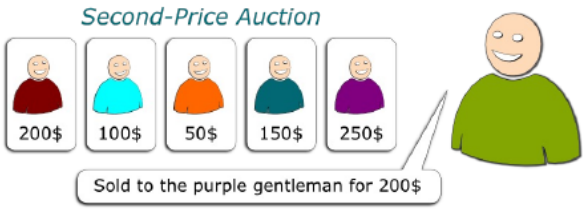
	1$	2$	3$	4$	5$
1$	0,0	**0**,**1**	**0**,0	**0**,-1	**0**,-2
2$	**1**,**0**	**0**,**0**	**0**,**0**	**0**,-1	**0**,-2
3$	0,**0**	**0**,**0**	**0**,**0**	**0**,-1	**0**,-2
4$	-1,**0**	-1,**0**	-1,**0**	**0**, **0**	**0**,-2
5$	-2,**0**	-2,**0**	-2,**0**	-2,**0**	**0**, **0**

- **Beste Antwort**: Für beide Spieler ist ein Gebot von 2$ die beste Antwort auf jede mögliche Aktion des anderen Spielers
- **Dominante Strategie**: Daraus folgt, dass ein Gebot von 2$ eine (schwach) dominante Strategie darstellt
- **Under-bidding**: Da das optimale Gebot (2$) geringer ist als der tatsächliche Wert (3$), führt die First Price Auction systematisch zu „Under-bidding“

8.1.2. Second Price Auction

DEFINITION: Auktion, bei der das höchste Gebot gewinnt, der Gewinner jedoch nur das zweithöchste Gebot zahlt

- **Payoff**: Wenn das Gebot b_i gewinnt und b_j das zweithöchste Gebot ist, beträgt der Nutzen $v_i - b_j$
- **Verhalten**: Das Bieten des tatsächlichen Wertes ($b_i = v_i$) ist hier eine schwach dominante Strategie
- Es gibt kein „Under-bidding“ mehr



8.1.2.1. Beweis: Bieten des wahren Wertes ($b_i = v_i$)

- Ein Abweichen vom wahren Wert (v_i) verbessert den Nutzen („Payoff“) nie, unabhängig von den Strategien der anderen

- **Fall 1 (Überbieten, $b > v_i$):** Relevante Änderung nur, wenn man dadurch gewinnt, statt zu verlieren. Das zweithöchste Gebot (b_j) ist dann grösser als v_i. Der Nutzen wird dadurch negativ oder null ($v_i - b_j \leq 0$)
- **Fall 2 (Unterbieten, $b < v_i$):** Relevante Änderung nur, wenn man dadurch verliert, statt zu gewinnen. Der vorher positive Nutzen ($v_i - b_j \geq 0$) fällt dadurch auf 0

8.1.2.2. Beispiel

- Wert des Objekts $v = 3\$$

	1$	2$	3$	4$	5$
1$	0,0	0,**2**	**0**,**2**	**0**,**2**	**0**, **2**
2$	**2**,0	0,0	**0**,**1**	**0**,**1**	**0**, **1**
3$	**2**,**0**	**1**,**0**	**0**,**0**	**0**,**0**	**0**, **0**
4$	**2**,**0**	**1**,**0**	**0**,**0**	**0**,**0**	**0**,-1
5$	**2**,**0**	**1**,**0**	**0**,**0**	-1,**0**	**0**, **0**

- **Dominante Strategie**: Für beide Spieler stellt ein Gebot von 3$ oder 4$ eine (schwach) dominante Strategie dar
- **Kein Under-bidding**: Dieses Gebot entspricht mindestens dem tatsächlichen Wert des Objekts (3$), was bedeutet, dass bei der Second Price Auction kein „Under-bidding“ auftritt

8.2. Zusammenfassung für Plattformbetreiber

- Plattformen wie Google möchten „Under-bidding“ vermeiden, um keine Einnahmen zu verpassen
- Bei einer Second Price Auction bieten die Akteure ihren wahren Wert
- Neben dem reinen Gebot muss Google auch sicherstellen, dass die Anzeigen relevant und von hoher Qualität sind, um die eigene Reputation zu wahren

8.3. Ausblick: AI in Suchmaschinen

- ChatGPT wurde im November 2022 veröffentlicht, woraufhin Google BARD im Februar 2023 ankündigte
- Microsoft Bing war die erste grosse Suchmaschine, die LLM-generierte Zusammenfassungen integrierte
- Es wird spekuliert, dass Google mit der Integration abgewartet hat, um das sehr profitable Werbegeschäft nicht zu gefährden

9. Game Theory IV - Sequential Games

9.1. Definitionen

Simultaneous Games: Spieler wählen ihre Aktion ohne das Wissen, was der andere Spieler macht
- Gefangenen Dilemma
- Schere-Stein-Papier

Sequential Games: Spieler kennen die vorherige Aktion / möglichen nachfolgenden Aktionen des anderen Spielers
- Schach
- Ultimatumspiel (Spieler A nimmt an oder nicht)

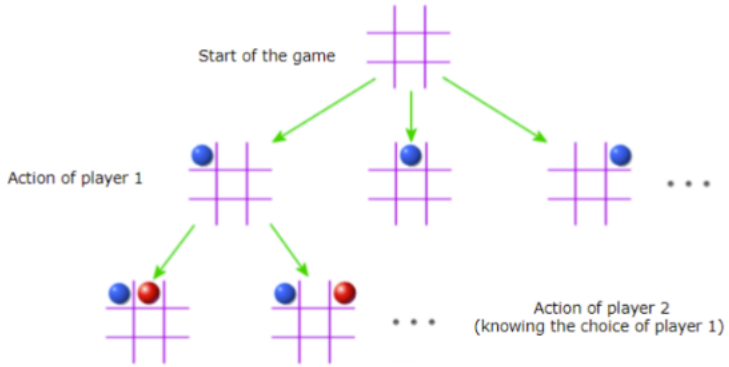
Endliche (finite) Games: finden auf einem begrenzten Spielfeld statt, Züge sind mathematisch definiert, da fixe Anzahl
- Tic-Tac-Toe
- Schach

Perfekte Information: jeder kennt die möglichen Spielzüge
- Schach

Imperfekte Informationen: die Züge des anderen sind nicht bekannt
- Poker
- verdeckte Karten

9.2. Spielbaum für endliche Spiele

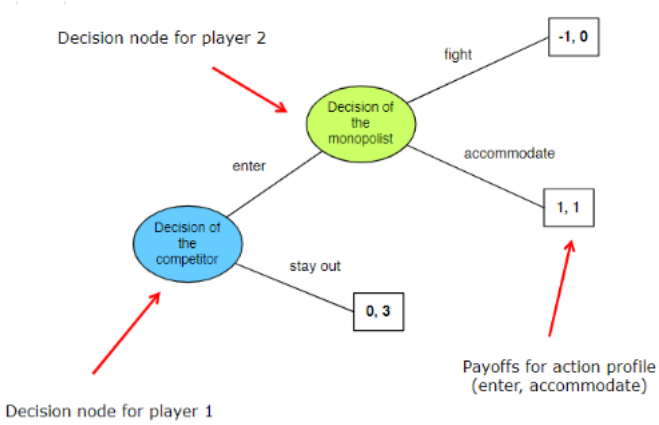
Endliche (finite) Spiele können als **Spielbaum** dargestellt werden.



9.3. Exercise: Entry Game

Ein Monopolist verdient aktuell **3 Mio.**. Ein potenzieller Wettbewerber entscheidet über einen Markteintritt:

- **Kein Eintritt:** Der Wettbewerber verdient **0**, der Monopolist behält seine **3 Mio.**.
- **Markteintritt:** Der Monopolist muss wählen, wie er reagiert:
  - **Akkommodieren (dulden):** Beide Firmen verdienen jeweils **1 Mio.** (Weil die Produkte billiger werden)
  - **Kampf (Preiskrieg):** Der Monopolist verdient **0**, der Wettbewerber verliert **1 Mio.** (hohe Marketing Kosten)



9.4. Theorem of Zermelo

Jede endliche zwei Spieler Spiel mit perfekten Information hat entweder:

1. Spieler 1 **gewinnt**, egal was Spieler 2 macht → **first-mover advantage**
2. Spieler 2 **gewinnt**, egal was Spieler 1 macht → **second-mover advantage**
3. **Unentschieden**

Beispiele:

- **first-mover advantage**: Vier-Gewinnt
- **second-mover advantage**: sequenzielles Schere-Stein-Papier

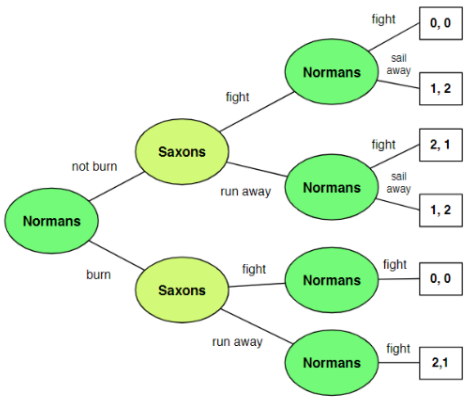
9.5. Backward Induction

DEFINITION: Lösungsverfahren für sequenzielle Spiele, bei dem vom Ende des Spielbaums (Payoffs) schrittweise rückwärts zum Anfang gerechnet wird.

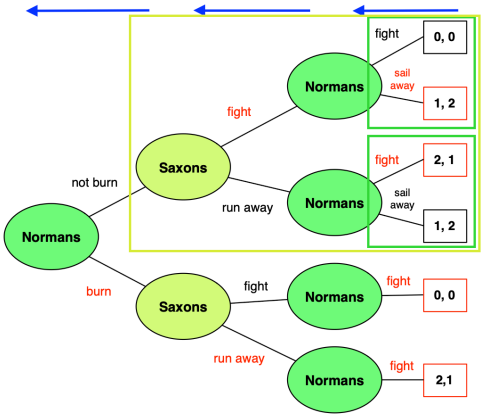
- Sequenzielle Spiele könne mit Backward Induction analysiert werden
- Es ist wichtig, dass beide Spieler rational entscheiden
- Spieler müssen die vorherigen Spielzüge wissen
  - Bsp. im 1066 Game müssen alle mitbekommen, dass die Schiffe abgebrannt wurden

9.5.1. The 1066 Game

Battle in 1066 zwischen Normans und Saxons. Die Normans greifen die Saxons an.

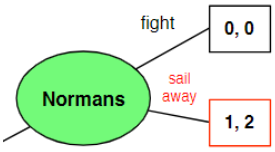


9.5.1.1. Backward Induction



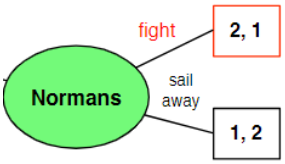
- Spieler 1: Normans
- Spieler 2: Saxons

Normans: oben rechts:



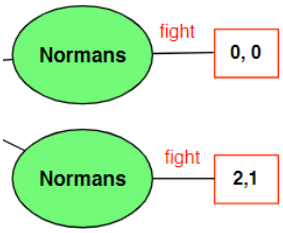
- fight: 0
- sail away: 1
- Normans wählen **sail away**

Normans: zweiter von oben rechts:



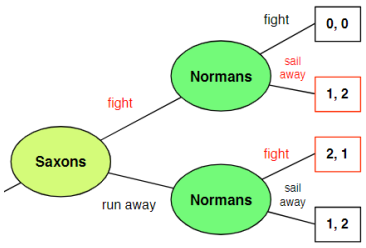
- fight: 2
- sail away: 1
- Normans wählen **fight**

Normans: dritter und vierter von oben rechts:



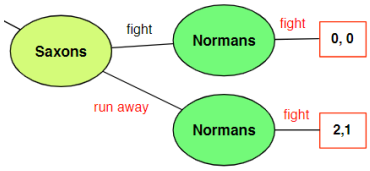
- können nur noch **fight** wählen, da sie nicht mehr fliehen können

Saxons: Mitte von oben:



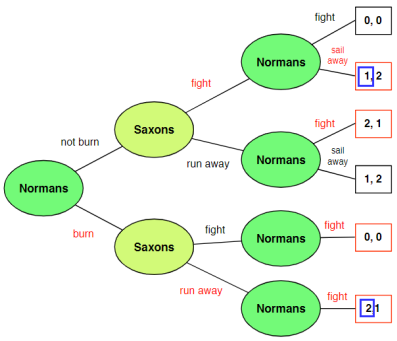
- fight: 2, da Normans dann sail away wählen
- run away: 1, da Normans dann fight wählen
- Saxons wählen **fight**

Saxons: Mitte von unten



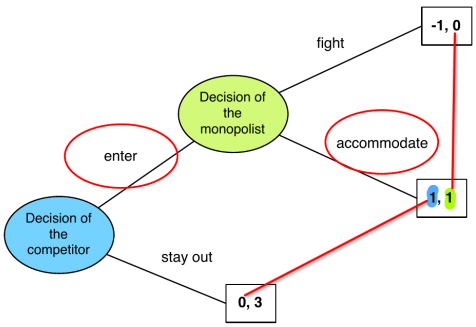
- fight: 0
- run away: 1
- Saxons wählen **run away**

Start node:



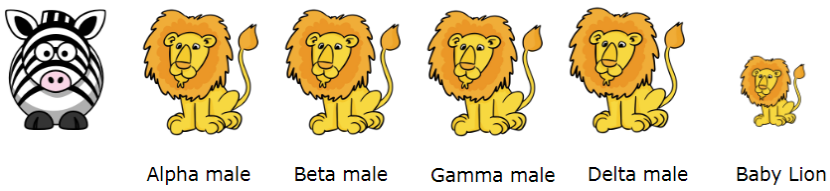
Normans wählen **burn**, da sie wissen, dass die Saxons dann **run away** wählen und sie durch **fight** 2 Punkte erhalten.

9.5.2. Entry Game



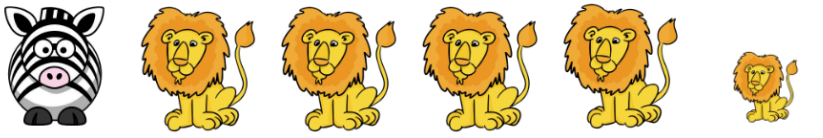
*Competitor wird **enter** wählen, da sie wissen das monopolist **accommodate** wählen wird und 1 besser als 0 ist*

9.5.3. The Lion Games



- Der Alpha male kann das Zebra fressen, dann macht er aber einen Verdauungsschlaf
- wenn Alpha male im Verdauungsschlaf ist, kann Beta male ihn fressen und fällt ebenfalls in den Verdauungsschlaf
- usw.
- Frage: Soll der alpha male das Zebra fressen oder nicht?

Lösung mit Backwards Induction:



- Baby Lion: wird den delta male essen, da er keine Feinde mehr hat
- Delta male: wird den gamma male nicht essen, da er sonst vom baby lion gegessen wird
- Gamma male: wird den beta essen, da der delta male ihn nicht fressen wird
- Beta male: wird den alpha male nicht essen, da er sons vom gamma male gefressen wird
- Alpha male: wird das Zebra fressen, da der beta male ihn nicht essen wird
- **Lösung: Alpha male kann das Zebra fressen**

9.5.4. The Pirate Game

- **Ausgangslage:** 5 Piraten müssen 100 Goldmünzen unter sich aufteilen.
- **Abstimmungsmodus:** Der ranghöchste Pirat macht einen Verteilungsvorschlag. Alle (einschliesslich des Vorschlagenden) stimmen ab.
- **Entscheidungsregeln:**
  - Bei einer Mehrheit wird der Plan angenommen.
  - Bei einem Unentschieden hat der Vorschlagende die entscheidende Stimme.
  - Wird der Plan abgelehnt, wird der Vorschlagende über Bord geworfen (stirbt), und der nächste Ranghöchste macht einen neuen Vorschlag.
- **Prioritäten der Piraten:**
  1. Überleben.
  2. Maximierung des eigenen Goldanteils.
  3. Anderen Piraten schaden (sie über Bord werfen), wenn das Ergebnis ansonsten gleich bleibt.
  4. Kein Vertrauen untereinander.

9.5.4.1. Lösung

A (älteste), B, C, D, E (jüngster)

1. zwei verbleibende Piraten: D = 100, E = 0
2. drei verbleibende Piraten: C = 99, D = 0, E = 1
3. vier verbleibende Piraten: B = 99, C = 0, D = 1, E = 0
4. alle fünf: A = 98, B = 0, C = 1, D = 0, E = 1

9.5.4.1.1. 2 Piraten (D und E)

- **D** macht den Vorschlag.
- Er braucht nur seine eigene Stimme (bei Gleichstand zählt seine Stimme doppelt).
- **Ergebnis:** D nimmt alles, E bekommt nichts.
- **Stand:** D: 100, E: 0.

9.5.4.1.2. 3 Piraten (C, D und E)

- **C** macht den Vorschlag. Er braucht eine weitere Stimme (neben seiner eigenen).
- Er weiss, wenn sein Plan abgelehnt wird, bekommt **E** im nächsten Schritt (siehe oben) garantiert **0**.
- C bietet **E** also **1 Münze** an. Das ist besser als 0, also stimmt E zu.
- **Ergebnis:** C: 99, D: 0, E: 1.

9.5.4.1.3. 4 Piraten (B, C, D und E)

- **B** macht den Vorschlag. Er braucht eine weitere Stimme für ein Unentschieden (2 von 4).
- Er schaut auf das Szenario mit 3 Piraten: Dort bekommt **D** genau **0**.
- B bietet **D** also **1 Münze** an. D stimmt zu, um überhaupt etwas zu bekommen.
- (C bekommt 0, da er B am liebsten über Bord werfen würde, um selbst 99 zu kassieren).

- **Ergebnis:** B: 99, C: 0, D: 1, E: 0.

9.5.4.1.4. Alle 5 Piraten (A bis E)

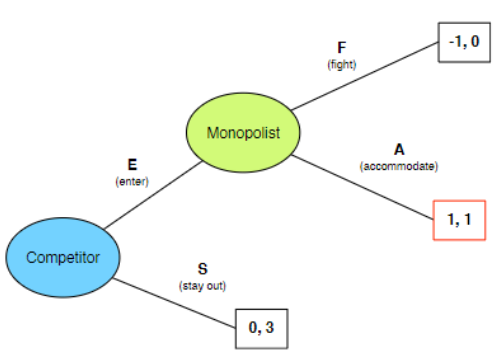
- **A** macht den Vorschlag. Er braucht zwei weitere Stimmen (insgesamt 3 von 5).
- Er schaut auf das 4-Piraten-Szenario: Dort gehen **C** und **E** leer aus (**0**).
- A bietet **sowohl C als auch E jeweils 1 Münze** an.
- Da 1 mehr ist als 0, stimmen beide zu, um ihr Überleben und einen kleinen Gewinn zu sichern.
- **Das Ergebnis:**
  - **A: 98 Münzen**
  - B: 0 Münzen
  - **C: 1 Münze**
  - D: 0 Münzen
  - **E: 1 Münze**

9.6. Nash Equilibrium in Sequenziellen Spielen

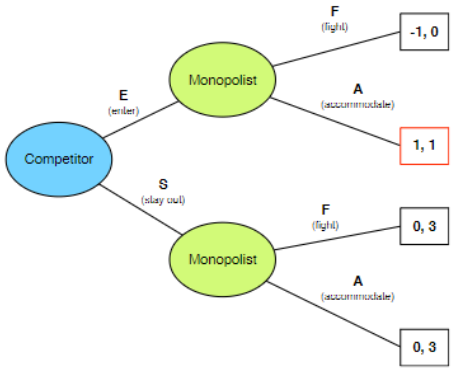
Das Nash Equilibrium in sequentiellen Games wird durch Backward Induction gefunden.

9.6.1. Analyse

- Ein sequential Game kann auch als simultaneous Game analysiert werden.
- Dafür muss der **game tree komplett** sein. Der Baum wird einfach ergänzt.



Redundancy-Free Game Tree



Complete Game Tree

Nash Equilibrium:

- Man findet **das gleiche** Nash Equilibrium mit der sequential und simultaneous game Analyse
- **Für simultaneous games findet man oft mehr als eines**

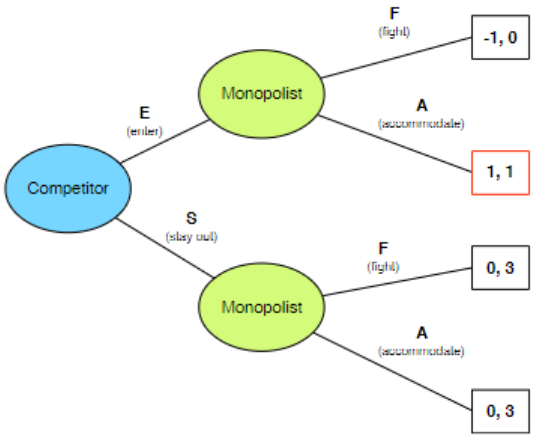
9.6.2. Sub - Game Perfect Nash Equilibrium

DEFINITION: Ein isolierter Ast des Spielbaums. Die Analyse entlarvt unglaubwürdige Drohungen, da eine Strategie in jedem Teilspiel zwingend rational sein muss.

- Ein Nash-Gleichgewicht, das für das gesamte Spiel und jedes einzelne Teilspiel (Sub-Game) gilt
- Schliesst unglaubwürdige Drohungen aus
- Die Strategie muss an jedem Knoten im Spielbaum rational sein
- Backward Induction findet dieses Gleichgewicht automatisch

9.6.3. Beispiele

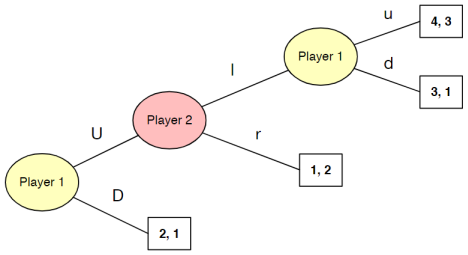
9.6.3.1. Entry game

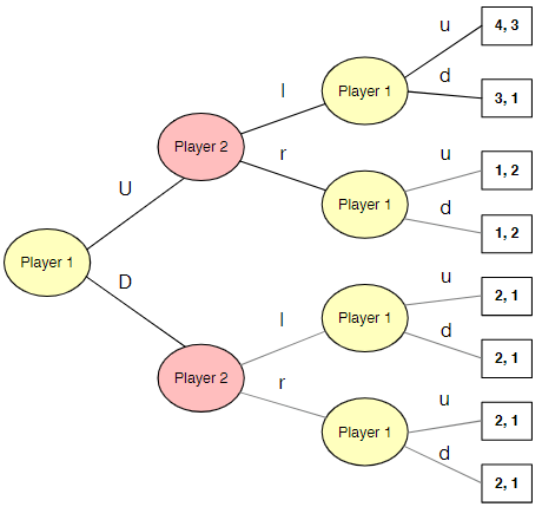


	Fight (F)	Accommodate (A)
Enter (E)	-1,0	**1**,1
Stay out (S)	**0**,3	0,3

sub-game perfect nash equilibrium: 1,1 nash equilibrium: 0,3 und 1,1

9.6.3.2. Don't Screw up



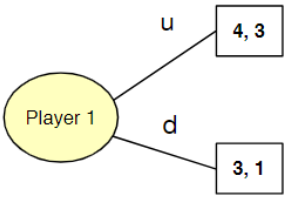


	l	r
Uu	**4**,3	1,2
Ud	3,1	1,**2**
Du	2,**1**	**2**,1
Dd	2,**1**	**2**,1

Nash Equilibria found in simultaneous game:

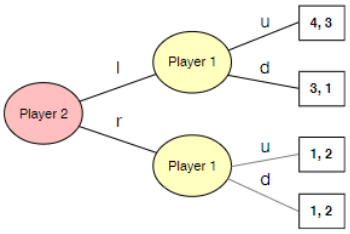
- (Uu, l) → compatible with backward induction
- (Du, r) → incompatible with backward induction
- (Dd, r) → incompatible with backward induction

Sub-games:



u	**4**,3
d	3,1

- Nash equilibrium: u
- Vergleich mit allen Nash equilibriums: (Uu, l), (Du, r), (Dd, r)
- (Dd, r) passt nicht mehr und wird somit entfernt



	l	r
u	**4**,3	**1**,2
d	3,1	**1**,2

- Nash equilibrium: (u,l), (d,r)
- Vergleich mit allen Nash equilibriums: (Uu, l), (Du, r), (Dd, r)
- (Uu, l), (Dd, r) passen noch
- (Du, r) passt nicht mehr und wird entfernt
- (Dd, r) ist bereits entfernt und somit gilt **(Uu, l) als Sub - Game Perfect Nash Equilibrium**

10. Constraint Programming 1 - Modelling with OR-Tools

10.1. Google OR-Tools

- State-of-the-art CP-Framework
- Gewann MiniZinc Competition (Weltmeisterschaft für Optimierungen)
- Apache License 2.0 (kommerziell nutzbar)
- In C++ implementiert
- Verfügbar für .NET, Java, Python

10.2. Constraint vs. Optimization Problems

Constraint Problem (CP):

- Besteht aus Variablen, Domains, Constraints
- **Variable**: Bekommt Werte
- **Domain**: Endliches Set an Werten
- **Constraint**: Definiert verbotene Zuordnungen
- Lösung: Gültige Zuordnung (befriedigt alle Constraints)
  - Kann 0, 1 oder mehrere Lösungen haben

Constraint Optimization Problem (OP):

- CP mit Zielfunktion
- Lösung: Befriedigt Constraints und optimiert Zielfunktion

→ OR-Tools behandelt beide

10.3. Implementierung

1. Neues Modell erstellen
2. Variablen definieren
3. Constraints hinzufügen (`Add`)
4. Solver erstellen und starten

```python
from ortools.sat.python import cp_model

# 1. Create model
model = cp_model.CpModel()

# 2. variables
r = model.NewIntVar(MIN Int, MAX Int, 'name')
p = model.NewIntVar(MIN Int, MAX Int, 'name2')

# 3. Constraints
model.Add(...)
model.Add(...)

# 4. Create and launch solver
solver = cp_model.CpSolver()
status = solver.Solve(model)

# Print solution if exists
if status == cp_model.OPTIMAL:
    print(f"name1: {solver.Value(var1)} name2: {solver.Value(var2)}")
```

10.3.1. Ausgabe einer Lösung

Manuell:

```python
solver = cp_model.CpSolver()
status = solver.Solve(model)
```

```
print(f"Product 1: ${solver.Value(p1)/100:.2f}")
# ...

print('Status:', status)
```

Default printing mit OR-Tools:

```python
solver = cp_model.CpSolver()
solver.parameters.enumerate_all_solutions = False

callback = cp_model.VarArraySolutionPrinter([p1, p2, p3, p4])
status = solver.Solve(model, callback)
print('Status:', status)
```

Solver-Status:
- 0: UNKNOWN
- 1: MODEL_INVALID
- 2: FEASIBLE
- 3: INFEASIBLE
- 4: OPTIMAL

Assert für Optimal:

```python
assert status == cp_model.OPTIMAL
```

10.3.2. Alle Lösungen ausgeben

```python
solver = cp_model.CpSolver()
solver.parameters.enumerate_all_solutions = True

callback = cp_model.VarArraySolutionPrinter([var1, var2, ...])
status = solver.Solve(model, callback)
print('Status:', status)
```

- Gibt symmetrische Lösungen aus
- Lösung: Symmetry Breaking hinzufügen

10.3.3. Symmetry Breaking

DEFINITION: Zusätzlicher Constraint, verhindert symmetrische Lösungen.

```python
model.Add(var1 <= var2)
model.Add(var2 <= var3)
# ...

solver = cp_model.CpSolver()
solver.parameters.enumerate_all_solutions = True

callback = cp_model.VarArraySolutionPrinter([var1, var2, ...])
status = solver.Solve(model, callback)
```

10.3.4. Alle Lösungen mit Custom Printing

- Sinnvoll für komplexe Ausgaben

```python
class MySolutionPrinter(cp_model.CpSolverSolutionCallback):
    def __init__(self, variables):
        cp_model.CpSolverSolutionCallback.__init__(self)
        self.__variables = variables

    def on_solution_callback(self):
        print(f"Var1: {self.Value(self.__variables[0])}")
        # ...

solver = cp_model.CpSolver()
solver.parameters.enumerate_all_solutions = True

callback = MySolutionPrinter([p1, p2, p3, p4])
status = solver.Solve(model, callback)
```

10.3.5. Global Constraints

DEFINITION: Built-in Constraints in OR-Tools für einfachere Implementierung.

10.3.5.1. Ausgewählte global constraints

→ nur zur Information

- `AllDifferent` : auf eine variable Anzahl an Variablen anwendbar
  - ähnlich wie `<=` oder `==`
- `Circuit` : Liste von Graph edged muss einem Hamiltonian path entsprechen
- `Element` : Einschränkung für Array-Werte mit einer Entscheidungsvariablen als Index
- `NoOverlap` : Stellt sicher, dass sich Intervalle zeitlich nicht überschneiden
- `Inverse` : Spiegel Array-Werte und Indizes
  - `variables[i] == j ⇔ inverse[j] == i`
- `MaxEquality` : Bindet den Maximalwert eines Arrays an einen Zielwert
  - `targe == maximum(variables)`
- `OnlyEnforceIf` : Wenn A eine Bedingung erfüllt, wird B gezwungen, einen bestimmen Wert anzunehmen

10.4. Beispiele

10.4.1. Hasen und Fasanen

Frage: In einem Feld mit Hasen und Fasanen hat es 20 Köpfe und 56 Beine. Wie viele Hasen und Fasanen hat es?

- Variablen: Hasen und Fasanen, min 0, max 20
- Values: Köpfe, Beine
- Constraints: Summe Hasen und Fasane = 20 Köpfe, Hasen x 4 + 2 x Fasane = 56 Beine

10.4.1.1. Code

```python
from ortools.sat.python import cp_model

# 1. Create model
model = cp_model.CpModel()

# 2. One variable for rabbits and pheasants (there cannot be more than 20)
r = model.NewIntVar(0, 20, 'rabbits')
p = model.NewIntVar(0, 20, 'pheasants')

# 3. Constraints
# Heads sum up to 20:
model.Add(r + p == 20)

# Legs sum up to 56:
model.Add(4 * r + 2 * p == 56)

# 4. Create and launch solver
solver = cp_model.CpSolver()
status = solver.Solve(model)

# Print solution when exists
if status == cp_model.OPTIMAL:
    print(f"Rabbits: {solver.Value(r)} Pheasants: {solver.Value(p)}")
```

10.4.2. Im Einkaufsladen

Aufgabe:

- Kind im Lebensmittellade $\rightarrow$ kauft **vier Artikel**
- Kind bezahlt: **7.11 Dollar**
- der **Preis bleibt gleich**, egal ob man die Artikel **addiert** oder **multipliziert**

Variablen:

- für jeden Preis der Produkte, p_1, p_2, p_3, p_4
- Integers bevorzugen (alle Preise mit 100 multiplizieren)
- Preise können nicht negativ sein

Values:

- Werte zwischen 0 und 711

Constraints:

- Preise addiert oder multipliziert ergibt immer 711 Doller
  - $p_1 + p_2 + p_3 + p_4 = 711$
  - $p_1 \times p_2 \times p_3 \times p_4 = 711 \times 100^3$
    - da die Preise mit 100 multipliziert wurden

10.4.2.1. Code

```python
from ortools.sat.python import cp_model

model = cp_model.CpModel()

# One variable for each product:
p1 = model.NewIntVar(0, 711, 'p1')
p2 = model.NewIntVar(0, 711, 'p2')
p3 = model.NewIntVar(0, 711, 'p3')
p4 = model.NewIntVar(0, 711, 'p4')

# Prices add up to 711:
model.Add(p1 + p2 + p3 + p4 == 711)

# Product of prices is 711:
model.Add(p1 * p2 * p3 * p4 == 711 * 100 * 100 * 100)

solver = cp_model.CpSolver()
status = solver.Solve(model)

if status == cp_model.OPTIMAL:
    print(f"Product 1: ${solver.Value(p1)/100:.2f}")
    print(f"Product 2: ${solver.Value(p2)/100:.2f}")
    print(f"Product 3: ${solver.Value(p3)/100:.2f}")
    print(f"Product 4: ${solver.Value(p4)/100:.2f}")
```

10.4.3. Cryptogram Puzzle

- Buchstaben: S, E, N, D, M, O, R, Y
- jeder Buchstabe ist eine Zahl von 0 bis 9
- Unterschiedliche Buchstaben haben unterschiedliche Nummern
- Zahlen dürfen nicht mit einer 0 starten
- Die folgende Gleichung muss gelten: SEND + MORE = MONEY

Variablen: Buchstaben S, E, N, D, M, O, R, Y **Values**: Zahlen von 0 bis 9 für die Buchstaben **Constraints**: Gleichung, Zahlen dürfen nicht mit 0 starten

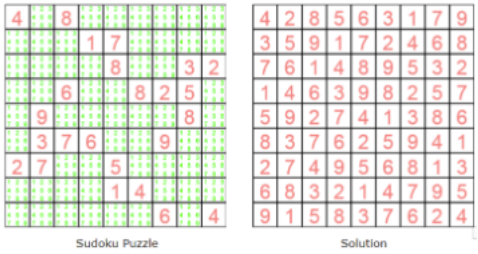
```python
model = cp_model.CpModel()

# Variablen: One decision variable for each character:
S = model.NewIntVar(0, 9, 's')
E = model.NewIntVar(0, 9, 'e')
N = model.NewIntVar(0, 9, 'n')
D = model.NewIntVar(0, 9, 'd')
M = model.NewIntVar(0, 9, 'm')
O = model.NewIntVar(0, 9, 'o')
R = model.NewIntVar(0, 9, 'r')
Y = model.NewIntVar(0, 9, 'y')

# SEND + MORE = MONEY:
# Stelle des Buchstabens muss aufmultipliziert werden
send  = S * 1000 + E * 100 + N * 10 + D
more  = M * 1000 + O * 100 + R * 10 + E
money = M * 10000 + O * 1000 + N * 100 + E * 10 + Y

model.Add(send + more == money)

# Leading characters must not be zero:
```

```
model.Add(S != 0)
model.Add(M != 0)
```

- das ergibt 155 nicht symmetrische Lösungen
- es fehlt die constraint, dass alle Buchstaben unterschiedliche Zahlen sein müssen
- dies kann mit `model.AddAllDifferent` implementiert werden
- dann findet es nur noch eine Lösung

```
model.AddAllDifferent([S, E, N, D, M, O, R, Y])
```
PYTHON

10.4.4. Sudoku



- Jede Zelle ist eine Entscheidungsvariable mit Werten zwischen 1 und 9
- Die Zahlen in einer Zeile müssen unterschiedlich sein
  - `AllDifferent` pro Zeile
- Zahlen in einer Spalte müssen unterschiedlich sein
  - `AllDifferent` pro Spalte
- Zahlen in einem 3x3-Block müssen unterschiedlich sein
  - `AllDifferent` pro Block

10.4.4.1. Code

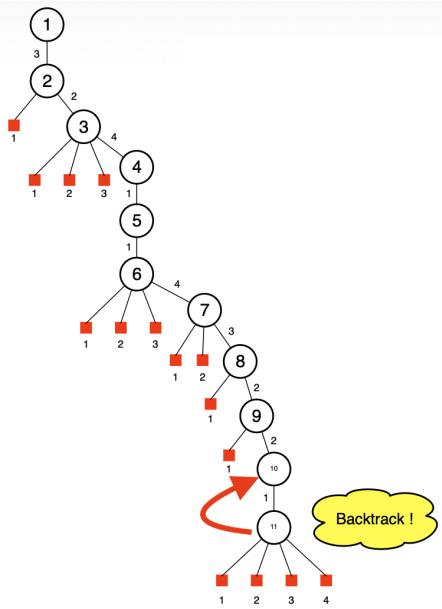
```
model = cp_model.CpModel()

# 9x9 Matrix of Decision Variables in {1..9}:
board = [[model.NewIntVar(1, 9, f"({i},{j})") for j in range(9)] for i in range(9)]

# Some Pre-Assignments:
model.Add(board[0][0] == 4)
model.Add(board[0][2] == 8)
model.Add(board[1][3] == 1)

# Each row / column contains only different values:
for i in range(9):
    model.AddAllDifferent([board[i][j] for j in range(9)]) # Rows
    model.AddAllDifferent([board[j][i] for j in range(9)]) # Columns

# Each 3x3 sub-grid contains only different values:
for i in range(3):
    for j in range(3):
        model.AddAllDifferent(
            [board[i * 3 + di][j * 3 + dj] for di in range(3) for dj in range(3)])

# Print solution when exists
# ...
```
PYTHON

11. CP Algorithms

11.1. Backtrack Search

DEFINITION: Ausprobieren von Werten in fester Reihenfolge, mit sofortigem Rücksprung (Backtracking) bei Bedingungsverletzungen.

1. **Variablen-Reihenfolge:** alle Variablen in festgelegter Reihenfolge besuchen
2. **Werte-Auswahl:** Nacheinander verfügbare Optionen testen.
3. **Constraint-Check:**
   - **Fehler:** Nächsten Wert prüfen. Falls keine Werte mehr übrig, **Rücksprung (Backtrack)** zur letzten Variable.
   - **Erfolg:** Weiter zur nächsten Variable.



11.1.1. Eigenschaften - Vorteile & Nachteile

- **vollständig**
- **Zeitkomplexität:** worst case ist exponentiell
- **Geschwindigkeit:** Oft schnell, bei häufigem Backtracking aber sehr langsam
- **Speicherkomplexität:** linear, daher ziemlich effizient
- **Parallelisierung** gut geeignet

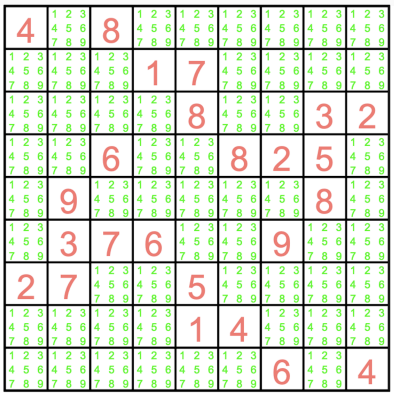
11.1.2. Problem

- **Früher Backtrack:** Schnelleres Finden von Fehlern spart Zeit.
- **Faustregel:** Weniger Optionen = schnellere Suche.
- **Strategie:** Wertebereich (Domain) pro Variable minimieren, um Sackgassen zu vermeiden. → Forward Checking

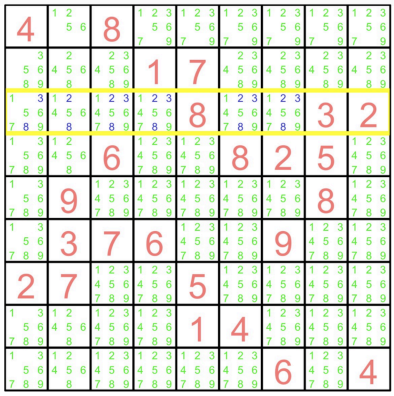
11.2. Forward Checking

DEFINITION: Frühzeitige Filterung unmöglicher Werte direkt nach jeder Variablenzuweisung zur Reduktion des Suchraums.

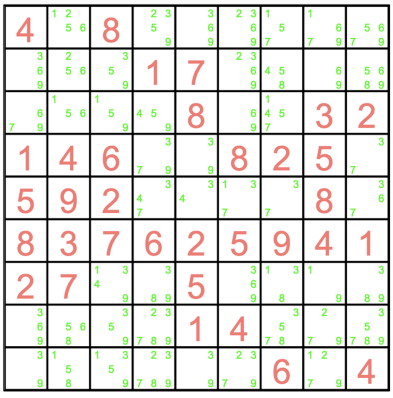
- Ungültige Werte sofort eliminieren (Constraints prüfen).
- Nur ein Wert übrig → Zuweisung
- **Unvollständig** (alleine keine garantierte Lösung).
- Drastische Vereinfachung **vor / während der Backtrack-Suche**
- **Workflow:** Reduzieren → Backtrack starten → erneut Reduzieren



Start Initial Forward Checking



Randomly selection situation in-between: Yellow frame indicates AllDifferent constraint that is correctly checked. Blue values are removed by forward checking



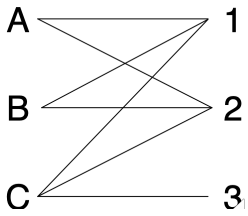
End Initial Forward Checking

11.3. Bounds Consistency

11.3.1. Constraint Problem mit Value Graphs

DEFINITION: **Value Graph** ist ein bipartiter Graph zur Visualisierung und Analyse von Constraint-Problemen

bipartiter Graph (zwei getrennte Mengen)
- **Linke Seite:** Variablen
- **Rechte Seite:** Mögliche Werte (Domain)



- **Linie**: „Dieser Wert ist für diese Variable zulässig."

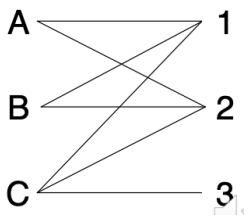
Ziel: Algorithmische Identifikation von fixen Zuweisungen und Lösungswegen.

Logik: Wenn eine Variable nur eine Kante hat, ist sie „determiniert" (festgelegt).

11.3.1.1. Beispiel

Szenario
- **Personen (Variablen):** Alice (A), Bob (B), Cecile (C).
- **Tage (Werte):** 1 (Fr), 2 (Sa), 3 (So).
- **Bedingung:** `AllDifferent` (Jeder Tag wird von einer anderen Person besetzt).

Einschränkungen (Frames):
- **A & B:** Können nur an Tag 1 oder 2 arbeiten ($A, B \in \{1, 2\}$).
- **C:** Kann an allen Tagen arbeiten ($C \in \{1, 2, 3\}$).

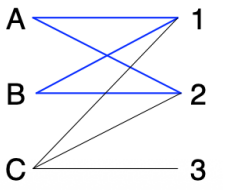


- Da A und B die Tage 1 und 2 unter sich aufteilen müssen, bleibt für **C nur Tag 3** übrig.
- **Ergebnis:** Zwei gültige Lösungen möglich:
  1. A=1, B=2, C=3
  2. A=2, B=1, C=3

11.3.2. Strongly Connected Component

DEFINITION: weitere Reduktion von Wertebereichen in einem Value Graph durch die **Identifikation abgeschlossener Teilmengen**.

- Identifikation abgeschlossener Teilmengen von Variablen und Werten.
  - blau markiert
- Variablen **ausserhalb** einer SCC dürfen **keine** Wert **innerhalb** dieser SCC nutzen.



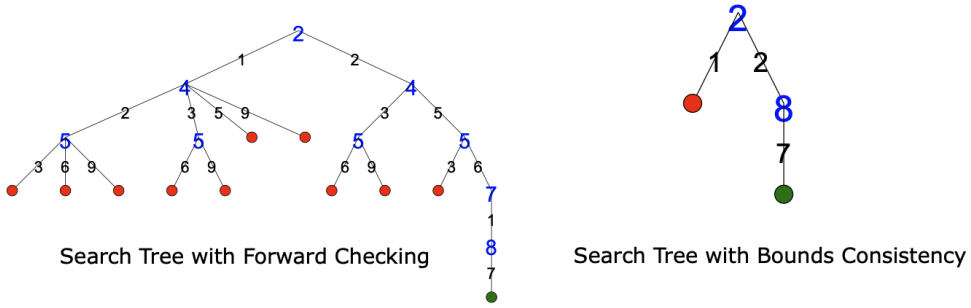
Komplexität
- **Alle Teilmengen:** $O(2^n)$, Exponentiell, zu langsam.
- **Alle Intervalle:** $O(n^2)$, Schneller, ignoriert „Löcher“ in Domains.
- **Hall Intervals:** $O(n \log n)$, Nutzt sortierte Listen & Bound-Updates; effizientester Standard.

11.3.3. Implementation Bounds Consistency

Überprüfung von Variablen-Domänen anhand ihrer **Min/Max-Werte**.
- **Identifikation:** Zähle Variablen, deren Domäne komplett in einem Werte-Intervall liegt.
- **Variablen = Werte:** Intervall für alle anderen Variablen **sperren** (Werte entfernen).
- **Variablen > Werte:** Konflikt $\rightarrow$ **keine Lösung**.
- **Variablen < Werte:** Keine Aussage möglich.

11.3.4. Search mit Bounds Consistency

- **Bounds Consistency eliminiert mehr Werte als *Forward Checking***
  - stärkere Filterung
- Nach jeder Entscheidung im Backtrack-Search wird Bounds Consistency erneut ausgeführt.
- Suchbaum wird massiv **verkleinert**
  - kürzerer Pfad zur Lösung, weniger Fehlversuche



11.3.5. Problem

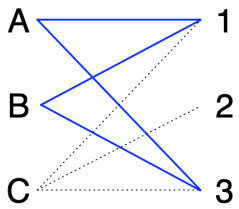
HINWEIS: Bounds Consistency (BC) arbeitet **nur mit Intervallen** (z. B. 1 bis 3).
- Ignoriert „Löcher“ in der Domäne

→ Domain Consistency

11.4. Domain Consistency

DEFINITION: Betrachtet **beliebige Teilmengen** von Werten statt nur Ober- und Untergrenzen.

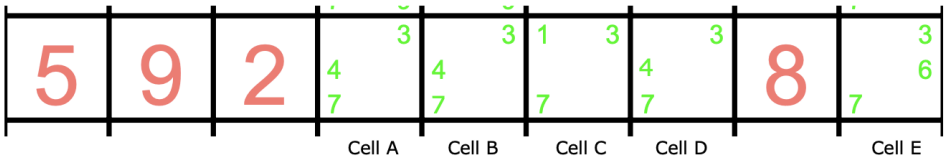
- Wenn n Variablen genau n Werte beanspruchen, sind diese Werte für alle anderen Variablen **blockiert**.
- Exklusivität von Wertekombinationen innerhalb von Teilmengen.



- $\{A, B\}$ benötigen $\{1, 3\}$
- C wird von $\{1, 3\}$ ausgeschlossen
- Resultat: $C = 2$

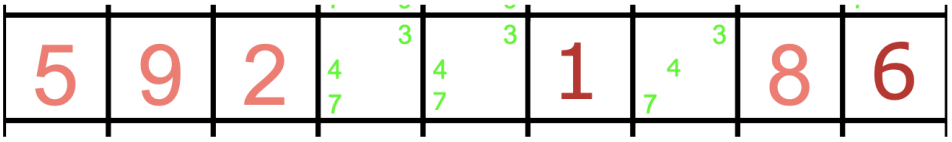
11.4.1. Beispiel Sudoku

Gruppe (A, B, D): belegen exklusiv $\{3, 4, 7\}$ (3 Zellen, 3 Werte).



Folge: Diese Werte werden in **C** und **E** gelöscht.

Ergebnis
- Zelle $C = 1$
- Zelle $E = 6$

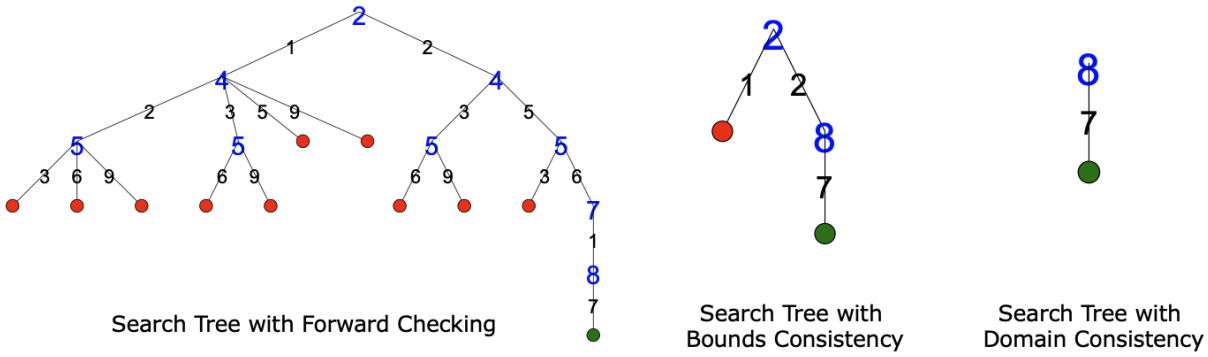


→ Findet Lösungen, die *Forward Checking* & Bounds Consistency übersehen.

11.5. Constraint Programming

- **Constraint Propagation Methods**:
  - Forward checking
  - bounds consistency
  - domain consistency
- **Constraint propagation Methoden sind incomplete**
- **Backtracking findet garantiert eine Lösung, ist aber zu langsam**
- **Synergie:** Propagation reduziert die Suchtiefe; Search findet den exakten Pfad.
- Tools wie **OR-Tools** nutzen eingebaute Algorithmen für globale Constraints.

11.6. Comparison of Search with Propagation



11.7. Propagation Levels

- **Forward Checking / Bounds Consistency** = bester Kompromiss aus Speed & Filterkraft.
- **Domain Consistency:** Oft zu teuer (**Overkill**), da Teilmengen-Prüfung komplex ist.
- **Software-Unterschiede:**
  - **OR-Tools:** Fokus auf **Bounds Consistency** + SAT-Learning.
  - **ECLiPSe:** Level manuell wählbar.
- **CP-SAT:** Nutzt dynamisches Lernen für effizientes Pruning.

12. CP - Optimization Problems

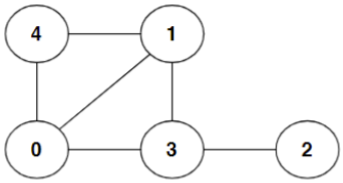
12.1. Grafenfärbung

Problem:

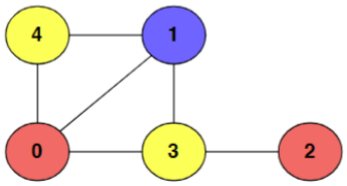
- Kanten sind eine Liste von Tuples (`graph`)
- Totale Anzahl Nodes (`size`)
- Anzahl verfügbare Farben $K > 0$ (`max_colors`)

Beispiel:

```txt
graph = (
(4, 1),
(1, 0),
(4, 0),
(1, 3),
(3, 0),
(3, 2)
)
```



Input Graph



Graph Colouring with K = 3

12.1.1. Model 1: 1D Array

- Idee: `color[n] = c <=>`
  - Node n hat Farbe c

```python
model1 = cp_model.CpModel()

# Eindimensionales Array der Entscheidungsvariablen (Farbzuweisung pro Knoten)
color = [model1.NewIntVar(0, max_colors - 1, str(i)) for i in range(size)]

# Benachbarte Knoten dürfen nicht dieselbe Farbe haben
for edge in graph:
    model1.Add(color[edge[0]] != color[edge[1]])

# Fixe Farbe für den ersten Knoten definieren, damit redundante Resultate durch blosse
Farbvertauschung vermieden werden
model1.Add(color[0] == 0)

# Modell lösen und alle gefundenen Lösungen ausgeben
solver = cp_model.CpSolver()
solver.parameters.enumerate_all_solutions = True

callback = cp_model.VarArraySolutionPrinter(color)
status = solver.SearchForAllSolutions(model1, callback)
```

12.1.2. Model 2: 2D Array

- Gleiche Lösung wie im Model 1
- Kann in ein Optimierungsproblem überführt werden
- 2D Array von binary Entscheidungsvariablen
- Matrix aus Wahrheitswerten
- **Zeilen Index**: Node `n`
- **Spalten Index**: Farbe `c`
- **Idee**: `board[n][c] = 1 <=>` Node n nimmt Farbe c

```python
model12 = cp_model.CpModel()

# 2D-Matrix aus binären Variablen (Zeilen=Knoten, Spalten=Farben)
board = [[model12.NewBoolVar(f"({_i},{_j})") for _j in range(max_colors)] for _i in
range(size)]

# Jeder Knoten erhält exakt eine Farbe (Zeilensumme == 1)
for n in range(size):
    model12.Add(sum([board[n][c] for c in range(max_colors)]) == 1)

# Benachbarte Knoten dürfen nicht dieselbe Farbe haben
for edge in graph:

# In keiner Farbspalte dürfen beide Nachbarn eine 1 haben (Summe < 2)
    for c in range(max_colors):
        model12.Add(board[edge[0]][c] + board[edge[1]][c] < 2)

# Fixe Farbe für Knoten 0 zur Symmetriebrechung (vermeidet Redundanz)
model12.Add(board[0][0] == 1)
```

Beispiel Array:

```python
# Korrekt (3 Knoten, 3 Farben): Jede Zeile exakt eine 1
board = [
    [1, 0, 0],  # Knoten 0 = Farbe 0
    [0, 1, 0],  # Knoten 1 = Farbe 1
    [0, 0, 1]   # Knoten 2 = Farbe 2
]

# Falsch: Bedingungen verletzt
board = [
    [1, 1, 0],  # FEHLER: Mehrere Farben (Zeilensumme = 2)
    [0, 0, 0],  # FEHLER: Keine Farbe (Zeilensumme = 0)
    [1, 0, 0]   # OK, aber kollidiert mit Knoten 0 bei Nachbarschaft
]
```

12.1.3. 1D Array vs. 2D Array

Merkmal	1D-Ansatz (Integer)	2D-Ansatz (Binär/Boolean)
Variablenanzahl	n (Anzahl Knoten)	$n \times k$ (Knoten $\times$ Farben)
Constraint-Typ	Alldifferent / NotEqual	Lineare Summen / Logik
Speicherverbrauch	Gering	Höher
Flexibilität	Gut für einfache Vergleiche	Sehr gut für komplexe Logik-Regeln

12.1.4. Minimale Anzahl Farben finden

- **Trial & Error ist ineffizient:** Das schrittweise Ausprobieren der Farbanzahl erfordert redundante Berechnungen durch den Solver.
- **Schlecht skalierbar:** Bei grösseren Graphen oder mehreren zu optimierenden Ressourcen führt dies zu enormen Wartezeiten.
- **Lösungsansatz:** Das Modell direkt als **Optimierungsproblem** formulieren, sodass der Solver das Minimum selbstständig und effizient sucht.

12.1.4.1. Channeling constraint

- Verwendung eines 2D Arrays
- Gleich wie model 2
- Man muss zählen wie oft eine Farbe verwendet wird
- Wenn A eine Bedingung erfüllt, wird B auf einen bestimmten Wert gesetzt
  - Ermöglicht if-then-else statements
  - Channeling constraints in OR-Tools: `model.Add(…).OnlyEnforceIf(…)`

12.1.4.2. Model 3: Optimierung

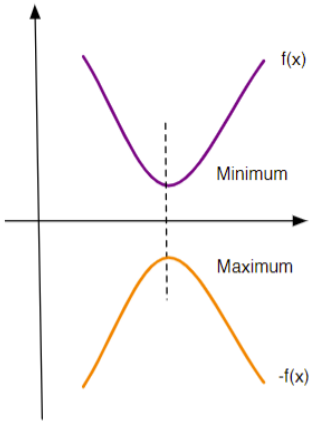
```python
# Binäre Variablen: Geben an, ob eine Farbe überhaupt verwendet wird
used = [model3.NewBoolVar(str(c)) for c in range(size)]

# Channel constraints
for c in range(len(used)):

    # Zählt, wie oft die Farbe 'c' über alle Knoten verwendet wird
    count = sum([board[n][c] for n in range(size)])

    # Wenn Farbe genutzt wird (count > 0), ist used[c] zwingend True
    model3.Add(count > 0).OnlyEnforceIf(used[c])

    # Wenn Farbe ungenutzt ist (count == 0), ist used[c] zwingend False
    model3.Add(count == 0).OnlyEnforceIf(used[c].Not())

# Zielfunktion: Minimiere die Gesamtzahl der verwendeten Farben
model3.Minimize(sum(used))
status = solver.Solve(model3)

if status == cp_model.OPTIMAL:
    # Optimale Lösung hier ausgeben ...

elif status == cp_model.INFEASIBLE:
    print("The model is over-constrained.")
```

12.1.4.3. Constraint Programming Approach to Optimization

1. Solver sucht die erste machbare Lösung x_0 (ignoriert die Optimierung)
2. Solver zählt die Farben f on $x_o \rightarrow f(x_0)$
3. Solver ergänzt das Model dynamisch mit einer zusätzlicher constraint $f(x) \leq f(x_0) - 1$
4. Loopt darüber und setzt immer eine neue constraint
5. Wenn es terminiert hat der Solver das minimum gefunden

12.2. Minimierung vs. Maximierung

- Minimierung und Maximierung ist eigentlich das Gleiche
- Minimum: $f(x)$
- Maximum: $-f(x)$



- Ein Solver muss somit nur das Minimum oder das Maximum implementieren (nicht beides)

12.3. Scheduling Problems

Analogie: Arbeiter/innen in Schichten einteilen (z.B. Spital)

12.3.1. Beispiel: Pflegepersonal

Ein Krankenhausleiter muss einen Wochenplan für vier Pflegekräfte erstellen:

- Jeder Tag ist in drei 8-Stunden-Schichten unterteilt
- Eine Pflegekraft darf nicht zwei Schichten am selben Tag arbeiten.
- Jede Pflegekraft arbeitet fünf oder sechs Tage pro Woche
- In keiner Schicht sind mehr als zwei verschiedene Pflegekräfte pro Woche im Dienst
- Wenn eine Pflegekraft an einem bestimmten Tag die Schichten 2 oder 3 arbeitet, muss sie dieselbe Schicht entweder am Vortag oder am Folgetag ebenfalls arbeiten (gilt nicht für Feiertage)

3D Array erstellen

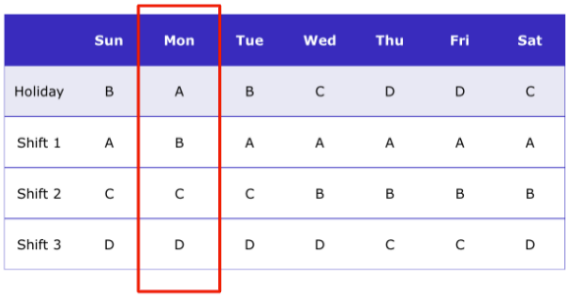
```python
# 4 Pflegekräfte (als Zahlen codiert statt Buchstaben)
num_nurses = 4

# 7 Tage pro Woche
num_days = 7

# 4 Schichten (0=Frei, 1=Frühschicht, 2=Spätschicht, 3=Nachtschicht)
num_shifts = 4

model = cp_model.CpModel()

# 3D-Matrix aus binären Variablen: SCHICHTEN x TAGE x PFLEGEKRÄFTE
# schedule[s][d][n] = 1 bedeutet: Pflegekraft n arbeitet an Tag d in Schicht s
schedule = [[[model.NewBoolVar(f"({_n},{_d},{_s})")
              for _n in range(num_nurses)]
              for _d in range(num_days)]
              for _s in range(num_shifts)]
```

12.3.1.1. Constraint 1: Jede Schicht muss genau einer Pflegekraft zugewiesen sein

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Holiday	B	A	B	C	D	D	C
Shift 1	A	B	A	A	A	A	A
Shift 2	C	C	C	B	B	B	B
Shift 3	D	D	D	D	C	C	D

```python
# Für jede Schicht ...
for s in range(num_shifts):

    # Für jeden Tag ...
    for d in range(num_days):

        # Exakt eine Pflegekraft pro Schicht und Tag zuweisen (== 1)
        model.Add(sum([schedule[s][d][n] for n in range(num_nurses)]) == 1)
```

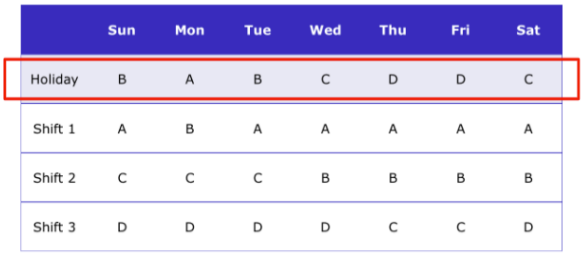
12.3.1.2. Constraint 2: Eine Pflegekraft darf nicht mehr als eine Schicht pro Tag arbeiten.



	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Holiday	B	A	B	C	D	D	C
Shift 1	A	B	A	A	A	A	A
Shift 2	C	C	C	B	B	B	B
Shift 3	D	D	D	D	C	C	D

```python
# Für jede Pflegekraft ...
for n in range(num_nurses):

    # Für jeden Tag ...
    for d in range(num_days):

        # Maximal eine Schicht pro Pflegekraft und Tag (<= 1)
        model.Add(sum([schedule[s][d][n] for s in range(num_shifts)]) <= 1)
```

12.3.1.3. Constraint 3: Jede Pflegekraft hat ein oder zwei Tag frei



	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Holiday	B	A	B	C	D	D	C
Shift 1	A	B	A	A	A	A	A
Shift 2	C	C	C	B	B	B	B
Shift 3	D	D	D	D	C	C	D

```python
# Für jede Pflegekraft ...
for n in range(num_nurses):

    # Bei fixer Pflegekraft (n) und Schicht 0 (frei): Zähle die freien Tage über alle
Tage (d)
    num_days_off = sum([schedule[0][d][n] for d in range(num_days)])

    # Mindestens 1 freie Tage erzwingen
    model.Add(num_days_off > 0)
    # Maximal 2 freie Tage erzwingen
    model.Add(num_days_off < 3)
```

12.3.1.4. Constraint 4: Jede Schicht wird als Ganzes von maximal zwei Pflegekräften übernommen

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Holiday	B	A	B	C	D	D	C
Shift 1	A	B	A	A	A	A	A
Shift 2	C	C	C	B	B	B	B
Shift 3	D	D	D	D	C	C	D

```python
# Für jede Schicht ausser Schicht 0 (frei) ...
for s in range(1, num_shifts):

    # Binäre Variablen: Arbeitet Pflegekraft in dieser Schicht?
    does_work = [model.NewBoolVar('') for _n in range(num_nurses)]

    # Für jede Pflegekraft ...
    for n in range(num_nurses):

        # Zähle die Einsätze der Pflegekraft (n) in dieser Schicht (s) über alle Tage
        var = sum([schedule[s][d][n] for d in range(num_days)])

        # Channeling: Falls Einsätze > 0, setze does_work[n] zwingend auf True
        model.Add(var > 0).OnlyEnforceIf(does_work[n])

        # Channeling: Falls Einsätze == 0, setze does_work[n] zwingend auf False
        model.Add(var == 0).OnlyEnforceIf(does_work[n].Not())

    # Maximal 2 verschiedene Pflegekräfte dürfen diese Schicht übernehmen
    model.Add(sum(does_work) <= 2)
```

12.3.1.5. Constraint 5: Pflegekräfte arbeiten an aufeinanderfolgenden Tagen in der Schicht 2 oder 3

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Holiday	B	A	B	C	D	D	C
Shift 1	A	B	A	A	A	A	A
Shift 2	C	C	C	B	B	B	B
Shift 3	D	D	D	D	C	C	D

such an assignment would be forbidden for shift 2 or 3

This one is tricky, indeed ☺

```python
# Für Schicht 2 und 3 ...
for s in [2, 3]:
    # Für jede Pflegekraft ...
    for n in range(num_nurses):
        # Für jeden Tag ...
        for d in range(num_days):

            # Arbeitet die Pflegekraft am Tag davor oder danach?
            before_or_after = model.NewBoolVar('')
            model.AddMaxEquality(before_or_after,
                [schedule[s][(d-1) % num_days][n], schedule[s][(d+1) % num_days][n]])

            # Falls an Tag d gearbeitet wird, muss auch am Tag davor oder danach
gearbeitet werden
            model.Add(before_or_after == 1).OnlyEnforceIf(schedule[s][d][n])
```

12.3.1.6. Fazit

- Kleines Problem (4 Pflegekräfte, 7 Tage, 3 Schichten) bringt viele Lösungen *to* 18'144
- Man kann somit noch viele weitere Constraints hinzufügen

Optimierungskriterien zur Lösungsreduktion

- **Gleichmässige Auslastung:** Arbeitszeit gerecht verteilen.
- **Präferenzen:** Individuelle Schichtwünsche priorisieren.
- **Kostenminimierung:** Teure Schichten oder Zuschläge reduzieren.
- **Ergonomie:** Belastende Schichtwechsel minimieren.

12.4. Subset Sum Variants and Knapsack Problems

12.4.1. Subset Sum Variants

Input:

- Items mit Gewichten
- Total Gewicht / maximum Gewicht N

Varianten:

- **Subset Sum als Constraint Problem**
  - **Gegeben**: Set von Items mit einem Gewicht und ein total Gewicht N
  - **Ziel**: Summe der ausgewählten Items **muss gleich N (Totales Gewicht)** sein
- **Subset Sum als Optimization Problem**
  - **Gegeben**: Set von Items mit einem Gewicht und ein maximum Gewicht N
  - **Ziel**: Summe der ausgewählten Items muss **maximiert sein ohne das maximal Gewicht N zu übersteigen**
- **Binary Subset Sum Problem**
  - jedes Item kann maximum einmal ausgewählt werden

Knapsack Variante

- Generalisierung von subset sum
- Items haben Werte
- **Ziel**: Summe der ausgewählten Items zu maximieren

12.4.2. Knapsack als Constraint Problem - Beispiel

- Schmuggler hat ein Rucksack mit einer Kapazität von 9 Stück
- Profit soll maximiert werden
- Welche Sachen sollen geschmuggelt werden?

Whiskey	4 units	$15 each
Perfume	3 units	$10 each
Cigarettes	2 units	$7 each

```python
names    = ["Whiskey", "Perfume", "Cigarettes"]
values   = [15, 10, 7] # Werte für Subset-Sum-Problem konstant auf 1 setzen
weights  = [4, 3, 2]
capacity = 9

# Schalter für die binäre Variante des Rucksackproblems
binary_variant = True

model = cp_model.CpModel()

if binary_variant:

    # Gegenstände können maximal einmal ausgewählt werden
    choices = [model.NewBoolVar(str(i)) for i in weights]

else:

    # Gegenstände können mehrfach ausgewählt werden
    choices = [model.NewIntVar(0, capacity, str(i)) for i in weights]

# Gesamtgewicht darf Kapazität nicht überschreiten
model.Add(cp_model.LinearExpr.WeightedSum(choices, weights) <= capacity)

# Maximiere die Summe der Werte
model.Maximize(cp_model.LinearExpr.WeightedSum(choices, values))
```

12.4.3. OR-Tools Knapsack Solver

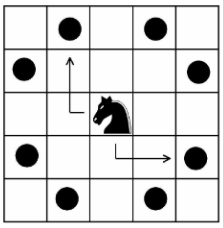
- Da das Knapsack Problem häufig in der Praxis auftaucht (z.B. bei Container Schiffen) gibt es spezialisierte solver von OR-Tools
- Das Model akzeptiert mehrere Containers und Container spezifische Gewichte
  - Es werden Gewichtsmatrixen und Liste von container Kapazitäten erwartet
- Beim Knapsack Problem sind andere Algorithmen (z.B. branch-and- bound) effizienter
  - Deshalb ist die Notation abweichend

```python
# Spezialisierter Knapsack-Solver mit Branch-and-Bound-Algorithmus
solver = pywrapknapsack_solver.KnapsackSolver(
    pywrapknapsack_solver.KnapsackSolver.
    KNAPSACK_MULTIDIMENSION_BRANCH_AND_BOUND_SOLVER, '')

# Problem initialisieren: Werte, Gewichtsmatrix und Kapazität
solver.Init(values, [weights], capacity)

# Berechnung des maximalen Gesamtwerts
best = solver.Solve()
print(f"Optimaler Zielwert: {best}")

# Ausgabe der gewählten Gegenstände
for i, _ in enumerate(values):
    print(f" - Gegenstand {i:>2} (Wert: {values[i]}, Gewicht: {weights[i]:>3}) "
          f"wurde {int(solver.BestSolutionContains(i))} mal gewählt")
```

13. AISO - 13 - Constraint Programming IV - Routing Problems

13.1. The Knight's Tour

- **Ziel**: Das Pferdchen (Springer) besucht jedes Feld auf dem Schachbrett genau einmal
- **einfache Lösung**: backtracking mit einer heuristic von Warnsdorff
  - **Warnsdorff**: Wenn man von der aktuellen Position des Springers mehrere Möglichkeiten hat, präferiert man immer an den Rand zu springen (da der Rand schwerer zu erreichen ist, als die Felder in der Mitte)
- Keine Lösung für ein 4 x 4 Schachbrett
- Viele Lösungen für ein 5 x 5 Schachbrett



13.1.1. Knight Tour Model

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

- Schachbrett durchnummerieren
- `board` : 64 x 64 array von boolean Variablen
- `board [i] [j] = True` : Springer darf von i zu j springen

Implementierung:

```python
    for each start position...
      for each end position...
        Add constraint board[start][end] == 0 when this move is illegal
```

13.1.2. Code

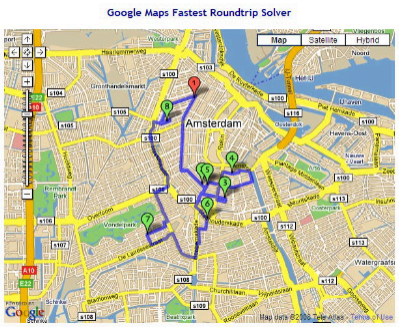
```python
model = cp_model.CpModel()

# board[i][j] = 1 bedeutet, der Springer springt von i nach j
board = [[model.NewBoolVar(f"({i},{j})") for j in range(64)] for i in range(64)]

# Von jedem Feld darf nur auf exakt ein anderes Feld gesprungen werden
for start in range(64):
    model.Add(sum([board[start][end] for end in range(64)]) == 1)

# Schachregeln: Für jede Startposition alle unmöglichen Ziele auf 0 setzen
for start in range(64):
    # Ungültige Zielpositionen ermitteln (Implementierung gemäss Jupyter Notebook)
    illegal = set(range(64)) - knight_moves(start)
    for unend in illegal:
        model.Add(board[start][unend] == 0)

# Circuit-Constraint erwartet Tripel aus: (Start, Ziel, Variable)
edges = [(start, end, board[start][end]) for start in range(64) for end in range(64)]

# Zwingt die Route, einen einzigen geschlossenen Weg (Hamiltonkreis) zu bilden
model.AddCircuit(edges)

solver = cp_model.CpSolver()
status = solver.Solve(model)
...
```

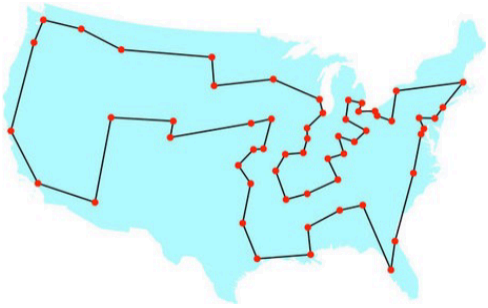
Constraints:

- von einer Position darf immer nur auf eine einzige Position gesprungen werden
- Schachregel: alle Positionen, die der Springen nicht erreichen kann auf 0 setzen
- edges als triples *to* Sequenz muss geschlossen sein

13.2. Travelling Salesperson Problem (TSP) and its Applications

- Liste von Städten (**points of interest**)
- Distanz (**costs**) zwischen allen Städten
- **Ziel**: kürzester möglicher Weg, damit man alle Städte genau einmal besucht und am Schluss zum Start zurückkommt





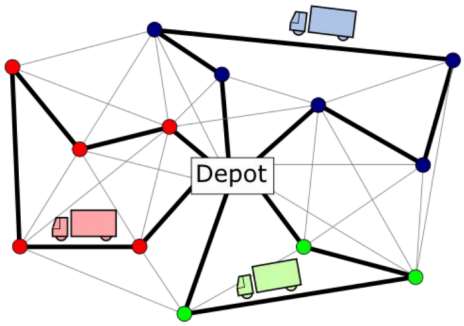
13.2.1. Node Routing vs. Arc Routing

- **Knoten (node) routing Problem**: muss jede Stadt einmal besuchen
  - Travelling Salesperson Problem
  - Produktionslinien Optimierung
  - Roboterarmwege verkürzen
- **Kanten (arc) Routing**: jede Strasse genau einmal befahren
  - Google Maps Auto

13.2.2. OR-Tools Routing Library

- OR-Tool stellt **spezielle API** für routing Probleme zur Verfügung
  - da es nicht so einfach ist, diese Problem mit herkömmlichen constraints zu lösen
  - die API ist ein layer über den ursprünglichen constraint programming APIs
- **depot**: Start und Ende
- Es können Wege mit **mehrere Fahrzeugen** geplant werden
  - solver kann definieren, wie viele Fahrzeuge es braucht um am effizientesten zu sein
- Es können weitere constraints definiert werden
  - pick-up und delivery
  - Fahrzeugkapazität
  - ...

Beispiel: 1 Depot und 3 Fahrzeuge



13.2.3. Code

- `RoutingIndexManager`: Verknüpft die Knoten des Graphen (Städte) mit den internen Indizes des Solvers.
- `RegisterTransitCallback`: Erstellt eine Funktion, die dem Solver sagt, wie hoch die „Kosten“ (Distanz) zwischen zwei Punkten sind.
- `SetArcCostEvaluatorOfAllVehicles`: Setzt diese Kostenfunktion als das Ziel fest, das minimiert werden soll.
- `AddCircuit` **(implizit):** Der Routing-Solver stellt sicher, dass alle Städte besucht werden und das Fahrzeug zum Depot zurückkehrt.

```python
# Index des Depots (Start- und Endpunkt)
depot = 0

# Anzahl der verfügbaren Fahrzeuge
vehicles = 1

# Städtenamen für die Ausgabe
names = ["New York", "Los Angeles", "Chicago", "Salt Lake City"]


# Distanz-Matrix
distances = [
    [0, 2451, 713, 1018],      # New York
```

```
    [2451, 0, 1745, 1524],     # Los Angeles
    [713, 1745, 0, 355],       # Chicago
    [1018, 1524, 355, 0]       # Salt Lake City
]

# Übersetzt zwischen den echten Knoten (Städten) und internen Solver-Indizes
manager = pywrapcp.RoutingIndexManager(len(names), vehicles, depot)

# Funktion zur Distanzberechnung zwischen zwei Knoten
def distance_func(ind_from, ind_to):
    return distances[manager.IndexToNode(ind_from)][manager.IndexToNode(ind_to)]

# Routing-Modell erstellen
routing = pywrapcp.RoutingModel(manager)

# Distanzfunktion als Callback für den Solver registrieren
callback = routing.RegisterTransitCallback(distance_func)

# Setzt die Distanz als Kosten, die vom Solver minimiert werden sollen
routing.SetArcCostEvaluatorOfAllVehicles(callback)

# Modell lösen
solution = routing.Solve()

# Hilfsfunktion, um das Resultat übersichtlich auszugeben
def print_solution(vehicles, cities, manager, routing, solution):
    sum_route_distance = 0
    for vehicle_id in range(vehicles):
        # Startpunkt des aktuellen Fahrzeugs abrufen
        index = routing.Start(vehicle_id)
        plan_output = f"Route for vehicle {vehicle_id}:\n"
        route_distance = 0
        # Solange das Ende der Route noch nicht erreicht ist
        while not routing.IsEnd(index):
            plan_output += f"{cities[manager.IndexToNode(index)]} > "
            previous_index = index
            # Den nächsten Knoten in der gefundenen Route ermitteln
            index = solution.Value(routing.NextVar(index))
            # Distanz für diesen Streckenabschnitt addieren
            route_distance += routing.GetArcCostForVehicle(previous_index, index,
vehicle_id)
        plan_output += f"{cities[manager.IndexToNode(index)]}\n"
        plan_output += f"Total distance of vehicle {vehicle_id}: {route_distance}
miles\n"
        print(plan_output)
        sum_route_distance += route_distance

    print(f"Total distance over all vehicles: {sum_route_distance} miles")

if solution:
    print_solution(vehicles, names, manager, routing, solution)
else:
    print("No solution found.")
```

13.2.4. Kosten maximieren

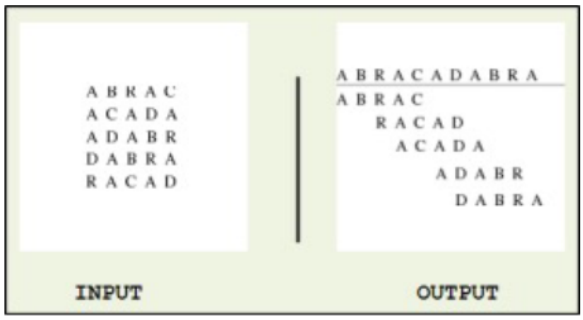
- OR-Tools für TSP kann nur eine Minimierung der Kosten berechnen
- um die Kosten zu maximieren, müssen die Distanzen negiert erfasst werden
- **Beispiele**
  - Taxifahrt: möglichst langer Weg um mehr Geld einzunehmen
  - DNA Sequencing

13.2.5. DNA Sequencing

Der Vorgang der Bestimmung der genauen Reihenfolge der Nukleotide innerhalb eines DNA-Moleküls, d.h. der Reihenfolge der vier Basen (Adenin, Guanin, Cytosin, Thymin) in einem DNA-Strang.

Computational Challenge: Shortest Superstring Problem

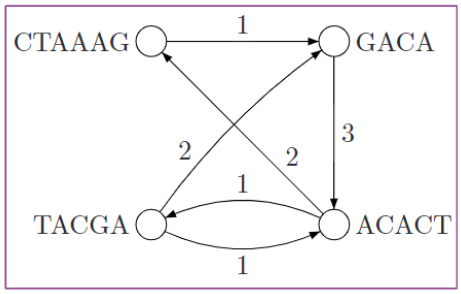
- finde für eine Menge von input strings die kürzeste superstring, die alle input strings als substring enthält



13.2.5.1. DNA Sequencing als TSP

- Input strings
  - `TACGA`
  - `ACACT`
  - `CTAAAG`
  - `GACA`
- Überlappungen in einem Graph definieren
  - 1 = ein Zeichen überlappt
  - 2 = zwei Zeichen überlappen
  - usw.
- **Pfeil**: upper *to* lower: definiert wie die Strings positioniert sind

Upper: `ACACT` **Lower**: `CTAAAG`



- Länge eines substrings = 20 - `#overlaps`
  - 20 = Anzahl Zeichen
- den kleinsten Superstring, also die grösste Überschneidung finden
- **ist ein TSP aber ohne Rückkehr zum Startpunkt**

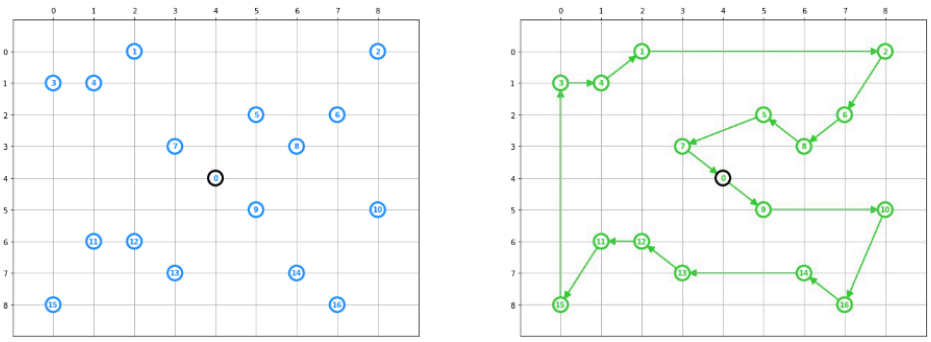
```txt
TACGA - - - - - - - - -
- - - - ACACT - - - - -
- - - - - - CTAAAG - - -
- - - - - - - - - GACA
```

`#overlaps` $= 1 + 2 + 1 = 4$

```txt
TACGA - - - - - - - - -
- - - - GACA - - - - - -
- - - - - ACACT - - - -
- - - - - - - CTAAAG
```

`#overlaps` $= 2 + 3 + 2 = 7$

13.2.6. Typisches TSP Problem



```txt
// Coordinates as Input: nodes = [(4, 4), (2, 0), (8, 0), (0, 1), (1, 1), (5, 2), (7, 2),
… (6, 7), (0, 8), (7, 8)])
```

- auch mit mehren möglichen Fahrzeugen, zeigt der solver eine Lösung mit nur einem Fahrzeug

13.2.7. Dimensions - Routing Problems mit zusätzlichen Constraints

Ein Routing Problem kann auch weitere Constraints haben, zum Beispiel:

- **Maximale** Anzahl an Orten, die jedes Fahrzeug anfahren kann
- Die **Gesamtnachfrage** (z. B. Paketgrössen) der Standorte auf der Route eines Fahrzeugs darf dessen **Kapazität** nicht überschreiten.
- Die Dauer (d. h. die **Zeit**) jeder Route darf eine festgelegte Grenze nicht überschreiten.
- Jeder Standort muss innerhalb eines **Zeitfensters** `[ai, bi]` bedient werden, und Wartezeiten sind zulässig
- Abholung am Standort X, Zustellung am Standort Y (**precedence constraints**)

13.2.7.1. TSP Dimensions

```python
# Parameter: Schrittweite, Maximalkapazität, Start bei 0, Name
def AddConstantDimension(self, value, capacity, fix_start_cumul_to_zero, name)
```

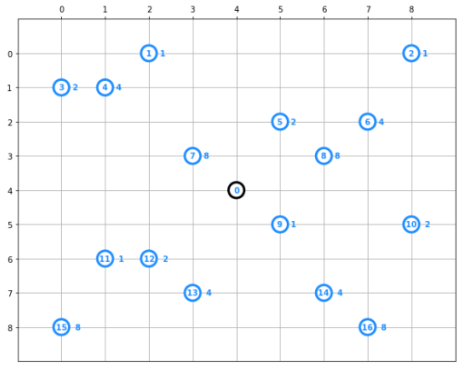
```python
# Zähler-Dimension "count" erstellen (+1 pro Stadt, limitiert auf Gesamtanzahl)
routing.AddConstantDimension(1, len(cities), True, "count")

# Referenz auf die Dimension für weitere Constraints holen
count = routing.GetDimensionOrDie("count")
```

- jede Route wird eine dimension variable zugeordnet
- Variablen werden mit 0 initialisiert
- Erhöhung um 1 bei jedem Schritt entlang der Route
- **Note**: Bewegung vom Start zum Ende (depot) zählt als einen Schritt
  - das Modell hat intern ein fiktives Zieldepot mit $n + 1$

```python
# 1/ Min. 1 Stadt pro Fahrzeug besuchen (Zugzähler > 1)
for i in range(vehicles):
    routing.solver().Add(count.CumulVar(routing.End(i)) > 1)

# 2/ Max. 5 Städte pro Fahrzeug (Zugzähler < 7 inkl. Depot)
for i in range(vehicles):
    routing.solver().Add(count.CumulVar(routing.End(i)) < 7)

# 3/ Städte 2 und 4 von unterschiedlichen Fahrzeugen anfahren
routing.solver().Add(routing.VehicleVar(2) != routing.VehicleVar(4))

# 4/ Stadt 1 zeitlich vor Stadt 3 besuchen
routing.solver().Add(count.CumulVar(1) < count.CumulVar(3))

# 5/ Stadt 6 zwingend direkt nach Stadt 7 besuchen
routing.solver().Add(count.CumulVar(7) + 1 == count.CumulVar(6))
```

13.2.7.2. Beispiel

- Jede Location definiert eine Nachfrage
- 4 Fahrzeuge, mit einer Kapazität von je 15 units



```python
vehicles   = 4
capacities = [15, 15, 15, 15] # max. Ladung pro Fahrzeug
demands    = [0, 1, 1, 2, 4, 2, 4, 8, 8, 1, 2, 1, 2, 4, 4, 8, 8] # Bedarf pro Stadt

# Funktion gibt den Bedarf eines bestimmten Knotens zurück
def demand(a):
    return demands[a]

# Bedarfsfunktion als Callback registrieren (nur vom Knoten abhängig)
d_callback = routing.RegisterUnaryTransitCallback(demand)

# Dimension für Fahrzeugkapazitäten erstellen
routing.AddDimensionWithVehicleCapacity(
    d_callback,
    0, # kein Puffer (slack)
    capacities, # Liste der maximalen Fahrzeugkapazitäten
    True, # Zähler startet bei 0
    "capacity"
)

# Modell lösen
solution = routing.Solve()

if solution:
    print_solution(vehicles, names, manager, routing, solution)
else:
    print("No solution found.")
```

- **`demands`** : Jedem Knoten (Stadt) wird eine Last zugewiesen, die abgeholt oder geliefert werden muss.
- **`RegisterUnaryTransitCallback`** : Registriert eine Funktion, die nur vom aktuellen Knoten abhängt (im Gegensatz zur Distanz, die von zwei Knoten abhängt).
- **`AddDimensionWithVehicleCapacity`** : Fügt dem Modell eine „Dimension“ hinzu, die sicherstellt, dass die Summe der Lasten auf einer Route die Kapazität des jeweiligen Fahrzeugs (`capacities`) nicht überschreitet.

13.2.7.2.1. Lösung

Totale Distanz: 64m mit Manhattan distance