Summary
Foundations of Artifical Intelligence
FS 2016

Marcel Neidinger
m.neidinger@unibas.ch

June 22, 2016

This is an unofficial summary for the *Foundations of Artifical Intelligence* lecture, hold by Prof. Helmert and Dr. Wehrle in the spring term 2016. The summary covers all topics of the 2016 lecture as well as the additional part on Monte Carlo Tree Search algorithms held by Dr. Keller.

As the lecture is held in English, this summary is in English as well. If you find any spelling errors, please feel free to mail me.

If you find any errors, feel free to mail me.

I want to specially thank Florian Spiess for proof-reading the summary as well as pointing out some gramatical errors.

Marcel Neidinger,
Basel, 14. June 2016

Contents

1 Introduction

There is no common definition, of what artificial intelligence really is. The four typical categories include

Acting Humanly A AI system can act as if it is a human($\rightsquigarrow$ *Turing Test*). The scientifical usefulness of this test is questionable, but the idea is wide spreed in pop media.

Thinking humanly Model a computer to replicate the human brain. Asks what cognitive abilities are necessary for intelligent behavior. Today, this is a seperate research area from AI ($\rightsquigarrow$ *cognitive science*)

Thinking rationally Thinking according to the laws of *logic*. This approach has the problem, that not all intelligent behavior is logical(relevance, uncertainty, contradictions)

Acting rationally Maximize utility, given available informations. Best suited approach for scientific methods.

Before AI, a lot of philosophers, mathematicians and linguists asked questions similair to AI, leading to things like the turing test. In the early years, approaches where systems tried to think humanly were taken. In the 1960 and 1970s *knowledge-based systems* were developed and advanced into expert systems and other AI approaches. However, these were less successful than hoped.

In the 1990s probabilistic methods and agent-oriented approaches combinied with a better understanding of theoretical complexity and the formalization of AI techniques lead to a more suffisticated AI world. Today, AI systems enter mainstream through

- Chess or pokerplaying systems
- Self-driving cars

1.1 Rational Agents

AI systems are used in very different tasks. To capture this diversity in a systematic framework **rational agents** in their **environments** are used.

Here, *agent programs* running on physical architecture compute *agent functions* maping sequences of observations to actions

$$f : \mathcal{P}^+ \to A \tag{1.1.1}$$

These happen in a *domain*. An example would be a vacuum cleaning roboter with two rooms A and B. here observations might be a tuple of location and cleannes ($\langle a, clean\rangle, \langle a, dirty\rangle, \dots$ and actions might be $left, right, suck, wait$

A *reflexive agent* is a very restricted and simple model that only relies on the last observation to compute the next action. The correspond to Mealy automatons with only 1 state.

1.2 Rationality

To answer the question, what the right action/agent function is, one could state that the robot has to behave *rationally*.

Rational Behaviour evaluates behaviour of agents with regards to a performance measure(e.g. utility or cost)

Perfect rationality means, that an agent always maximizes utility given available informations and taking into account expected value of future performance.

This however does not mean, that the agent is omniscience, as only incomplete information are available. Also, there is no perfect prediction of future due to uncertain behaviour in the environment. This means, that perfect rationality is rarely achievable due to limited computational power.

1.3 Environemnt and Problem Solving Methods

An AI problem always consits of three parts

Performance Measure A way to quantify an agents utility based on its action and environment

Agent model Which actions an agent can make and what observations the agent can access.

Environemnt Which aspects of the world are relevant to the agent and the reactions the world is doing, based on an agents actions.

An environment is classified by its properties.

- **static vs. dynamic** Is the environment state changing, while the agent is contemplating its next action? (Does the world wait for the agent to make its turn)
- **deterministic vs. non-deterministic vs. stochastic** Is the next state of an environment fully determined by the current state and the agents move(deterministic)? If not is the next step affected by randomness(stochastic)?
- **completely vs. partially vs not observable** Do the agents observations fully determine the state of the environment(completely)? If not, can the agent at least determine some aspects of the state of the environment(partially)
- **discrete vs. continuous** Is the environments state given by discrete or by continous parameters?
- **single vs. multi-agent** Must other agents be considered? If so, do they behave cooperatively, selfishly or are they adversaries?

A problem can be solved by

1. implementing a problem-specific solution
2. Creating a problem description and use a general algorithm(solver)
3. Learn aspects fo algorithm from experience

Combinations of these methods are possible. For the second approach, three things are neceassary

- A **model** to classify, define and understand the problems in terms of possible/optimal solutions and problem instance
- A **language** to represent the problem instance in
- A **algorithm** to find a solution

The key is, to implicitly describe complex models in a declerative language. This allows for a compact model description as well as an algorithmic exploitation of the problems structure.

2 State Space Search

(Classical) state-space search problems are amongst the simplest and most important classes of AI problems. The objective of an agent is to reach a goal state from an initial state by applying a sequence of actions. A common example is the 15-Puzzle where one has to swap tiles on the board in order to get the numbers in line. Other examples include scheduling of events, flights and manufacturing tasks, query optimization in databases or behavior of NPCs in computer games.

Classically, state space search problems are

- single-agent
- fully observable
- static
- discrete (Only a finite number of states/actions
- deterministic effect of actions on the state

2.1 Formalization

To study search problems, a formal model is needed. State spaces are **(labeled, directed) graphs** where **paths** to the goal state represent solutions, while the shortest path corresponds to the optimal solution. Formally, this is given by a 6-tuple

$$S = \langle S, A, cost, T, s_0, S_* \rangle \tag{2.1.1}$$

Here

S is a finite set of states

A is a finite set of actions

$cost : A \to \mathcal{R}_0^+$ returns an actions cost

$T \subseteq S \times A \times S$ A transition relation. We call the triple $\langle s, a, s' \rangle \in T$ a (state) transition. This means that we use action a to go from state s to s'. We can write $s \xrightarrow{a} s'$ or $s \to s'$ if the action does not matter. This relation is deterministic, so it is not possible to have both $s \xrightarrow{a} s_1$ and $s \xrightarrow{a} s_2$ with $s_1 \neq s_2$

$s_0 \in S$ the initial state

$S_* \subseteq S$ set of goal states

As state-spaces are often depicted as directed graphs, we use common terminology from graph theory. With $S = \langle S, A, cost, T, s_0, S_* \rangle$ and $s, s' \in S$ with $S \to s'$. We call

- s a predecessor of s'
- s' is a successor of s

If $s \xrightarrow{a} s'$, then a is applicable in s.

With $s^{(0)}, \ldots, s^{(n)} \in S$ and $\pi_1, \ldots, \pi_n \in A$ such that $s^{(0)} \xrightarrow{\pi_1} s^{(1)}, \ldots, s^{(n-1)} \xrightarrow{\pi_n} s^{(n)}$ we call

- $\pi = \langle \pi_1, \ldots, \pi_n \rangle$ a path from $s^{(0)}$ to $s^{(n)}$
- The length of this path π $|\pi| = n$
- The cost of π : $cost(\pi) = \sum_{i=1}^{n} cost(\pi_i)$

A state s is called **reachable** if a path from s_0 to s exists. Every path from $s \in S$ to $s_* \in S_*$ are **solutions for/from** s. A solution from s_0 is called **solution for** S. Optimal solutions have minimal cost among all solutions.

State-space search is the algorithmic problem of finding solutions in state spaces or proving that no solution exists.

2.2 Representation of State-Space Searches

Practically interesting state spaces can become huge($10^{10}, 10^{20}, 10^{100}$ states)Three main options for representing such state spaces exist.

1. **Explicit (directed) graphs** where vertices are states and directed arcs describe transitions. These can be represented as an adjacency matrix. For small state spaces, solutions could be computed in $O(|S| \log |S| + |T|)$ using Dijkstra's algorithm. For large state spaces the required space is too much.
2. **declarative representation** of state spaces allow a compact description with algorithms operating directly on description. This allows automatic reasoning about the problem itself such as reformulation, simplification or abstraction.
3. **Black Box approach** with an abstract interface for state spaces that implements the following methods
   - `init()` generates initial state
   - `is_goal(s)` tests if s is a goal state
   - `succ(s)` generate applicable actions and successors of s(sequence of pairs $\langle a, s' \rangle$ with $s \xrightarrow{a} s'$
   - `cost(a)` gives the cost of action a

2.3 Examples of State-Spaces

2.4 Basic Algorithms

This section introduces some basic algorithms and suitable data structures for search algorithms.

The general idea of a search algorithm ist to start with an **initial state** repetedly expand a state by generating its successor and stop when either a goal state has been reached, or all reachable states have been considered (no solution).

2.4.1 Data Structures for Search

Three basic data structures are needed for search.

search node Stores a state that has been reached, how it was reached and at which cost. Collectively, they form the search-tree. A search node n has the following attributes

- `n.state` The state associated with this node
- `n.parent` The nodes parent. **none** for the root
- `n.action` The action leading from **n.parent** to this node
- `n.path_cost` The cost of the path from initial state to this state.

Such a datastructure yields three basic operations `make_root_node(), make_node()` and `extract_path`.

```
function MAKE_ROOT_NODE()
    node := newSearchNode
    node.state := init()
    node.parent := none
    node.action := none
    node.path_cost := 0
    return node
```

```
function MAKE_NODE(parent,action, state)
    node := newSearchNode
    node.state := state
    node.parent := parent
    node.action := action
    node.path_cost := parent.path_cost + cost(action)
    return node
```

```
function EXTRACT_PATH(node)
    path := ⟨⟩
    while node.parent ≠ none do
     path.append(node.action) node := node.parent
    path.reverse()
```

open list also called frontir organizes the leaves of a search tree. It determines and removes the next node to expand as well as inserts a new node that is a candidate for expansion. Despite it's name, it should never be implemented as a simple list. The open list should implement

- `open.is_empty()` to test if the open list is empty
- `open.pop()` removes and returns the next node to expand
- `open.insert(`n`)` inserts node n into the open list

The implementation of `pop()` highly depends on the search algorithms decision which node to return next. The choice of a suitable data structure depends on this.

closed list remembers expanded states to avoid duplicated expansion of the same state. It shall efficiently insert a node whose state is not yet in the closed list and test if a node with a given state is in the closed list. If so, return this node.

- `closed.insert(n)` insert node n into *closed*. If a node with this state already exisits in the list, replace it.
- `closed.lookup(s)` test if a node with state s exists in the closed list. Return it if so, and return **none** otherwise

Some implementations support the modification of an open list entry, if a shorter path to this state is found. Instead of this complicated implementation, one can also eliminate duplicates later. For closed list, hash tables with states as keys can serve as efficient implementations

2.4.2 Tree-Search

In Tree Search all possible paths are organized in a tree, the *search tree*. Search nodes correspond one-to-one to paths from initial state. Duplicates are possible, due to the existence of multiple nodes with identical state. The **depth** of search tree is unbounded.

A generic search tree algorithm looks like this

This is an generic template for tree search algorithms. Concrete implementations often differ in details for efficiency reasons.

2.4.3 Graph-Search

In difference to tree search, graph search

- Recognizes duplicates by keeping only one search node when a state is reachable by multiple paths.
- search nodes correspond one-to-one to **reachable states**(in difference to search-trees where they correspond to **paths from initial state**)
- Search tree is bounded, as number of states is finite.

Some algorithms might not eliminiate duplicates in order to e.g. find an optimal solution later.

```
function GENERIC_GRAPH_SEARCH
    open := newopenList
    open.insert(make_root_node())
    closed := newClosedList
   while not open.is_empty() do
    n := open.pop()
      if closed.lookup(n.state) == none then
    closed.insert(n)
        if is_goal(n.state) then
    return extract_path(n)
        for all ⟨a, s'⟩ ∈ succ(n.state) do
    n' := make_node(n,a,s')
    open.insert(n')
   return unsolvable
```

2.4.4 Evaluating Search Algorithms

In order to evaluate search algorithms, four criterias are important

Completeness Is the algorithm guaranteed to find a solution if one exists (semi-complete) and does it terminate if no solution exists(complete)

Optimality Are solutions, returned by the algorithm guranteed to be optimal.

Time complaxity How much time does it take (usually in worst case) for the algorithm to terminate. This is usually measured in generated nodes, given as a function of

- **branching factor** b the maximum number of successors for a state
- **search depth** d The length of the longest path in the generated search tree

Space Complexity How much memory does the algorithm need, usually measured in (concurrently) stored nodes. Usually a function of branching factor and search depth.

2.4.5 Breadth-first Search

Blind search algorithms(such as Breadth-first Search(BFS)) use no information about state spaces. They are also called **uninformed** search algorithms.

A breadth-first search expands nodes in order of generation(FIFO). An open list can then be implemented as a linked list or deque.

SKETCH

The algorithm searches the state space **layer by layer** and finds shallowest goal state first.

```
function BFS-Tree
    open := newDeque
    open.push_back(make_root_node())
   while not open.is_empty() do
    n := open.pop_front()
    ToDo: Algorithm
```

This version allready implements the fact that the first generated goal node is always the first expanded goal node(due to the FiFo principle). Therefor it is more efficient to test for goal states at generation rather than upon expanding.

The BFS tree search is **semi-complete** since it might generate new nodes endlessly due to no duplicate eliminiation.

```
function BFS-Graph
```

BFS-Graph is **complete** since there is only a finite number of states.

For both variants of BFS, the time complexity is

$$1 + b + b^2 + b^3 + \cdots + b^d = O(b^d) \tag{2.4.1}$$

Since we're adding b^n additional search nodes for the n-th layer. This is also the same for memory complexity.

Also, both variants of BFS are **optimal** if all actions have the same cost but not in general(?).

Generally, BFS-Graph is much more efficient if there are many duplicates and is therfor preferable. BFS-Tree is better if it is known, that there is a negligible number of dupliacates in the state space.

2.4.6 Uniform Cost Search

BFS is only optimal, if all actions have the same cost. The idea of **Uniform cost search** is, to always expand a node with **minimal path cost**(that is n.path_cost() or $g(n)$), so in **ascending path cost order**. To implement this, use a priority queue(min-heap), ordered by $g(n)$ for the open list.

ALGORITHM

Early goal tests/eraly updates of the closed list are not a good idea, because ?.

The algorithm is complete since it will terminate after all nodes are in the closed list and is also optimal, as it will find the action with the least costs using the Min-Heap

For space and timecomplexity, no simple and accurate bounds can be given, as they depend on the distribution of action cost. For $\epsilon := \min_{a \in A}\{cost(a)\}$ and c^* beeing the optimal solution cost, the time(and space) complexity is

$$O\left(b^{\frac{c^*}{\epsilon}+1}\right) \tag{2.4.2}$$

at most.

2.4.7 Depth-first search

While BFS expands nodes in order of generation(FiFo), **Depth-first search** expands nodes in opposite order of generation(LiFo). This means, that the deepest node is expanded first. The open list is implemented as **a stack**.

SKETCH

DFS is almost always implemented as a tree search and is neither complete nor semi-complete or optimal. It is only complete if the state space is acyclic, so if the state space is a directed tree.

---

function DEPTH-FIRST SEARCH (NON-RECURSIVE( )

It is easier, to implement this a s recurisve algorithm. The CPU stack is serving as an implicit open list and no search node data structure is needed.

---

function DEPTH_FIRST_SEARCH(s)
function DEPTH-FIRST SEARCH (RECURSIVE)
  return depth_first_search(init())

For state spaces with paths of length m, DFS might generate $O(b^m)$ nodes even if shorter solutions exist.

The best case solution (with path length l) can be found with $O(b\,l)$ generated notes, ?. This can be improved using incremental successor generation to $O(l)$

The algorithm only needs to store all nodes along(node on path and their children) so the space complexity is $O(b\,m)$ where m is the maximal search depth.

2.4.8 Iterative Deepening Depth-first Search

The idea of iterative deepening depth-first search is to performe a sequence of depth-limited searchs with increasing depth limit. The algorithm searches over an expanding number of layers in the tree.

---

function ITERATIVE DEEPENING DFS
  for depth_limit $\in \{0, 1, 2, \dots\}$ **do**
   solution := depth_limited_search(init(), depth_limit)
    if solution $\neq=$ **none then**
   return solution

This combines the advantages of breadth-first and depth-first search.

- **semi-complete** (almost like BFS)
- **optimal** if all actions have same cost (like BFS)
- only needs to **store nodes along the path** which leads to a space complexity of $O(b\,d)$ where d is the minimal solution length.

For the time complexity of iterative deepening with d beeing the minimal solution length we get

$$(d+1) + d\,b + (d-1)\,b^2 + (d-2)\,b^3 + \cdots + 1\,b^d = O(b^d) \tag{2.4.3}$$

?

2.4.9 Summary of blind search algorithms

	breadth first	uniform cost	depth first	depth limited	iterative deepening
complete	yes*	yes	no	no	semi
optimal	yes**	yes	no	no	yes**
time	$O(b^d)$	$O(b^{\frac{c^*}{\epsilon}+1})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
space	$O(b^d)$	$O(b^{\frac{c^*}{\epsilon}+1})$	$O(b\,m)$	$O(b\,l)$	$O(b\,d)$

2.5 Heuristic Algorithms

So far, all algorithms were blind. This means that they use no problem specific knowledge to find a more scalable solution. The idea of a heuristic is to provide a **problem-specific** criteria that distinguishes good and bad states to expand.

A heuristic function h in a state space S is defined as

$$h : S \to \mathbb{R}_0^+ \cup \{\infty\} \tag{2.5.1}$$

So it maps each state to a non-negative number of infinity. The idea is that $h(s)$ **estimates** the distance (= cost of cheapest path) from s to it's closest goal state. Heuristics can be arbitrary functions. However, the closer h is to true goal distance, the more efficient is the search using h.

Good heuristics enable the algorithm to focuse its search on promising states rather than evaluating all states.

2.5.1 Properties of Heuristics

A perfect heuristic for a givne state space $\mathcal{S}$, written as h^* maps each state $s \in S$ to the cost of an optimal solution for s and $h(s) = \infty$ if no solution exists from this state.

A heuristic h for the same state space and same states is

Safe if $h^*(s) = \infty \forall s \in S$ where $h(s) = \infty$. This means that the heuristics correctly predicts dead ends.

Goal-aware if $h(s) = 0$ for all goal states s. This means that the heuristic identifies goal states correctly.

admissible If $h(s) \leq h^*(s) \forall s \in S$. This means that h never overestimates a state. The heuristic may underestimate states.

consistent if $h(s) \leq cost(a) + h(s') \forall s \xrightarrow{a} s'$ This means that the heuristic tracks transitions and their action cost correctly(Or at least doesn't over-estimate them).

If a heuristic h is ...

- ...**admissible** than h is also **safe** and **goal aware**. Why?
- ...**goal aware** + **consistent** than h is also **admissible**

How to show that a heuristic has all 4 properties?

2.5.2 Best-first search algorithms

A best-first search algorithm is a heuristic search algorithm that evaluates search nodes with an evaulation function f and always expands a node n with minimal $f(n)$ value.

The implementation is essentially the same as in uniform cost search, where the different choices for f yield different algorithms.

- **Greedy best-first** where $f(n) = h(n.state)$ so only the heuristic counts.
- **A*** where $f(n) = g(n) + h(n.state)$ so combination of path cost and heuristic count.
- **Weighted A*** where $f(n) = g(n) + w \cdot h(n.state), w \in \mathbb{R}_0^+$ so we interpolate between greedy best-first search and A*

An algorithmic implementation of this would be

```
function BEST-FIRST SEARCH WITHOUT REPOPENING
    open := newMinHeap ordered by ⟨f, h⟩
  if h(init()) < ∞ then
   open.insert(make_root_node())
  closed := newHashSet
  while not open.is_empty() do
   n := open.pop_min()
     if n.state ∉ closed then
   closed.insert(n)
         if is_goal(n.state) then
   return extract_path(n)
         for all ⟨a, s'⟩ ∈ succ(n.state) do
             if h(s') < ∞ then
   n' := make_node(n,a,s')
   open.insert(n')
  return unsolvable
```

For a **safe** heuristic h, this algorithm is complete, while the optimality depends on f.

Since uniform cost search expands nodes in order of increasing g values, this guarantees that cheapest path to state of a node has been found when the node is expanded. with an arbitrary evaluation function f this does not hold in general. This means, that we want to expand duplicate nodes when a cheaper path to their states are found. This is called **reopening**.

ALGORITHM REOPENING

2.5.3 Greedy best-first search

Greedy best-first search **only considers the heuristic** so

$$f(n) = h(n.state) \tag{2.5.2}$$

With a safe heursitic, this algorithm is **clompete** but solutions can be arbitarirly bad. The algorithm is, in practice, very fast(one of the fastes in practice). Also, the behaviour is not affected by monotonic transformations of h. WHY INTERESTING

2.5.4 A*

A* combines greedy best-first search with uniform cost search

$$f(n) = g(n) + h(n.state) \tag{2.5.3}$$

It is a trade-off between path cost($g(n)$) and proximity to goal($h(n.state)$). So $f(n)$ **estimates the overall cost of cheapest solution from initial state via n to goal state**.

Again, the algorithm is **complete** with a **safe** heuristic and

- **with reopening: optimal** with **admissible** heuristic
- **without reopening optimal** with **admissible and consistent** heuristics.

Common implementation problems of A* are

- **no reopening** with inconsistent heuristic
- **duplicate test too early** on generation of search node
- **goal test too eraly** on generation fo search nodes

2.5.5 Weighted A*

A* with a more heavily weighted heuristic

$$f(n) = g(n) + w \cdot h(n.state) \tag{2.5.4}$$

where $w \in \mathbb{R}_0^+ \geq 1$. A weight of less then 1 is conceivable, but not a good idea WHY

The weighting parameter w controls **greediness** of search.

$w = 0$ like uniform cost search

$w = 1$ like A*

$w \to \infty$ like greedy best-first search

With $w \geq 1$ the properties are analogous to A*

- **h admissible** So the **found** solution is guranteed to be at most w times as expensive as optimum when reopening is used
- **h admissible** and **consistent** So the **found** solution is guaranteed to be at most w times as expensive as optimum. no reopening is needed.

2.5.6 Optimality proof for A* with reopening

Some definitions needed for this are

solvable A state s of state space is **solvable** if $h*(s) < \infty$

Cost of optimal cheapest path $g^*(s)$ In a state space we write $g^*(s)$ for the cost of optimal path from s_0 to s(∞ if unreachable)

- $g^*(n.state) \leq g(n)$ since $g^*(n.state)$ describes the best path.

Settled A state s is settled at a given point during the execution of A* (with or without) reopening if s is included in *distances* an $distances[s] = g^*(s)$. This means that we have found the optimal path to a state.

PROOF Optimality (S-18 + S-19)

2.5.7 IDA*

The main drawback of best-first graph search is space complexity. To solve this problem, apply concepts of iterative deepening depth-first search. This leads to **Iterative-Deepening A***

Bounded depth-first search with an increasing bound. Instead of bounding the depth we **bound** f.This implements a **Tree search** unlike the previous best-first search algorithms.

ALGORITHM

Some important properties of depth-first search and A* are

- **semi-complete** if h is safe and $cost(a) > 0$ for all actions a
- **optimal** if h is admissible
- **space complexity** $O(l\,b)$ where l is length of the longest path and b the branching factor.

IDA* has a considerable overhead because of **no duplicates** are detected which may be exponentially slower in many state spaces and is therefor often combined with partial duplicate estimation.

The overhead due to iterative increases of f bound often negligible but not always, especially if the action cost vary a lot. This means that a new bound for f only reaches a small number of new serach nodes.

This means that IDA* is most useful when there are few duplicates.

3 Combinatorial Optimization

Previously, in state-space search, we tried to find action sequence from initial state to goal state. The difficulty is the state explosion where a large number of states occures.

With **combinatorial optimization** there are no actions or transitions. we **don't search for path** but for a **configuration**(a state) with low cost or highest quality.

3.1 Problem Definition

A combinatorial optimization problem(COP) is given by a tuple $\langle C, S, opt, v\rangle$ consisting of

- a set of (solution) candidates C
- a set of solutions $S \subseteq C$
- an objective sense $opt \in \{min, max\}$
- an objective function $v : S \to \mathbb{R}$

Here, a **problem** is ment in the sense of an instance which is different to its usual usage in computer science. In normal COPs the number of candidates is usually too large to be enumerated explicitly.

An **optimal solution quality** v* for $\mathcal{O} = \langle C, S, opt, v\rangle$ is defined as

$$v^* = \begin{cases} \min_{c\in S} v(c) & \text{, if opt=min} \\ \max_{c\in S} v(c) & \text{, if opt=max} \end{cases} \tag{3.1.1}$$

The idea of COPs is to find a **solution** of good or ideally optimal quality for our COP $\mathcal{O}$ or to prove that no solution exists at all. The measure how good a solution is defined by its closeness to v*.

They have two aspects

- **search aspect** where we want to find a solution from set S amongst all candidates C
- **optimization aspect** where we to find, among all solutions S, the one of highest quality.

For **pure search problems** all solutions are equally good and the challenge is to find a solution at all. v is then a constant function, and opt can be choosen arbitrarily.

For **pure optimization problems** all candidates are solutions in the difficulty is to find solutions of high quality. Formally $S = C$.

EXAMPLE 8-Queens

COPs can be solved with any of the methods presented in the lecture, but this and the next chapter focuse on **local search**.

The main idea is, to apply the same concept of heuristic search where states ≈ candidates, but there are **no actions** so the designer of the algorithm chooses the neighbors of a given candidate. This is a criticial aspect. The path to goal however is irrelevant. .

3.2 Local Search

A heuristic h estimates the quality of a candidate. For pure optimization, one could choose the objective function v itself. For pure search often the distance to closest solution is used (as in state space search). The method **does not remember** paths only candidates and is very memory efficient, as only one current candidate. This however means that the algorithm is neither **optimal nor complete**.

The solution is **improved iterativly** by **hill climbing**

---

function HILL CLIMBING (FOR MAXIMIZATION PROBLEMS
  current := a random candidate
  repeat
  next := a neighbor of current with maximum h value
 if h(next) ≤ h(current) **then**
  return current
 current := next

The search is a **walk** uphill towards a solution.

3.2.1 Properties of Hill Climbing

The algorithm always terminates if candidate set is finite because the algorithms sees every state, given enough time, however there is **no guarantee** that a result is a solution.

If the algorithm finds a solution, this solution is **locally optimal** with respect to h but no global qualities are guaranteed.

3.3 Advanced techniques

3.3.1 Dealing with local Optima

Due to a situation, where all neighbors are worse than current candidate (local optima) or a situation where many neighbors are equally good as the current candidate, the algorithm might get stuck at a candidate.

Possible solutions of this problem are

- **Allowing stagnation** so steps without improvements. Limit number of steps to guarantee terminiation at some point and return the best visited candidate at the end.
- **Include random aspects** in the search neighborhood. This combines exploration(by randomness) with exploitation(heuristic guidance)
- **make random steps**
- **breadth-first search** to better candidates
- **restarts** with new initial candidates

OUTLOOK?

4 Constraint Satisfaction Problems

constraint Satisfaction Problems are classified as **static**, **deterministic**, **fully observable**, **single-agent** and the problem solving method is **general**

Constraint is a condition that every solution to a problem must satisfy.

This means that, given a *set of variables* with corresponding domains and a *set of constraints* we want to find an **assignment** to the variables, that satisfie all constraints.

Some examples of constraint satisfaction problems are

- **The 8-queens problem**
- **Sudoku** with its constraints to a number in a certain cell
- **Graph coloring**
- **satisfiability in propositional logic**

4.1 Constraint Networks

A **constraint network** is (informally) defined by

- A finite set of variables
- A finite set of domains for each variable
- A set of constraints (here: binary relations)

Formally, this yields

Binary Constraint Network A binary constraint network is a 3-tuple $\mathcal{C} = \langle V, dom, (R_{uv})\rangle$ where

- V is a non-empty and finite set of variables
- dom is a function that assigns a non-empty and finite domain to each variable $v \in V$.
- $(R_{uv})_{u,v\in V,u\neq v}$ is a family of binary relations (**constraints**) over V where for all $u \neq v$ ($R_{uv} \subseteq dom(u) \times dom(v)$. For a binary constraint, R_{uv} expresses which joint assignments to u and v are allowed in solutions. This means that the constraint is trivial if $(R_{uv}) = dom(u) \times dom(v)$ since every combination of assignments is allowed.

Additionally, one might use **unary constraints** to restrict a single variable. Here R_v restricts the values of $dom(v)$.

These networks allow for **compact encoding** of large sets of assignments. For a network with n variables and domains of size k we have k^n assignments.

For the description, we have at most $\binom{n}{2}(\Rightarrow O(n^2)$ constraints) where every constaint consists of $O(k^2)$ pairs, so we only have $O(n^2k^2)$ for the encoding size.

4.1.1 Assignments and Consistency

For a constraint network $\mathcal{C}$

partial Assignment of $\mathcal{C}$ is a function

$$\alpha : V' \to \cup_{v\in V} dom(v) \tag{4.1.1}$$

with $V' \subseteq V$ and $\alpha(v) \in dom(v) \forall v \in V'$. This assigns values to some or all variables in V. A **total assignment** are defined on all variables, so $V' = V$.

Incosistents A parital assignment α is incosistent if there are variables u, v such that α is defined for both u and v but $\langle \alpha(u), \alpha(v) \rangle \notin R_{uv}$ so α violates the constraint R_{uv}.

Solution A **consistent** and **total** assignment of $\mathcal{C}$ is called a solution of C. If a solution exists, then $\mathcal{C}$ is solvable else it is **incosistent**.

Note that, just because a partial assignemtn α is consistent does not mean that this set can be expanded into a solution. It only means that, so far, no constraints where violated.

Thightness Let $\mathcal{C}' = \langle V, dom', R'_{uv} \rangle$ be annother constraint network on the same variable set V as $\mathcal{C}$. We call $\mathcal{C}$ **tighter** than C', in symbols $\mathcal{C} \sqsubseteq \mathcal{C}'$, if

- $dom(v) \subseteq dom'(v) \forall v \in V$
- $R_{uv} \subseteq R'_{uv} \forall uv, \in V$

Equivalent We call two networks with equal variable sets **equivalent**$(C \equiv C')$ if they have the same solution.

4.2 CSP Algorithms

The basic concept of CSP solving algorithms consists of

- **search** check partial assignments systematically
- **backtracking** discard inconsistent partial assignments
- **inference** derive equivalent, but teighter constraints to reduce the size of the search space

4.2.1 Naive Backtracking

A very simple approach (without inference) can be made to backtracking with the following algorithm

ALGORITHM NaiveBacktracking

This is a **depth-first search** in a state-space where

- **states** are consistent partial assignments
- **initial state** is the empty assignment
- **goal states** Are consistent total assignments
- **actions** $assign_{v,d}$ that assigns a value $d \in dom(v)$ to variable v.
- **action cost** of 0
- **transitions** for each non-total assignment α choose variable $v = select(\alpha =)$ that is unassigned in α and transition $\alpha \xrightarrow{assign_{v,d}} \alpha \cup \{v \to d\}$ for each $d \in dom(v)$

Depth-first search is particularly well-suited here because of the bounded path length(by number of variables). Also solutions are located at the same depth, and there is guarantee for no duplicates since the state space is a directed tree (WHY)

The problem is, that naive backtracking often **explores similar searrch paths(partial assignments)** that are identical. So critical variables are not recognized and decisions that necessarily lead to constraint violations are only recognized when all variables involved in the constraint have been assigned.

This leads to focuse on critical decisions and by inference of consequences of previous decisions.

The Backtraking does not specify the order in which variables are considered for assignment however, this can **strongly influence** the state space size.

For variable orders on commonly uses

minimum remaing values which preferes variables that have small domains as this leads to fewer subtrees which corresponds to a smaller overall tree

most constraining variable which preferes variables contained in many nontrivial constraints as this leads to constraints beeing **tested early on**. This leads to a smaller tree as well as inconsistencies are recognized early on .

Alternativly these can be combined by using minimum remaing values criterion and break ties with most constraining variables criterion.

Also, the value order is not specified. This is not importent for subtrees without solutions WHY but if there is a solution than ideally a value that leads to a solution should be chosen first.

Conflict In a constraint network $\mathcal{C}$ for variables $v \neq v'$ and values $d \in dom(v)$, $d' \in dom(v')$ the assignemnt $v \to d$ is in conflict with $v' \to d'$ if $\langle d, d' \rangle \notin R_{vv'}$

So prefere values $d \in dom(v)$ such that $v \to d$ causes as few conflicts as possible with unassigned variables in α. We can either use

- **static** fixed order prior to the search
- **dynamic** orders that select orders based on the search state

4.3 Inference

Inference Derive additional constraints that are implied by the given constraints i.e. that are satisfied in all solutions.

An example would be a constraint network with variables $V = \{v_1, v_2, v_3\}$, Domain $\{1, 2, 3\}$ and constraints $v_1 < v_2$ and $v_2 < v_3$

It follows, that v_2 cannot be equal to 3, since v_3 has to be bigger than v_2 and the only value that satisfies this is if v_2 is smaller than 3.

This is a new unary constraint that tightenes the domain of v_2.

Formally, **Inference** replaces a constraint network $\mathcal{C}$ with a tighter but equivalent constraint network, using inference. This yields a much smaller state space to search in, but the more complex the inference the higher **the complexity per search node**

You can either apply Inference **once in preprocessing before search** or **combined with search** before recursive calls during the backtracking procedure.

For an already assigned variable $v \to d$ corresponds to $dom(v) = \{d\}$ more inferences are possible. However, the derived constraints have to be **retracted** during the backtracking phase since they were based on a given assignment.

This procedure is very powerful but might become expensive.

ALGORITHM BACKTRACKING WITH INFERENCE

In itself, **inference** is a placeholder as different methods can be applied. They can recognize unsolvability (given α) and indicate this by clearing the domain of a variable.

Implemented efficiently, inferences are often **incremental**, so previously assigned variable/value pair is taken into account.

4.3.1 Forward Checking

Forward Checking is a very simple inference method

Forward Checking Given a partial assignment α, remove all values from the domains of unassigned variables v in α that are in conflict with already sassigned variable/value pairs in α.

This method is a correct inference, as it retains equivalence and **acts on the domains**. So it affects **the unary constraints** not binary constraints.

Also, the consistency checking at the beginning of the backtracking is no longer needed as the forward checking is doing this incrementally. This method is considerably **cheap** but still useful.

4.3.2 Arc Consistency

Arc Consistency For a constraint network $\mathcal{C}$ a variable $v \in V$ is **arc consistent** with respect to $v' \in V$ if for every value $d \in dom(v)$ there exists a value $d' \in dom(v')$ with $\langle d, d' \rangle \in R_{vv'}$

The idea is that, for every assignment to a variable u there must be a suitable assignment to every other variable v. If there exists no such assignment, remove values of u for which no suitable partner assignment for v exists. This introduces a **tighter unary constraint** on u.

ALGORITHMS AND PROOF

4.3.3 Path consistency

The model of Arc consistency can be enhanced to path consistency. For every joint assignment to variables u, v there must be a suitable assignment to every third variable w. If not, remove all pairs of values from u and v for whom no partner assignments for w exist. This introduces a **tighter binary constraint on u and v**

This can be further enhacned into $i-$consistency, however in practice cases beyond $i = 3$ (which is path consistency) is bareley used.

4.4 Problem structure

To solve a constraint network of n variables and k values k^n assignments must be considered. Inference may allevite combinatorial explosion but will not always avoid it. Therefor, the **strucutre** has to be taken into account in order to further solve practically relevant constraint networks.

4.4.1 Constraint Graph

Constraint Graph For a constraint network $\mathcal{C}$ the constraint graph of $\mathcal{C}$ is graph with vertices V and which contains an edge between u and v iff R_{uv} is a non-trivial constraint.

This holds a huge optimization as one can state that, if the constraint graph has multiple connected components, these **subproblems** can be solved separetly. the union of the resulting solutions of all subproblems is a solution for $\mathcal{C}$.

For a network with $k = 2$ and $n = 30$ that decomposes into three subprblems of equal size this becomes $3 \cdot 2^{10} = 3072$ instead of $2^{30} = 1073741823$ assignments.

4.4.2 Trees as constraint Graphs

If the Graph does not contain cycles (so it technically is a tree) this can be optimized even further. We then can solve $\mathcal{C}$ in $O(nk^2)$.

The algorithm is as follows

1. Build a directed tree with a arbitrary variable as the root node
2. Order variables $v_1, \ldots, v_n$ such that parents are ordered before their children.
3. For $i \in \langle n, n-1, \ldots, 2 \rangle$ call `revise(`$v_{parent}(i), v_i$`)` So each variable is arc consistent with respect to its children.
4. If a domain becomes empty, the problem is unsolvable
5. Otherwise, solve with `BacktrackingWithInference` with variable order $v_1, \ldots, v_n$ and forward checking.

This means that the solution is found **without backtracking steps**

4.4.3 Decomposition Methods

The previous chapter showed that constraint graphs can efficiently be solved if they are either a tree or decompose in several components. If this is not the case there exists two concepts to deal with the constraint graph.

Conditioning Apply backtracking with forward checking until the constraint graph **restricted to the remaining unassigned variables** decompoes or is a tree

Tree Decomposition Consider a constraint network $\mathcal{C}$ with variables V. A tree decomposition of $\mathcal{C}$ is a graph $\mathcal{T}$ where

- Every vertex of $\mathcal{T}$ corresponds to a subset of variables V Such a vertex is called a subproblem of $\mathcal{C}$
- Every variable of V appears at least once in a subproblem in $\mathcal{T}$
- For every contrivial constraint R_{uv} of $\mathcal{C}$ the variables u and v appear together in at least one subproblem of $\mathcal{T}$
- $\mathcal{T}$ is acylci(a tree/forest)

With Tree decomposition meta variables, a group of orginial variable that jointly cover all variables and constraints, are used. Their values correspond to consistent assignment to the groups. Then use contraints between overlapping groups to ensure compatibility.

5 Propositional logic

Propositional logic is a way of modeling and representing problems and knowledge that allows for automated reasoning.

A satisfiable problem in propositional logic can be seen as a non-binary CSP over $\{F, T\}$, where the formula encodes constraints.

To model a state space, states are represented as truth values of atomic propositons. An atomic propositon for the missionaries and cannibals problem would be `two-missionaries-on-left-shore`. These propositions can not be divided furhter and logical connections of these propositions form propositional formulas.

5.1 DLPP Algorithm

5.2 Local Search

6 Automated Planning

The goal of planning is to find a sequence of actions that lead from an initial state to a goal state.

Classical planning uses a general approach to find solutions for state space search problems.

So given a state space description in terms of suitable problem description language we either want a plan or a proof that no plan exists. We distinguish between optimal planning and suboptimal planning that differ int the optimality of allowed solutions.

New in this chapter is, that we're searching for a general algorithm that does not know the task the algorithm is going to solve! This means that there is **no problem specific heuristic** and that we need an **input language** to model the planning task.

6.1 Compact Description

Instead of viewing a state space as a black box we introduce state varaible. A state is an assignment to state variables. This means that n binary state variables can describe 2^n states. Transitions and goals are described with logic-based formalisms.

6.2 Planning Formalisms

We call a description language for state spaces a **planning formalism**

6.2.1 STRIPS

STRIPS stands for Stanford Research Institute Problem Solver and is the most simple common planning formalism. Its state variables are binary.

States s(based on agiven set of state variables V) can be represented as an assignment $s : V \to \{F, T\}$ or as sets $s \subseteq V$ where s encodes the set of state variables that are true in s.

Goals and preconditions of actions are given as sets of variables that must be true.

Effects of actions are given as sets of variables thare are set to true or false after the action.

STRIPS Planning Task A STRIPS planning task is a 4-tuple $\sqcap = \langle V, I, G, A \rangle$ where

- V finite set of state variables
- $I \subseteq V$ the initial state
- $G \subseteq V$ the set of goals
- A a finite set of actions where for all actions $a \in A$ we know
  - $pre(a) \subseteq V$ the preconditions for a
  - $add(a) \subseteq V$ the add effects of a
  - $del(a) \subseteq V$ the delete effects of a
  - $cost(a) \in \mathbb{N}_0$ the cost of a

This implies a state-space $S(\sqcap) = \langle S, A, cost, T, s_0, S_* \rangle$

- Set of states $S = 2^V$
- actions A as defined in $\sqcap$
- action costs as defined in $\sqcap$
- transitions $s \xrightarrow{a} s'$ for states s, s' and action a iff $pre(a) \subseteq s$(preconditions satisfied) and $s' = (sdel(a)) \cup add(a))$ (effects are applied)
- Initial state $s_0 = I$
- Goal states $s \in S_*$ for states s iff $G \subseteq s$(goals reached)

STRIPS is a particularly simple yet powerful description that is cumbersome for users

6.2.2 ADL

In addition to STRIPS, ADL allows for **arbitrary logic formulas** to describe actions and goals. So an action is applicable or a goal reached in states that satisfy the formula. Additionally, there are **conditional effects**. A variable v is only set to true/false if a given logical formula is true in the current state.

6.2.3 SAS$^+$

Instead of binary domains as in STRIPS, SAS$^+$ allows for finite domains. States are assignments to these variables. Preconditions and goals are given as **partial assignments** and effects are assignments to subset of variables. So for the successor state s' the effect $\{v_1 \to b, v_2 \to c\}$ means that $s'(v_1) = b$ and $s'(v_2) = c$ while the rest retains their values.

6.2.4 PDDL

PDDL is the standard language used in practice. The description is made even more compact by using a restricted predicate logic instead of pure propositional logic.

It also introduces features like numeric variables and derived variables for defining macros. These are formulas that are automatically evaluated in every state and can be used in preconditions.

6.3 Planning Heuristics

The STRIPS planner uses the number of goals not yet satisfied in a STRIPS planning task as a heuristic. The intuiton shows that feqer unsatisfied goals is closer to goal state.

Hower, the heuristic is **rather uninformed**.

- For state s if there is no action a in s such that applying a in s satisfies strictly more (or fewer) goals, then all successor states have the same heuristic value as s.
- It ignores almost the whole task structure since the heuristic value does not depend on the actions.

6.3.1 Delete relaxation

The basic idea of **delete relaxation** is to estimate solution costs by considering a simplified planning task where all negative action effects are ignored.

In STRIPS tasks, `add` effects are always useful as they get us closer to fully satisfying all variables of the goal state while `delete` effects are always harmful. The idea of relaxation is to ignore all of these negative effects.

relaxation of actions The relaxation a^+ of STRIPS action a is the action with $pre(a^+) = pre(a)$, $add(a^+) = add(a)$, $cost(a^+) = cost(a)$ and $del(a^+) = \emptyset$

Relaxation of a planning taks for a STRIPS planning task $\sqcap = \langle V, I, G, A\rangle$ is the task $\sqcap^+ = \langle V, I, G, \{a^+ | a \in A\}\langle$

The same pattern applies for an action sequence π. STRIPS planning tasks without delete effects are called relaxed planning tasks. Plans for these are called **relaxed plans**.

$h^+(\sqcap)$ denotes the cost of an optimal plan for $\sqcap^+$, so an optimal relaxed plan. Analogously, $h^+(s)$ denotes the cost of optimal relaxed plan starting in state s. We then call h^+ **the optimal relaxation heuristic**.

For general STRIPS planning tasks h^+ is an admissible and consistent heuristic.

However, calculating the optimal solution for h^+ is hard. In fact, NP-hard as it is an reduction from SET COVER. A suboptimal solution can be found quite simply, so in practice, heuristics approximate h^+ from below or above.

A **relaxed planning graph** represents which variables in $\sqcap^+$ can be reached and how. We have graphs with variable laysers V^i and action layers A^i where variable layer V^0 contains the variable vertex $v^0 \forall v \in I$. The action layer A^{i+1} contains the action vertex a^{i+1} for action a if V^i contains the vertex $v^i \forall v \in pre(a)$

6.3.2 Maximum and Additive Heuristics

h^{max} and h^{add} are the most simple RPG heuristics. The vertex annotations are numerical values, where the vertex values estimate the cost to

- make a given variable true
- reach and apply a given action
- reach the goal

6.4 Abstraction

The Idea of **abstraction** is to estimate cost by considering a smaller planning task.

This uses SAS$^+$ instead of STRIPS, so

- state variables v are not binary but with **finite domain** $dom(v)$
- Therfore preconditions, effects and goals are specified as partial assignments.

Formally,

SAS$^+$ planning task is a 5-tupple $\sqcap = \langle V, dom, I, G, A\rangle$ where

- V finite set of state variables
- dom domain; $dom(v)$ must by finite and non-empty $\forall v \in V$. States are **total assignments** for V according to dom
- I the initial state(where a state is a total assignment)
- G goals (partial assignments)

- A finite set of actions a with
  - $pre(a)$ it's preconditions (partial assignment)
  - $eff(a)$ it's effects (partial assignment)
  - $cost(a) \in \mathbb{N}_0$ the cost of an action

SAS$^+$ state space A SAS$^+$ planning task $\sqcap$ induces a state space $S(\sqcap) = \langle S, A, cost, T, s_0, S_* \rangle$ with

- set of states: total assignments of V according to dom
- actions A defined in $\sqcap$
- action cost $cost$ as defined in $\sqcap$
- transition $s \xrightarrow{a} s'$ for states s, s' and action a iff
  - $pre(a)$ complies with s(precondition satisfied)
  - s' complies with $eff(a)$ for all variables mentioned in eff; complies with s for all variables (effects are applied)
- initial state $s_0 = I$
- goal states $s \in S_*$ for state s iff G complies with s

6.4.1 State Space Abstraction

State space abstractions drop distinctions between certain states but preserve the state space behavior as well as possible.

It is defined by an abstraction function α that determines which states can be distinguished in the abstraction. We use this information to compute the abstraction state space $\mathcal{S}^\alpha$ of $\mathcal{S}$ which is similar but smaller.

Abstraction Heuristic Use abstract goal distances(goal distances in $\mathcal{S}^\alpha$ as a heuristic for concrete goal distances in $\mathcal{S}$. Abstraction heuristic h^α.

Induced abstraction Let $\mathcal{S} = \langle S, A, cost, T, s_0, S_* \rangle$ be a state space and $\alpha : S \to S'$ a surjective function. The **abstraction of $\mathcal{S}$ induced by** α denoted as $\mathcal{S}^\alpha$ is the state space $\mathcal{S}^\alpha = \langle S', A, cost, T', s'_0, S'_* \rangle$ with

- $T' = \{\langle \alpha(s), a, \alpha(t) \rangle | \langle s, a, t \rangle \in T\}$
- $s'_0 = \alpha(s_0)$
- $S'_* = \{\alpha(s) | s \in S_*\}$

Every abstraction heuristic is **admissible and consistent**. the choice of the abstraction function α is of vital importance. **Every** α yields a admissible and consistent heuristic, but most of them lead to poor heuristics.

Ideas for heuristics

- **one state abstraction** $\alpha(s) = const.$ compact and easy to compute but very uninformed heuristic.
- **identify abstraction** $\alpha(s) = s$ perfect heuristic and easy to compute, but it has too many abstract states so computation of h^α is too hard.

These ideas lead to bad heuristics. Instead use either

- Pattern databases
- merge-and-shrink

6.4.2 Pattern Databases

Pattern database heuristic is among the best known heuristics for many search problems. In a PDB, some aspects **are preserved with perfect precision** while other aspects are not preserved at all.

pattern database heuristics Let P be a subset of variables of planning tasks. the abstraction heuristic induced by the **projection** π_P on P is called pattern databse heuristic with pattern P noted h^P

In practice, the question remains on how to automatically find good patterns or combine multiple PDB heuristics.

A fundamental restriction of PDBs is **that patterns must be kept small**.

6.4.3 Merge-and-Shrink

Merge and shrink abstractions are a proper generalization of PDBs. They represent PDBs (with a polynomial overhead) and can sometimes represent abstractions compactly where this is impossible with PDBs.

In comparisson to PDBs that represent **a few state variables perfectly** in the abstract state space while merge-and-shrink represents **all state variables, but in a potentially lossy fashion**

6.5 Landmarks

Landmarks are something that must be part **of every solution** . We can then estimate solution cost by the number of landmarks unsatisfied.