Kontext-basierte Suche für klassische Handlungsplanung

Bachelorarbeit

Lukas Songajlo

Betreuer: Manuel Heusner, Dr. Martin Wehrle

Prüfer: Prof. Dr. Malte Helmert

1. September 2014

Was ist klassische Handlungsplanung?

Vorgehensweise:

- Agent beginnt in einem initialen Zustand
- Angewandte *Aktionen* wechseln Zustände
- Agent erreicht (wenn möglich) einen Zielzustand

Plan: Sequenz von Aktionen

Ziel: *Automatisches* Lösen von Planungsproblemen

SAS$^+$-Planungsaufgaben (formal)

Eine Planungsaufgabe ist ein 4-Tupel $\Lambda = \langle \mathcal{V}, \mathcal{A}, s_0, G \rangle$:

- $\mathcal{V}$ - Menge von Zustandsvariablen mit jeweils endlichen Wertebereich
- $\mathcal{A}$ - Menge von Aktionen mit jeweils *pre*, *eff* und cost
- s_0 - Anfangszustand
- G - Menge von belegten Zustandsvariablen

Heuristische Suche

Heuristikfunktion h(s) liefert Informationen über den Zustandsraum:

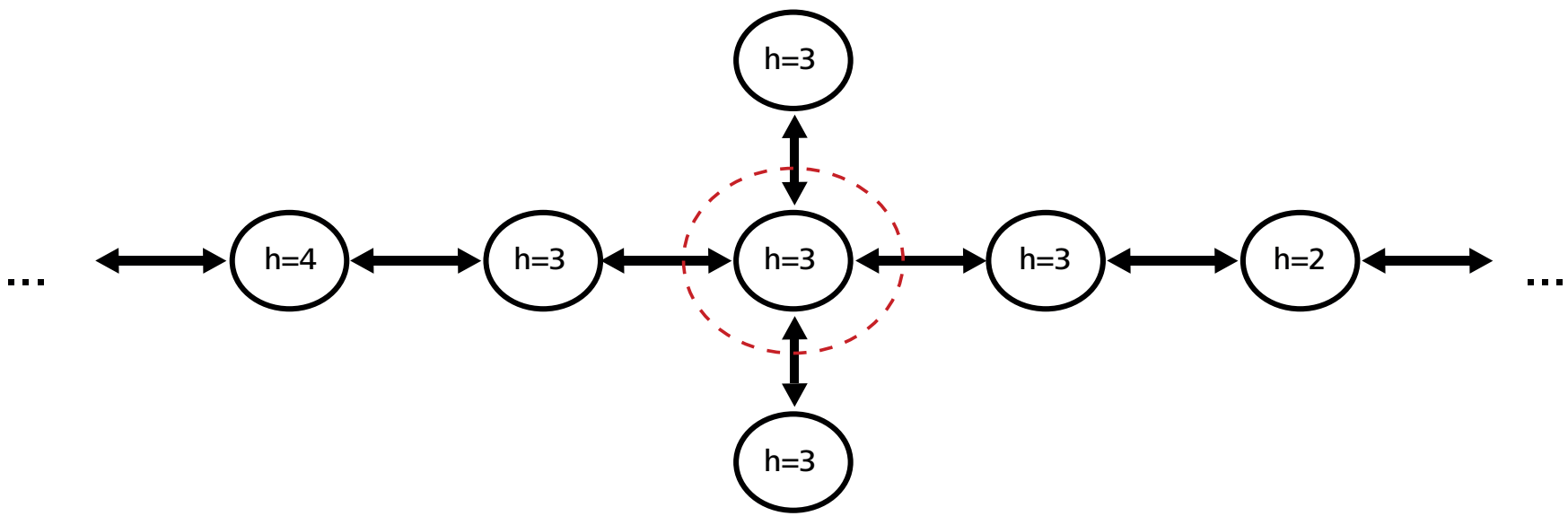
- Schätzt Abstand von Zustand s zu einem Zielzustand
- Weist potentiellem Nachfolge-Zustand einen numerischen Wert zu

Bezug zur Handlungsplanung?

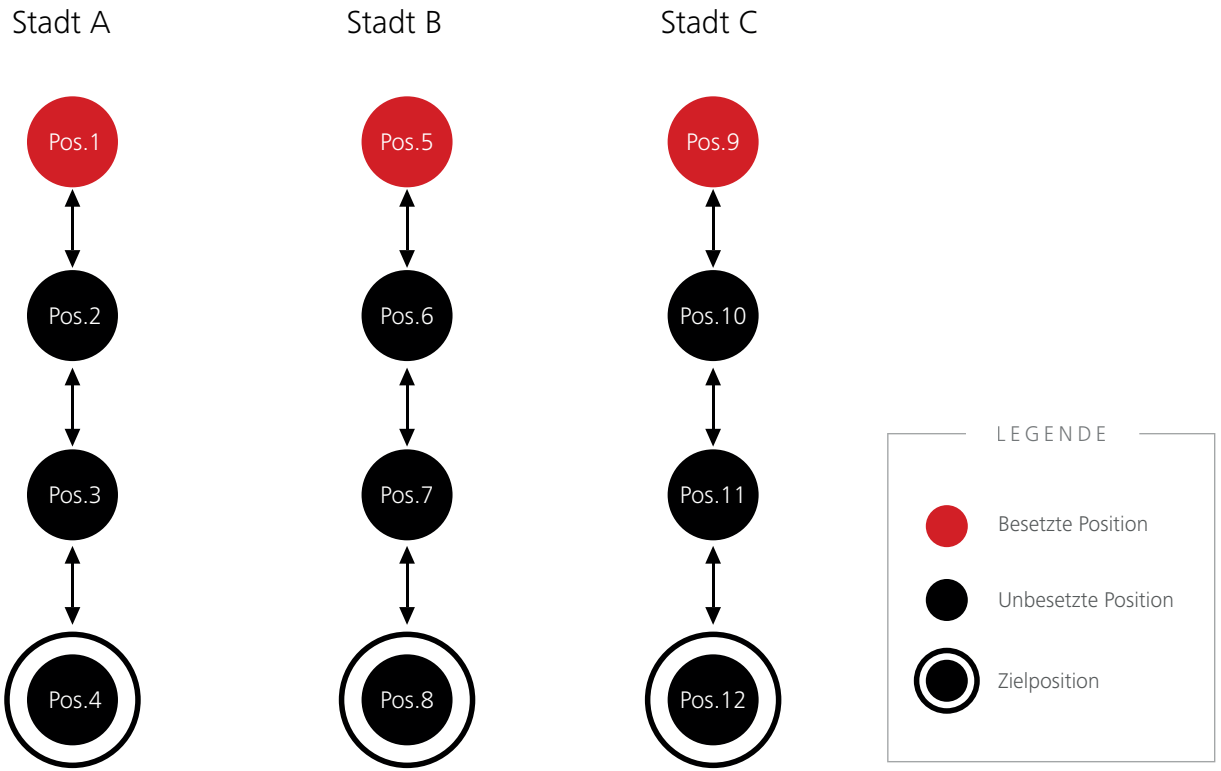
Heuristische Werte werden automatisch berechnet, ohne Kenntnis über Domäne (z.B. h^{max}, h^{FF}, ...)

Bekanntes Problem - Suchplateaus

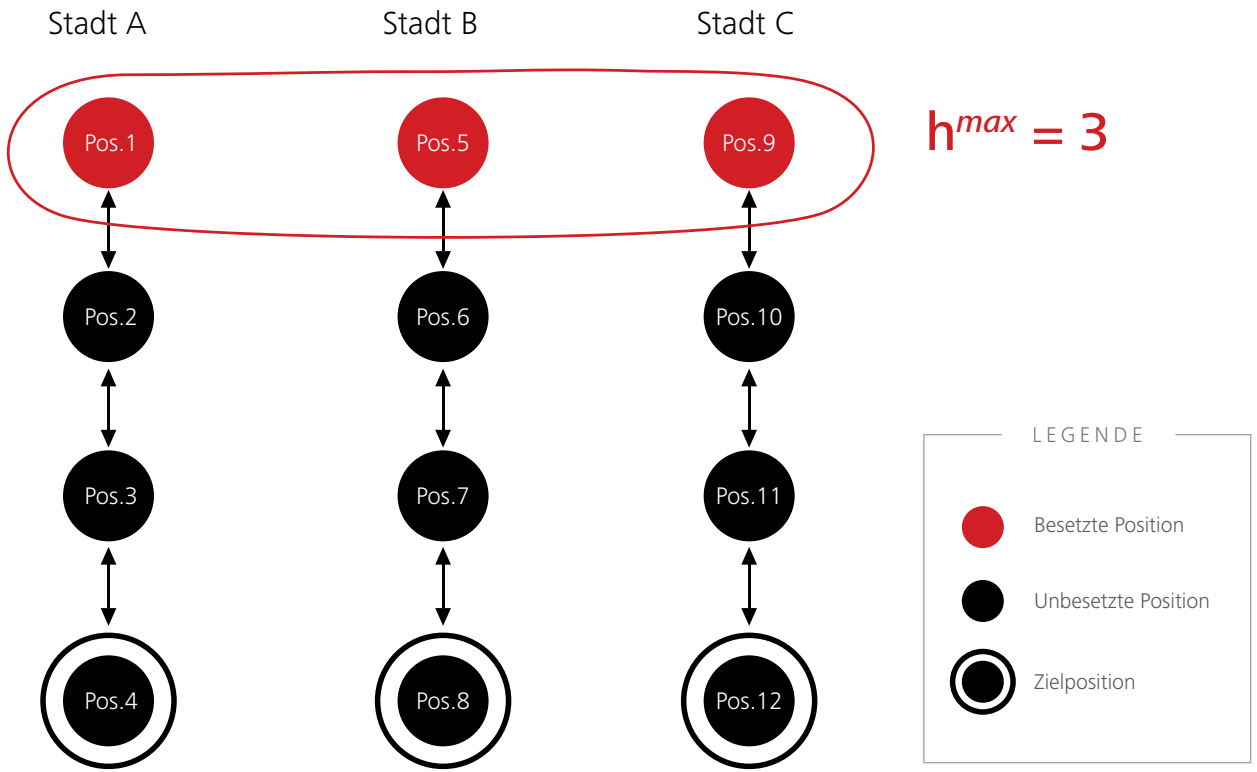
- Bei gleichbleibendem heuristischen Wert können **Suchplateaus** entstehen
- Suche verläuft unter „Gierige Bestensuche“ blind



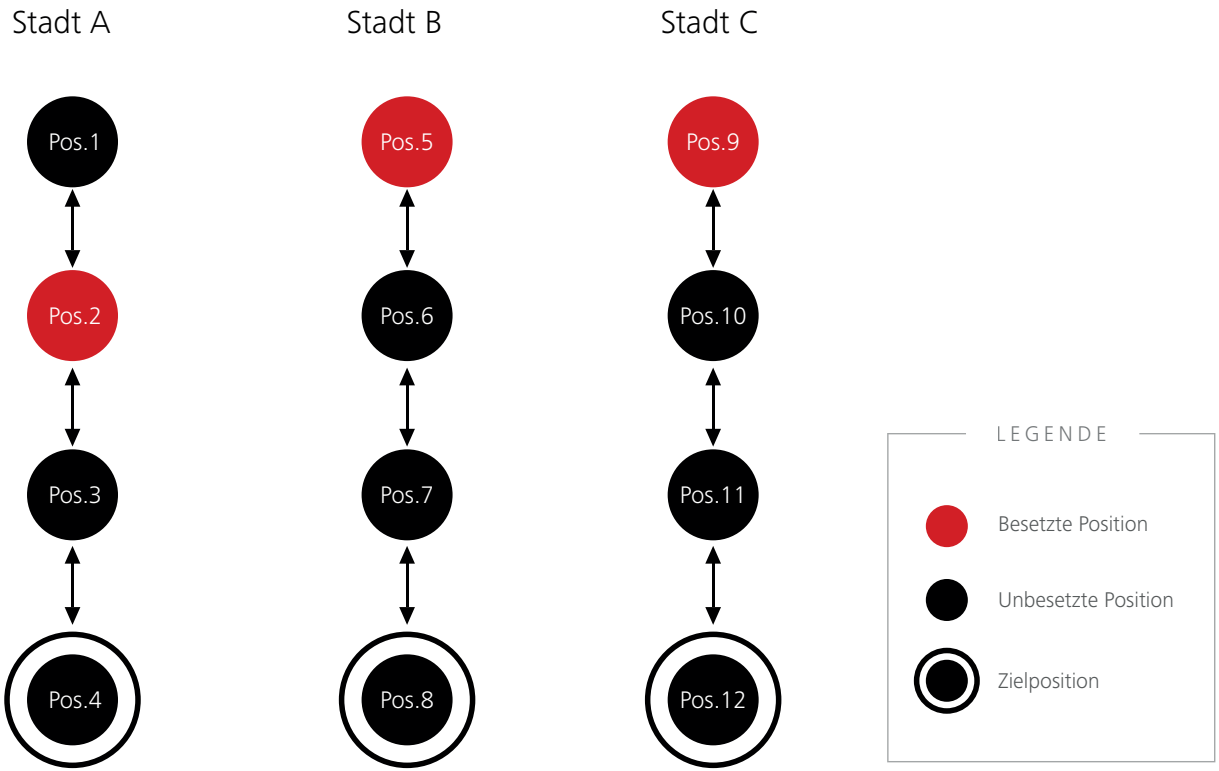
Motivation - Logistikproblem



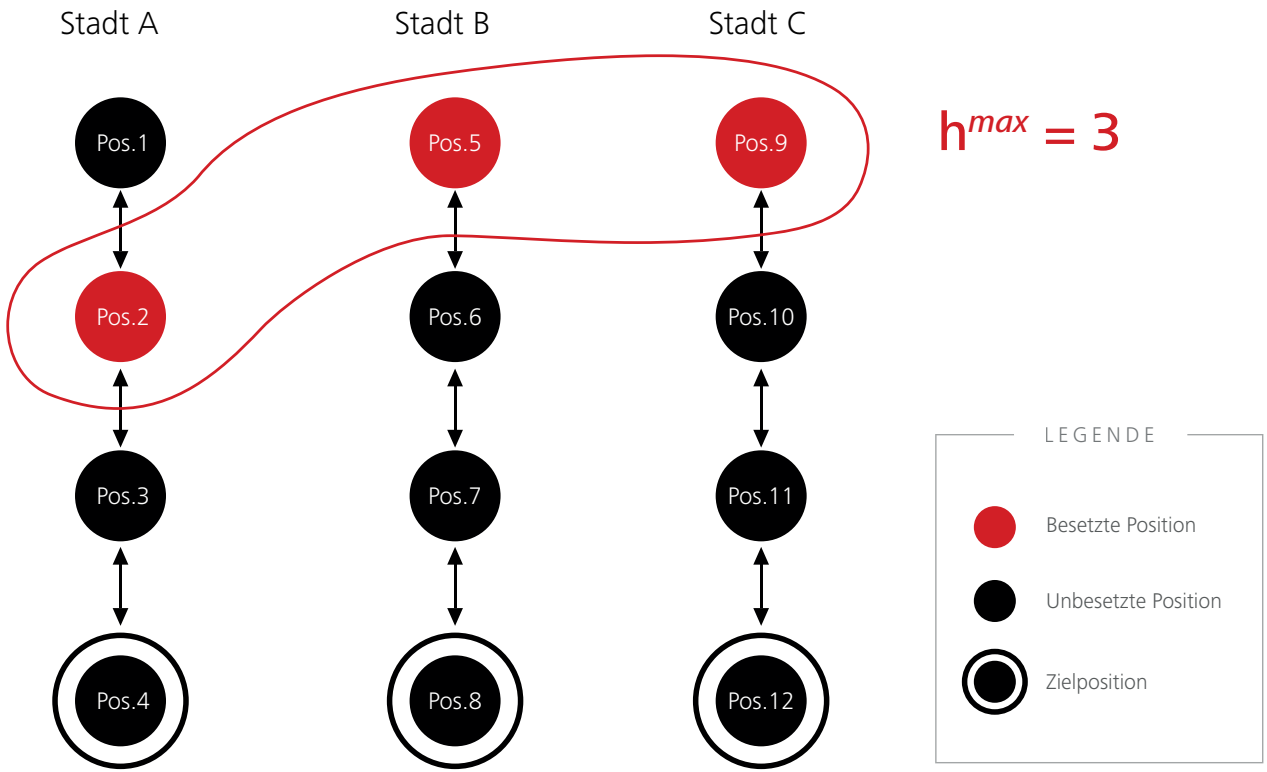
Motivation - Logistikproblem



Motivation - Logistikproblem



Motivation - Logistikproblem



Mehr Orientierung! Thema dieser Arbeit

Gezieltere Suche durch zusätzliche Information:

- Ausnutzung von *Aktions*-Abhängigkeiten
- Erweiterung der gierigen Bestensuche

Suchalgorithmus der gierigen Bestensuche

```
 1: openlist := new priority queue ordered by h
 2: openlist.initialize()
 3: closedlist := { }
 4: while openlist not empty do
 5:     node = openlist.getMinimum()
 6:     if closedlist.contains(node.state) = none then
 7:         closedlist.add(node.state)
 8:         if node.state is a goalstate then
 9:             return getPath(node)
10:         for each successor s' of node.state do
11:             if h(s') < ∞ then
12:                 openlist.add(createNode(s'))
13: return unsolveable
```

Idee der kontext-basierten Suche

- Aktion bevorzugen, die von der **Vorgänger**-Aktion profitiert
- Pfad solange folgen bis Teilziel erreicht ist

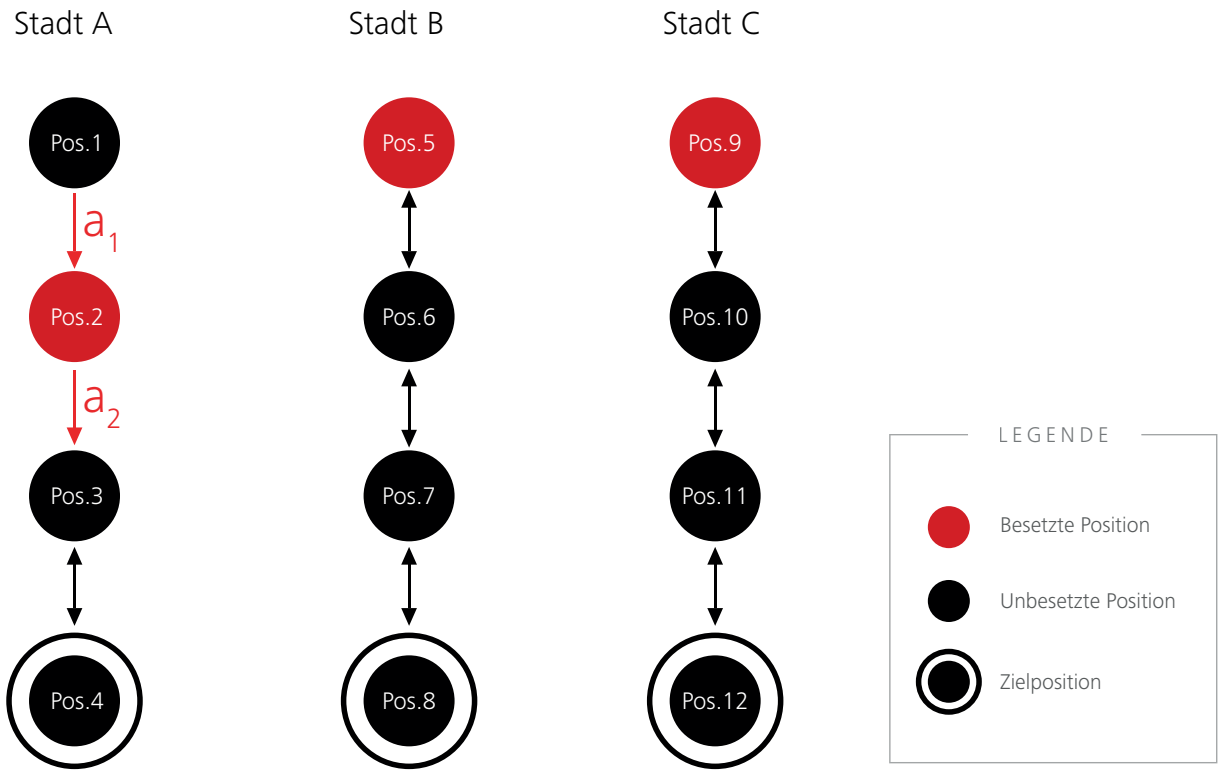
Abhängigkeiten zwischen Aktionen

Gegeben: $a_1, a_2 \in \mathcal{A}$

Interferenz

- pre(a_1) und eff(a_2) enthalten gemeinsame Variablen
- pre(a_2) und eff(a_1) enthalten gemeinsame Variablen
- eff(a_1) und eff(a_2) enthalten gemeinsame Variablen

Abhängigkeiten zwischen Aktionen



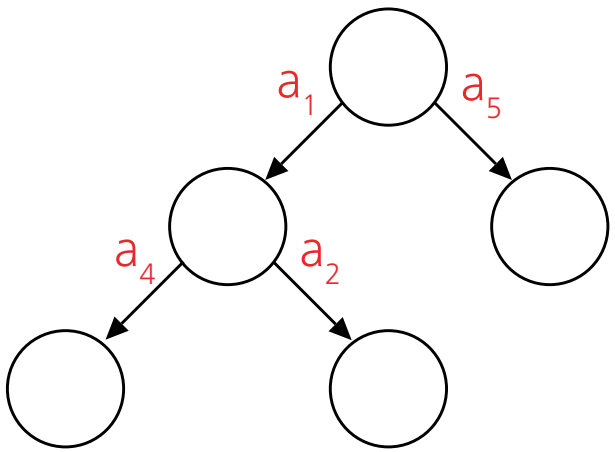
Abhängigkeiten zwischen Aktionen

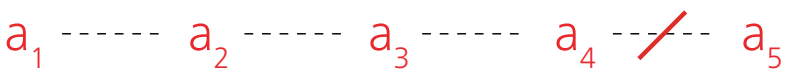
Gegeben: $a_1, a_2 \in \mathcal{A}$ und $d \in \mathbb{N}$

Interferenz-Distanz

- **Fall 1**: $a_1 = a_2$, dann gilt $d = 0$
- **Fall 2**: a_1, a_2 interferieren über paarweisen Interferenzen miteinander, dann gilt $d = \#$ paarw. Interferenzen
- **Fall 3**: a_1, a_2 interferieren überhaupt nicht, dann gilt $d = \infty$

Abhängigkeiten zwischen Aktionen





Umsetzung der Interferenz-Distanzen

- Repräsentation als Matrix
- Berechnung mit Flyod's Algorithmus im Vorfeld der Suche

Priorisierung von Zuständen

1. Priorisierung: Interferenz-Distanz

2. Priorisierung: Heuristischer Wert (Tiebreaker)

Suchalgorithmus

```
1:  openlist := new priority queue ordered by h
2:  openlist.initialize()
3:  closedlist := { }
4:  while openlist not empty do
5:      node = openlist.getMinimum()
6:      if closedlist.contains(node.state) = none then
7:          closedlist.add(node.state)
8:          if node.state is a goalstate then
9:              return getPath(node)
10:         for each successor s' of node.state do
11:             if h(s') < ∞ then
12:                 openlist.add(createNode(s'))
13: return unsolveable
```

Suchalgorithmus

Funktion: add(Node n)

```
 1: priority-queues := given N + 1 priority queues
 2: a := action to current state s
 3: a' := applicable action to switch state from s to n.state
 4: if a not writes variable corresponding to G then
 5:     if D(a, a') ≠ ∞ then
 6:         priority-queues[D(a, a')].add(n)
 7:     else
 8:         prune n.state
 9: else
10:     priority-queues[0].add(n)
```

Suchalgorithmus

Funktion: Node getMinimum()

```
1: best_list = 0
2: for each queue do
3:     if queue is not empty then
4:         best_list = queue.getIndex()
5:         break
6: return priority-queues[best_list].getMinimum()
```

Resultate

- basierend auf Instanzen von Fast-Downward

Gliederung

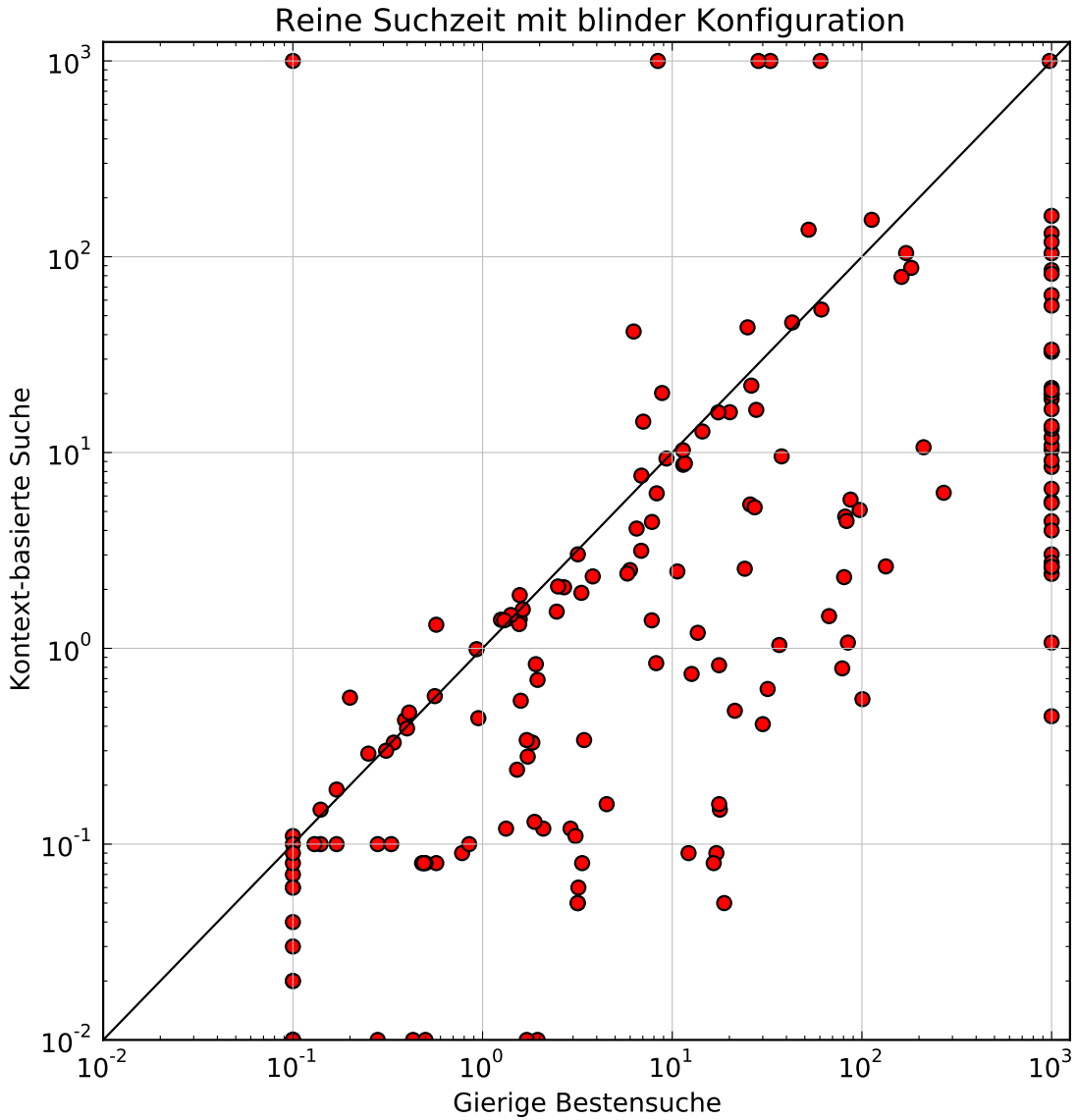
Vergleich GBS und KBS mit...

- ...blinder Suche
- ...h^{max}
- ...h^{FF}

Ergebnisse mit blinder Suche

	GBS	KBS
Gelöste Probleme - Summe	260	**288**
Evaluationen - im geom. Mittel	78408.44	**36075.39**
Expansionen - im geom. Mittel	61134.44	**15744.18**
Generierte Knoten - im geom. Mittel	379596.23	**99261.75**
Vorberechnungszeit - im geom. Mittel	**0.10**	1.02
Reine Suchzeit - im geom. Mittel	1.04	**0.36**
Gesamte Suchzeit - im geom. Mittel	1.04	**0.87**

Ergebnisse mit blinder Suche



Ergebnisse mit h^{max}

	h^{max}-**GBS**	h^{max}-**KBS**
Gelöste Probleme - Summe	**458**	416
Evaluationen - im geom. Mittel	**4208.74**	4442.11
Expansionen - im geom. Mittel	1388.40	**1331.32**
Generierte Knoten - im geom. Mittel	10007.14	**9354.75**
Vorberechnungszeit - im geom. Mittel	**0.10**	1.02
Reine Suchzeit - im geom. Mittel	0.33	**0.27**
Gesamte Suchzeit - im geom. Mittel	**0.33**	0.67

Ergebnisse mit h^{FF}

	h^{FF}-**GBS**	h^{FF}-**KBS**
Gelöste Probleme - Summe	**764**	492
Evaluationen - im geom. Mittel	**377.54**	1783.43
Expansionen - im geom. Mittel	**90.98**	535.90
Generierte Knoten - im geom. Mittel	**591.55**	3613.31
Vorberechnungszeit - im geom. Mittel	**0.10**	1.02
Reine Suchzeit - im geom. Mittel	**0.13**	0.17
Gesamte Suchzeit - im geom. Mittel	**0.13**	0.60

Verbesserungen

Berechnung der Interferenz-Distanzen

- on-the-fly während Suche
- Hohe Interferenz-Distanz approximieren

Priorisierung

- Umkehrung der Priorisierung

Danke für Ihre Aufmerksamkeit!

Q & A