Grundlagen der Künstlichen Intelligenz

14. Klassische Suche: IDA*

Malte Helmert

Universität Basel

31. März 2014

Klassische Suche: Überblick

Kapitelüberblick klassische Suche:

- 3.–5. Einführung
- 6.–9. Basisalgorithmen
- 10.-17. heuristische Algorithmen
  - 10. Heuristiken
  - 11. Analyse von Heuristiken
  - 12. Bestensuche als Graphensuche
  - 13. Gierige Bestensuche, A*, Weighted A*
  - 14. IDA*
  - 15. A*: Optimalität, Teil I
  - 16. A*: Optimalität, Teil II
  - 17. A*: Vollständigkeit und Komplexität

14.1 IDA*: Idee

IDA*

Der Hauptnachteil der vorgestellten Bestensuchalgorithmen ist ihr Speicheraufwand.

Idee: verfolge einen ähnlichen Ansatz wie bei der iterativen Tiefensuche

- beschränkte Tiefensuche mit steigenden Schranken
- statt der Tiefe beschränken wir f (in diesem Kapitel $f(n) := g(n) + h(n.\text{state})$, wie bei A*)

$\rightsquigarrow$ IDA* (iterative-deepening A*)

- ist eine Baumsuche, anders als die vorigen Bestensuchalgorithmen

14.2 IDA*: Algorithmus

Anmerkungen zum Algorithmus (1)

- Wir beschreiben eine IDA*-Implementierung mit expliziten Suchknoten.
- Effizientere Implementierungen lassen Suchknoten implizit, so wie bei der Tiefensuche in Kapitel 9.

Anmerkungen zum Algorithmus (2)

- Unsere rekursiven Aufrufe liefern zwei Werte:
  - *f_limit*, die nächste sinnvolle f-Schranke für den Teilbaum, der in dem Aufruf betrachtet wurde (oder **none**, wenn eine Lösung gefunden wurde)
  - *solution*, die gefundene Lösung (bzw. **none**)
- Effizientere Implementierungen speichern diese Werte in Instanzvariablen oder einer Closure, um Zeit für Werteübergabe zu sparen

IDA*: Pseudo-Code (Hauptprozedur)

IDA*: Hauptprozedur

n_0 := make_root_node()
f_limit := 0
while *f_limit* $\neq \infty$:
    $\langle$*f_limit*, *solution*$\rangle$:= recursive_search(n_0, *f_limit*)
    if *solution* $\neq$ **none**:
        return *solution*
return unsolvable

IDA*: Pseudo-Code (Tiefensuche)

function recursive_search(n, *f_limit*):
if $f(n) >$ *f_limit*:
    return $\langle f(n),$ **none**$\rangle$
if is_goal(n.state):
    return $\langle$**none**, extract_path(n)$\rangle$
next_limit := ∞
for each $\langle a, s' \rangle \in$ succ(n.state):
    if $h(s') < \infty$:
        n' := make_node(n, a, s')
        $\langle$*rec_limit*, *solution*$\rangle$:= recursive_search(n', *f_limit*)
        if *solution* $\neq$ **none**:
            return $\langle$**none**, *solution*$\rangle$
        next_limit := min(*next_limit*, *rec_limit*)
return $\langle$*next_limit*, **none**$\rangle$

14.3 IDA*: Eigenschaften

IDA*: Eigenschaften

Erbt wichtige Eigenschaften von A* und iterativer Tiefensuche:

- semi-vollständig, wenn h sicher
  und $cost(a) > 0$ für alle Aktionen a
- optimal, wenn h zulässig
- Speicheraufwand $O(\ell b)$, wobei
  - ℓ: Länge des längsten erzeugten Pfads
    (bei Einheitskosten: beschränkt durch optimale Lösungskosten)
  - b: Verzweigungsgrad

$\rightsquigarrow$ Beweise?

IDA*: Diskussion

- im Vergleich zu A* potenziell erheblicher Overhead, da keine Duplikate erkannt werden
  - ⇝ in vielen Zustandsräumen exponentiell langsamer
  - ⇝ oft mit teilweiser Duplikateliminierung kombiniert (Zykelerkennung, Transpositionstabellen)
- Overhead durch iteratives Anheben der f-Schranke ist oft vernachlässigbar, aber nicht immer
  - besonders problematisch, wenn Aktionskosten stark variieren: dann kann es leicht passieren, dass jede neue f-Schranke nur wenige neue Suchknoten erreicht

14.4 Zusammenfassung

Zusammenfassung

- IDA* ist eine Baumsuchvariante von A* basierend auf iterativer Tiefensuche
- Hauptvorteil: niedriger Speicheraufwand
- Nachteil: wiederholte Arbeit kann signifikant sein
- am nützlichsten, wenn es wenige Duplikate gibt (Duplikateliminierung würde Speichervorteil verlieren)