Alternative Programmierkonzepte (T3INF4271)

Logische Programmierung

05
Prolog
Graphensuche

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Antonius van Hoof

AVH 2021

„Uninformierte“ Traversierungsalgorithmen

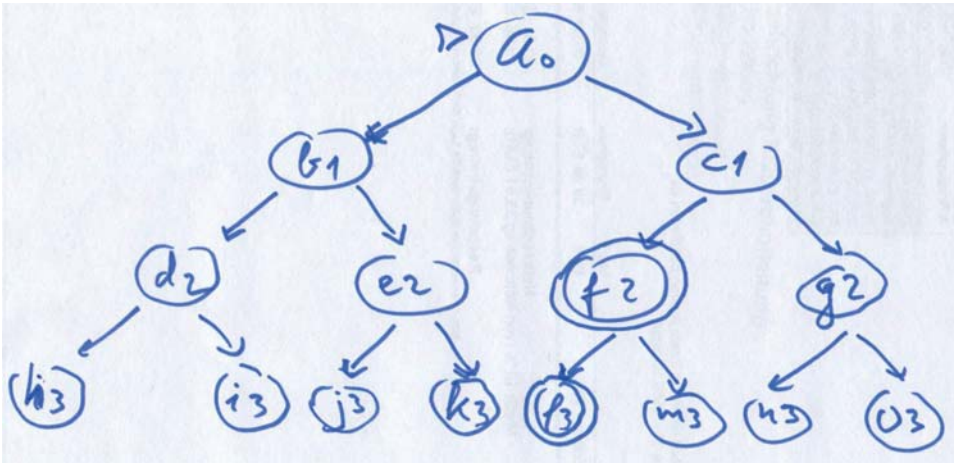
- **Depth First**
  Pass sehr natürlich zu Prologs Exekutionsmodell
- **Breadth First**
  Hierzu müssen alle Pfade in einem Argument mitgeführt werden
- **Bounded Search**
  Hier wird ein Zähler in der Tiefensuche mitgeführt, der ein vorgebenenes Limit (Bound) nicht überschreiten darf
  Wenn der Bound erreicht wird ohne Erfolg, wird die Suche abgebrochen
- **Iterative Deepening**
  Wie Bounded Search: In einer Tiefensuche (Depth First) wird bis zum Erfolg mit jeweils um eins erhöhten Bound gesucht (also Suche mit Bound 0,1,2,3,..,n)

Setup für uninformierte Suche

```
edge(a0,b1).
edge(a0,c1).
edge(b1,d2).
edge(b1,e2).
edge(d2,h3).
edge(d2,i3).
edge(e2,j3).
edge(e2,k3).
edge(c1,f2).
edge(c1,g2).
edge(f2,l3).
edge(f2,m3).
edge(g2,n3).
edge(g2,o3).

start(a0).
solution(f2).
solution(l3).

move(From,To) :- edge(From,To).
goal(State) :- solution(State).
```




```
% All examples will be shown around the simple framework:
%
% find_solution(X) :-
%     start(Start),
%     <search_algorithm>(Start,...,X,...),
%     solution(X).
%
```


DF unter implizite Benutzung von Prologs Backtracking

```
%%      simple_df_star(+Begin,-End) is nondet.
%
%       Compute in End a node in the transitive closure of Begin
%       using simple depth first strategy
%
simple_df_star(X,X).
simple_df_star(X,Z) :-
        move(X,Y),
        simple_df_star(Y,Z).
```

```
%%      simple_df_star(+Begin,-End,-Solution:list) is nondet.
%
%       Compute in End a node in the transitive closure of Begin using
%       simple depth first strategy and showing in Solution the path
%       from Begin to End. (Solution is NOT the actual graph traversal of
%       the algorithm).
%
simple_df_star(X,X,[]).
simple_df_star(X,Z,[Y|Ys]) :-
        move(X,Y),
        simple_df_star(Y,Z,Ys).
```

DF mit eigenem Stack

```
%%      df_star_open(+Open:list,-End) is nondet.
%
%       Compute in End a node in the transitive closure of the nodes in
%       Open, using a simple depth first strategy. Open is a list of
%       nodes (At the start the singleton list [Start]).
%
df_star_open([X|_],X).
df_star_open([X|Open1],Z) :-
        findall(Next,move(X,Next),NextL),
        append(NextL,Open1,Open2),
        df_star_open(Open2,Z).
```

```
%%      df_star_open(+Open:list,-Closed:list,-End) is nondet.
%
%       Compute in End a node in the transitive closure of the nodes in
%       Open, using a simple depth first strategy and showing in Closed
%       the complete search path taken from the Head of Open to End in the
%       search for End (This is NOT the solution path from Head to End).
%
df_star_open([X|_],[],X).
df_star_open([X|Open1],[X|Ys],Z) :-
        findall(Next,move(X,Next),NextL),
        append(NextL,Open1,Open2),
        df_star_open(Open2,Ys,Z).
```


Breitesuche mit Queue

```
%%      simple_bf_star(+OpenPaths:list,-ClosedNodes:list,-End,-Solution:list)
%%      is nondet.
%
%       call it with OpenPaths instantiated to [[StartNode]].
%
%       Compute in End a node in the transitive closure of the nodes in
%       Open, using a simple breadth first strategy and showing in Path the
%       complete path taken from the Head of Open to End in the search for
%       End (most often this is not the shortest path from Head of Open to
%       End).
%
simple_bf_star([[X|Path]|_],[],X,[X|Path]).
simple_bf_star([[X|Path]|Open1],[X|Ys],Z,Sol) :-
        findall([Next,X|Path],move(X,Next),NextPaths),
        append(Open1,NextPaths,Open2),
        simple_bf_star(Open2,Ys,Z,Sol).
```

Iterative DF

```
%%          bounded_df_star_open(+Bound:int,+OpenPaths:list,-ClosedNodes:list,-End,-Solution:list) is nondet.
%
%           As df_star_open, but with a Bound (should be >= 0) on the allowed
%           search depth.
%
%           call it with OpenPaths instantiated to [[StartNode]].
%
bounded_df_star_open(_,[[X|Path]|_],[],X,[X|Path]).
bounded_df_star_open(N,[[X|Path]|Open1],[X|Ys],Z,Sol) :-
        length(Path,NN), NN =< N,
        findall([Next,X|Path],move(X,Next),NextPaths),
        append(NextPaths,Open1,Open2),
        bounded_df_star_open(N,Open2,Ys,Z,Sol).
```

```
%% Merke auf, dass wir in dem Prädikat iterative_df_star_open/5
% das Argument Bound einfach weglassen können: Es berichtet lediglich in
% welcher Tiefe eine Lösung gefunden wurde. Falls man Bound beim Aufruf
% instanzieren würde, prüft es lediglich ob es in der Tiefe eine Lösung
% gibt. Achtung: falls nicht, dann verschwindet die Suche ins Nirwana
% des Stack Overflow! (Wie kann man das einfach reparieren?)
%
% Weiterer Grund auf Bound zu verzichten: Solution ist ein Pfad und der
% Pfad hat eine Länge die direkt mit dem Bound korrespondiert!
```

```
%%      iterative_df_star_open(+Begin,-End,-Bound,-PathLength:int,-Solution:list)
%       is nondet.
%
%       Using bounded_df_star_open/5 to find a path from Begin to End of
%       length PathLength by iterative deepening in a depth first search
%       strategy
%
iterative_df_star_open(X,Z,Bound,Closed,Solution) :-
        iterative_df_star_open(0,[[X]],Z,Bound,Closed,Solution).

iterative_df_star_open(Bound,X,Z,Bound,Closed,Solution) :-
        bounded_df_star_open(Bound,X,Closed,Z,Solution).
iterative_df_star_open(Bound,X,Z,NewBound,Closed,Solution) :-
        Bound2 is Bound + 1,
        iterative_df_star_open(Bound2,X,Z,NewBound,Closed,Solution).
```


Bemerkung: iterative BF geht natürlich auch! (Wofür wäre das ggf. sinnvoll?)

```
%%          bounded_bf_star_open(+Bound:int,+OpenPaths:list,-ClosedNodes:list,-End,-Solution:list)
%%          is nondet.
%
%           As bounded_df_star_open, but breadth first search with a Bound
%           (should be >= 0) on the allowed search depth.
%
%           call it with OpenPaths instantiated to [[StartNode]].
%
bounded_bf_star_open(_,[[X|Path]|_],[],X,[X|Path]).
bounded_bf_star_open(N,[[X|Path]|Open1],[X|Ys],Z,Sol) :-
        length(Path,NN), NN =< N,
        findall([Next,X|Path],move(X,Next),NextPaths),
        append(Open1,NextPaths,Open2),
        bounded_bf_star_open(N,Open2,Ys,Z,Sol).
```

```
%%          iterative_bf_star_open(+Begin,-End,-Bound,-PathLength:int,-Solution:list)
%%          is nondet.
%
%           Using bounded_bf_star_open/5 to find a path from Begin to End of
%           length PathLength by iterative deepening a breadth first strategy
%
iterative_bf_star_open(X,Z,Bound,Closed,Solution) :-
        iterative_bf_star_open(0,[[X]],Z,Bound,Closed,Solution).

iterative_bf_star_open(Bound,X,Z,Bound,Closed,Solution) :-
        bounded_bf_star_open(Bound,X,Closed,Z,Solution).
iterative_bf_star_open(Bound,X,Z,NewBound,Closed,Solution) :-
        Bound2 is Bound + 1,
        iterative_bf_star_open(Bound2,X,Z,NewBound,Closed,Solution).
```

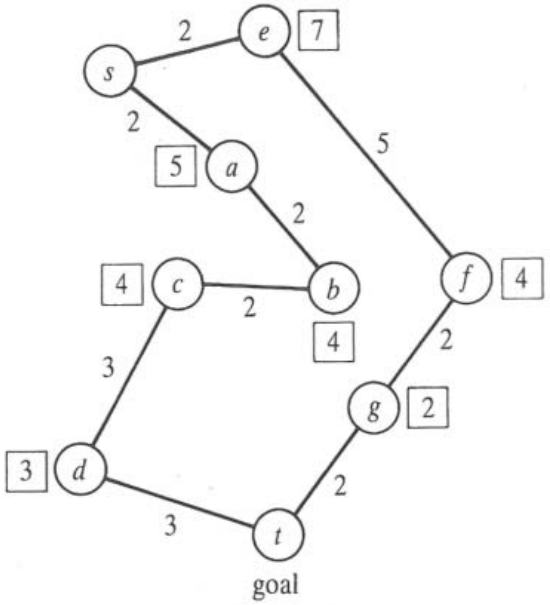
WARNUNG!!!!

```
/*
 * Die obigen Prädikaten funktionieren nur für DAGs (d.h. Bäume). Für
 * vollen Graphen braucht es:
 *
 * (1) das mitführen der besuchten Knoten
 * (hier als Closed, bereits in einigen Prädikaten vorhanden) und
 *
 * (2)
 * die Prüfung ob eine neu hinzu zu fügendene Knote nicht bereits
 * besucht wurde. Machen!
 *
 */
```


„Heuristische“ Suche

- Hierbei wird die Suche durch den Graphen geleitet von Informationen die an Knoten oder Kanten vorhanden (oder Errechenbar) ist
- Ein sehr bekannter Algorithmus ist der ***A*-Algorithmus***
- **A*** führt zu einen modifizierten Breitensuche, wobei jeweils der am meisten Erfolg versprechenden Weg verfolgt wird (Errechnet aus reale Kosten und weiter zu erwartenden Kosten), bis man Erfolg hat oder ein anderer Weg wieder mehr Erfolg verspricht
- **A*** kann jedoch auch mittels Iterative Deepening implementiert werden Vorteil: kein exponentieller Platzverbrauch

Beispiel für A* nach Bratko



Source: Bratko, *Prolog Programming for AI*

```
move(s, a, 2).      estimate(a, 5).
move(a, b, 2).      estimate(b, 4).
move(b, c, 2).      estimate(c, 4).
move(c, d, 3).      estimate(d, 3).
move(d, t, 3).      estimate(e, 7).
move(s, e, 2).      estimate(f, 4).
move(e, f, 5).      estimate(g, 2).
move(f, g, 2).
move(g, t, 2).      estimate(s, 100).
goal(t).            estimate(t, 0).
```


Literatur

- **Für diese Vorlesungsinhalt greifen wir zu auf**

- Ulle Endriss
  Universität von Amsterdam
- Vorlesung Prolog
- https://staff.science.uva.nl/u.endriss/teaching/prolog/
- Kapitel 8 Search, Lecture 10

Endriss behandelt auch in dem Kapitel auch noch:
- Lecture 11: adversarial search with MiniMax
- Lecture 12: Alpha-Beta Pruning and Heuristic Evaluation

All dies ist in Prolog leicht umsetzbar.

A* in Prolog

```
% Users of this algorithm will have to implement the following
% application-dependent predicates themselves:

%    * move(+State,-NextState,-Cost).
%      Given the current State, instantiate the variable NextState
%      with a possible follow-up state and the variable Cost with the
%      associated cost (all possible follow-up states should get
%      generated through backtracking).

%    * goal(+State).
%      Succeed if State represents a goal state.

%    * estimate(+State,-Estimate).
%       Given a State, instantiate the variable Estimate with an
%      estimate of the cost of reaching a goal state. This predicate
%      implements the heuristic function h.

% Further down this file/on the previous slice, you can find an example definition of these
% predicates, taken from Bratko's book Prolog Programming for AI.
```


The "User Interface"

```
% Now we are not only going to maintain a list of paths (as in
% breadth-first search, for instance), but a list of (reversed) paths
% labelled with the current cost g(n) and the current estimate h(n):
%      General form: Path/Cost/Estimate
%      Example: [c,b,a,s]/6/4
% Our usual "user interface" initialises the list of labelled paths with
% the path consisting of just the initial node, labelled with cost 0 and
% the appropriate estimate:

solve_astar(Node, Path/Cost) :-
            estimate(Node, Estimate),
            astar([[Node]/0/Estimate], RevPath/Cost/_),
            reverse(RevPath, Path).

% That is, for the final output, we are not interested in the estimate
% anymore, but we do report the cost of solution paths.
```

Moves

```
% The following predicate serves as a "wrapper" around the move/3
% predicate supplied by the application developer:

move_astar([Node|Path]/Cost/_,  [NextNode,Node|Path]/NewCost/Est) :-
        move(Node, NextNode, StepCost),
        \+ member(NextNode, Path),
        NewCost is Cost + StepCost,
        estimate(NextNode, Est).

% After calling move/3 itself, the predicate (1) checks for cycles,
% (2) updates the cost of the current path, and (3) labels the new
% path with the estimate for the new node.
% The predicate move_astar/2 will be used to generate all
% expansions of a given path by a single state:

expand_astar(Path, ExpPaths) :-
        findall(NewPath, move_astar(Path,NewPath), ExpPaths).
```


Getting the best path

```
% The following predicate implements the search strategy of A*: from
% a list of labelled paths, we select one that minimises the sum of the
% current cost and the current estimate.

get_best([Path], Path) :- !.
get_best([Path1/Cost1/Est1,   _/Cost2/Est2 | Paths],  BestPath) :-
        Cost1 + Est1 =< Cost2 + Est2,
        !,
        get_best([Path1/Cost1/Est1 | Paths], BestPath).
get_best([_|Paths], BestPath) :-
        get_best(Paths, BestPath).

% Remark: Implementing a different bestfirst search algorithm only
% involves changing get_best/2; the rest can stay the same.
```

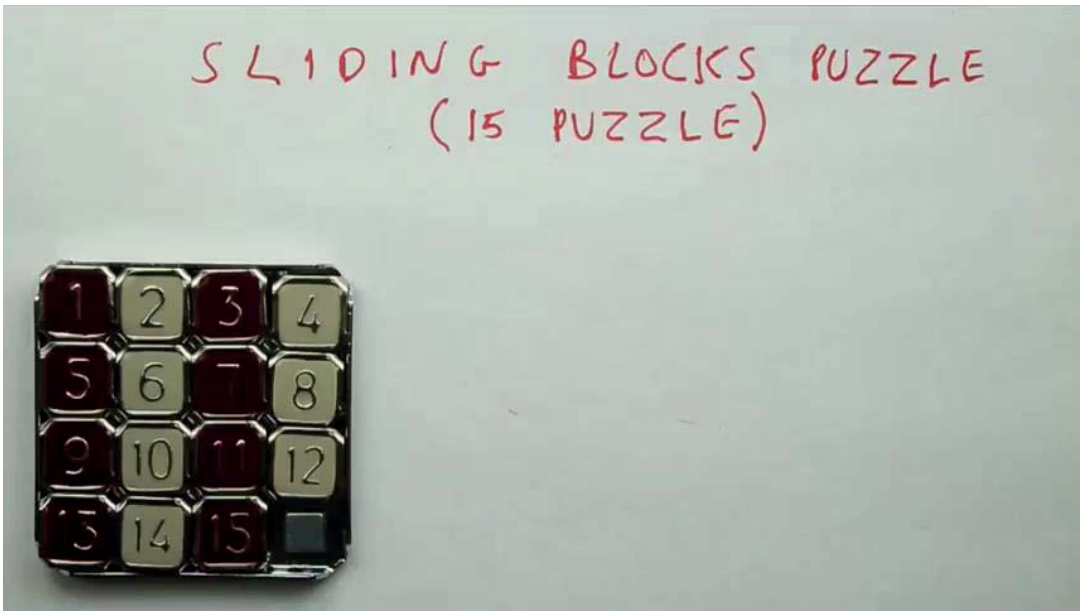
The main algorithm (called from the "user interface")

```
% Stop in case the best path ends in a goal node:
astar(Paths, Path) :-
        get_best(Paths, Path),
        Path = [Node|_]/_/_,
        goal(Node).
% Otherwise, extract the best path, generate all its expansions, and
% continue with the union of the remaining and the expanded paths:
astar(Paths, SolutionPath) :-
        get_best(Paths, BestPath),
        select(BestPath, Paths, OtherPaths),
        expand_astar(BestPath, ExpPaths),
        append(OtherPaths, ExpPaths, NewPaths),
        astar(NewPaths, SolutionPath).
```


IDA*, die Tiefensuchevariante von A*

- **Vorherige Behandlung von A* ging aus von einer Queue, ähnlich wie in der Breitensuche**
  - Queue wird gewandelt in eine Priority Queue
    - Durch get_best/2 in astar/2
- **Es ist aber möglich, A* ebenso in einer iterierten Tiefensuche zu realisieren → IDA***
  - Liefert das Beste aus beiden Welten:
    - Eine optimale Lösung (falls eine existiert)
    - Bei minimaler Verbrauch an Speicher
      - Wir verwenden kein explizites Argument für die Open-Liste, sondern verwenden den impliziten Prolog Backtracking-Mechanismus
    - Dadurch können "größere" Probleme gelöst werden als bei A* mit expliziten Queue
- **Behandeln wir hier nicht mehr ausführlich → Selbststudium**
  - Sehen Sie die Beispielimplementierung in *graph_search_ida-star.pl*

Aufgabe 4: Schiebepuzzle



https://classroom.udacity.com/courses/cs271/lessons/48678875/concepts/487015090923

Alternative Programmierkonzepte (T3INF4271)

Logische Programmierung

06
Prolog
Definite Clause Grammars
DCG

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Antonius van Hoof

DCGs

- Prolog ist sehr geeignet für das Scannen und Parsen von Eingaben
- In Prolog ist standardmäßig die Möglichkeit eingebaut, Definite Clause Grammars (DCG), kontextfreie Grammatiken mit semantischen Aktionen zu beschreiben
- Daraus bildet Prolog dann selbst einen Recursive Descent Parser (Top-down Parser mit Backtracking)
- Zu bedenken ist, dass diese Art von Parser nicht mit Linksrekursion umgehen kann (Führt zu Endlosschleife). Diese ist in der Grammatik zuerst zu beseitigen (Wie man sowas macht lernen Sie in der Vorlesung Formale Sprachen und Compilerbau)

- Mechanismus ist geeignet u.a. für Natural Language Processing, aber auch für die Formulierung von Prolog Sprachmakros und vieles mehr…

DCGs sind eine schlaue Anwendung von Difference Listen

https://www.youtube.com/watch?v=6egAF4-HVzw

Erste Annäherung

- **Für die Erklärung und Behandlung von DCGs verwenden wir ein Skript von Temur Kutsia, Uni Linz**
  https://www3.risc.jku.at/education/courses/ws2019/logic-programming/slides/Chapter9.pdf

- **Als Beispiel für die Sprachverarbeitung gibt es zwei Prologbeispielprogramm:**
  - `parser_fliegen.pl`:
    Ein Parser für eine lustige Untermenge der Deutschen Sprache
  - `parser_aussagenlogik.pl`:
    Ein Scanner und Parser für Aussagenlogik.

Aber DCGs sind mächtiger!

- **Nicht nur für das Parsen von Input,**
  sondern auch das Generieren von Output!
  - Verwendung in umgekehrter Richtung (Ist Prolog nicht toll?!!!)
- **Nicht nur für (natürliche) Sprachkonstrukte sondern allgemein**
- **Siehe ausführliche Beschreibung in:**
  https://www.metalevel.at/prolog/dcg
- **Tutorial zu DCGs:**
  https://github.com/Anniepoo/swipldcgtut/blob/master/dcgcourse.adoc
  Using Definite Clause Grammars in SWI-Prolog

Sonstige Anwendungsbeispiele von DCGs

```
%
% Towers of Hanoi
%
% DCG Version (funktioniert in allen Richtungen)
%

move(0,_,_,_) --> {}, [].
move(N,Start,Aux,Goal) -->
   { N #> 0,
     NN #= N-1},
   move(NN,Start,Goal,Aux),
   [disk_from_to(Start,Goal)],
   move(NN,Aux,Start,Goal).


hanoi(N,Start,Aux,Goal,Moves) :-
   phrase(move(N,Start,Aux,Goal),Moves),
   maplist(writeln,Moves),
   writeln('FERTIG!!!\n').
```


DCGs for Pure Input

https://www.youtube.com/watch?v=Dqpxy4W7fAo

DCGs for State Space Problems

https://www.youtube.com/watch?v=vdabv9EkYrY