Applications of Artificial Intelligence

Sebastian Iwanowski
FH Wedel

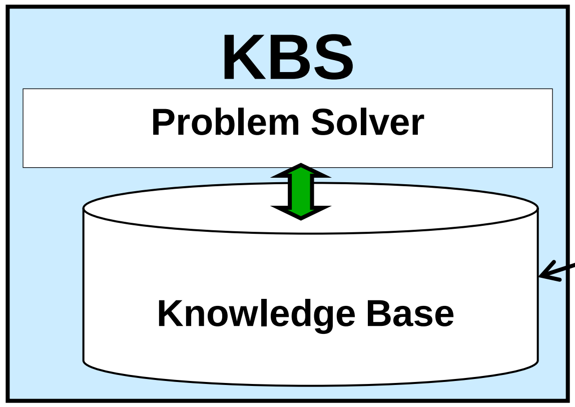
Chapter 3:
Algorithmic Methods of AI

Search Strategies

Relevance of search strategies for logic problems:

Search for a solution of the satisfiability problem

Relevance of search strategies for knowledge-based systems:



if the knowledge is not case-based

The problem solver nearly always has to solve a satisfiability problem for constraints of the knowledge base!

➔ All problem solvers search

Example for a knowledge-based search engine: PROLOG

PROLOG is knowledge-based:

- **Knowledge base**

  Facts and rules, dynamically extensible

- **Inference engine („Problem Solver“)**

  deriving facts and rules automatically

- **Dialog component**

  Input: Query
  Output: yes / no, specification of unification applied in case of success,
  write as a „side effect“

  Yes: The predicate of the query can be concluded from knowledge base.
  No: The predicate of the query cannot be concluded from knowledge base.
  „No“ does not imply that it can be concluded that the predicate is false.

Application: Class Scheduling

Given finite sets Courses**, R**ooms**, T**ime slots

Task: Generate an injective (one-to-one) function C $\rightarrow$ RxT

Strict Constraints (must be fulfilled in any case):

- **Certain courses must not take place at the same time.**
- **For some courses, certain time slots are not admitted.**
- **For some courses, certain rooms are not admitted.**

Soft constraints (may be violated):

- **Certain courses should not take place at some times.**
- **Certain courses should take place successively.**
- **Certain courses should not take place on the same day.**

Optimisation function:

- **fewest violations of soft criteria**
- **fewest free periods for certain study programmes**
- **most uniform distribution on different days for ...**

Application: Traveling Salesman Problem (TSP)

Given: Graph with node set V and weighted edges between the nodes

Task: Find a round trip traversing the graph edges reaching each node at least once.

Constraints:

- **Only edges of the graph are to be used.**

Optimisation function:

- **Minimise the global edge costs !**

Generalisation in logistic applications:

Constraints:

- **Load and destribute goods obeying capacity restrictions !**
- **Consider time windows in which delivery may take place !**

Soft criteria (may be violated):

- **Certain edges have to be avoided.**
- **Certain time windows are unfavourable.**

Application: Shortest Path Problem

Given: Graph with node set V and weighted edges between the nodes

Task: For two selected nodes S and T, find a path through the graph.

Constraints:

- **Only edges of the graph are to be used.**

Optimisation function:

- **Minimise the global edge costs !**

Generalisation in transport applications (public or individual):

Constraints:

- **Edge costs depend on the time used.**
- **Travelors are subject to individual contraints that may value certain edges in a different way or make them even unusable.**

Soft criteria (may be violated):

- **Certain edges have to be avoided**
- **Certain time windows are unfavourable**

Constraint Satisfaction Problem (CSP)

Specification of a CSP:

- **set of variables**
- **domains of definition**
- **constraints: relations between variables (strict or soft)**
  (nomally, equations or inequalities)
- **optimisation criterion**
  (normally, a real-valued function on the variables which has to be minimised or maximised)

valid solution:

assignment of values to all variables such that all strict constraints are satisfied

optimal solution:

valid solution optimising the optimisation criterion

Constraint Solvers are programs which find a valid or even optimal solution for a given CSP automatically.

Traversing search graphs

1. search method: Find a global solution via partial solutions

- **Node: describes state in search domain**
  - **State: Assigning values to variables**

    Each state has got an evaluation.
- **Edge: transition of a state into a subsequent state**
  (usually feasible in one direction only)
  - **Subsequent state: Assign a value to a new variable keeping the values for the already assigned variables**
- **Initial node: initial state**
  (is always unique)
  - **Initial node: No variable has got a value.**
- **Final node: final state wanted (problem solution)**
  (several ones are admissible)
  - **Final node: All specified variables have got admissible values.**

Traversing search graphs

Different search goals are possible:

1) **Find some solution or detect that there is none.**
2) **Find further solutions or detect that there are none.**
3) **Find all solutions.**
4) **Find an optimal solution or at least a rather good one.**

- **Expansion of a node: Compute all subsequent resp. adjacent nodes**

Different search strategies differ in:

Which node has to be expanded next?

Special case:

- **Search graph is a search tree**

(makes the path from initial node to each final node unique)

Example for search trees in CSP

Constraint system:

```
1)  (2 < x < 4)
2)  (0 < y < 6)
3)  (x + y > 7)
4)  (x · y < 10.5)
```

Domain of definition for valid solutions:

x,y ∈ **Q,**
at most k positions after the decimal point

Optimisation criterion:

Minimise |y – x|

for bounded k:

- finite search space
- several valid solutions
- always 1 optimal solution

for unbounded k:

- infinite search space
- infinitely many valid solutions
- no optimal solution

Example for search trees in CSP

Constraint system:

```
1)  (2 < x < 4)
2)  (0 < y < 6)
3)  (x + y > 7)
4)  (x · y < 10.5)
```

Domain of definition for valid solutions:

$x,y \in \mathbf{Q}$,
at most k positions after the decimal point

Optimisation criterion:

Minimise $|y - x|$

CSP variables:

- Assign values to the 8 variables x_0, x_1, x_2, x_3 and y_0, y_1, y_2, y_3 where $x = x_0 . x_1 x_2 x_3$ and $y = y_0 . y_1 y_2 y_3$ and x_i and y_i are the respective decimal digits (integer numbers between 0 and 9).

Nodes and successor definitions:

- Each node in the search network assigns either the same number of digits for x as for y with values (type 1) or one digit more for y than for x (type 2).
- A successor of type 1 is a node of type 2 with the same values as the predecessor plus one more value for a digit for y.
- A successor of type 2 is a node of type 1 with the same values as the predecessor plus one more value for a digit for x

Example for search trees in CSP

Constraint system:

```
1) (2 < x < 4)
2) (0 < y < 6)
3) (x + y > 7)
4) (x · y < 10.5)
```

Domain of definition for valid solutions:

x,y ∈ **Q,**
at most k positions after the decimal point

Optimisation criterion:

Minimise |y – x|

Nodes and successor definitions:

- Each node in the search network assigns either the same number of digits for x as for y with values (type 1) or one digit more for y than for x (type 2).
- A successor of type 1 is a node of type 2 with the same values as the predecessor plus one more value for a digit for y.
- A successor of type 2 is a node of type 1 with the same values as the predecessor plus one more value for a digit for x

depends on good luck

Optimum expansion strategy for this problem:

- The initial node assigns x_0=2, y_0=4 (type 1). All other digits are not yet assigned.
- Expand the initial node and the successors such that you come to the optimal solution (x=2.176, y=4.825) fastest possible.

Uninformed Search Strategies

In general, only *blind (uninformed) search* is possible:

There is no information about good search search directions (the target is only recognised on arrival)

Possible expansion strategies:

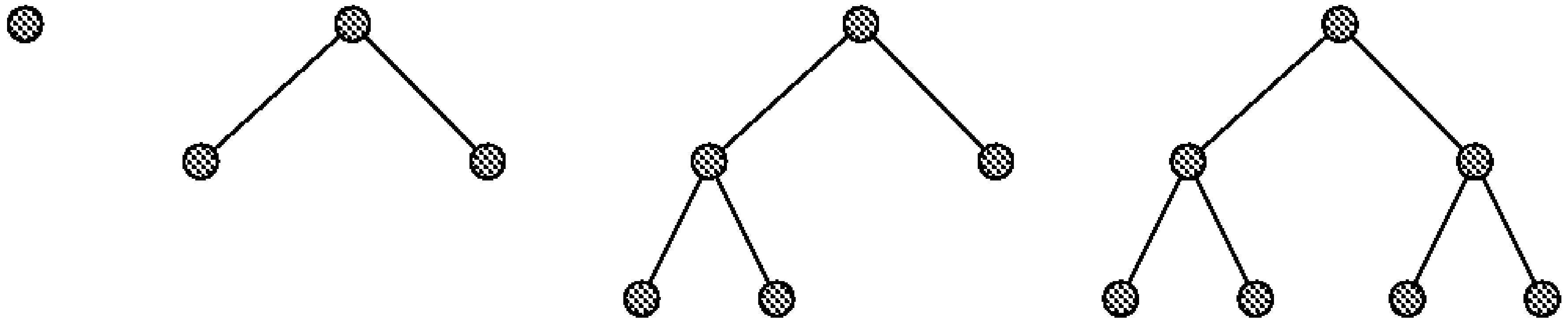
- Valid nodes are expanded first.
- The rightmost valid node on the next level is expanded.
- ...

Systematic search strategies:

1. breadth first search
2. depth first search
3. best first search

Uninformed Search Strategies

breadth first search:



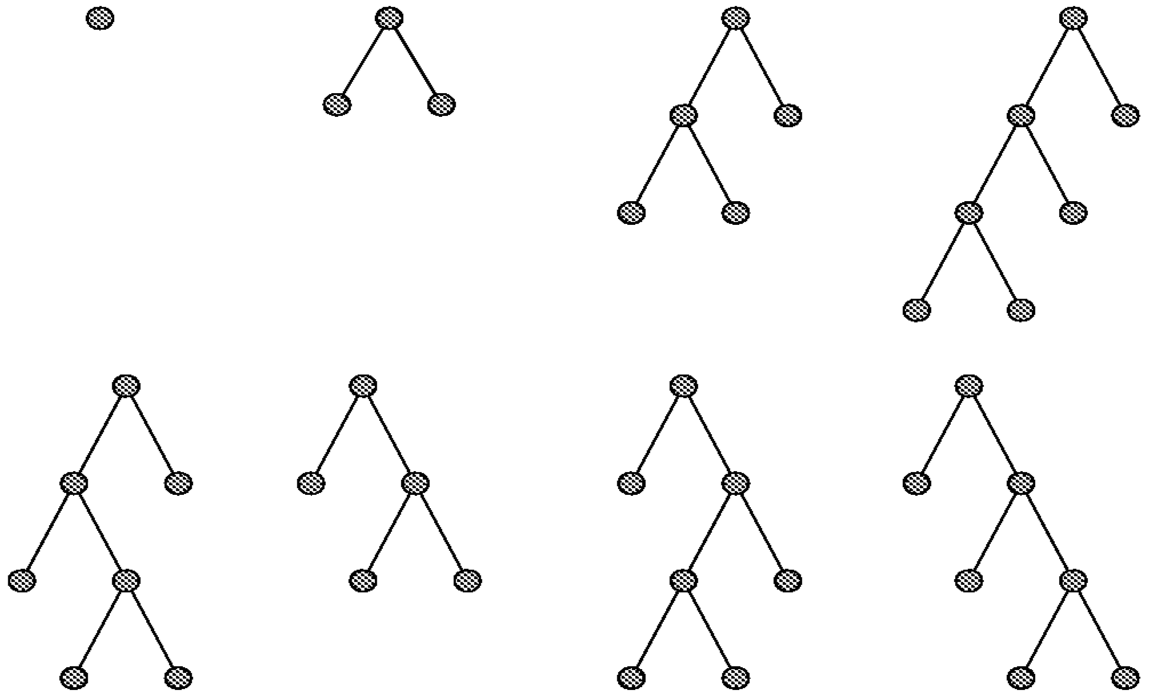
problem size: depth of search tree

Exponential time and space

for AI search procedures not relevant in most cases

Uninformed Search Strategies

depth first search:



problem size: depth of search tree

Exponential time

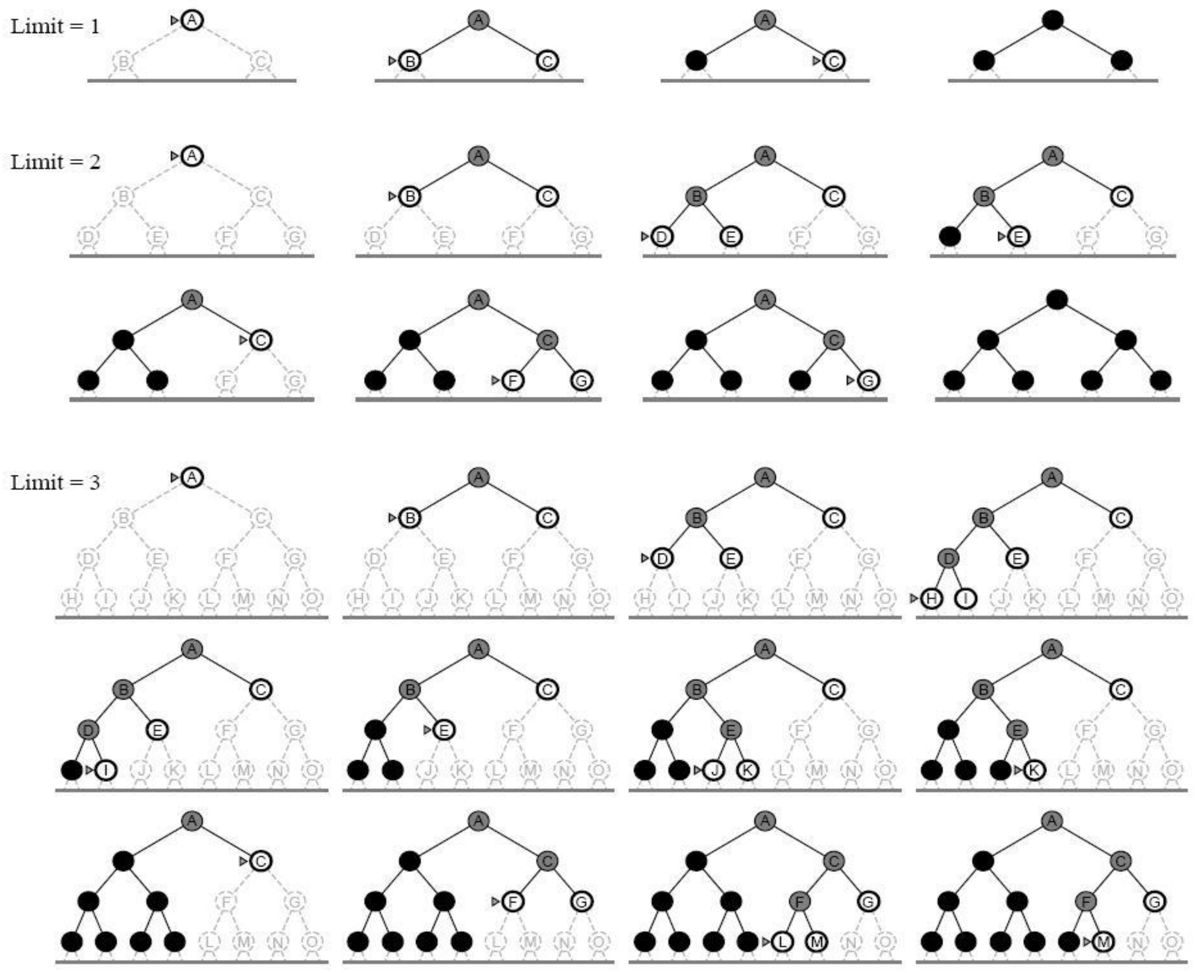
Linear space

The „normal case“ for standard AI procedures

Uninformed Search Strategies

bounded depth first search:

- Execute depth first search only up to limited search level.
- If not successful, increase limit for search level and start depth first search again.



Uninformed Search Strategies

best first search:

- **Additional information: Evaluation label for the nodes.**
- **Search target: Find the best solution first (and the others later).**
- **Expand the node with best evaluation first.**

→ *Mixture of depth first and breadth first searches*

In the *worst case* this is no better than breadth first search:

Exponential effort for time and space

Problem size:

Depth of search tree

For good evaluation functions, *the avarage case* is much better!

For special problems, even the worst case is much better:

Example: Special case „Shortest Path Problem":

Dijkstra's algorithm (quadratic effort for time, linear for space)

Problem size: Number of nodes

Uninformed Search Strategies

Dijkstra‘s algorithm for weighted graphs

(special case of best first search)

For all edges (u,v) there is a weight function:
length (u,v) := length of an edge from node u to node v

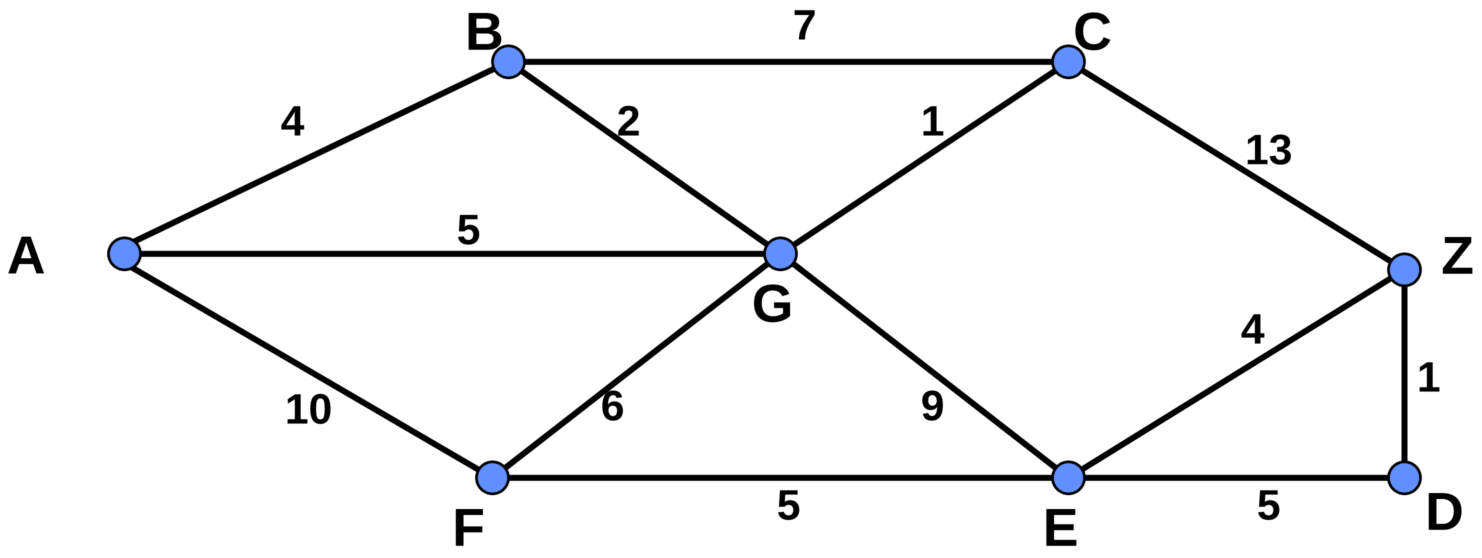
Requirement for edge weights: All lengths have to be nonnegative.

Algorithm for the search of a path from S to T having minimal global edge length:

```
• Put S into the set Done. Label S by distance(S) := 0.
  Put all other nodes into the set YetToCompute.
  Label all neighbors N of S by distance (N) := length (S,N)
  and all other nodes by distance (V) := ∞.

• Repeat:
      Choose node V from YetToCompute with minimum distance (V)
                and shift V to the set Done.
      Update all neighbors N of V that are still in YetToCompute:
          distance (N) := min {distance (N), distance (V) + length (V,N)}.
  until V = T
```

Proposition: This algorithm expands all nodes with a path length shorter than to T.

Example for Dijkstra‘s algorithm



Shortest path from G to Z: G → E → Z (13 units)

Node (distance from G, direct predecessor):

A(5,G)		A(5,G)		$\boxed{\text{A(5,G)}}$						
B(2,G)		$\boxed{\text{B(2,G)}}$								
$\boxed{\text{C(1,G)}}$										
D(∞)	→	D(∞)	→	D(∞)	→	D(∞)	→	D(∞)	→	D(14,E)
E(9,G)		E(9,G)		E(9,G)		E(9,G)		$\boxed{\text{E(9,G)}}$		
F(6,G)		F(6,G)		F(6,G)		$\boxed{\text{F(6,G)}}$				
Z(∞)		Z(14,C)		Z(14,C)		Z(14,C)		Z(14,C)		$\boxed{\text{Z(13,E)}}$

Informed (Heuristic) Search Strategies

Given the following kind of information for weighted graphs:

Distance function h(state) being an *estimated* measure for the real distance to the target

- easily computable
- but accurate enough not to lead the search procedure to the wrong target

h() provides a nonnegative value: The smaller the value, the closer the target

Application: „Hill climbing“

- Informed add-on to **depth first search**:
- Among the possible candidates, expand the node with best heuristic value.
- In case of backtracking expand the next best node respectively.

Main problem: Long halt in local maxima

Informed (Heuristic) Search Strategies

Given the following kind of information for weighted graphs:

Distance function h(state) being an *estimated* measure for the real distance to the target

- easily computable
- but accurate enough not to lead the search procedure to the wrong target

h() provides a nonnegative value: The smaller the value, the closer the target

Application: Optimistic hill climbing

- Special case of informed add-on to **depth first search**
- Expand only the node with best heuristic value.
- Backtracking is omitted: If heuristic value was wrong, the best result will not be found.

Main problem: Getting stuck in local maxima

Informed (Heuristic) Search Strategies

Given the following kind of information for weighted graphs:

Distance function h(state) being an *estimated* measure for the real distance to the target

- easily computable
- but accurate enough not to lead the search procedure to the wrong target

h() provides a nonnegative value: The smaller the value, the closer the target

Application: A* algorithm

- Informed add-on to **best first search**
- Expand the node where the sum of node label **plus** heuristic function is minimum.

Weitere Infos für die Anwendung von A* in öffentlichen Verkehrsnetzen:
Seminarvortrag und Ausarbeitung von Stefan Görlich, SS 2005, Nr. 5
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

Informed (Heuristic) Search Strategies

A* algorithm for weighted graphs

(Generalisation of Dijkstra's algorithm) *(State evaluation = Node evaluation)*

Requirement for edge weights: All edge lengths must be nonnegative.

Requirement for heuristic function $h_T(u)$ for estimating the real distance $d_T(u)$ to target node T**:**

Admissibility: $h_T(u) \leq d_T(u)$

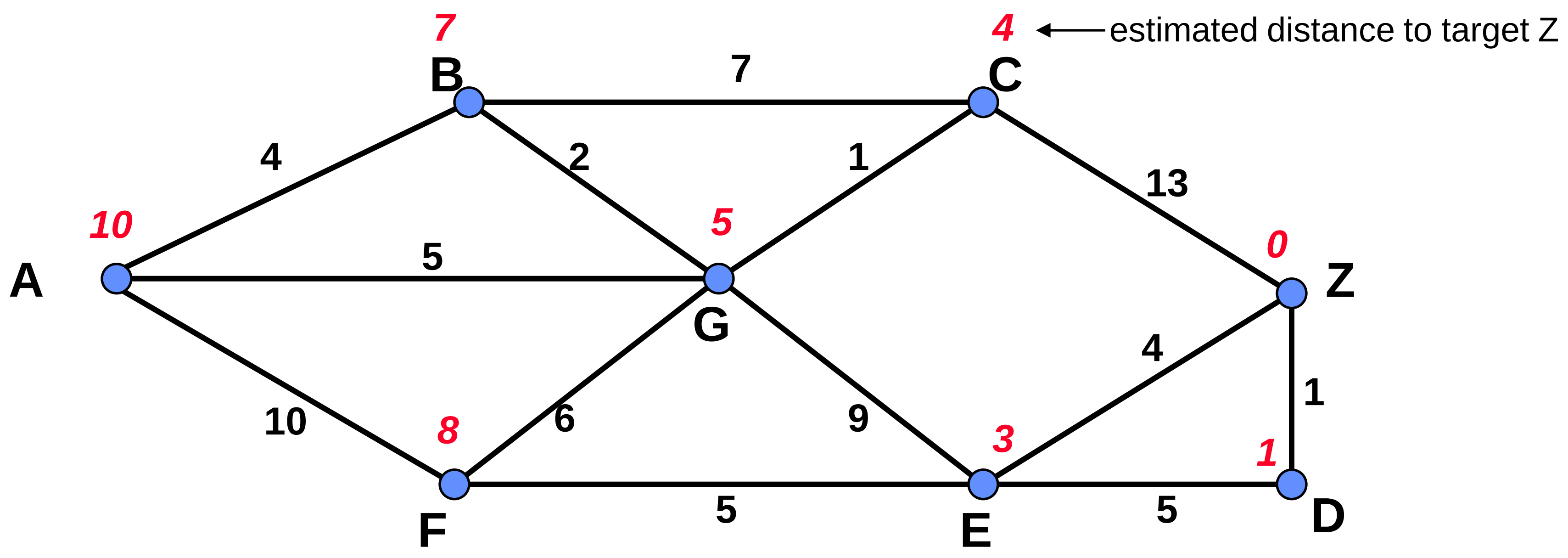
Monotonicity: $h_T(u) \leq h_T(v) + \mathit{length}(u,v)$

Algorithm for the search of a path from S to T having minimal global edge length:

```
•   Put S into the set Done. Label S by distance(S) := 0.
    Put all other nodes into the set YetToCompute.
    Label all neighbors N of S by distance (N) := length (S,N) and
                          estimatedTotal (N) := distance (N) + $h_T$(N)
    and all other nodes by distance (V) := ∞ and estimatedTotal (V) := ∞.

•   Repeat:
        Choose node V from YetToCompute with minimum estimatedTotal (V)
                      and shift V to the set Done.
        Update all neighbors N of V that are still in YetToCompute:
            distance (N) := min {distance (N),  distance (V) + length (V,N)}.
            estimatedTotal (N) := distance (N) + $h_T$(N)  (if update is necessary).
    until V = T
```

Proposition: This algorithm expands all nodes with an estimatedTotal shorter than to T.

Example for A* algorithm



Shortest path from G to Z: G → E → Z (13 units)

Node (real distance from G, direct predecessor, estimated total to target):

A(5,G,15)		A(5,G,15)		A(5,G,15)		A(5,G,15)
B(2,G,9)		$\boxed{\text{B(2,G,9)}}$				
$\boxed{\text{C(1,G,5)}}$						
D(∞)	→	D(∞)	→	D(∞)	→	D(14,E,15)
E(9,G,12)		E(9,G,12)		$\boxed{\text{E(9,G,12)}}$		
F(6,G,13)		F(6,G,14)		F(6,G,14)		F(6,G,14)
Z(∞)		Z(14,C,14)		Z(14,C,14)		$\boxed{\text{Z(13,E,13)}}$

Informed (Heuristic) Search Strategies

A* algorithm for weighted graphs

(Generalisation of Dijkstra's algorithm)

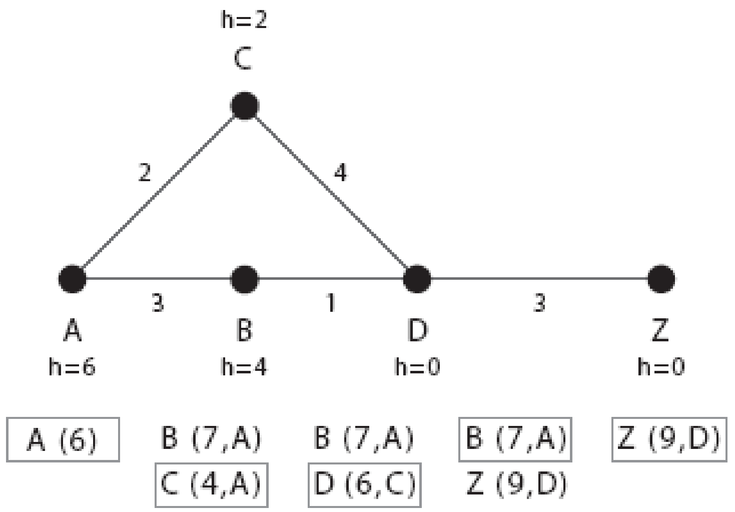
Requirement for edge weights: All edge lengths must be nonnegative.

Requirement for heuristic function $h_B(u)$ for estimating the real distance $d_B(u)$ to target node B**:**

Admissability: $h_B(u) \leq d_B(u)$

What happens if monotonicity is abandoned ? $h_B(u) \leq h_B(v) + \textit{length}(u,v)$

Example:



Aus: Diplomarbeit Andre Keller (SS 2008)

Error: D will not be updated anymore because it is already in **`Done`**

Informed (Heuristic) Search Strategies

A* algorithm for weighted graphs

(Generalisation of Dijkstra‘s algorithm) ***(State evaluation = Node evaluation)***

Requirement for edge weights: All edge lengths must be nonnegative.

Requirement for heuristic function $h_T(u)$ for estimating the real distance $d_T(u)$ to target node T**:**

Admissability only: $h_T(u) \le d_T(u)$

Algorithm for the search of a path from S to T having minimal global edge length:

```
•  Put S into the set Done. Label S by distance(S) := 0.
   Put all other nodes into the set YetToCompute.
   Label all neighbors N of S by distance (N) := length (S,N) and
                       estimatedTotal (N) := distance (N) + h_T(N)
   and all other nodes by distance (V) := ∞ and estimatedTotal (V) := ∞.

•  Repeat:
      Choose node V from YetToCompute with minimum estimatedTotal (V)
                and shift V to the set Done.
      Update all neighbors N of V from Done and YetToCompute:
         distance (N) := min {distance (N),  distance (V) + length (V,N)}.
         estimatedTotal (N) := distance (N) + h_T(N)  (if update is necessary).
         If an update occurred to a neighbor N* of Done: Shift N* back to YetToCompute
   until V = T
```

Pruning search space strategies

For the 1. search method introduced so far: Approaching global solutions via partial solutions:

Any strategy must backtrack to earlier assignment stages in the search tree when no solution can be found with the current assignments. It should be avoided to do this only when all assignments have been explicitely performed.

Strategies for pruning the search space:

1. Partial Testing

- Test constraints having variables only that have already assigned values.
- States in which some constraints are violated already *may* not be expanded further, but rather traced back. If there are no descendant nodes anymore and no solution is found, the inference **must** trace back.

2. Forward Checking

- Reduce all domains for variables not assigned such that the future assignment still has a chance to be feasible.
- Trace back if this leads to empty domains.

Pruning search space strategies

Example for not pruning at all:

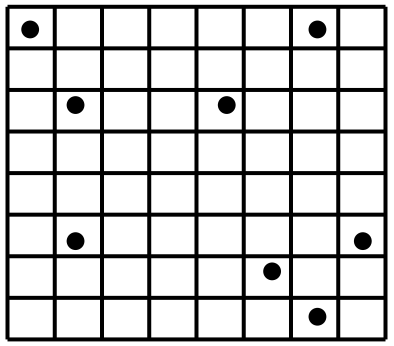
8-queens-problem (solution by Bratko, 2nd method)

Knowledge base:

```
queens2(YList) :-
   permutation([1,2,3,4,5,6,7,8], YList),
   admissible(YList).

permutation([],[]).
permutation([First|Tail],ResultList) :-
   permutation(Tail,ResultTail),
   del(First,ResultList,ResultTail).

admissible([]).
admissible([Y1|Others]) :-
   admissible(Others),
   conflictFree(Y1,Others,1).

conflictFree(_,[],_).

conflictFree(Y, [Y1|YTail], XDiff) :-
   YDiff is Y1-Y,
   YDiff =\= XDiff,
   YDiff =\= -XDiff,
   XDiff1 is XDiff + 1,
   conflictFree(Y,YTail,XDiff1).
```

Query:

```
?-queens2(YList).
```



Pruning search space strategies

Example for partial testing:

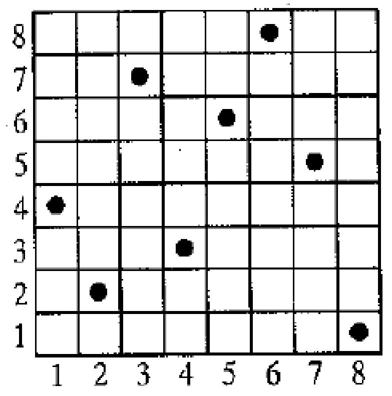
8-queens-problem (solution by Bratko, 1st method)

Knowledge base:

```
queens1([]).

queens1([X/Y | Others]) :-
   queens1(Others),
   member(Y,[1,2,3,4,5,6,7,8]),
   conflictFree(X/Y,Others).

conflictFree(_,[]).

conflictFree(X/Y, [HeadX/HeadY | Others]) :-
   Y =\= HeadY,
   DiffY is HeadY - Y,
   DiffY =\= HeadX - X,
   DiffY =\= X - HeadX,
   conflictFree(X/Y,Others).

template([1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]).
```

Query:

```
?- template(S), queens1(S).
```



Pruning search space strategies

Example for forward checking:

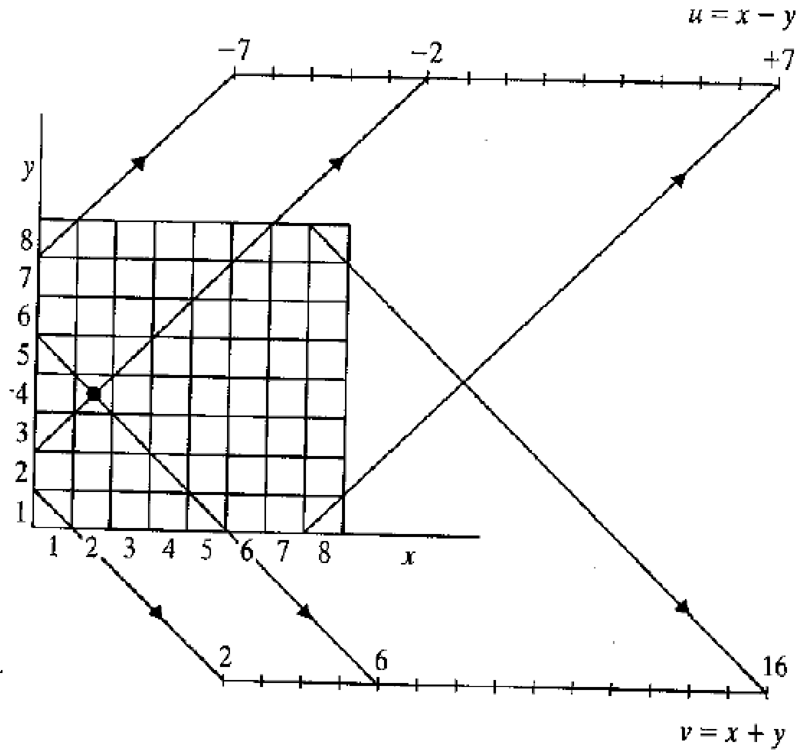
8-queens-problem (solution by Bratko, 3rd method)

Knowledge base:

```
queens3(YList) :-
   sol(YList, [1,2,3,4,5,6,7,8],
              [1,2,3,4,5,6,7,8],
              [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7],
              [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]).

sol([],[], DomainY, DomainU, DomainV).

sol([Y |YTail], [X | XTail], DomainY, DomainU, DomainV) :-
   del(Y,DomainY,ReducedDomainY),
   U is X - Y,
   del(U,DomainU,ReducedDomainU),
   V is X + Y,
   del(V,DomainV,ReducedDomainV),
   sol(YTail, XTail, ReducedDomainY, ReducedDomainU,
ReducedDomainV).

del(Item, [Item|List], List).
del(Item, [First|Tail],[First|ResultTail]) :-
   del(Item,Tail,ResultTail).
```

Query:

```
?-queens3(YList).
```



Traversing search graphs

Alternative 2. search method:

Systematic improvement of preliminary (global) solutions

- **Node: describes state in search domain**
  - **State: Assignment of values to all variables (not all of them need be admissible)**
    Each state has got an evaluation.
- **Edge: Transition of a state into an adjacent state**
  (usually feasible in both directions)
  - **Adjacent state: New values for certain variables keeping all values for the other variables**
- **Initial node: initial state** (is always unique)
  - **Initial node: Start with any assignment to the variables.**
    (or apply a reasonable starting heuristic)
- **Final node: final state wanted (problem solution)**
  (several ones are admissible)
  - **Final node: No adjacent state has got a better evaluation than the present one.**
    (or some heuristic function is achieved)

General Optimisation Methods for CSP

For the 2. search method of systematic improvement:

Min-Conflicts procedure:

Idea:

- Start with an arbitrary assignment of values (valid or not).
- Assign new values for certain variables such that the new assignment bares fewer conflicts than the old one.

Advantages:

- happens to show good run time behaviour
- „repair strategy“ if something changes dynamically

Disadvantages:

- „Getting stuck“ in local minima
  - counter measures: random walk, tabu list, ...

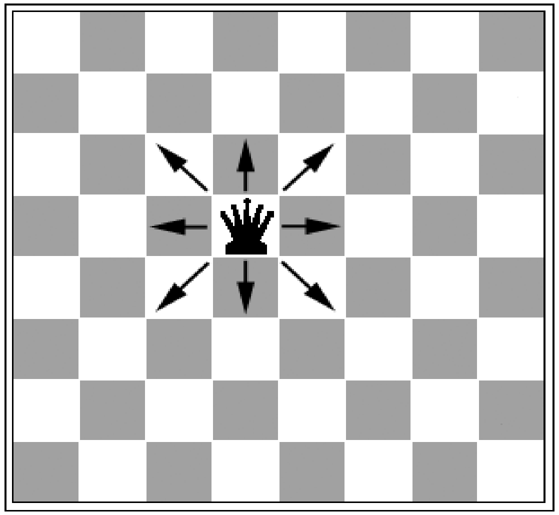
Weitere Details zum Thema Constraintsysteme:
Seminarvortrag und Ausarbeitung von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

General Optimisation Methods for CSP

For the 2. search method of systematic improvement:

Min-Conflicts procedure:

Application: 8-queens-problem



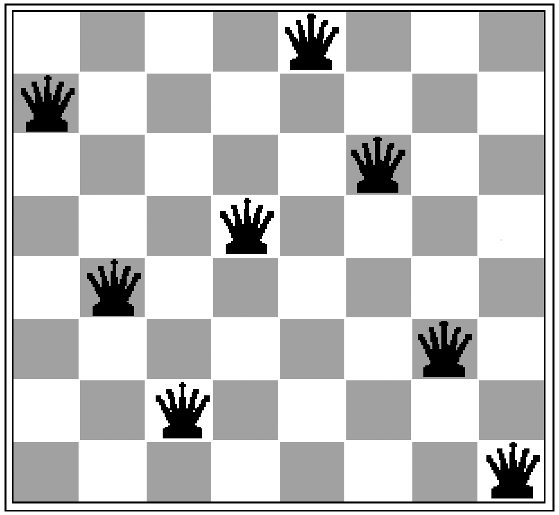
Quelle: Seminarvortrag von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

General Optimisation Methods for CSP

For the 2. search method of systematic improvement:

Min-Conflicts procedure:

Application: 8-queens-problem



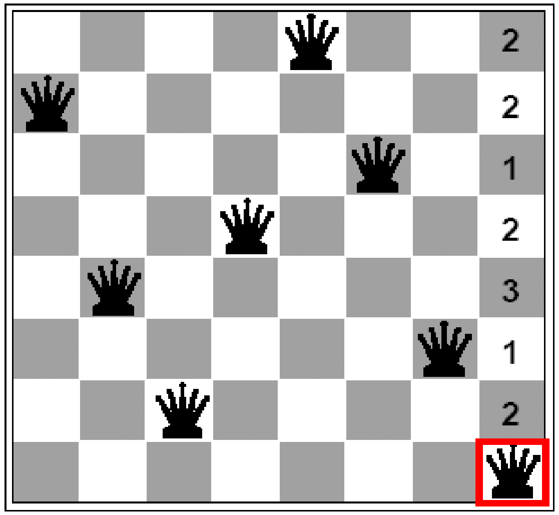
Quelle: Seminarvortrag von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

General Optimisation Methods for CSP

For the 2. search method of systematic improvement:

Min-Conflicts procedure:

Application: 8-queens-problem



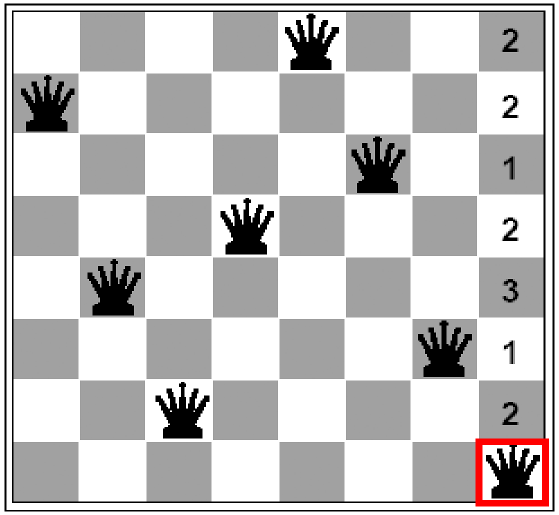
Quelle: Seminarvortrag von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

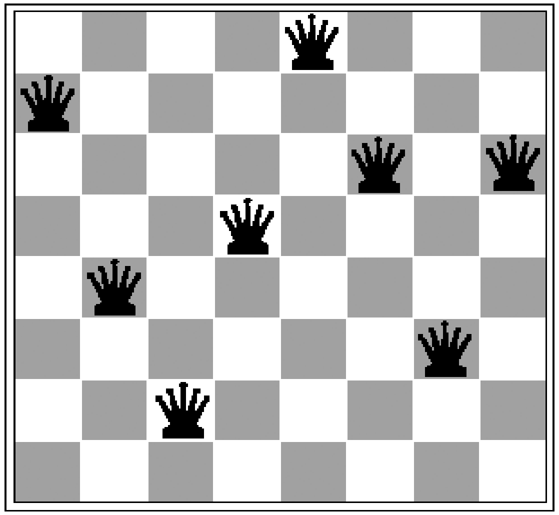
General Optimisation Methods for CSP

<u>For the 2. search method of systematic improvement</u>:

<u>Min-Conflicts procedure:</u>

Application: 8-queens-problem





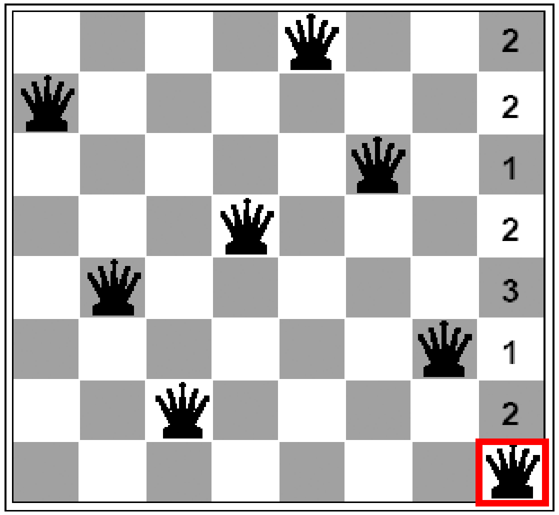
Quelle: Seminarvortrag von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

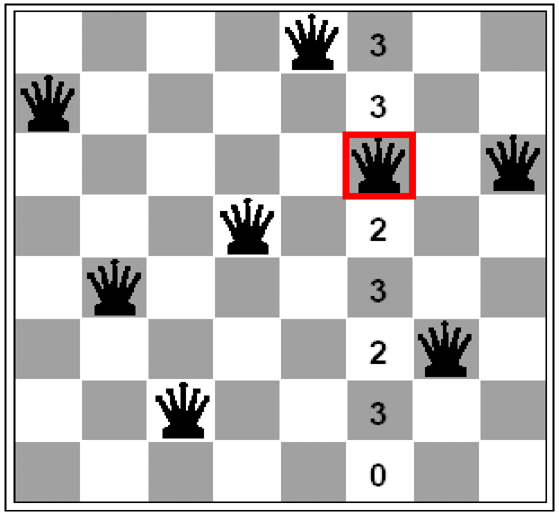
General Optimisation Methods for CSP

For the 2. search method of systematic improvement:

Min-Conflicts procedure:

Application: 8-queens-problem





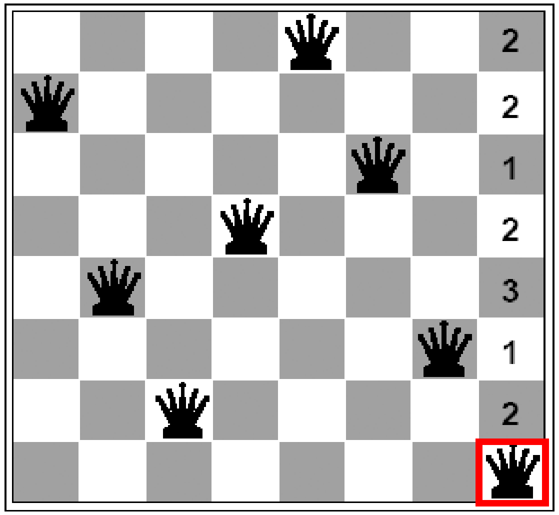
Quelle: Seminarvortrag von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

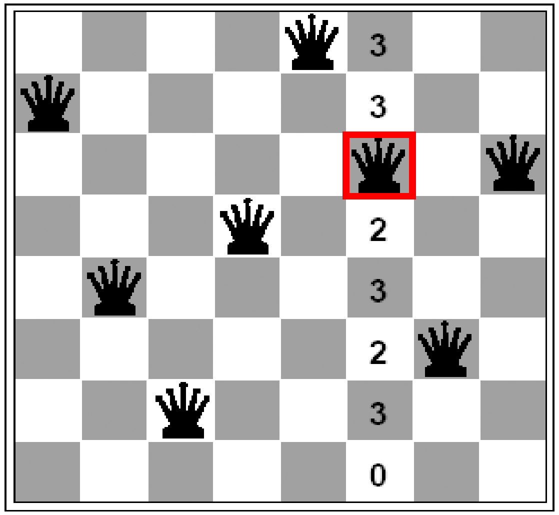
General Optimisation Methods for CSP

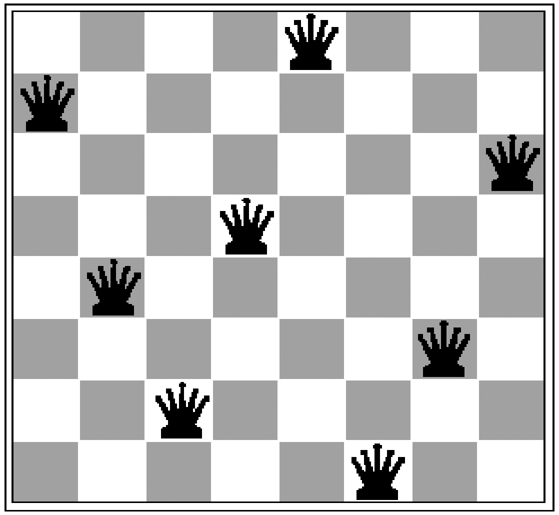
<u>For the 2. search method of systematic improvement</u>:

<u>Min-Conflicts procedure:</u>

Application: 8-queens-problem







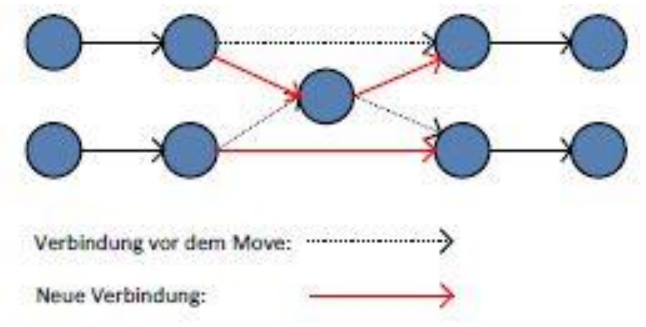
Quelle: Seminarvortrag von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

General Optimisation Methods for CSP

For the 2. search method of systematic improvement:

Working with tabu lists in search graphs:

- Determine a certain validity range for the algorithm, e.g. by a given number of operations
- Protocol all edges used in a transition from one state to another
- All edges used within the previous validity range are not to be used again, neither their counterdirection.



Further enhancement: Simulated annealing

- Admit temporary deteriorations.
- Diminish the tolerance bound for deterioration in the course of algorithmic progress gradually.

These methods will mainly be used in improvements of global solutions

- Good results in logistics (TSP generalisations)