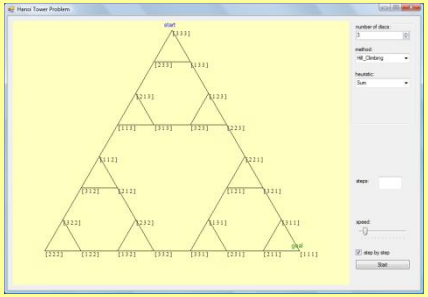
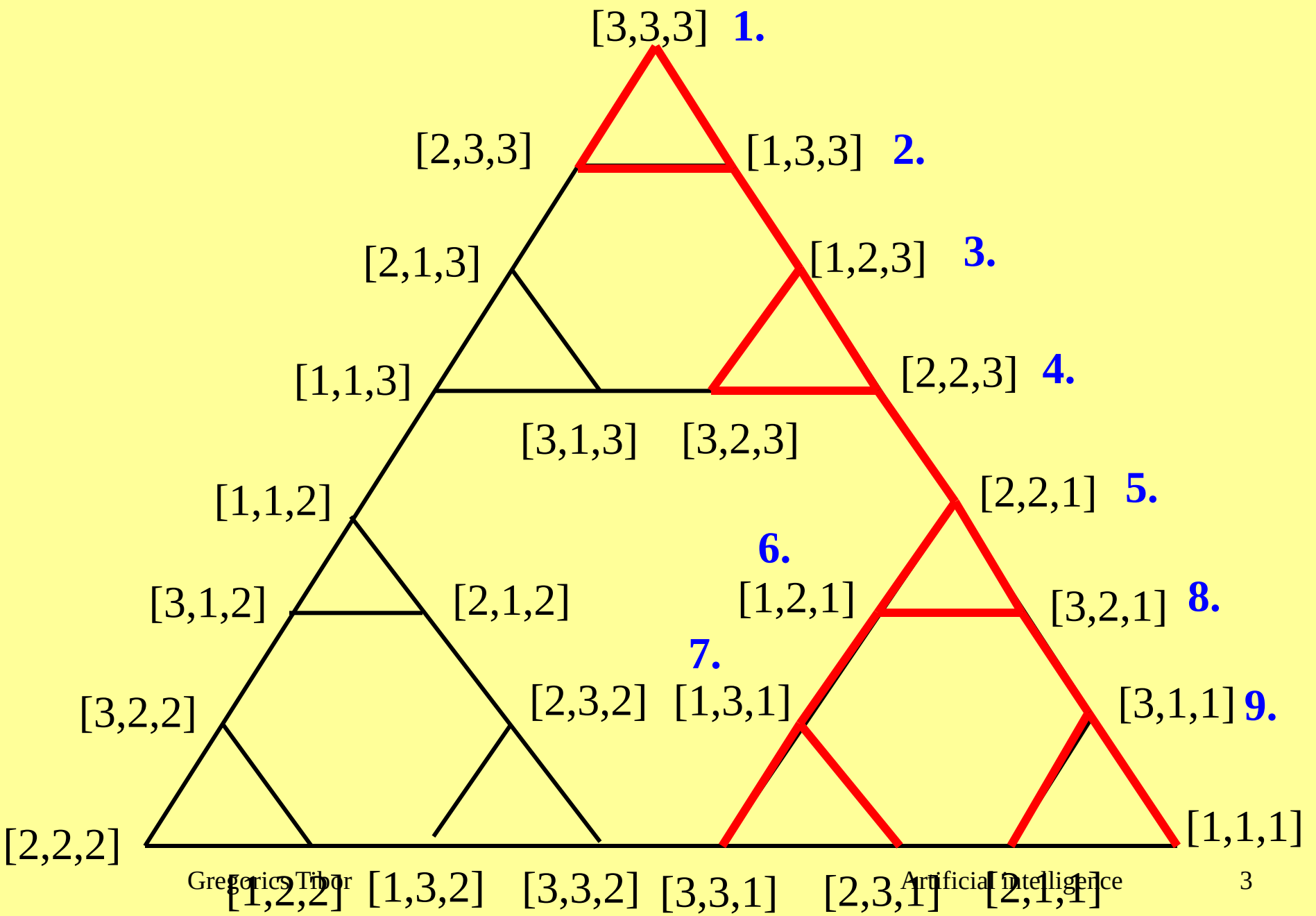
3. Graph-search

- It is a search system
  - global workspace: stores all paths going from the start in part (this is a subgraph of the representation graph) and separately records the nodes, they are called open nodes, that have already been achieved but their successors have not been discovered yet
    - initial value: start node
    - termination condition: goal node appears or the algorithm gets stuck
  - searching rules: expand an open node
  - control strategy: selects an open node to be expanded

3.1. General graph-search



- search graph (G) : the subgraph of the representation graph that has been discovered
- set of open nodes ($OPEN$) : they are waiting for their expansions because their descendents are not known or not well-known
- closed nodes : the nodes of the search graph that has already been expanded
- expansion (Γ) : generating all successors of a node with its outgoing arcs
- f: $OPEN \rightarrow \mathbb{R}$ evaluation function: helps to select the appropriate open node to be expanded.


[3,3,3] 1.
[2,3,3]
[1,3,3] 2.
[2,1,3]
[1,2,3] 3.
[1,1,3]
[2,2,3] 4.
[3,1,3]
[3,2,3]
[1,1,2]
[2,2,1] 5.
6.
[3,1,2]
[2,1,2]
[1,2,1]
[3,2,1] 8.
7.
[3,2,2]
[2,3,2]
[1,3,1]
[3,1,1] 9.
[2,2,2]
[1,1,1]
[1,2,2]
[1,3,2]
[3,3,2]
[3,3,1]
[2,3,1]
[2,1,1]

DATA := *initial value*
while ¬ *termination condition*(DATA) **loop**
    SELECT R FROM *rules that can be applied*
    DATA := R(DATA)
endloop

First (wrong) version

Procedure *GK0*

1. $G := (\{start\}, \emptyset)$: $OPEN := \{start\}$
2. **loop**
3. **if** *empty*(*OPEN*) **then return** *cannot find solution*
4. $n := min_f(OPEN)$
5. ***if*** *goal*(*n*) **then return** *there is a solution*
6. $OPEN := OPEN - \{n\} \cup \Gamma(n)$
7. $G := G \cup \{(n,m) \in A \mid m \in \Gamma(n)\}$
8. **endloop**

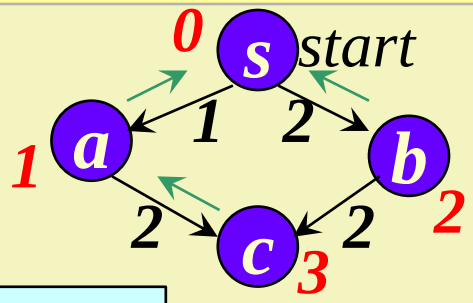
end

Faults

- At termination the solution path cannot be got.
  - The traces of the paths should be signed.
- The optimal solution is not guaranteed (neither solution)
  - The costs of the discovered paths should be stored.
    If several paths to the same node were found, the best path cost needs to be stored.
- Circles cause fault
  - Storing the costs of the paths can help to avoid the circle because in δ-graph a path with a circle is always more expensive than its acyclic version.

Functions of the graph-search

- $\pi: N \to N$ ***parent pointer function***
  - $\pi(n)$ = one parent of n in G,
    $\pi(start) = nil$



π records an unique path to each node from *start* in G.
It determines a directed spanning tree in G with *start* as a root node.

  - If only the π spanning tree preserved the optimal paths to any node from *start* in G: π spanning tree was optimal

correct nodes,
correct search graph

- $g: N \to \mathbb{R}$ ***cost function***
  - $g(n) = c^{\alpha}(start,n)$ − cost of a discovered path $\alpha \in \{start \to n\}$
    - If only $g(n)$ was the cost of the path $start \to n$ denoted by π for each node n : π and g were consistent

Maintaining the correctness of the search graph

- Initially: $\pi(start) := nil,\ g(start) := 0$
- for all $m \in \Gamma(n)$ (after expansion of the node n) :
  - 1. m is a new node

    if $m \notin G$ **then** $\pi(m) := n,\ g(m) := g(n)+c(n,m)$

    $OPEN := OPEN \cup \{m\}$
  - 2. m is an old node to that a cheaper path has been found

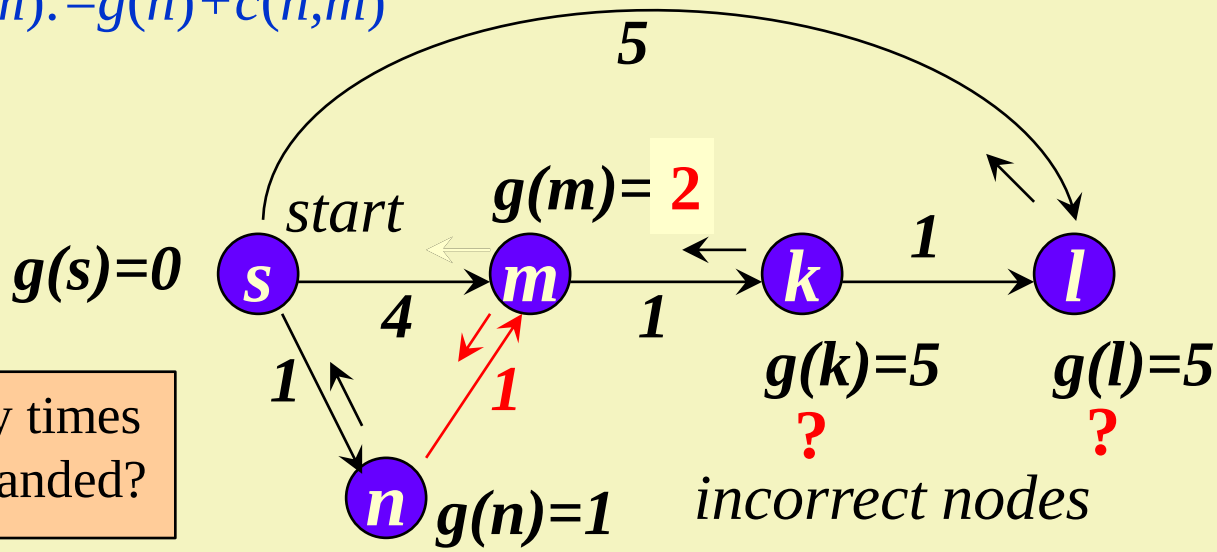
    if $m \in G$ and $g(n)+c(n,m)<g(m)$ **then**

    $\pi(m) := n,\ g(m) := g(n)+c(n,m)$
  - 3. m is an old node to that a not cheaper path has been found

    if $m \in G$ and $g(n)+c(n,m) \geq g(m)$ **then** *SKIP*

The search graph does not even preserve its correctness

If $m \in G$ and $g(n)+c(n,m)<g(m)$ then
$\pi(m):=n$, $g(m):=g(n)+c(n,m)$



Danger: how many times will a node be expanded?

- Possible answers:
  1. The pointers and costs of all descendants of the node *m* should be modified using some traversal method.
  2. Such a case could be avoided with a good evaluation function.
  3. Do not care of this just put the node *m* back into *OPEN*.

DATA := *initial value*
while ¬ *termination condition*(DATA) **loop**
SELECT R FROM *rules that can be applied*
DATA := R(DATA)
endloop

Algorithm of general graph-search

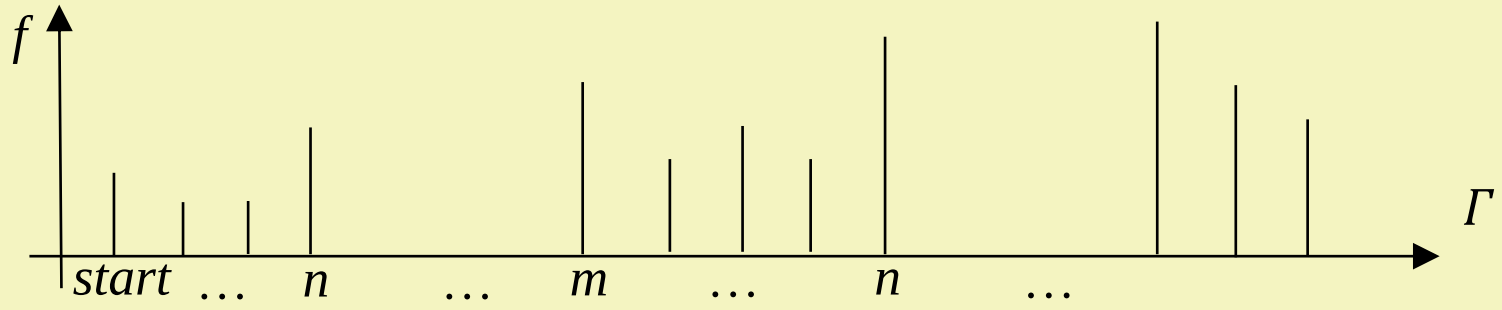
1. $G := (\{start\}, \emptyset)$: $OPEN := \{start\}$: $\pi(start) := nil$: $g(start) := 0$
2. **loop**
3. **if** *empty*(*OPEN*) **then return** *cannot find solution*
4. $n := min_f(OPEN)$
5. **if** $goal(n)$ **then return** solution (n, π)
6. $OPEN := OPEN - \{n\}$
7. **for** $\forall m \in \Gamma(n)$ **loop**
8. **if** $m \notin G$ or $g(n)+c(n,m)<g(m)$ **then**
9. $\pi(m) := n$, $g(m) := g(n)+c(n,m)$, $OPEN := OPEN \cup \{m\}$
10. **endloop**
11. $G := G \cup \{(n,m) \in A \mid m \in \Gamma(n)\}$
12. **endloop**

Summarization of execution and outcomes

It can be proved:

- Any node is expanded only finite times in δ-graphs.
- The general graph search always terminates in finite δ-graphs.
- The general graph search finds a solution in finite δ-graphs if there exists a solution

Execution diagram



- The expanded nodes with their evaluation function values are enumerated in order of their expansions (the same node can occur several times).

3.2. Famous graph-search algorithm

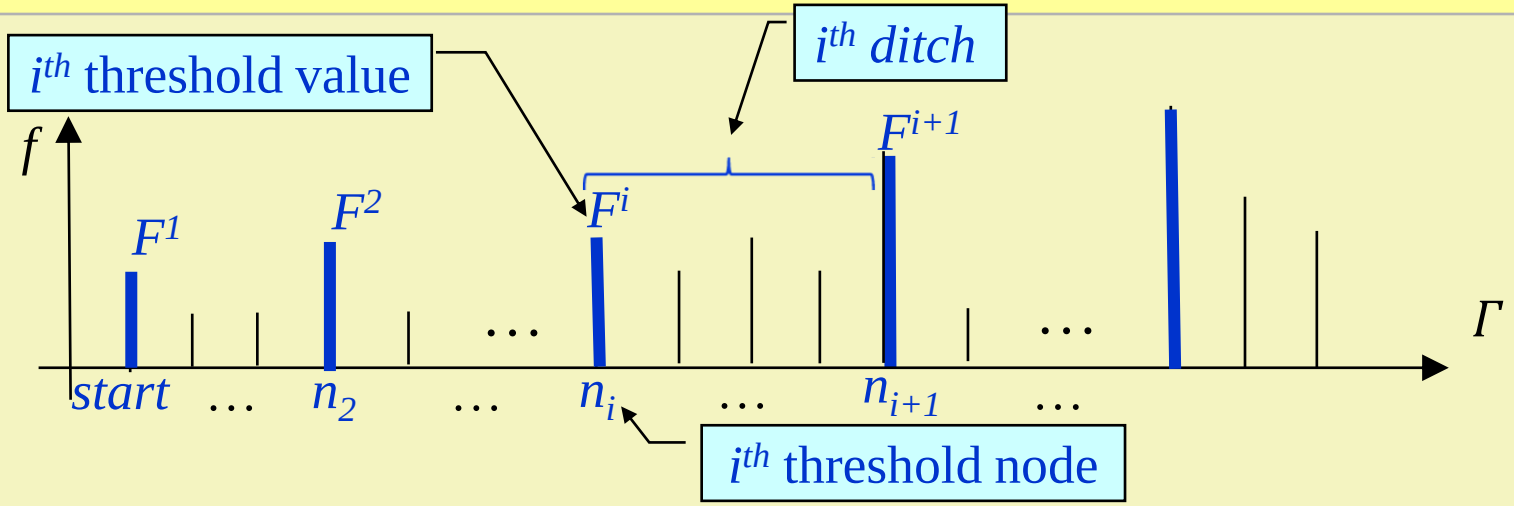
- How can we define the evaluation function?

Non-informed	Heuristic
Depth graph-search	Look forward (best-first)
Breadth graph-search	A, A^*, A^c
Uniform-cost graph-search	B

Decreasing evaluation function

- An evaluation function is decreasing if its value on a node never increases but it always decreases when a cheaper path to this node has been found.
- It can be proved that the graph-search with a decreasing evaluation function re-establishes automatically the correctness of the search graph over and over again.

About the correctness of the search graph with decreasing evaluation function



- A monotone increasing subsequence F^i ($i=1,2,\ldots$) is constructed from the values of the diagram so that it starts with the first value and then always the closest non smaller one must be selected.
- It can be shown that the graph-search with a decreasing evaluation function has correct search graph at expansion of a threshold node, and never expands incorrect nodes.

Non-informed graph-search

not identical to the backtracking

depth first graph-search	$f = -g,\ c(n,m) = 1$	in infinite δ-graphs a depth bound is needed
breadth first graph-search	$f = g,\ c(n,m) = 1$	• finds optimal (the shortest) solution if there exists one even in infinite δ-graph • any node is expanded at most once
uniform-cost graph-search	$f = g$	• finds optimal (the cheapest) solution if there exists one even in infinite δ-graph • any node is expanded at most once

similar to Dijkstra's shortest path algorithm

Heuristics in graph-search

- The heuristic function $h:N \rightarrow \mathbb{R}$ estimates the cost of the cheapest path from n to the goal.

- $h(n) \approx \min_{t \in T} c^*(n,t) = c^*(n,T) = h^*(n)$ $\quad h^*:N \rightarrow \mathbb{R}$

remaining optimal cost

- Examples:
  - *8*-puzzle : *W*, *P*
  - *0* (zero function) ~ fake heuristic function

Properties of heuristic function

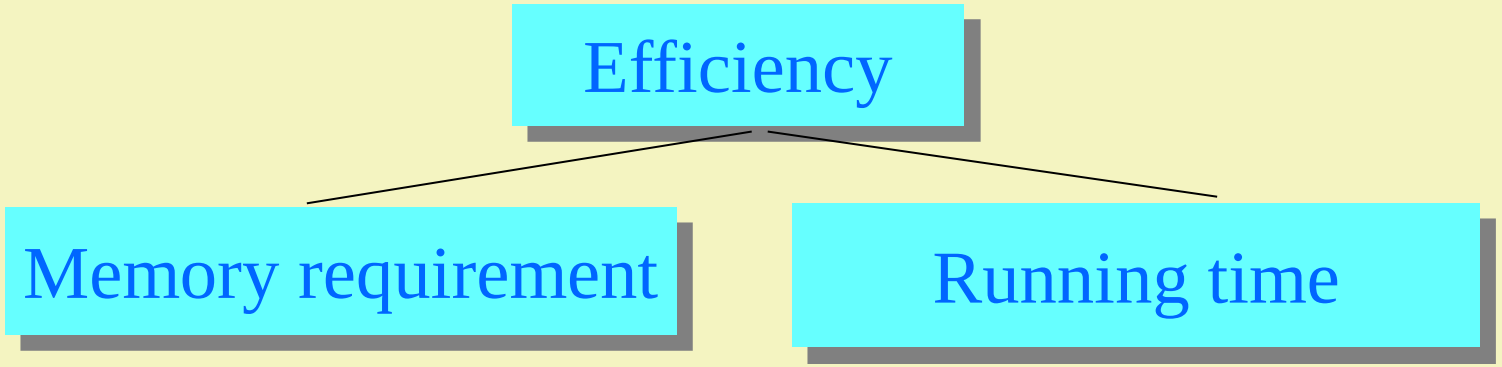
- Famous properties:
  - Non-negative: $h(n) \geqq 0 \quad \forall n \in N$
  - Admissible: $h(n) \leqq h^*(n) \quad \forall n \in N$
  - Monotone (consistent): $h(n) - h(m) \leqq c(n,m) \quad \forall (n,m) \in A$
- Remarks
  - *8*-puzzle : *W*, *P* is non-negative, admissible and monotone.
  - Zero function is non-negative, admissible and monotone.

Outcomes of the heuristics graph-search

look forward graph-search	$f = h$	
algorithm A	$f=g+h$, $0 \leqq h$	• finds solution if there exists one (even in infinite δ-graph)
algorithm A^*	$f=g+h$, $0 \leqq h$, $h \leqq h^*$ (admissible)	• finds optimal solution if there exists one (even in infinite δ-graph)
algorithm A^c	$f=g+h$, $0 \leqq h$, $h \leqq h^*$ $h(n)-h(m) \leqq c(n,m)$ (monotone)	• finds optimal solution if there exists one (even in infinite δ-graph) • expands a node at most once • if h is monotone and gives zero on goal, then it is admissible

3.3. Efficiency of *algorithm A**

Those problems are focused on that have got a solution and have got an admissible heuristic function because only these conditions can guarantee that *algorithm A** terminates with an optimal solution.



Black&White problem

f	*Graph-search*	*solution*	*G*	*Γ*
-g	*Depth first*	5	8	5
g	*Breadth first*	4	10	8
I	*Look forward*	5	8	5
g+I	*algorithm A*	4	9	7
*g+2*I*	*algorithm A*	4	8	6
*g+2*I–1(if...)*	*algorithm A*	4	7	5

3.3.1. Analysis of memory requirement

- $CLOSED_S$ ~ the set of the nodes closed (expanded) by the path-finding algorithms S until its termination

- Fix a problem. Let X and Y be two path-finding algorithms.
  X is not worse than Y if $CLOSED_X \subseteq CLOSED_Y$
  X is better than Y if $CLOSED_X \subset CLOSED_Y$

- Using these definitions we can compare

1. two *algorithms* A^* using different admissible heuristics on the same problem.
2. *algorithm* A^* and another path-finding algorithm using the same admissible heuristics on a given subset of problems.

Comparing two algorithms A^* using different heuristics

- Let A_1 (with heuristics h_1) and A_2 (with heuristics h_2) be *algorithms* A^*.

  A_2 is more informed than A_1 if for all nodes $n \in N \setminus T$: $h_1(n) < h_2(n)$.

  $h_1(n) < h_2(n) \leqq h^*(n)$

- It can be proved that a more informed A_2 is not worse than a less informed A_1, i.e.,

  $CLOSED_{A_2} \subseteq CLOSED_{A_1}$

- In practice, $CLOSED_{A_2} \subset CLOSED_{A_1}$ even if $h_1(n) \leqq h_2(n)$.

15-puzzle

f =	g+0	g+W	g+P
6 steps solution	117	7	6
13 steps solution	not enough memory	119	13
21 steps solution	not enough memory	3343	145
30 steps solution	not enough memory	not enough memory	1137
34 steps solution	not enough memory	not enough memory	3971

Comparing algorithm A^* with other graph-searches

- Our aim is to show that algorithm A^* does not require much more memory than the other graph-searches on the same problems.

- An admissible problem is the path-finding problem that has got a solution and has got an admissible heuristic function.
  - In this investigation the heuristic function will be embedded in the problem rather than in the control strategy.
  - From this perspective there is no difference between *algorithm A*, A^* and A^c since the heuristics of the problem determines that an *algorithm A* could be A^* or A^c.

X dominates Y

- Let *X* and *Y* be two sets of path-finding algorithms.
- *X dominates Y* relative to a given subset of admissible problems if for all problems and for each member *y* of *Y* there exists a member *x* of *X* so that $CLOSED_x \subseteq CLOSED_y$
- *X strictly dominates Y* relative to a given subset of admissible problems if *X* dominates *Y* but *Y* does not dominate *X*.

A non-deterministic algorithm can be treated as a set of its deterministic versions. In a graph-search, a secondary (tie-breaking) rule can choose from the open nodes that have got the same evaluation function value. Thus an algorithm A^* can be mapped to many deterministic graph-searches that are defined by their tie-breaking rules.

Admissible path-finding algorithms

- A path-finding algorithm is admissible if it can find an optimal solution on each admissible problem.
- Examples:
  - *Uniform-cost graph-search*
  - *Algorithm* A^*
  - *Algorithm* A^{**} : $f(n)=\mathbf{max}_{m \in start \rightarrow n}(g(m)+h(m))$ tie-breaking rule: prefers the goal node
  - *Algorithm IDA** (it is backtracking algorithm using a special cutting process, embedded in a loop that calls it with an increasing depth bound repeatedly)

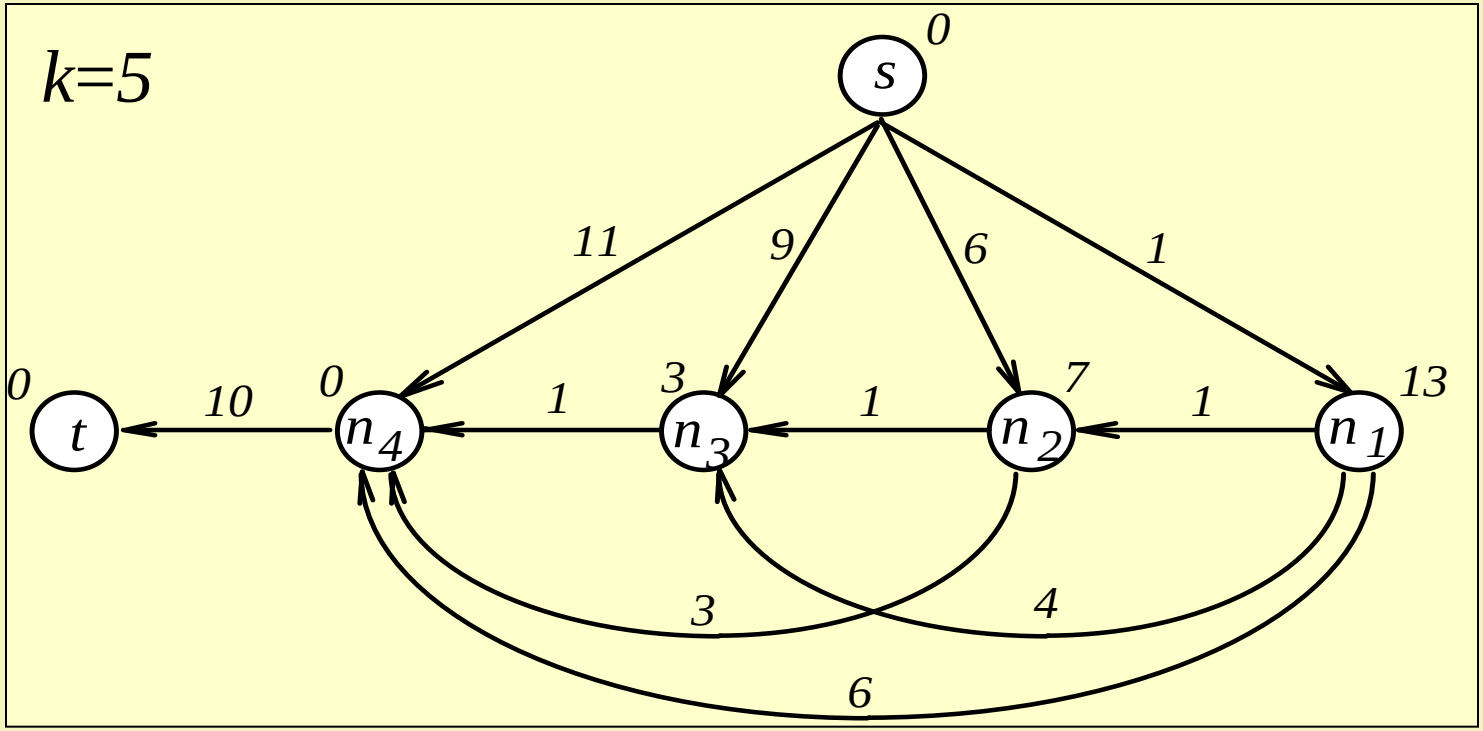
Provable results

- *Algorithm A** dominates all admissible path-finding algorithms on the problems that have got monotone admissible heuristics.
- No admissible path-finding algorithm dominates all other admissible path-finding algorithms on all admissible problems.
- *Algorithm A*** strictly dominates *algorithm A** on all admissible problems.

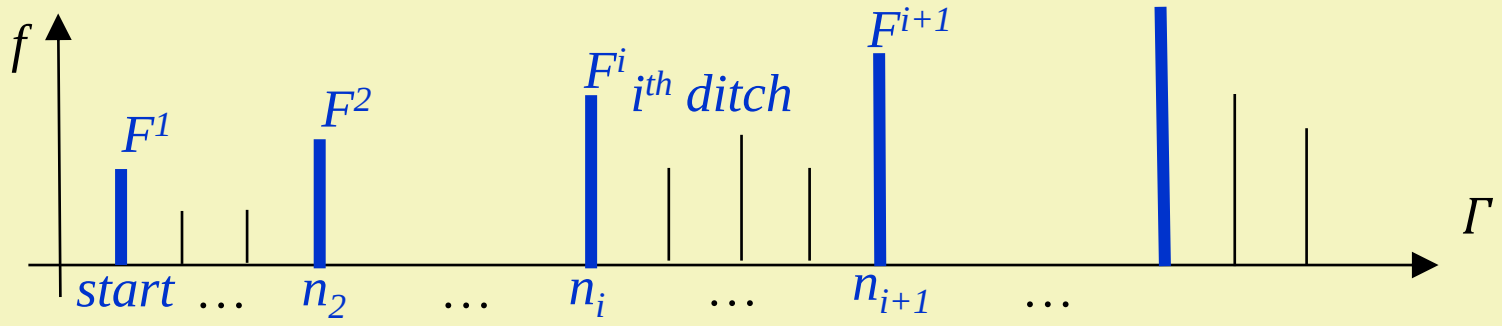
3.3.2. Analysis of running time

- Denote the number of the closed nodes as k
- Lower limit on the number of the iterations: k
  - If algorithm A^* uses a monotone heuristics, (so it is an algorithm A^c) it expands a node at least once thus the number of the closed nodes is equal to the number of expansions.
- Upper limit on the number of the iterations: 2^{k-1}
  - See Martelli's problem

Martelli's example



Discussion



- One node – even in the same ditch – can be expanded several times.
- A secondary evaluation function is introduced in the ditches. It can be proved that the set of the nodes expanded in a ditch does not depend on this inner function, it influences only the order and the number of expansions of the nodes in the ditch. (Thus neither the threshold nodes nor threshold values nor their order change.)
- Martelli suggested using the cost function *g* as an inner evaluation function.

Algorithm B

- *Algorithm B* is derived from *algorithm A*.

  Introduce the variable *F* to store the current threshold value. Change the step 1 and step 4 of the algorithm:

  - Step *1.* + $F := f(s)$
  - Step *4.* **if** $min_f(OPEN)<F$
  - **then** $n := min_g(m \in OPEN \mid f(m)<F)$
  - **else** $n := min_f(OPEN); F \leftarrow f(n)$
  - **endif**

Running time of algorithm B

- *Algorithm B* works in the same way as *algorithm* A^* except that it expands a node in a ditch only once.
  - In the worst case all closed nodes are threshold nodes and their first expansions are the thresholds because of the decreasing evaluation function.
  - The first ditch consists of only the start node, in the second ditch only the second threshold node can be found, the i^{th} ditch contains at most the previous $i-1$ threshold nodes (that is, all nodes except for the start node), thus the last k^{th} ditch has got at most $k-1$ nodes.
  - It follows that the number of expansions is at most $\frac{1}{2} \cdot k^2$.