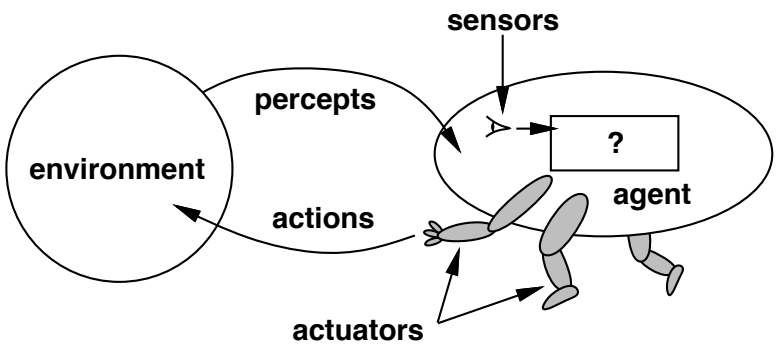
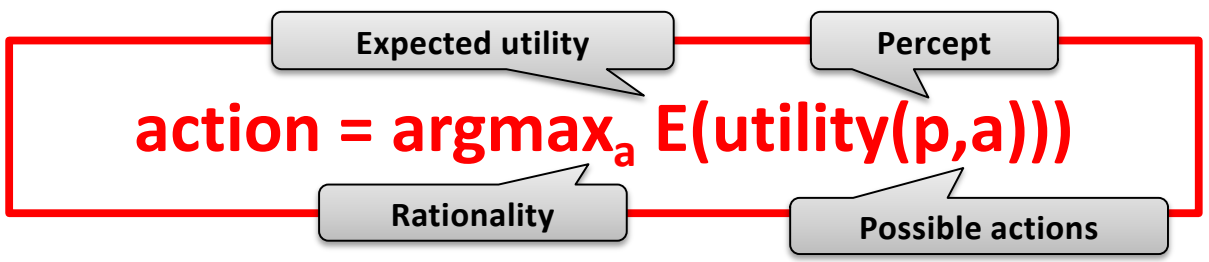
Introduction to Artificial Intelligence

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**.



A rational agent should select an action that is expected to maximize its performance measure.



Reflex agent

Selects an action based on current state of the world (that is obtained via perception).

Simple reflex agent:

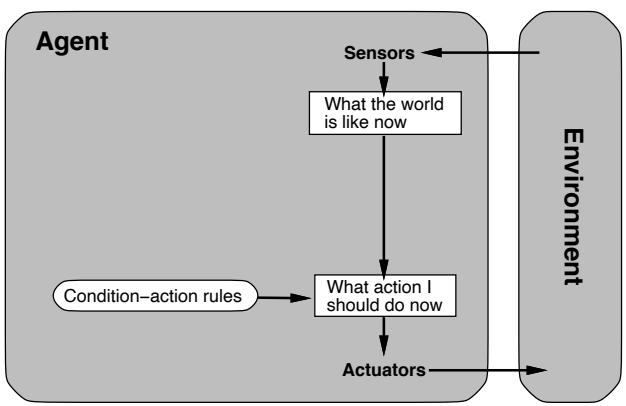
Observation → Action

Model-based reflex agent:

(PastState, PastAction, Observation) → State

State → Action

- Works for partially observable environments (observations are "stored" in the state).



Goal-based agent

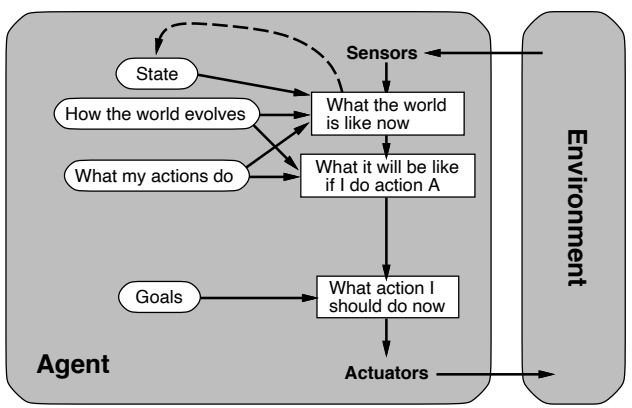
Agents can be more flexible if they assume goal (desirable states) in addition to the current state.

Transition model

(PastState, PastAction, Observation) → State

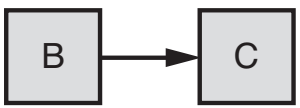
(State, Goal) → Action

- This may involve reasoning about future (what actions do), if goal is not reachable immediately (via single action).
- Search and planning.



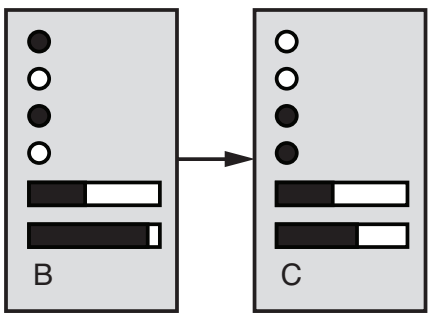
Atomic

- state is indivisible (no internal structure, black box)
- used in search



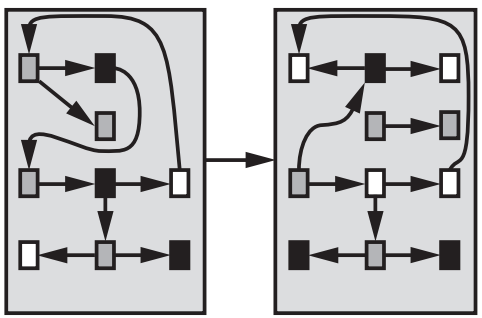
Factored

- state is a vector of values (properties of the world)
- used in constraint satisfaction, propositional logic, planning



Structured

- state is a set of objects (with own attributes) connected via various relations
- used in first-order logic



Problem solving agent is a type of goal-based agent

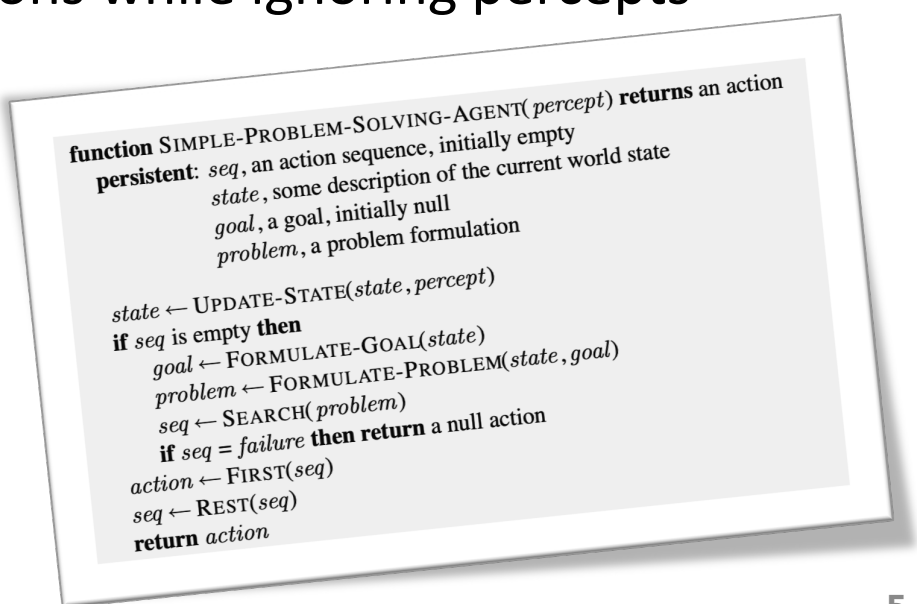
- uses **atomic representation** of states
- goal is represented by a set of **goal states**
- actions describe **transitions** between states

The task is to find a **sequence of actions** that reaches the goal state (from the initial/current) state.

- realized via **search**
- agent then executes the actions while ignoring percepts (an **open-loop system**)

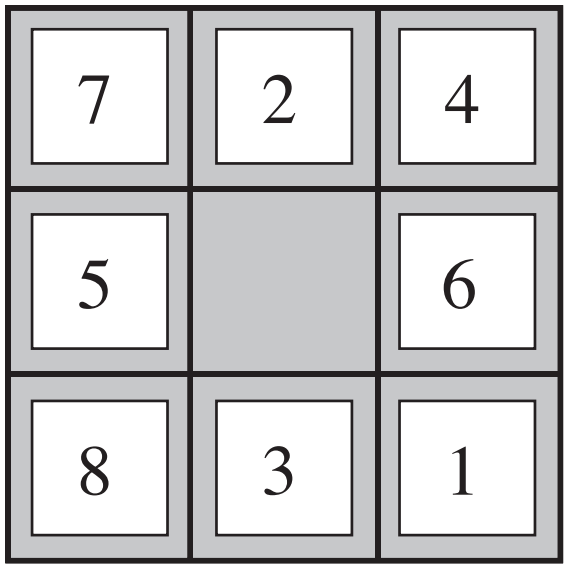
We will assume **environment**:

- fully-observable
- discrete
- static
- deterministic

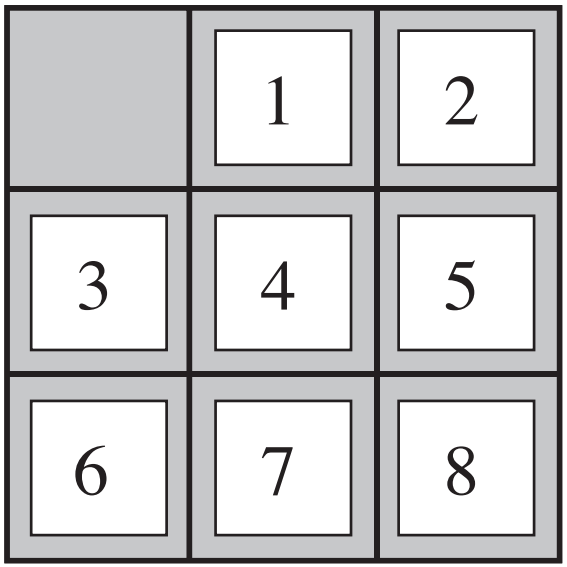


```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```



Start State



Goal State

8-puzzle (on a 3x3 board) has 9!/2 = 181 440 reachable states
15-puzzle (on a 4x4 board) has around 10^{13} states
24-puzzle (on a 5x5 board) has around 10^{25} states

Process of deciding what actions and states to consider, given a goal.

We need **abstraction** - removing detail from a representation

- abstraction of world states (which properties of world are important for problem solving)
- abstraction of actions (to reach the goal and to minimize search effort)

What is the **appropriate level of abstraction**?

- abstraction should **valid** (we can expand any abstract solution into a solution in the more detailed world)
- the abstraction should **useful** (carrying out each of the actions in the solution is easier than the original problem)

Intelligent agents would be completely swamped by the real world without using abstractions!

Well-defined problem:

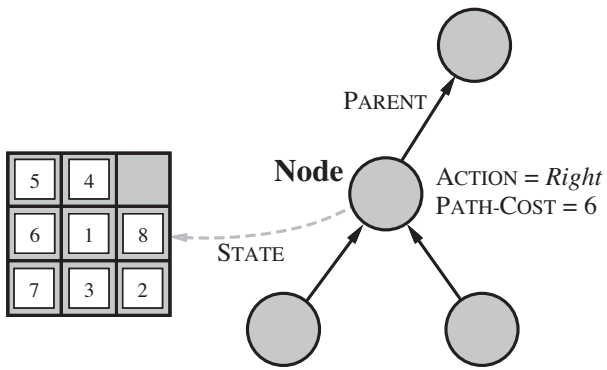
- **initial state**
- **transition model**: (state, action) → state
  - state space is defined implicitly (to describe huge state spaces)
- **goal test**

A **solution** of the problem is a **sequence of actions**.

Search algorithms consider various possible action sequences.

Search tree:

- initial state at the **root**
- **branches** are actions
- **nodes** correspond to states

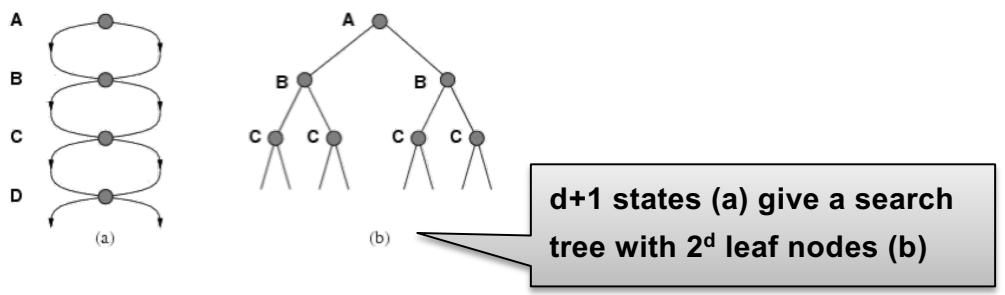


Core structure of a search algorithm:

1. put the root node to a **frontier** (a.k.a. **open list**)
2. **select a node** from frontier using some **search strategy**
3. **expand the node** (add successor states to the frontier)
4. repeat until a goal node is found (or the frontier is empty)

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

Can explore **redundant paths** (repeats a state at different nodes)



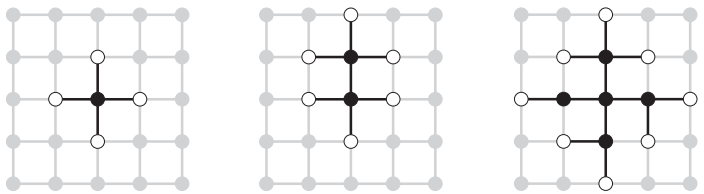
Sometimes, this redundancy can be avoided (for example, in constraint satisfaction).

Augment the tree-search algorithm with a data structure called an **explored set** (a **closed list**), which remembers every expanded node.

```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored set
```

Search tree contains at most one copy of each state.

Frontier separates explored and unexplored regions.



Uninformed search algorithms have no additional information beyond the problem formulation.

Breadth-first search

- shallowest unexpanded node is chosen for expansion (FIFO)

Properties:

- **Complete** (for finite branching factor)
- **Optimal** (if path-cost is a non-decreasing function of the depth)
- **Time and space complexity** $O(b^{d+1})$, where **b** is branching factor, **d** is depth of the goal node and goal test is done during expansion

Memory requirements are a bigger problem than is the execution time!

Depth-first search

- deepest unexpanded node is chosen for expansion (LIFO)

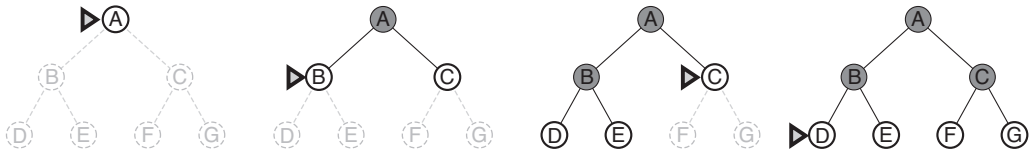
Properties:

- **Complete** for **graph-search**
- **Incomplete** for **tree-search** (could be made complete with no extra memory cost by checking new states against those on the path to the root)
- **Sub-optimal** (could be made optimal via **branch-and-bound**)
- **Time complexity** $O(b^m)$, where **m** is maximum depth of any node (it could be m >> d)
- **Space complexity** O(bm)

Backtracking - only one successor is generated at a time rather than all, O(m) memory

Breadth-first search

- shallowest unexpanded node is chosen for expansion (FIFO)



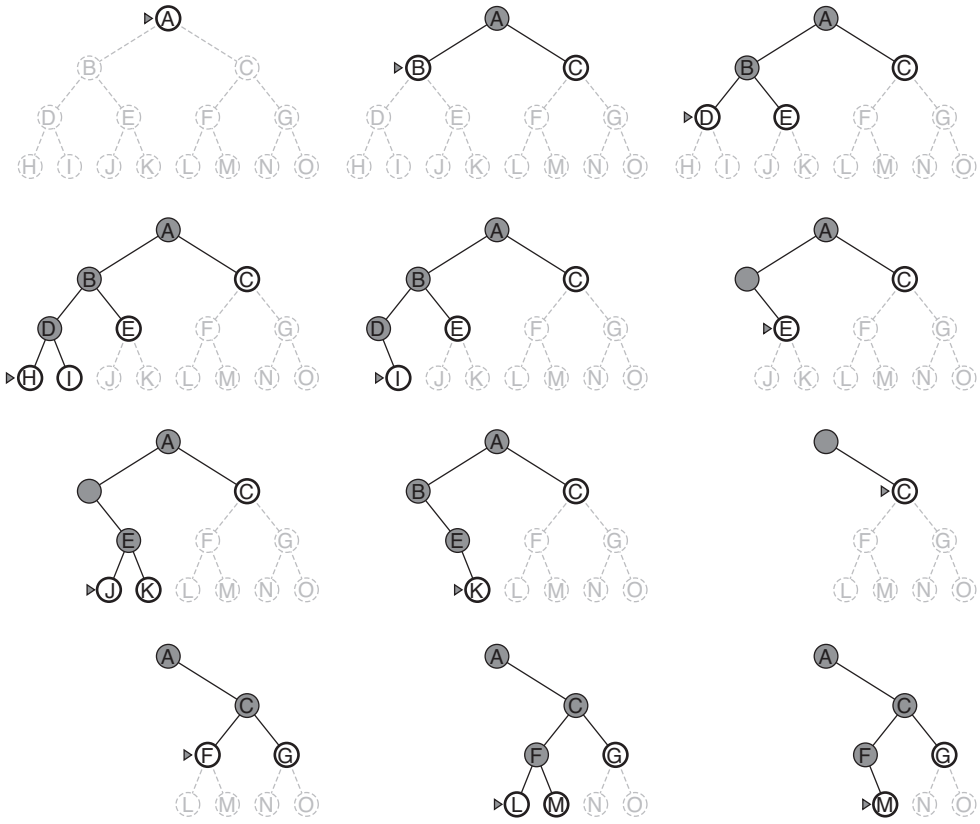
Depth-first search

- deepest unexpanded node is chosen for expansion (LIFO)
- if backtracking is used, then a single branch is kept in memory

  A – B – D – H

  A – B – D – I

  …

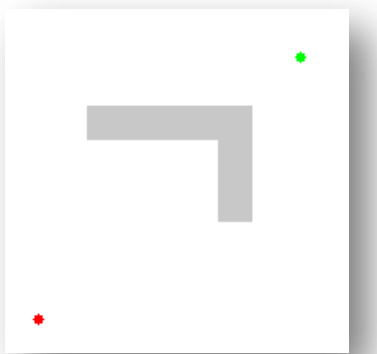


Extension of breadth-first search to work with **any step-cost function**.

Expand the node with the **lowest path cost g(n).**

- **Goal test** is applied to a node when selected for expansion rather than when the node is first generated (otherwise suboptimal path can be found).
- New test in case a **better path** is found to a node currently on the frontier.

Also known as **Dijkstra's algorithm**



```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Using **problem-specific knowledge** beyond the problem formulation.

Best-first search

- expand the node with the lowest **evaluation function f(n)**
- identical to uniform-cost search, except for the use of **f** instead of **g**

Frequently, a component of **f** is a **heuristic function**, denoted **h(n)**

- h(**n**) = estimated cost of the cheapest path from the state at node **n** to a goal state
- h(**n**) is an arbitrary non-negative function, such that, if **n** is a goal node then h(**n**)=0

- Heuristics functions are the most common form of additional knowledge.

Greedy best-first search

- expand the node closest to the goal, f(n) = h(n)
- not optimal
- incomplete (the graph-search version is complete in finite spaces)

The most widely known form of best-first search.

Created as part of the **Shakey project** (for path planning).

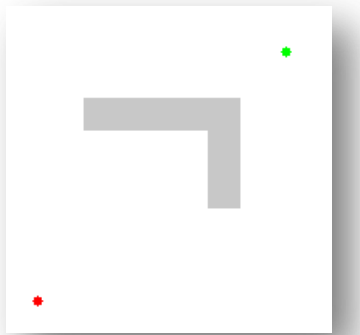
f(n) = g(n) + h(n)

f(n) = estimated cost of the cheapest solution through n

Identical to uniform-cost search except that A* uses g+h instead of g.

Optimal and **complete** (provided that the heuristic function h(n) satisfies certain conditions).

A usually runs out of space long before it runs out of time!*

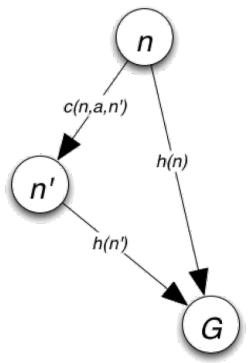


admissible heuristic h(n)

- **h(n) ≤ "the cost of the cheapest path from n to goal"**
- an optimistic view (the algorithm assumes a better cost than the real cost)
- function f(**n**) in A* is a lower estimate of the cost of path through **n**

monotonous (consistent) heuristic h(n)

- let **n'** be a successor of **n** via action **a** and c(**n**,**a**,**n'**) be the transition cost
- **h(n) ≤ c(n,a,n') + h(n')**
- this is a form of triangle inequality



Monotonous heuristic is admissible.

let n_1, n_2,..., n_k be the optimal path from n_1 to goal n_k, then

$h(n_i) - h(n_{i+1}) \leq c(n_i,a_i,n_{i+1})$, via monotony

$h(n_1) \leq \sum_{i=1,..,k-1} c(n_i,a_i,n_{i+1})$, after „sum"

For a monotonous heuristic, the values of f(n) are non-decreasing over any path.

Let n' be a successor of n, i.e. g(n') = g(n) + c(n,a,n'), then

f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') ≥ g(n) + h(n) = f(n)

If h(n) is an admissible heuristic, then the algorithm A* in TREE-SEARCH is optimal.

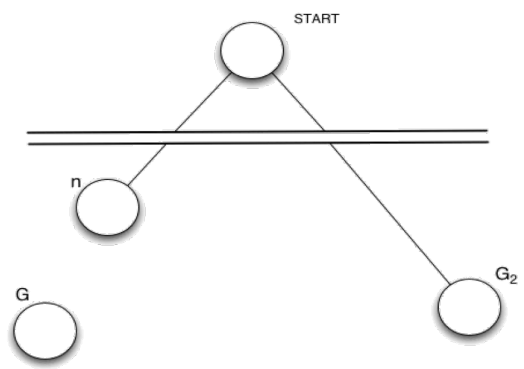
- in other words, the first expanded goal is optimal
- let G_2 be sub-optimal goal from the frontier and C* be the optimal cost

  $f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$, because $h(G_2) = 0$
- let **n** be a node from the frontier and being on the optimal path

  $f(n) = g(n) + h(n) \leq C^*$, via admissibility of h(n)
- together

  $f(n) \leq C^* < f(G_2)$,

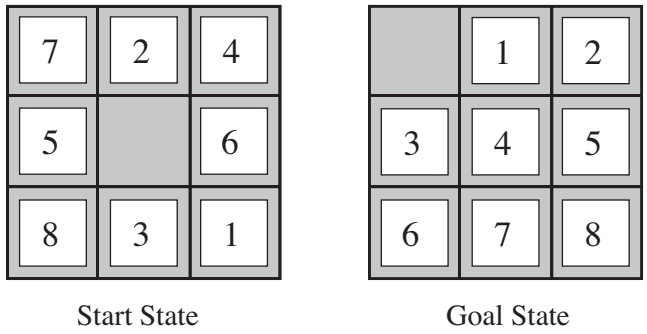
  i.e., the algorithm must expand n before G_2 and this way it finds the optimal path.



If h(n) is a monotonous heuristic, then the algorithm A* in GRAPH-SEARCH is optimal.

- Possible problem: reaching the same state for the second time using a better path – classical GRAPH-SEARCH ignores this second path!
- Possible solution: selection of the better of the two paths leading to the closed node (extra bookkeeping) or using monotonous heuristic.
  - for monotonous heuristics, the values of f(n) are non-decreasing over any path
  - A* selects for expansion the node with the smallest value of f(n), i.e., the values f(m) of other open nodes m are not smaller, i.e., among all "open" paths to n there cannot be a shorter path than the path just selected (no path can shorten)
  - hence, the first closed goal node is optimal

Example: sliding-block puzzle



Possible (admissible) heuristics:

h_1 = „the number of misplaced tiles“
= 8

h_2 = „the sum of the distances of the tiles from the goal positions“
= 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18

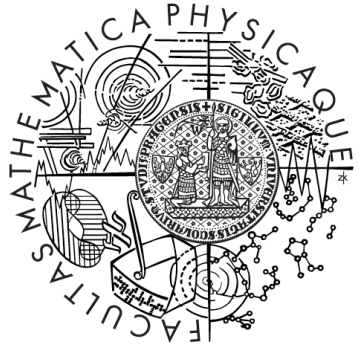
a so-called **Manhattan heuristic** (the optimal solution needs 26 steps)

Which heuristic is better?

notice that $\forall n\ h_2(n) \geq h_1(n)$ – we say that **h_2 dominates h_1**

A* with h_2 never expands more nodes than A* with h_1

- A* expands all nodes such that f(n) < C*, so h(n) < C* - g(n)
- In particular, if the algorithm expands a node using h_2, then the same node must be expanded using h_1.



Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz