Methods of Knowledge Based Software Development

Tanel Tammet, Juhan Ernits
Department of Computer Science
Tallinn University of Technology
tanel.tammet@ttu.ee, juhan.ernits@ttu.ee
2015

Organisation of the course

- Lectures: Mondays 17:45 SOC-211B,C
- Lab sessions: Tuesdays 17:45 ICT-401
- Course web page: http://courses.cs.ttu.ee/pages/ITI8600
- 4 hand in problems each worth of 10% of exam mark. Involves coding!
  - Some problems will be split in 2 subtasks.
  - Safety net: should you fail, you will have a chance to get an additional problem, but max mark per problem will go down to 10%.
  - Missing deadlines will lose you 1% of max mark per day! Try to be on time! (If you have welfare reasons for delays, please let me know as they emerge, you can get deadline extensions, but upon prior agreement).
  - 2 bonus points if you hand in the report 3 days before the deadline.
- Written exam. Worth 60% of the mark!
  - NB! Make sure to do the home assignments, otherwise your maximum will be “0”!
  - You will have to have at least 5% for each home assignment to get to the exam.
- Goes without saying: try to solve the hand in problems yourself. You will have to explain what the code does upon request. If multiple solutions look too much alike, all involved parties will get 0% for that problem.

Hand in problems

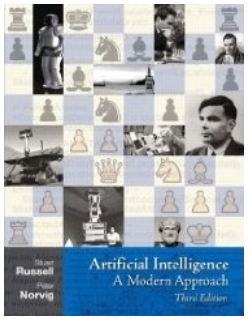
- Involve programming.
  - Main language used in the course is Python, but C++ and F# are encouraged as well. Yes, Java is OK too!
- It is important that you are able to encode the problem in such a way that you can solve it by writing a program!
- Pair programming will be part of the learning process.
- Submission to university GIT repositories (no e-mails)

Questions and answers sessions

- There will be a limited number of Q&A sessions offered so that you will be able to come and ask questions that have arisen during reading the links /the book or solving homework problems.
- If you feel that you cannot even formulate a question, come to the sessions!

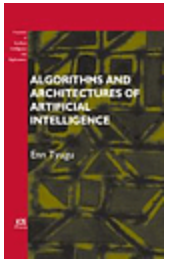
Exam

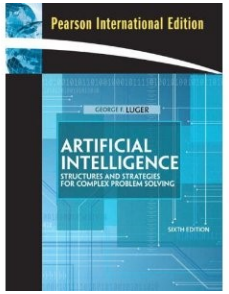
- 60% of the final mark
- You are asked to solve problems from various topics covered in the course on paper.
  - It can be quite hard, but the homeworks are meant to help you



Literature

- Web resources.
- Main book (for search and learning modules):
  - Russell and Norvig. Artificial Intelligence: A Modern Approach. 3rd edition 2010. http://tallinn.ester.ee/record=b2881231~S1*est http://tallinn.ester.ee/record=b3000919~S1*est
- Additional reading:
  - Enn Tõugu. Algorithms and architectures of artificial intelligence. IOS Press, Amsterdam 2007. http://tallinn.ester.ee/record=b2291064~S1*est
  - George F Luger. Artificial Intelligence: Structures and Strategies for complex problem solving. 6th edition, 2009. http://tallinn.ester.ee/record=b2881423~S1*est
- This and that from other sources too (check the web page).





Perspective

- The course covers many topics – could be split into several courses, but:
  - Machine learning is addressed in depth in another course taught during the Spring semester.
  - Logic can be pursued in a separate logics course
  - Data mining is a separate course
- Too few copies of the book at the library:
  - You can also lease the AIMA book electronically at http://www.mypearsonstore.com/bookstore/artificial-intelligence-a-modern-approach-coursesmart-9780136067337.

Methods of Knowledge Based Software Development

Main topics of AI

- Natural language processing
- Knowledge representation
- Automated reasoning
- Machine learning
- Computer vision
- Robotics

Topics covered in other courses

- Robotics: IBX0020 – Robotics (Maarja Kruusmaa)
- Machine learning: ITI8565 Machine learning (Sven Nõmm)
- Natural Language Processing (thesis projects available from Dept. of Computer Science and Institute of Cybernetics (Tanel Alumäe, Einar Meister)
- Logic for Computer Science (Tarmo Uustalu)
- Data Mining (Innar Liiv)

State of the art

- Deep Blue defeated the reigning world chess champion Garry Kasparov in 1997
- Proved a mathematical conjecture (Robbins conjecture) unsolved for decades
- No hands across America (driving autonomously 98% of the time from Pittsburgh to San Diego)
- 100000s of miles of autonomous driving (Google car)
- IBM DeepBlue played Chess at grandmaster level
- NASA's on-board autonomous planning program controlled the scheduling of operations for a spacecraft
- `Proverb` solves crossword puzzles better than most humans
- IBM Watson won in Jeopardy! in 2011.
- Medical diagnosis (e.g. diagnosis of diabetes)
- Robotics: Mars rovers, underwater robots, factories
- Learning to play ATARI games with deep reinforcement learning

30 July 2014 Last updated at 10:29 GMT

UK to allow driverless cars on public roads in January

COMMENTS (47)

The BBC's Jon Ironmonger finds out how to 'drive' a driverless car

The UK government has announced that driverless cars will be allowed on public roads from January next year.

It also invited cities to compete to host one of three trials of the tech, which would start at the same time.

In addition, ministers ordered a review of the UK's road regulations to provide appropriate guidelines.

The Department for Transport had originally pledged to let self-driving cars be trialled on public roads by the end of 2013.

Related Stories

Google to build self-driving cars

UK to road test driverless cars

FBI: Driverless cars could be lethal

A peek into ethical questions

- AI has the same ethical problems as other, conventional artifacts.
  - Understanding what a realistic experience of AI is, is likely to help us better judge what it means to be human
- Developers of AI have an obligation NOT to exploit people's ignorance and make them think AI is human
- Robots are not really your friends
  - We build them with certain goals. Both buyers and builders should pick goals sensibly

Solving problems by searching

Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms

Problem-solving agents

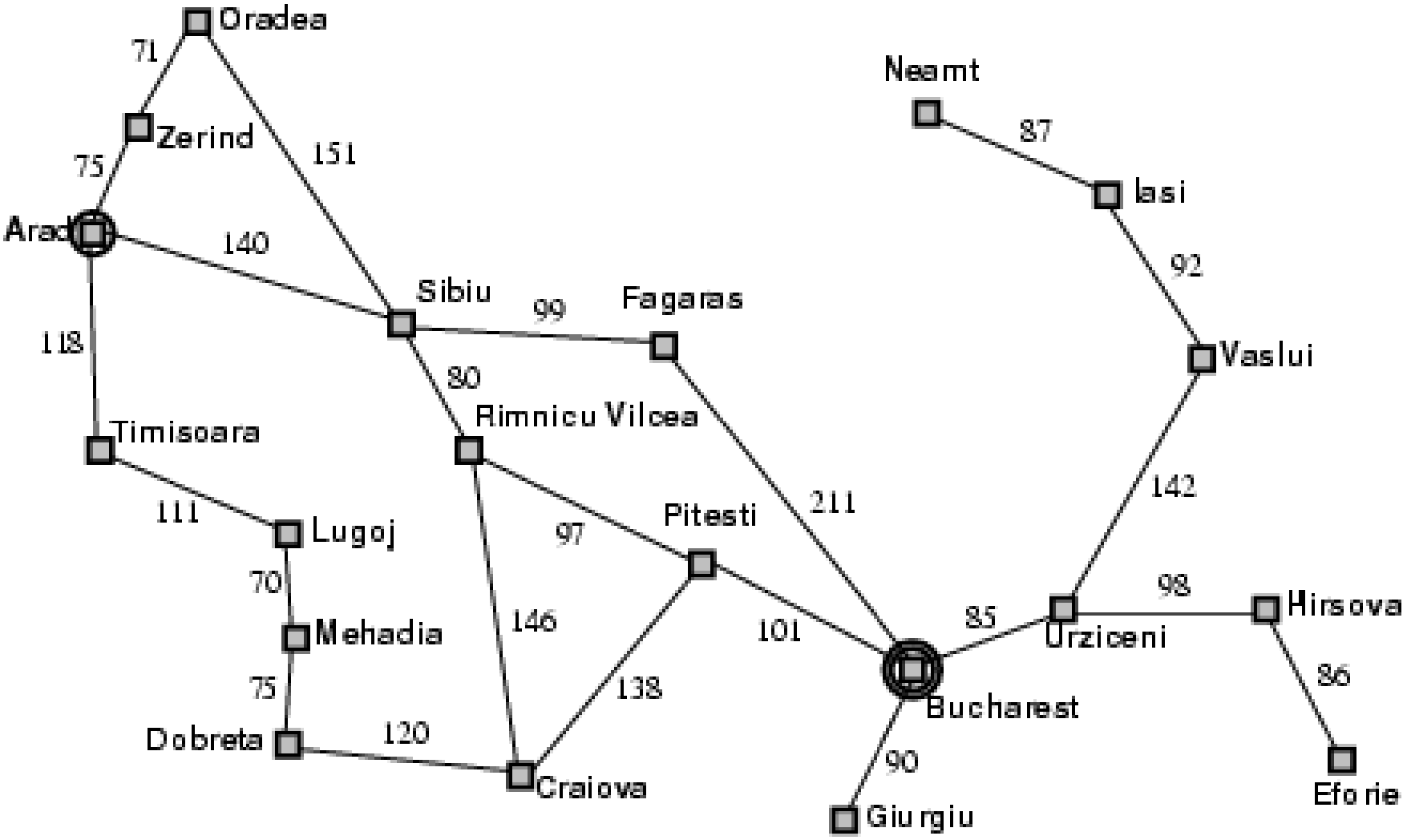
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

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

Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania

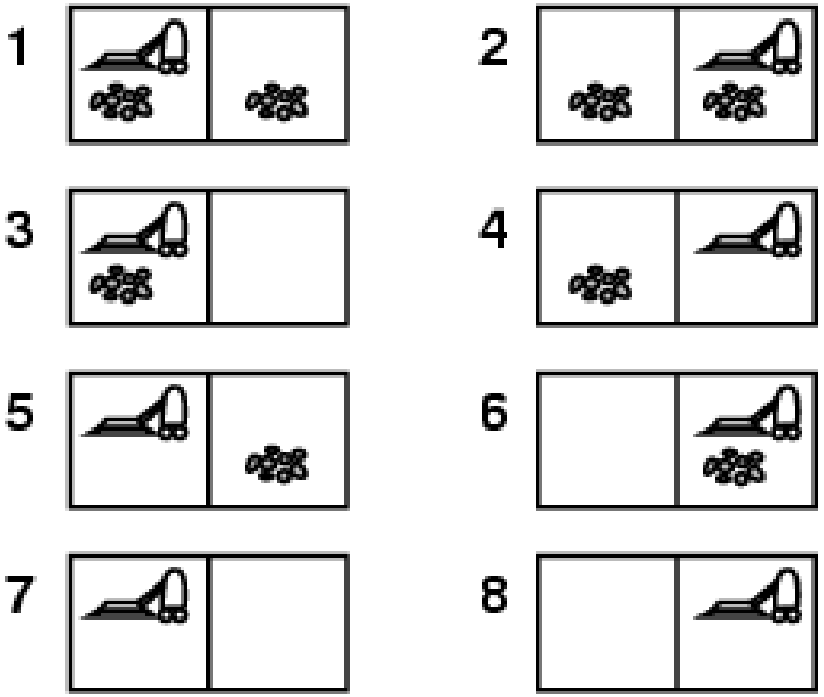


Problem types

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable → contingency problem
  - percepts provide new information about current state
  - often interleave search and execution
- Unknown state space → exploration problem

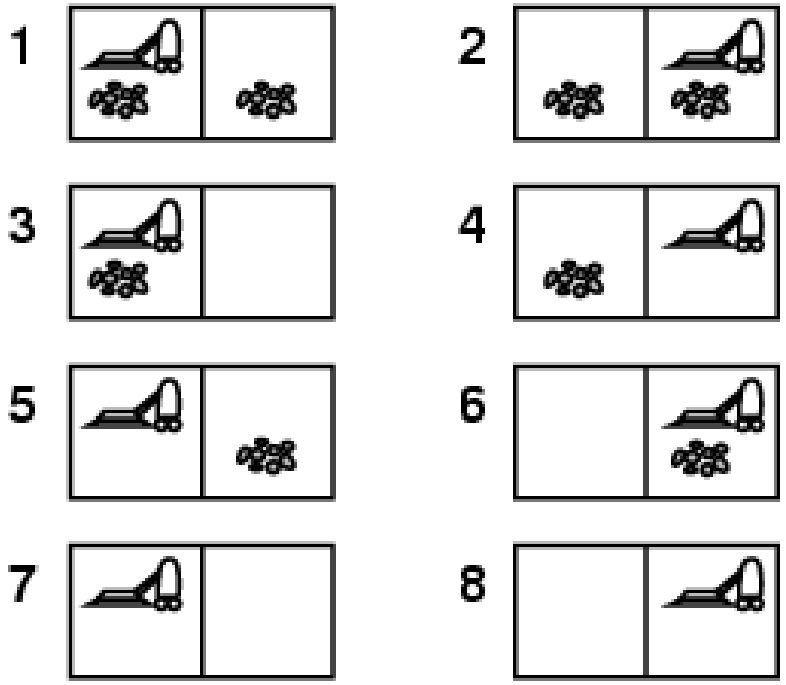
Example: vacuum world

- Single-state, start in #5.
  Solution?



Example: vacuum world

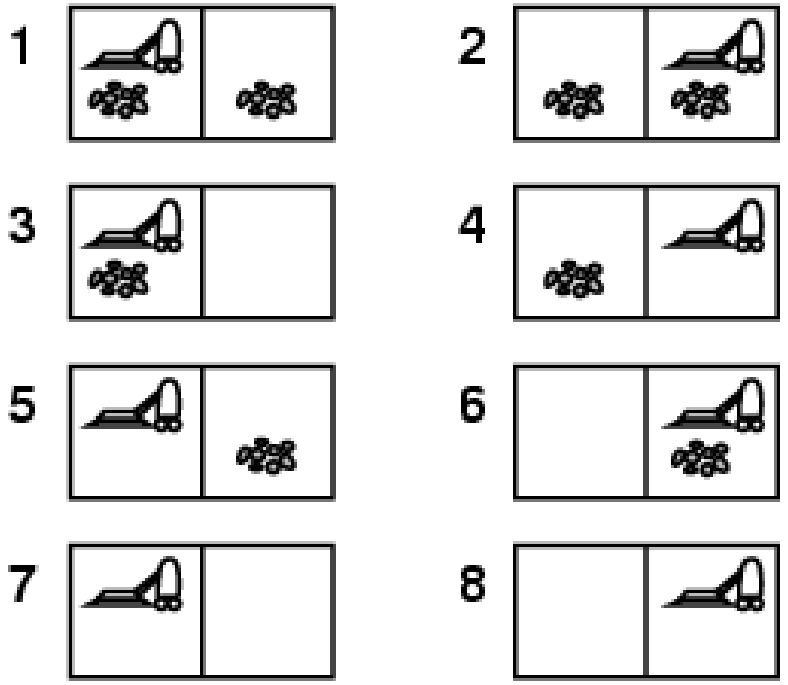
- Single-state, start in #5. Solution? *[Right, Suck]*
- Sensorless, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*} Solution?



Example: vacuum world

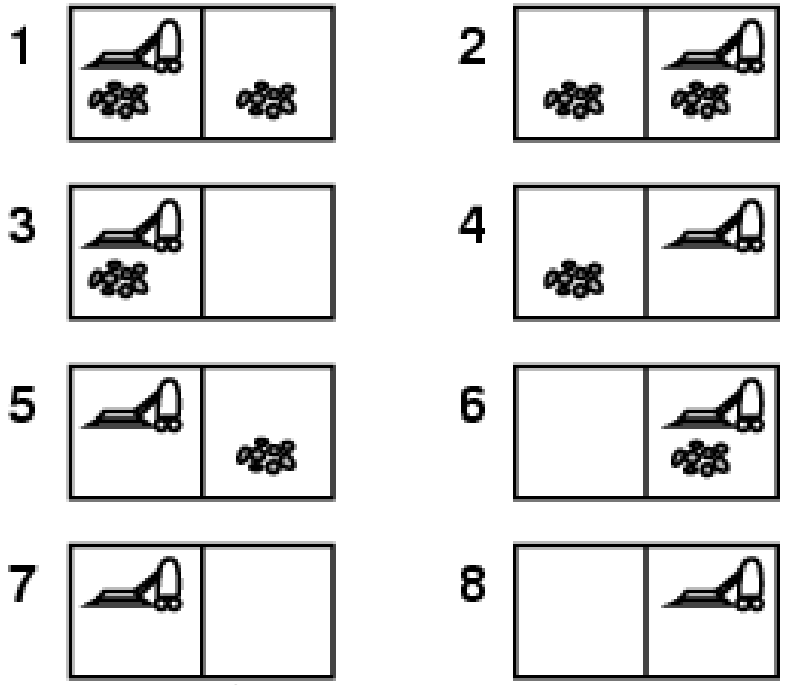
- Sensorless, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*}
  Solution?
  [Right,Suck,Left,Suck]

- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt
  - Percept: *[L, Clean],* i.e., start in #5 or #7
    Solution?



Example: vacuum world

- Sensorless, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*}
  Solution?
  [Right,Suck,Left,Suck]



- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean],* i.e., start in #5 or #7
    Solution? *[Right, **if** dirt **then** Suck]*

Single-state problem formulation

A problem is defined by four items:

1. initial state e.g., "at Arad"
2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad)* = {<*Arad* → *Zerind, Zerind*>, … }
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0

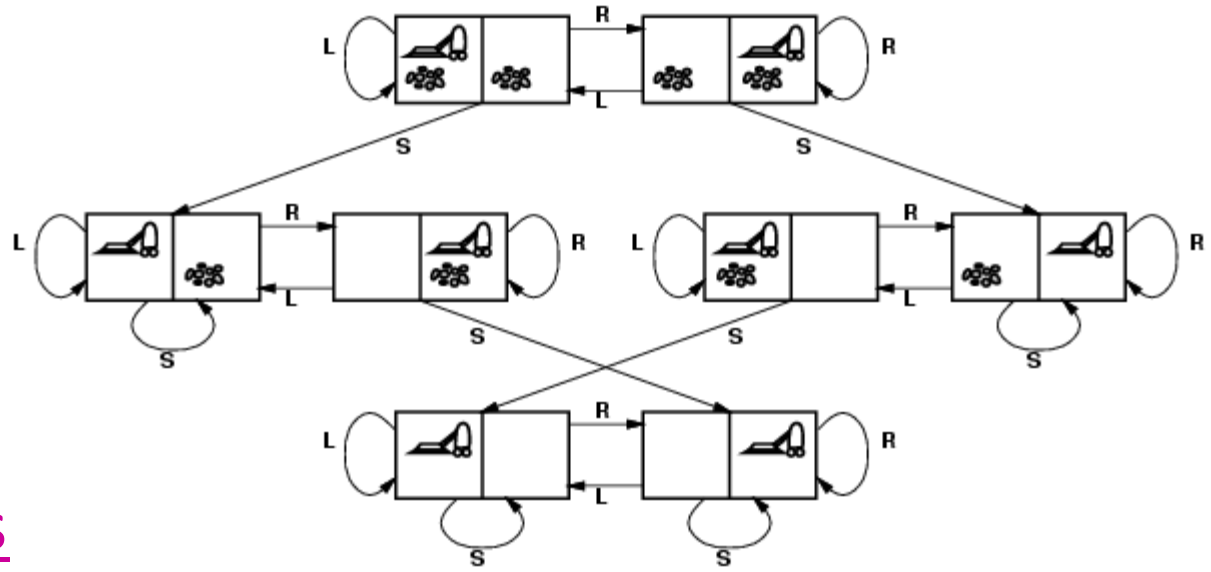
- A solution is a sequence of actions leading from the initial state to a goal state

Selecting a state space

- Real world is absurdly complex
  → state space must be abstracted for problem solving

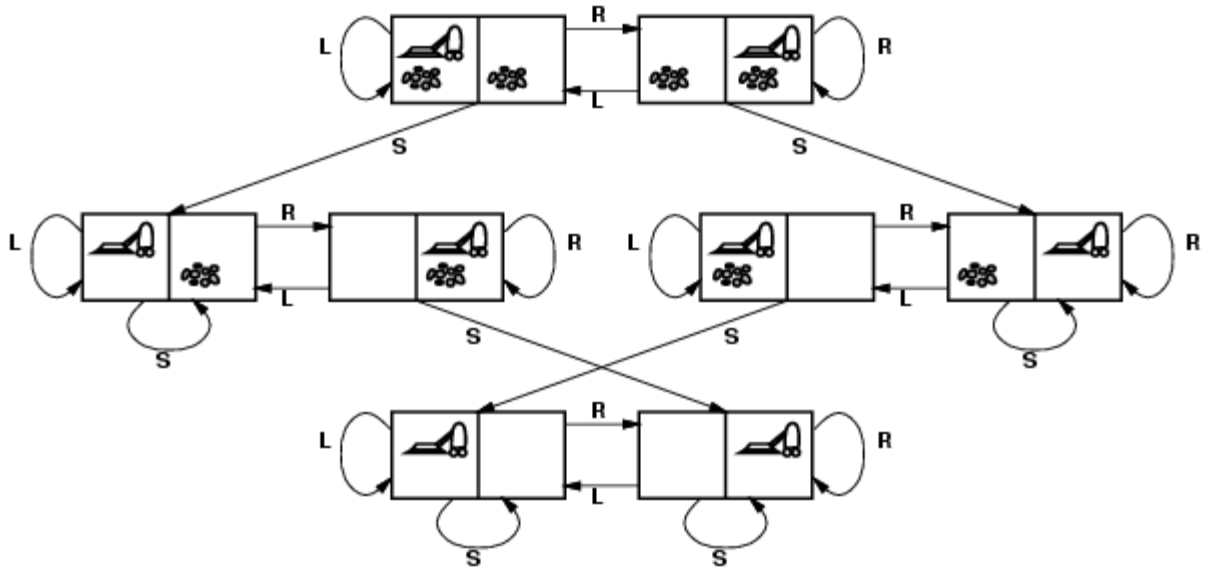
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad“ must get to some real state "in Zerind"
- (Abstract) solution =
  - set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

Vacuum world state space graph



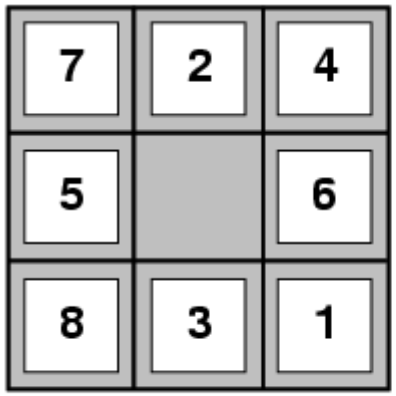
- states
- actions?
- goal test?
- path cost?
-

Vacuum world state space graph

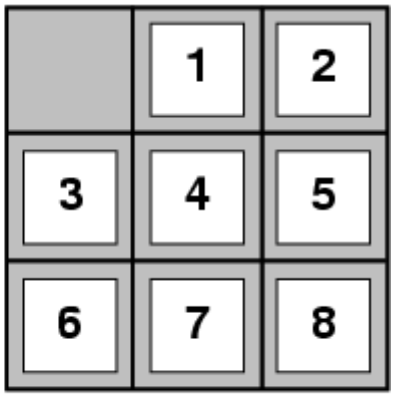


- states? integer dirt and robot location
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

Example: The 8-puzzle



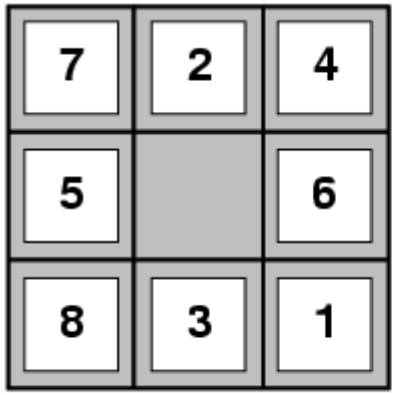
Start State



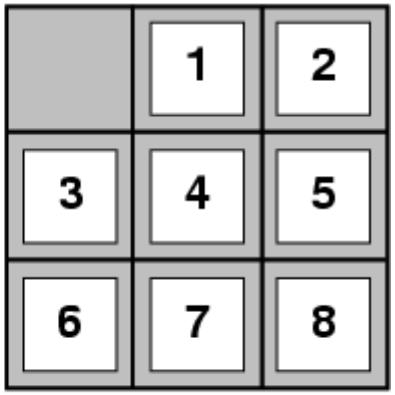
Goal State

- states?
- actions?
- goal test?
- path cost?

Example: The 8-puzzle



Start State

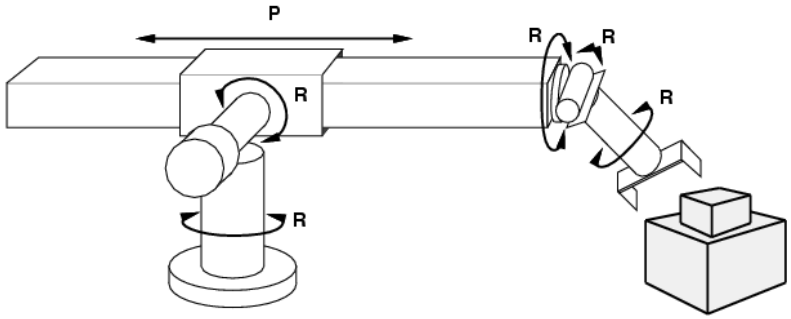


Goal State

- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

[Note: optimal solution of *n*-Puzzle family is NP-hard]

Example: robotic assembly



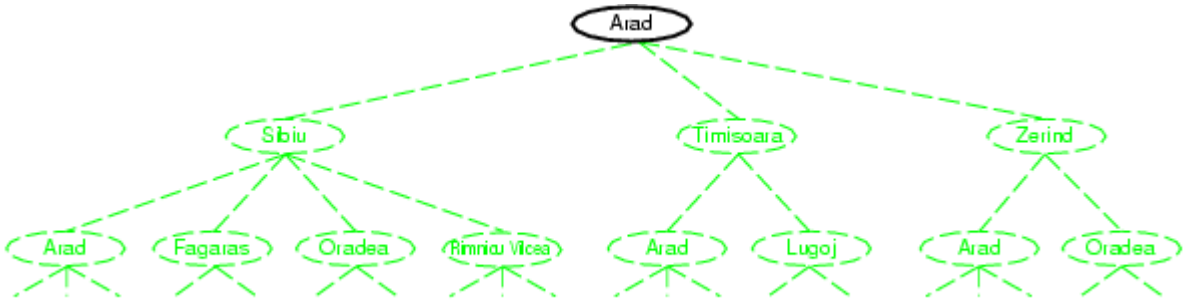
- states?: real-valued coordinates of robot joint angles parts of the object to be assembled
- actions?: continuous motions of robot joints
- goal test?: complete assembly
- path cost?: time to execute

Tree search algorithms

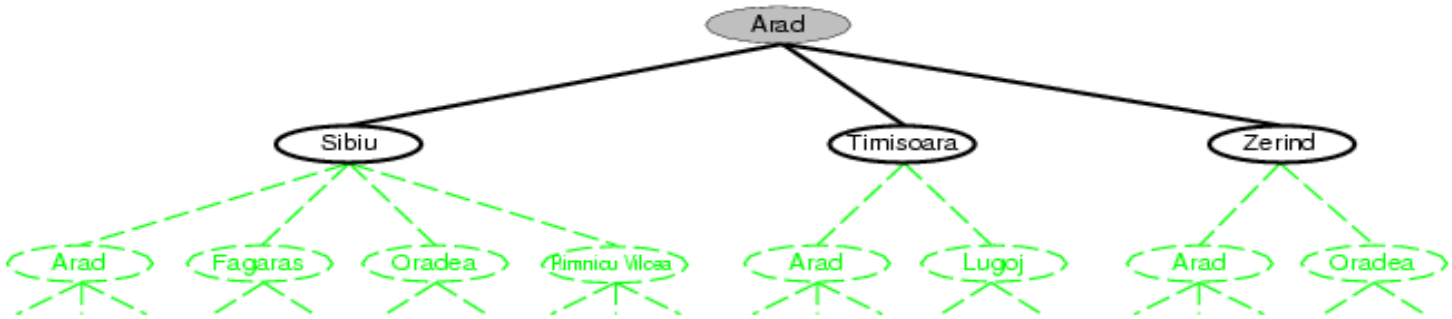
- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

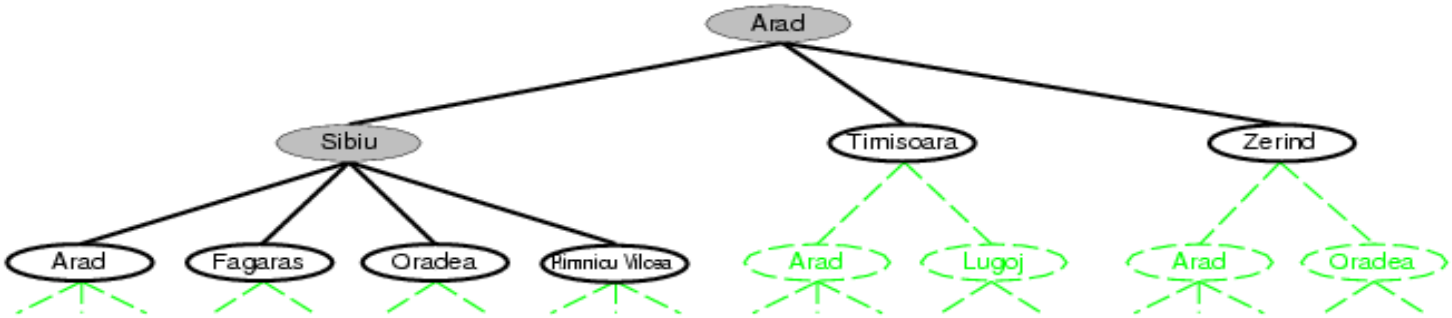
Tree search example



Tree search example



Tree search example



Implementation: general tree search

```
function TREE-SEARCH( problem, frontier ) returns a solution, or failure
 frontier ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), frontier)
   loop do
       if frontier empty then return failure
       node ← REMOVE-FRONT( (frontier)
       if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
     frontier ← INSERTALL(EXPAND(node, problem), frontier)
```

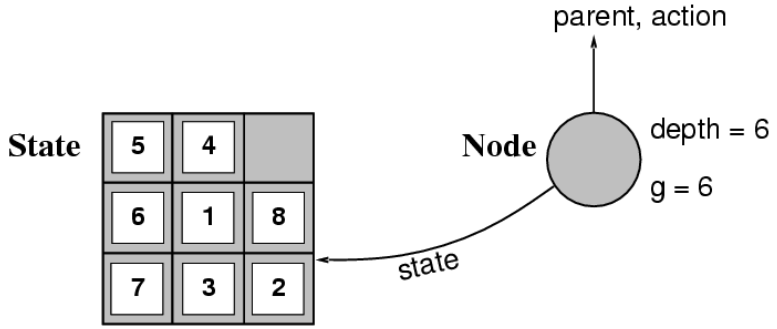
```
function EXPAND( node, problem) returns a set of nodes
   successors ← the empty set
   for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
       s ← a new NODE
       PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
       PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
       DEPTH[s] ← DEPTH[node] + 1
       add s to successors
   return successors
```

Tree search in Python

```
def tree_search(problem, frontier):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Don't worry about repeated paths to a state. [Fig. 3.7]"""
    frontier.append(Node(problem.initial))
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None
```

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost *g(x)*, depth



- The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.

Graph search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Frontier is sometimes called *fringe.*

Graph search in Python

```
def graph_search(problem, frontier):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    If two paths reach a state, only use the first one. [Fig. 3.7]"""
    frontier.append(Node(problem.initial))
    explored = set()
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        explored.add(node.state)
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored
                        and child not in frontier)
    return None
```

Acknowledgements

- This set of slides contains several prepared by Hwee Tou Ng and Stuart Russell, available from the AIMA pages.