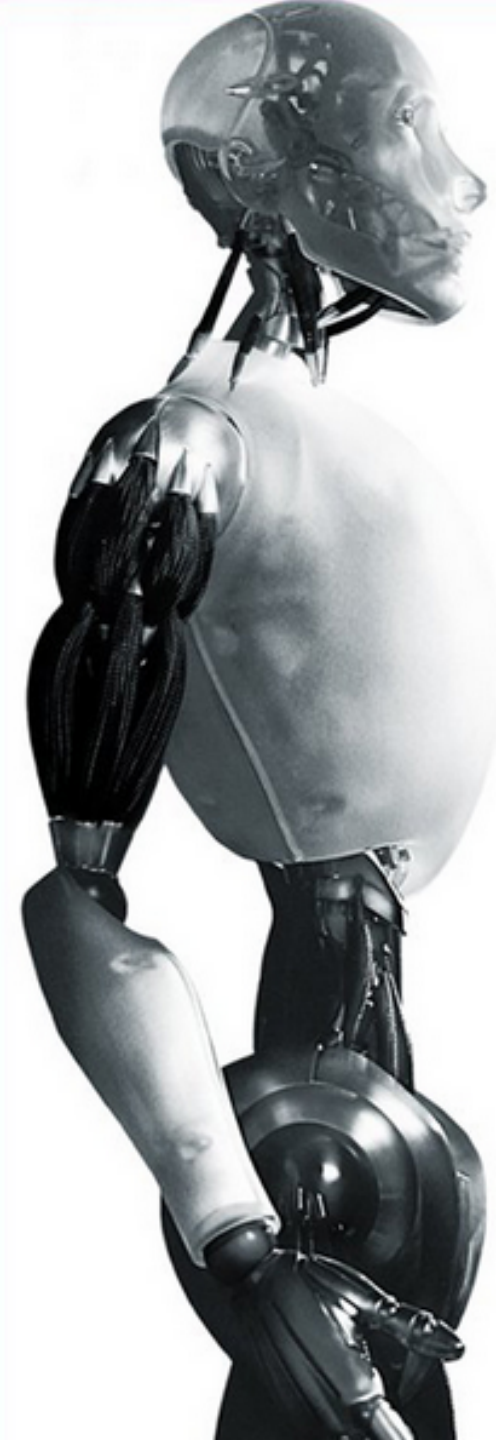
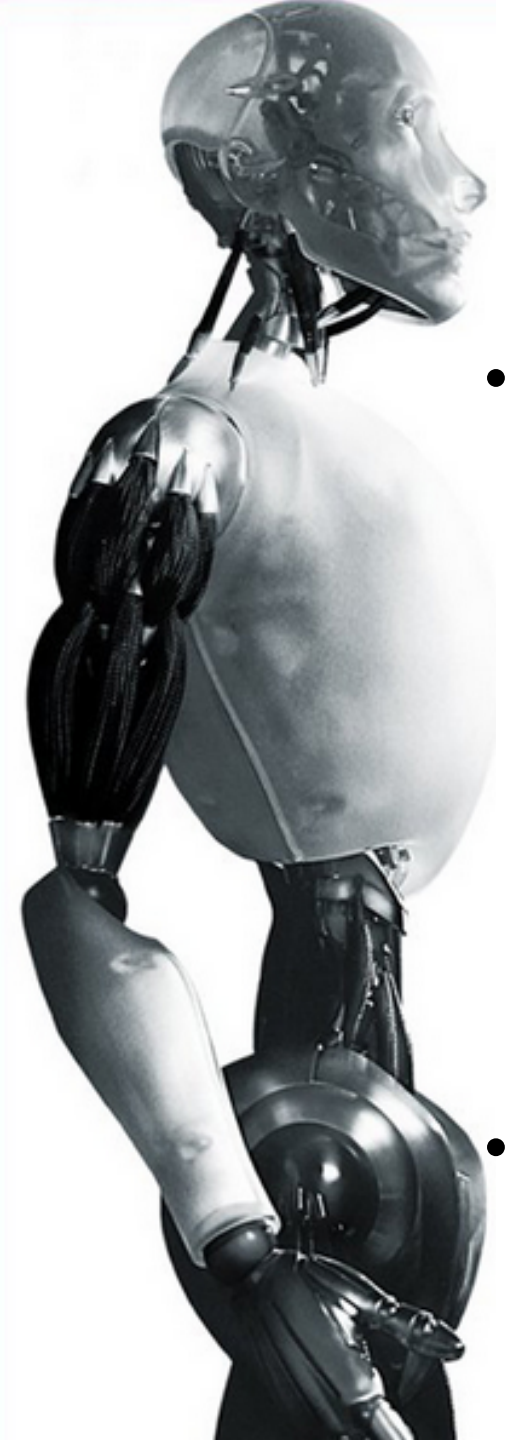


Artificial Intelligence

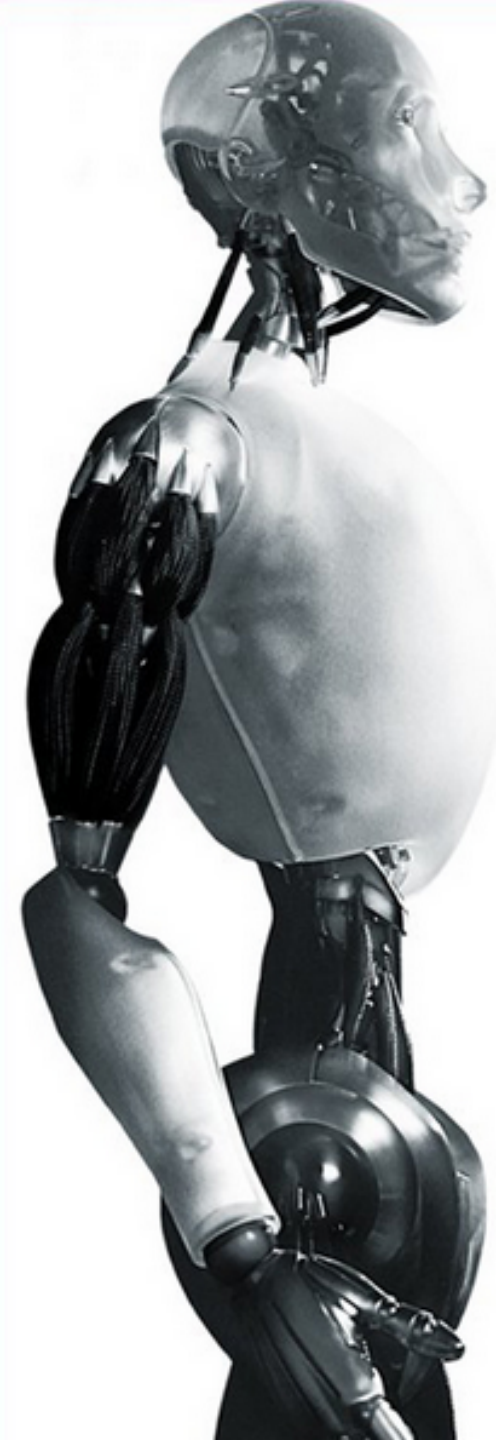
Searching

Dr Alexiei Dingli



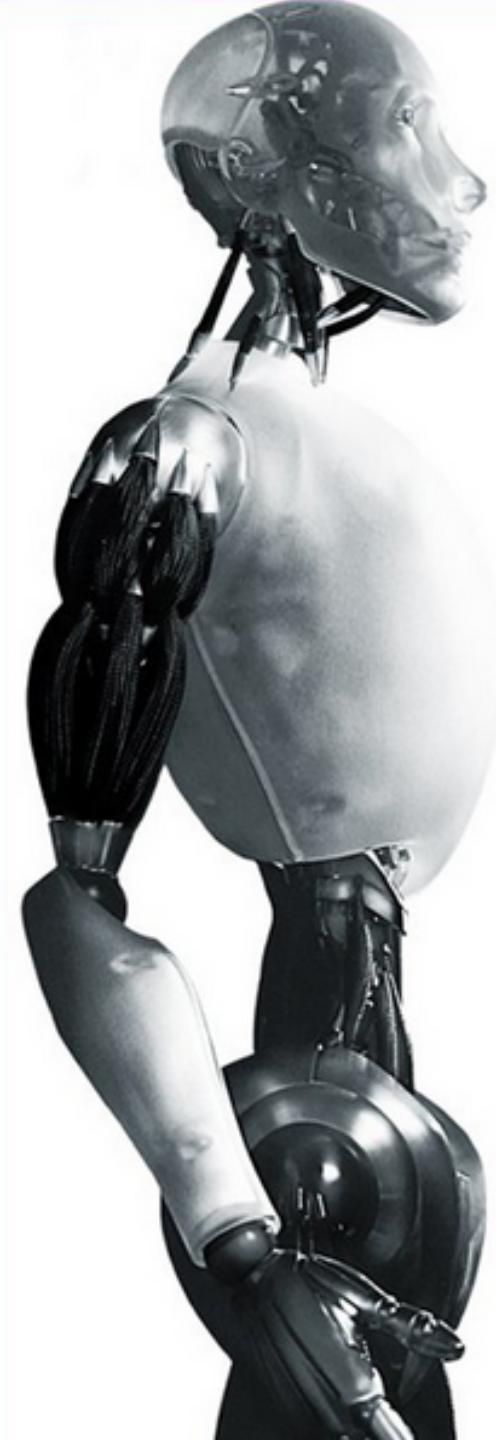
Search and AI

- In solving problems, we sometimes have to search through many possible ways of doing something.
  - We may know all the possible actions our robot can do, but we have to consider various sequences to find a sequence of actions to achieve a goal.
  - We may know all the possible moves in a chess game, but we must consider many possibilities to find a good move.
- Many problems can be formalised in a general way as search problems.



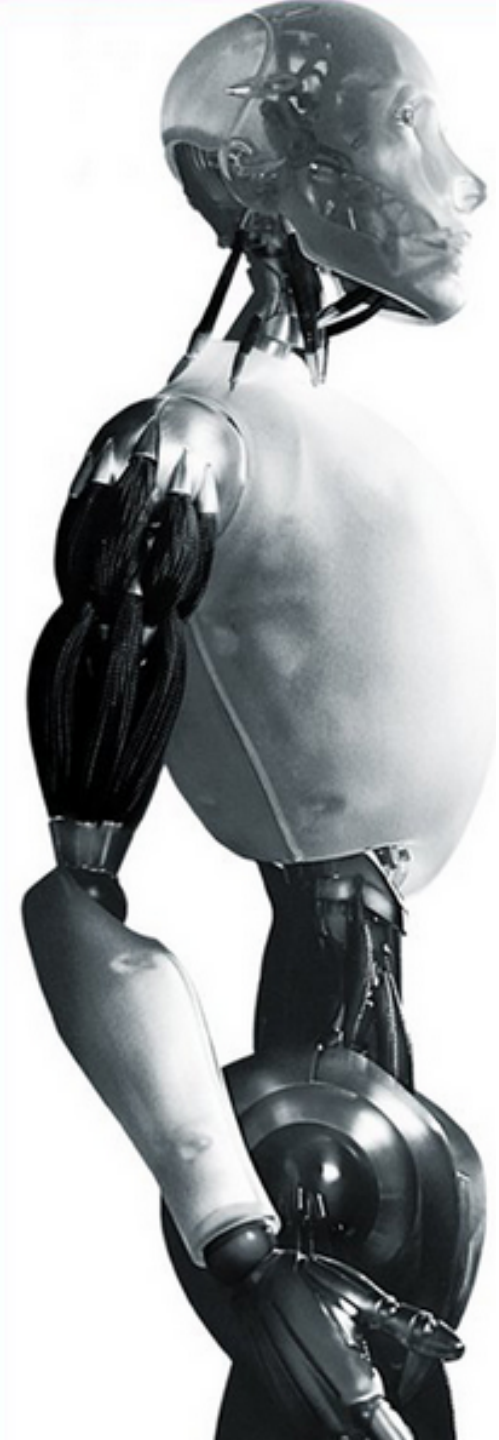
Different searches

- Two classes:
  - Algorithms that only evaluate complete solutions such as exhaustive search, etc.
  - Algorithms that require the evaluation of partially constructed or approximate solutions



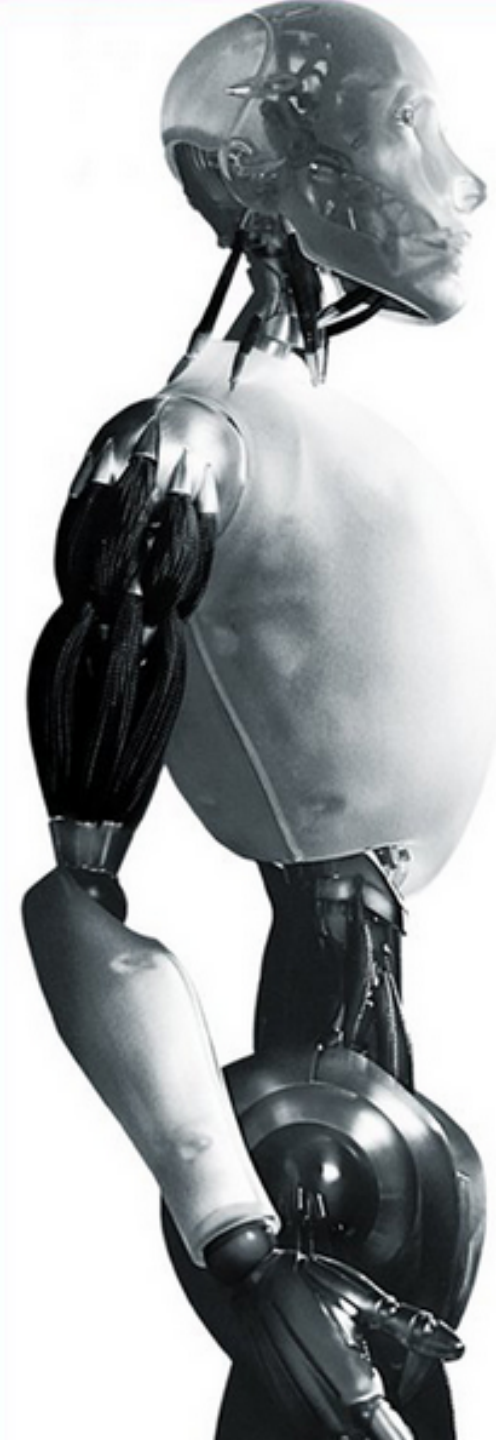
Exhaustive Search

- Checks every solution in the search space until the best solution is found
- Can be used only for small instances
- Exhaustive algorithms are simple
- Search space can be reduced by backtracking
- Some optimization methods such as A* are based on an exhaustive search



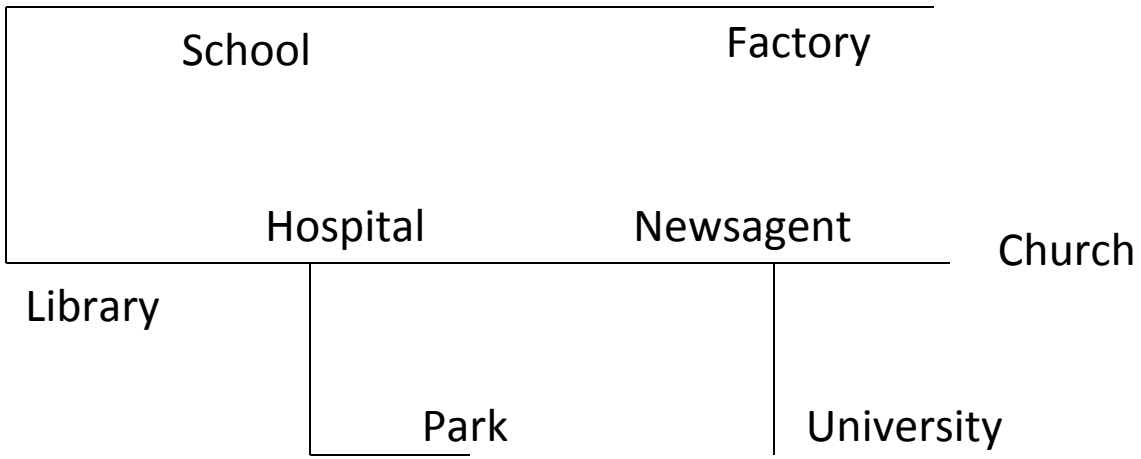
Search as Problem Solving

- Search problems described in terms of:
  - An initial state. (e.g., initial chessboard, current positions of objects in world, current location)
  - A target state.(e.g., winning chess position, target location)
  - Some possible actions, that get you from one state to another. (e.g. chess move, robot action, simple change in location).
- Search techniques systematically consider all possible action sequences to find a *path* from the initial to target state.

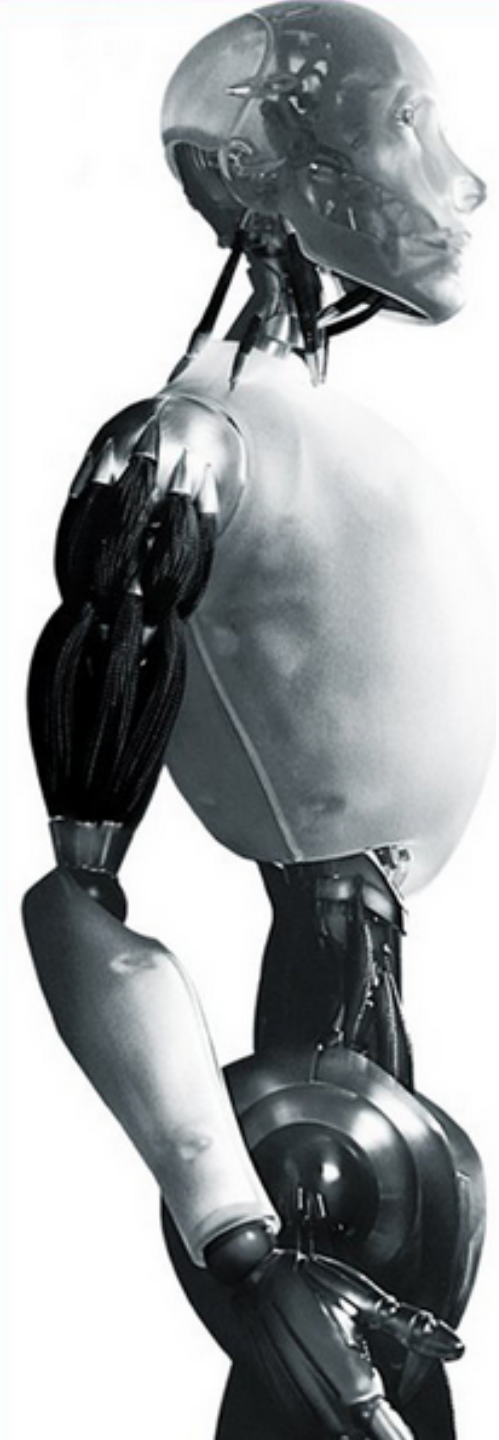


Simple Example

- Easiest to first look at simple examples based on searching for route on a map.

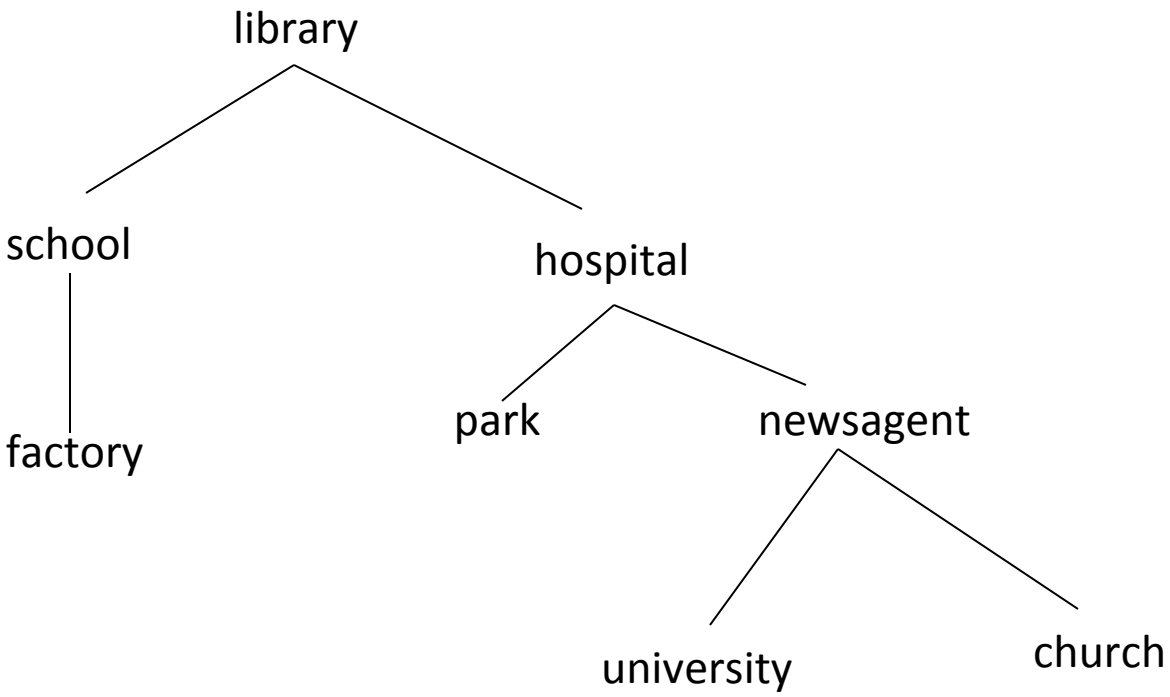


- How do we systematically and exhaustively search possible routes, in order to find, say, route from library to university?

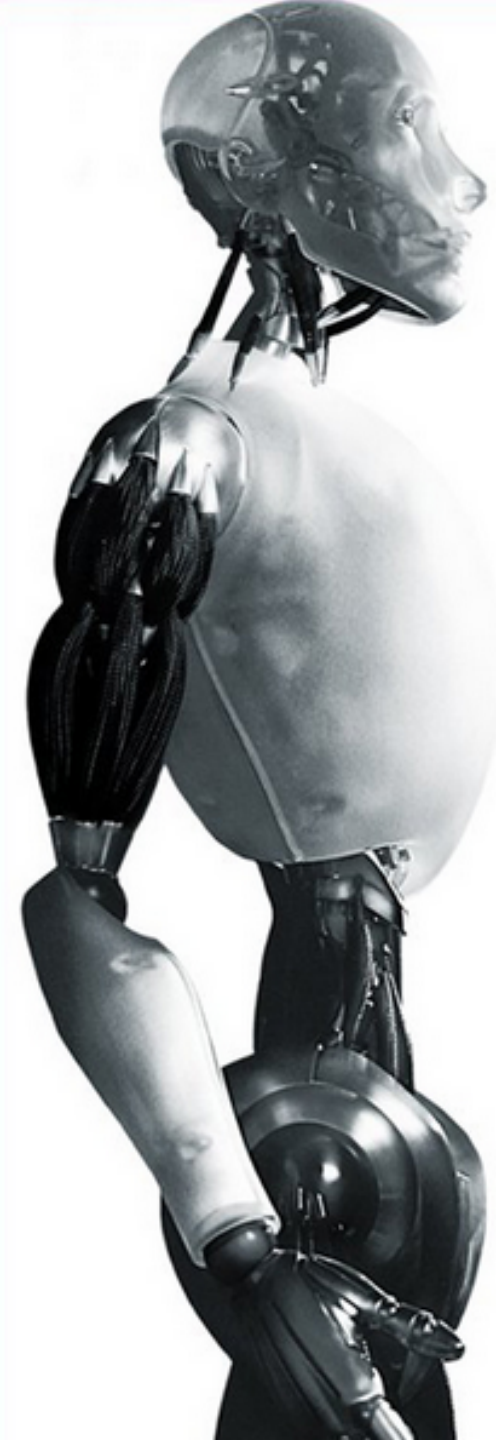


Search Space

- The set of all possible states reachable from the initial state defines the *search space*.
- We can represent the search space as a tree.

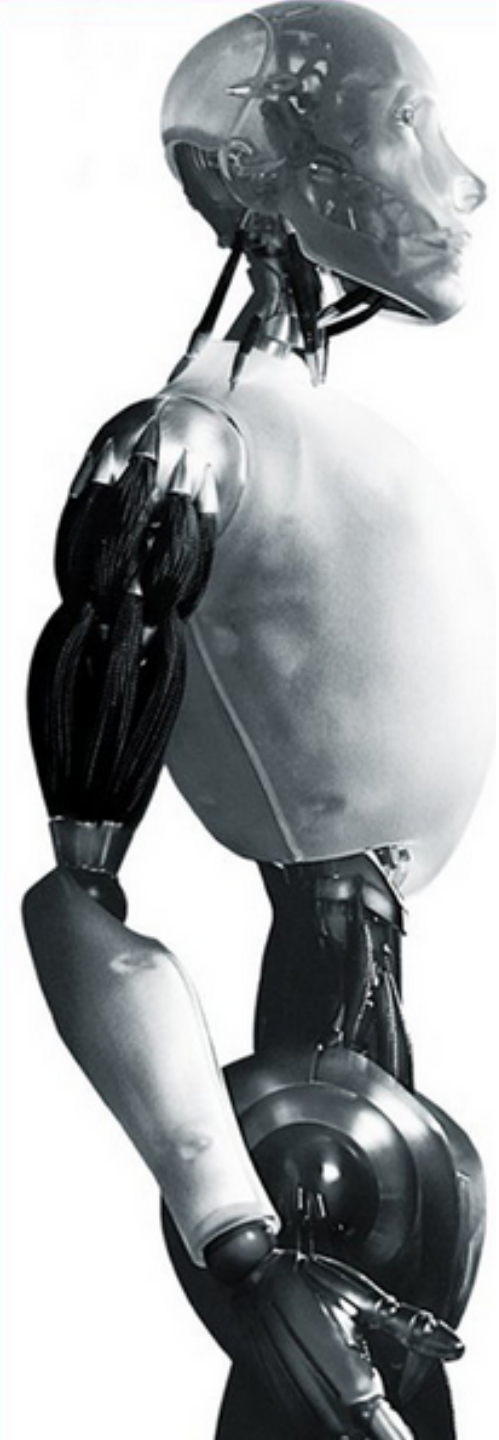


- We refer to nodes connected to and "under" a node in the tree as "successor nodes".



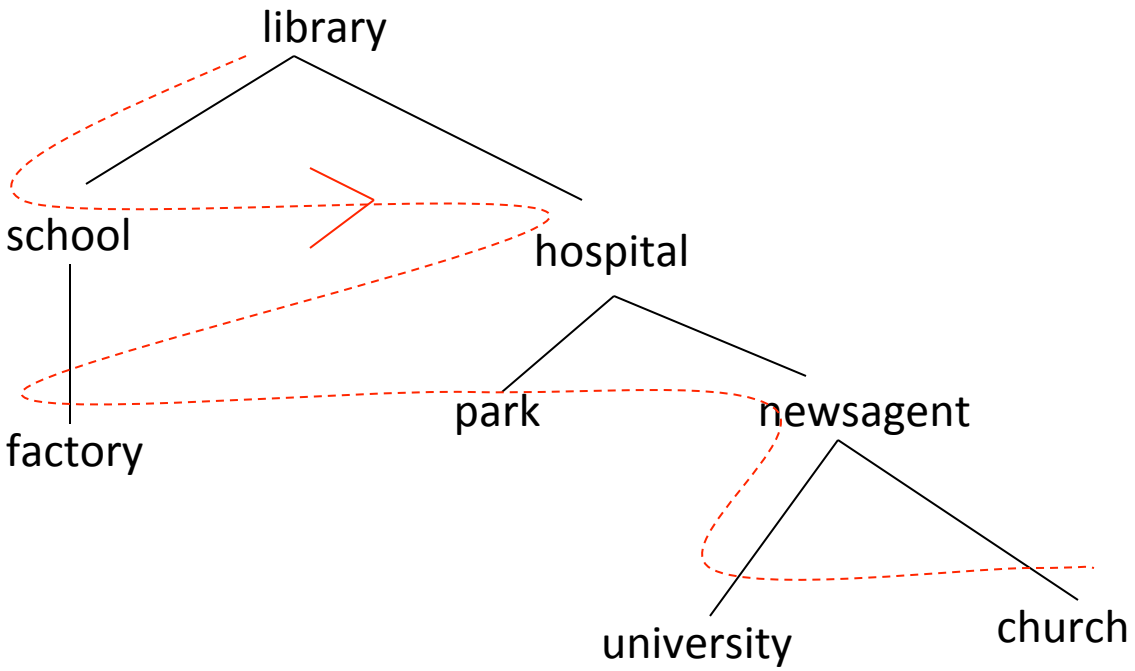
Simple Search Techniques

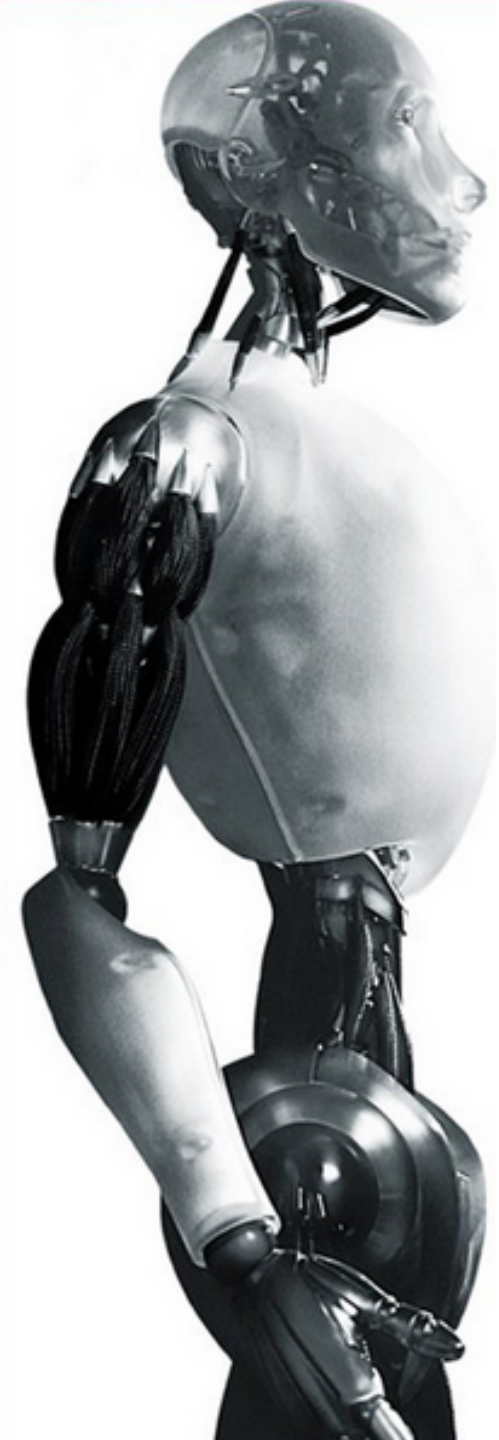
- How do we search this tree to find a possible route from library to University?
- May use simple systematic search techniques, which try every possibility in systematic way.
- Breadth first search - Try shortest paths first.
- Depth first search - Follow a path as far as it goes, and when reach dead end, backup and try last encountered alternative.



Breadth first search

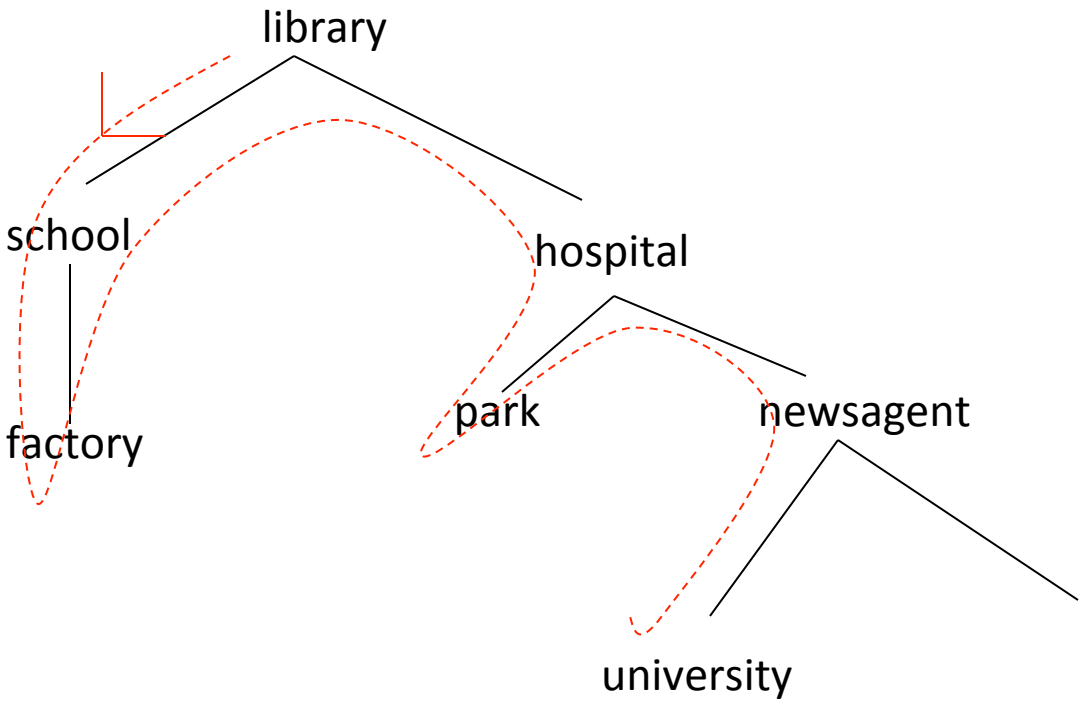
Explore *nodes* in tree order: library, school, hospital, factory, park, newsagent, uni, church. (conventionally explore left to right at each level)

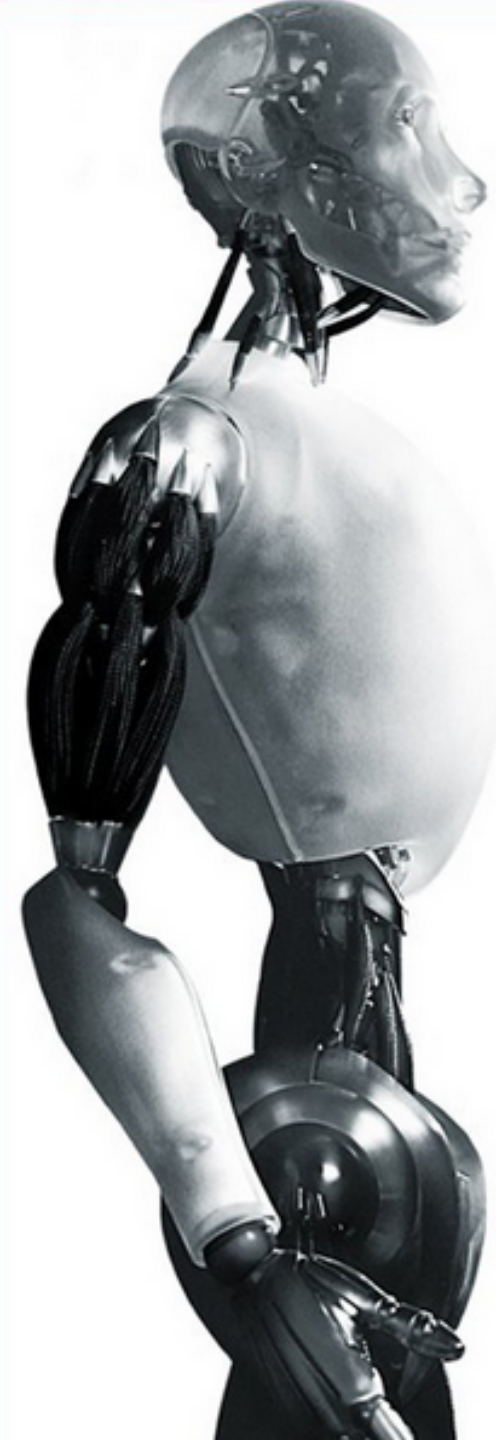




Depth first search

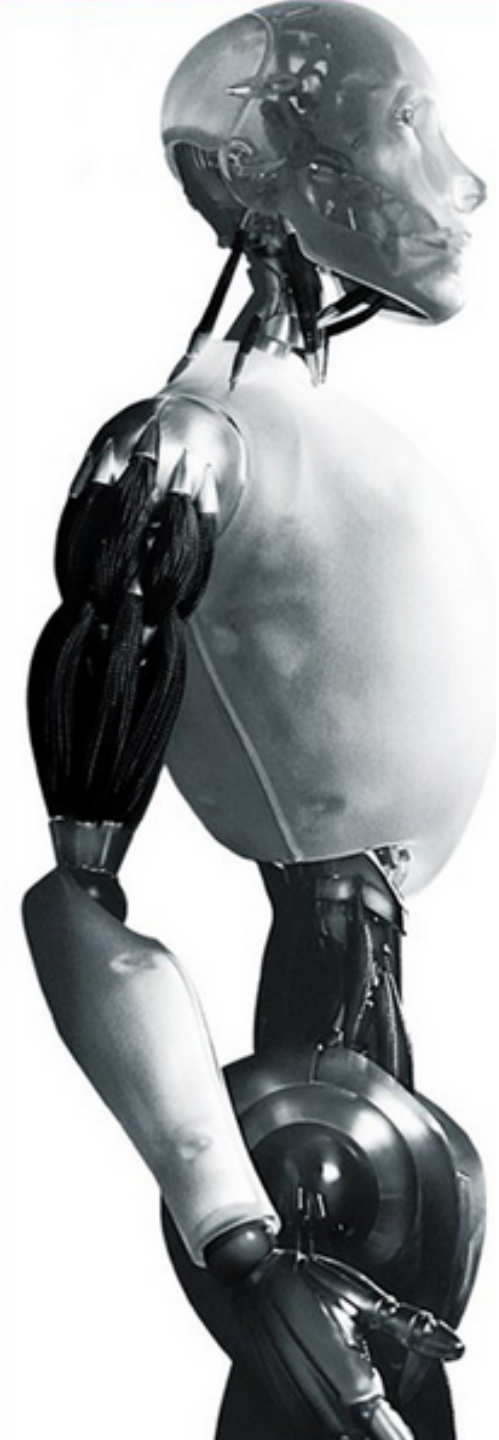
- Nodes explored in order: library, school, factory, hospital, park, newsagent, university.





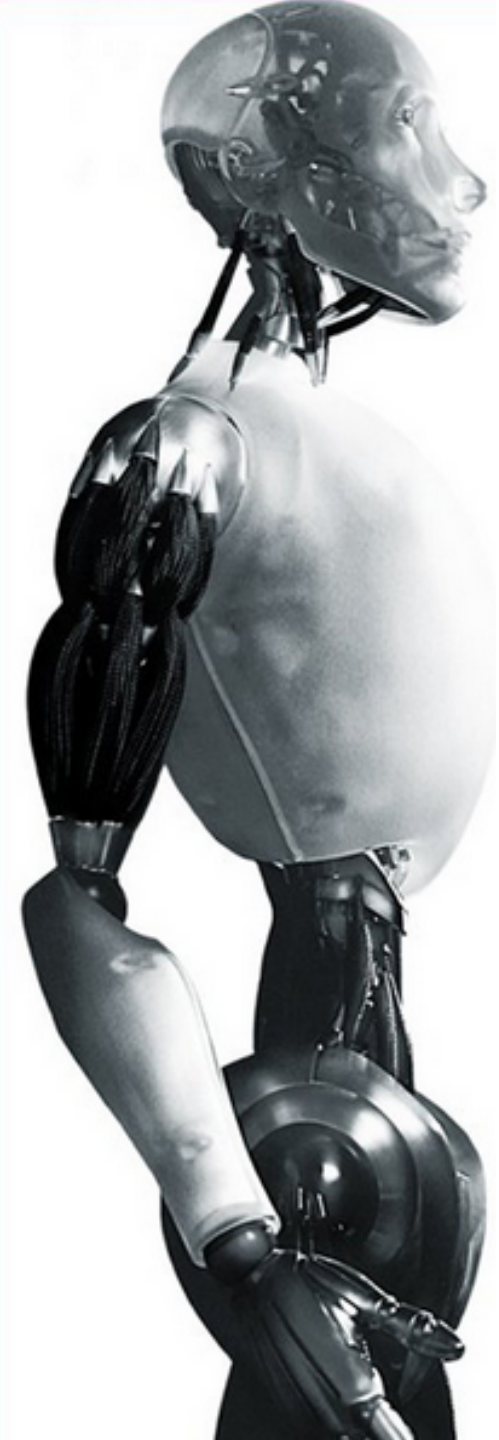
Algorithms for breadth first and depth first search.

- Very easy to implement algorithms to do these kinds of search.
- Both algorithms keep track of the list of nodes found, but for which routes from them have yet to be considered.
  - E.g., [school, hospital] -have found school and hospital in tree, but not yet considered the nodes connected to these.
- List is sometimes referred to as an agenda. But implemented using stack for depth first, queue for breadth first.



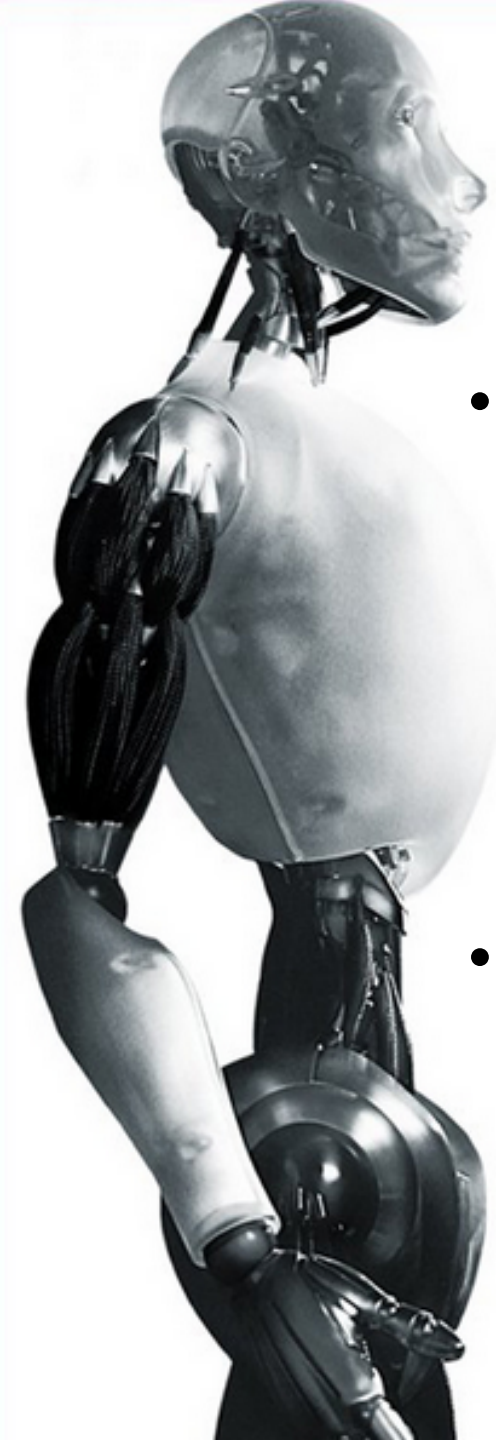
Algorithm for breadth first:

- Start with queue = [initial-state] and found=FALSE.
- While queue not empty and not found do:
  - Remove the first node N from queue.
  - If N is a goal state, then found = TRUE.
  - Find all the successor nodes of N, and put them on the end of the queue.



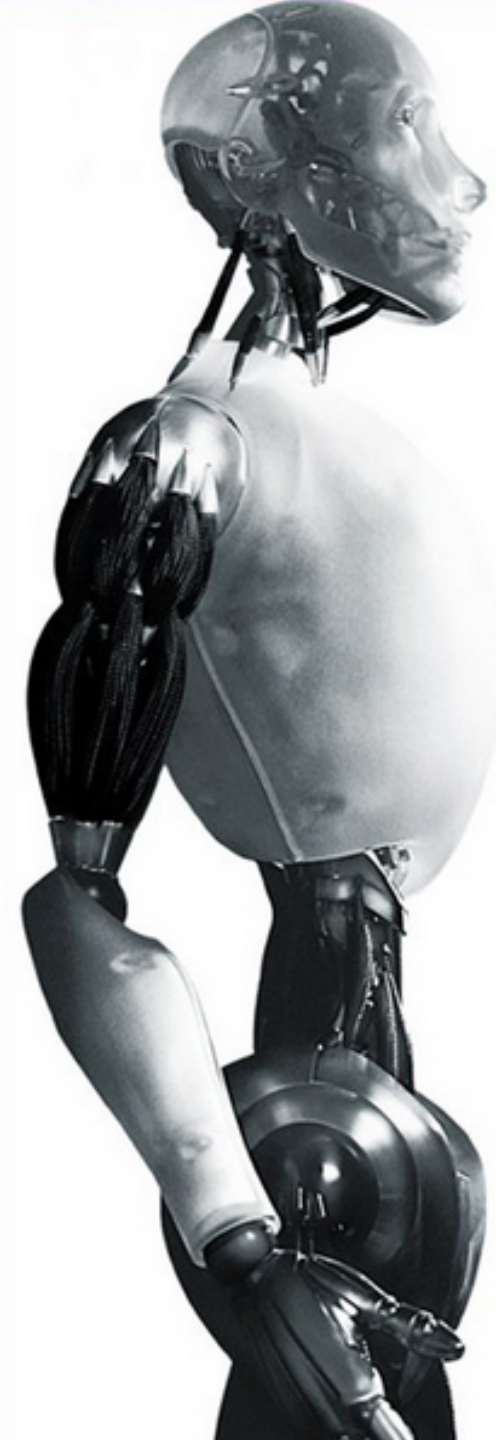
Algorithm for depth first:

- Start with stack = [initial-state] and found=FALSE.
- While stack not empty and not found do:
  - Remove the first node N from stack.
  - If N is a goal state, then found = TRUE.
  - Find all the successor nodes of N, and put them on the top of the stack.



Extensions to basic algorithm

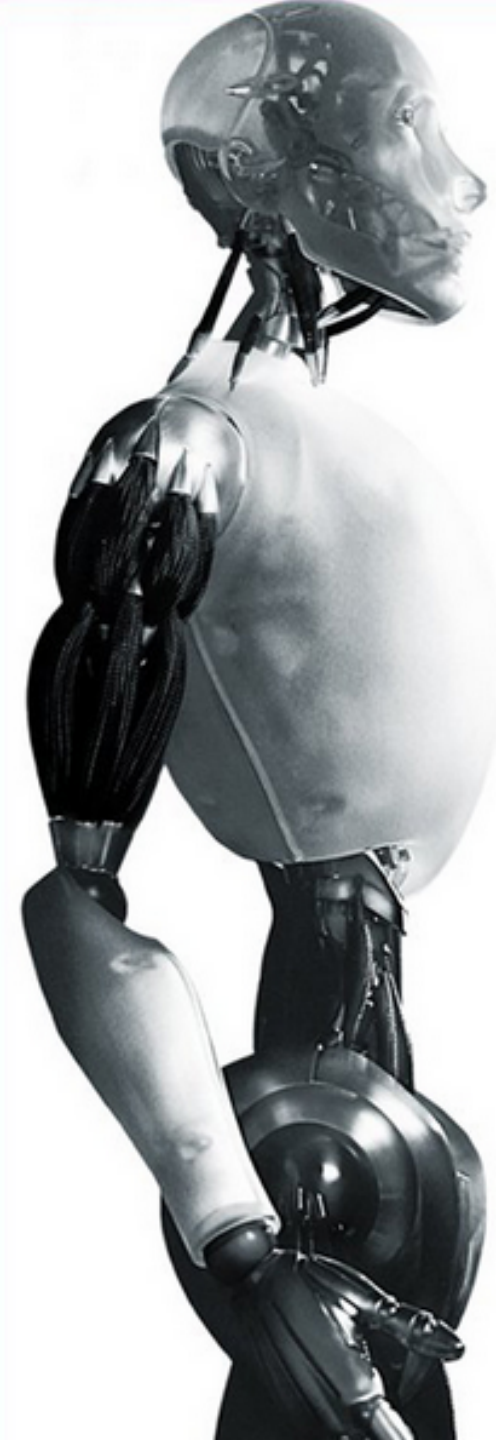
- Loops: What if there are loops (ie, we are searching a *graph*)? How do you avoid (virtually) driving round and round in circles?
  - Algorithm should keep track of which nodes have already been explored, and avoid redoing these nodes.
- Returning the path: How do you get it to actually tell you what the path it has found is!
  - One way: Make an item on the agenda be a path, rather than a node.



Problem solving as search

- How can we formulate more interesting problems as search?
- Have to think of problems in terms of initial state, target state, and primitive actions that change state.

- Consider:
  - Game playing: actions are moves, which change the board state.
  - Planning robot behaviours: actions are basic moves, like “open door”, or “put block1 on top of block2”, which change situation/state.

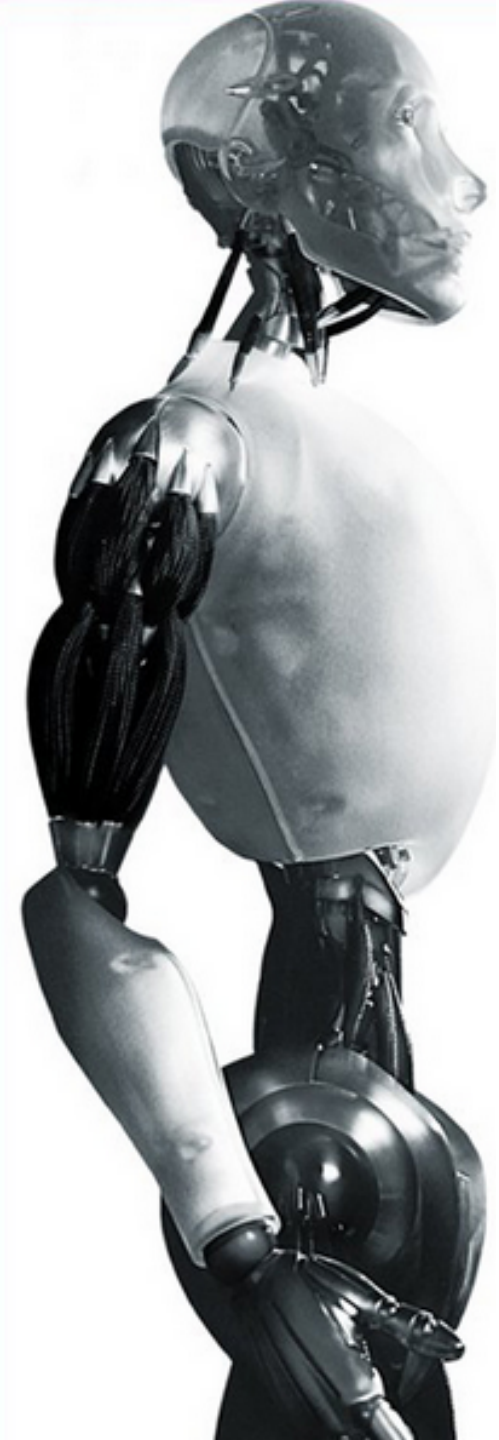


Robot planning problem.

- Consider pet robot (or not very intelligent flat-mate) in small flat with two rooms. You and your robot are in room1, your beer is in room 2, the door is closed between the rooms.

- Actions:
  - move(robot, Room, AnotherRoom)
  - open(robot, door)
  - pickup(robot, Object).

- Initial state:
  - in(robot, room1) etc.

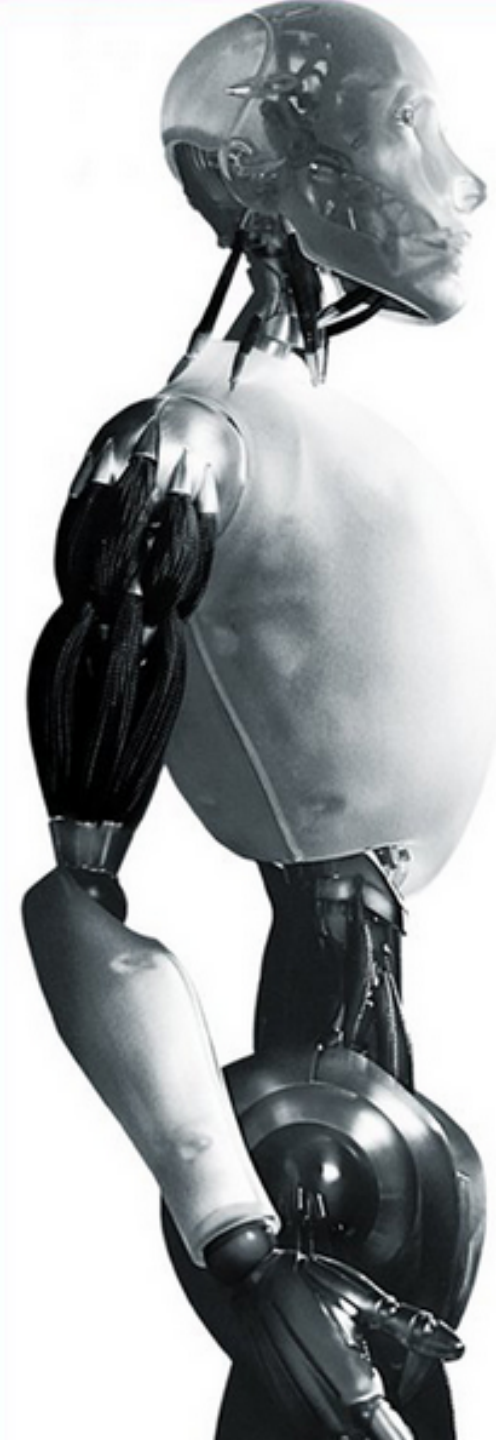


Or.. To solve a puzzle

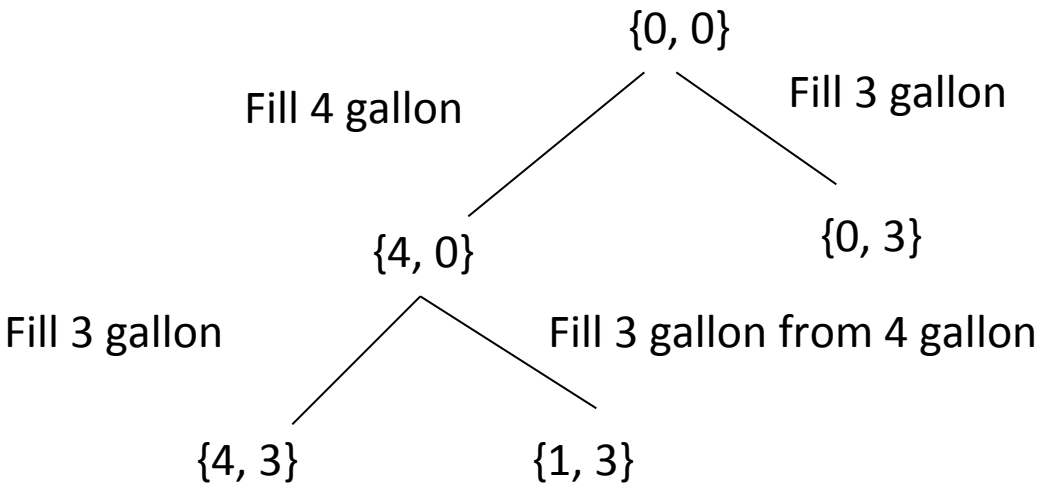
- "You are given two jugs, a 4 gallon one, and a 3 gallon one. Neither has any measuring markers on it. There is a tap that can be used to fill the jugs with water. How can you get exactly 2 gallons of water in the 4 gallon jug?"

- How do we represent the problem state? Can represent just as pair or numbers.
  - {4, 1} means 4 gallons in 4 gallon jug, 1 gallon in 3 gallon jug.

- How do we represent the possible actions.
  - Can give simple rules for how to get from old to new state given various actions.

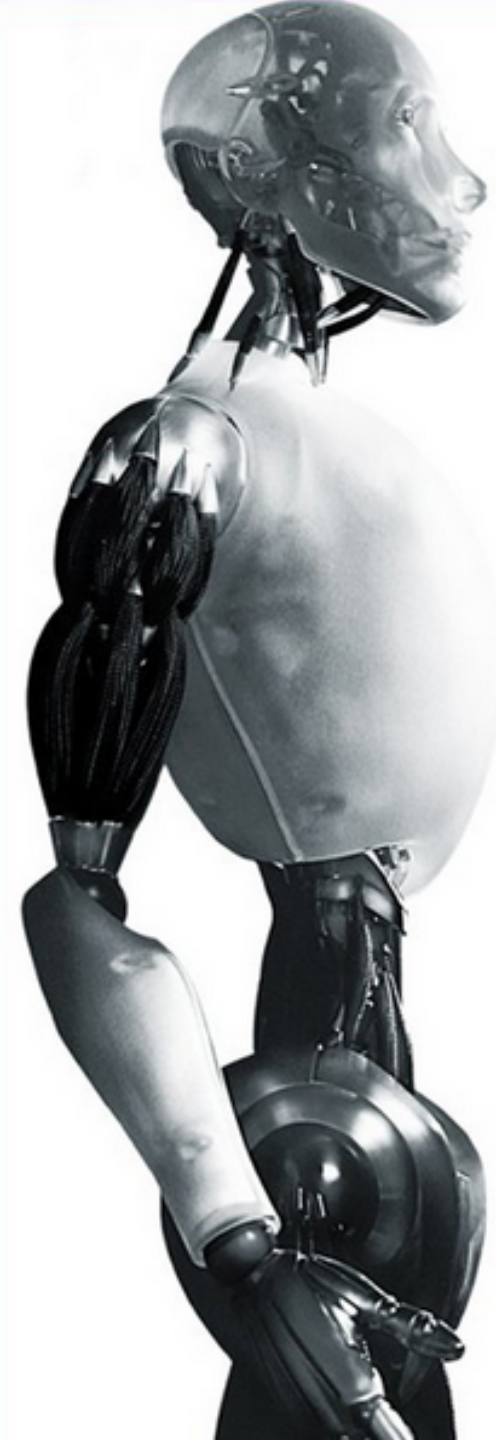


Search Tree for Jugs



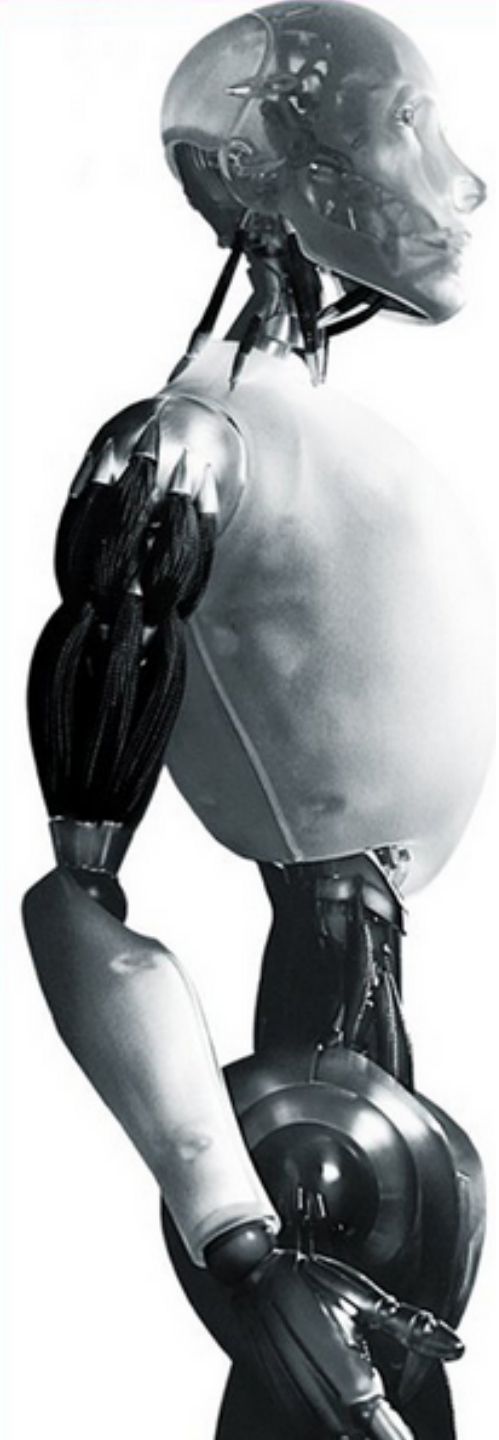
and so on …

Write the complete search tree!

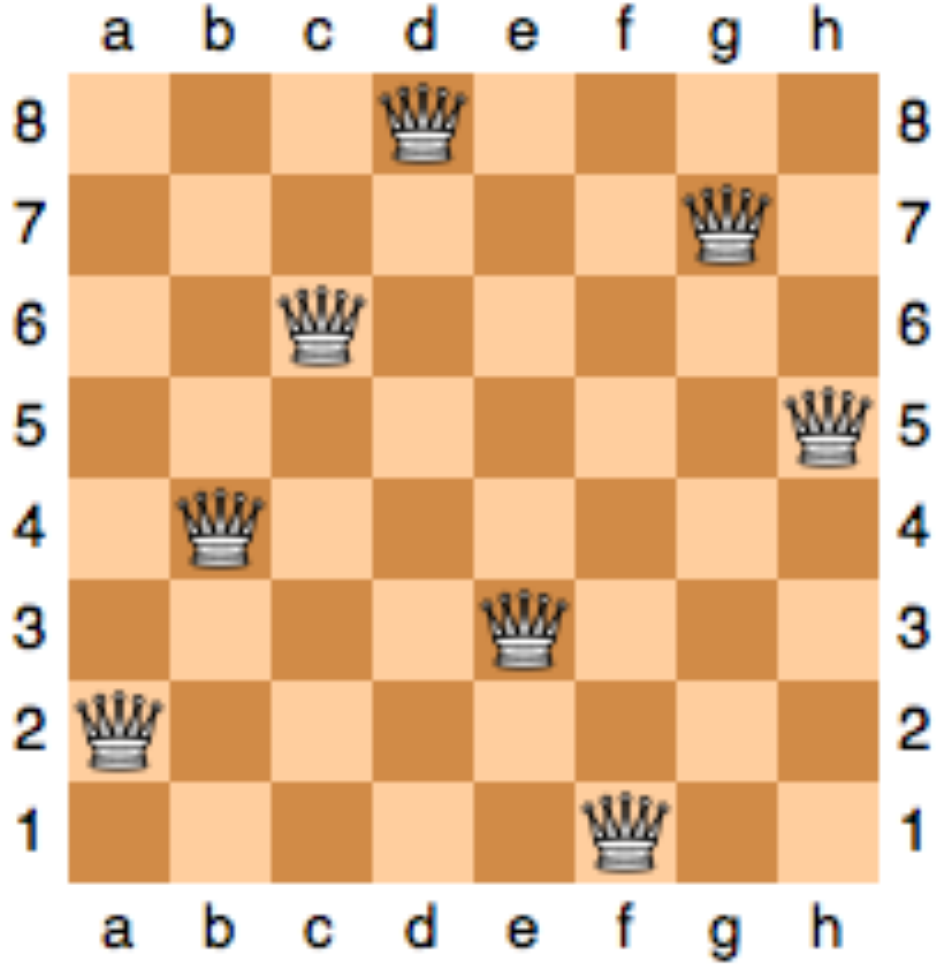


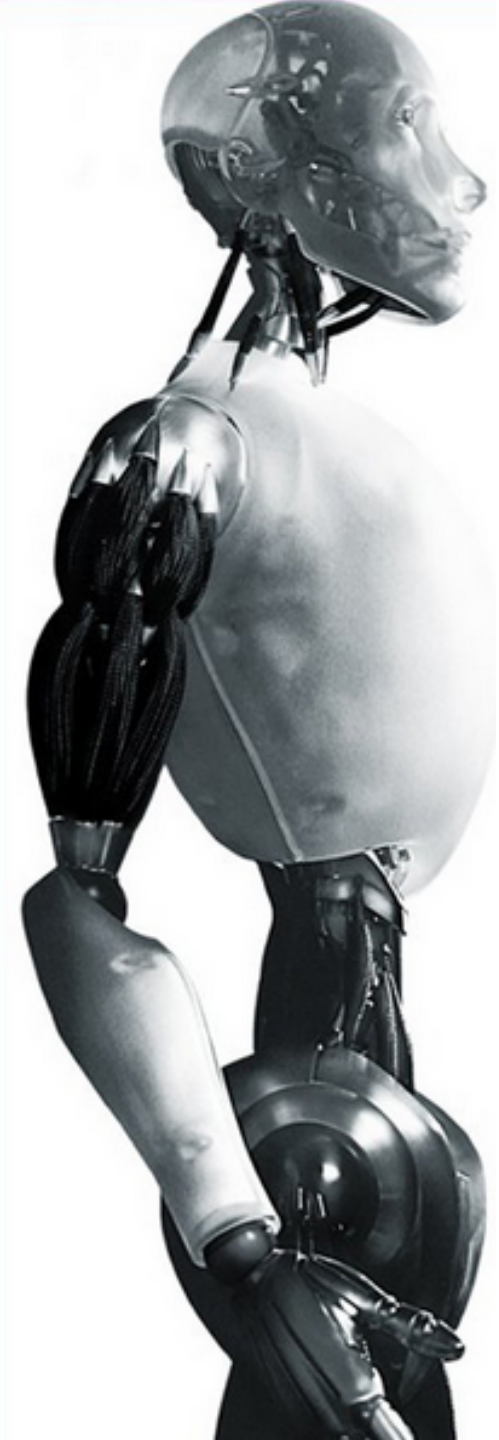
So..

- To solve a moderately complex puzzle what we can do is:
  - Express it in terms of search.
  - Decide how "problem state" may be expressed formally.
  - Decide how to encode primitive actions as rules for getting from one state to another.
  - Use a standard tree/graph search algorithm/program, which uses a general "successor state" function which you define for your problem.



The 8-Queens Puzzle

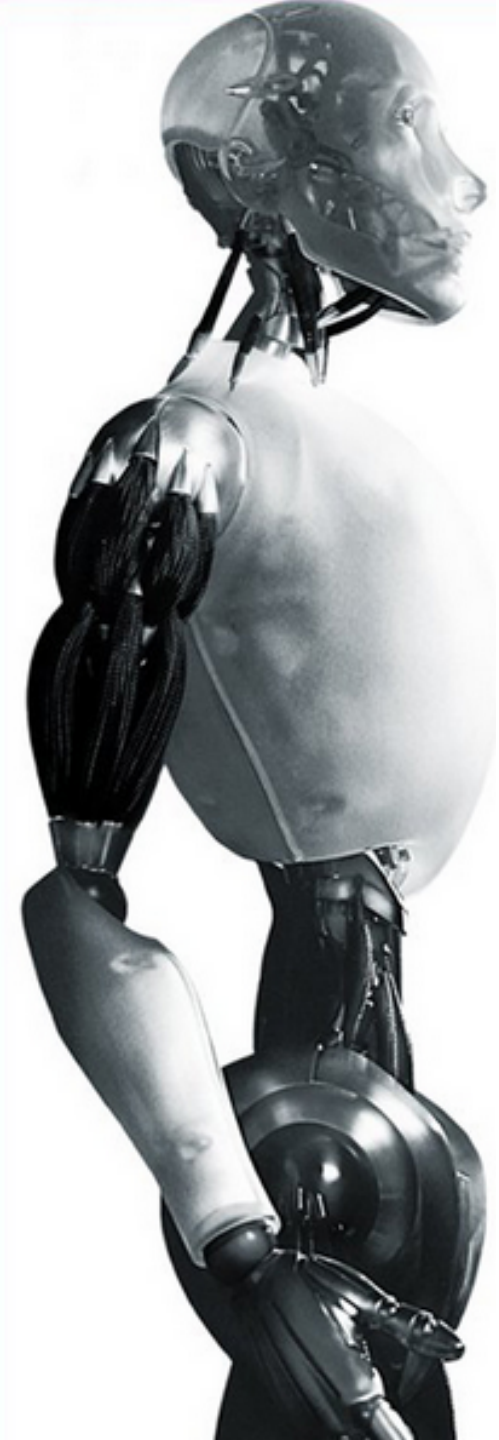




The 8-Queens Puzzle

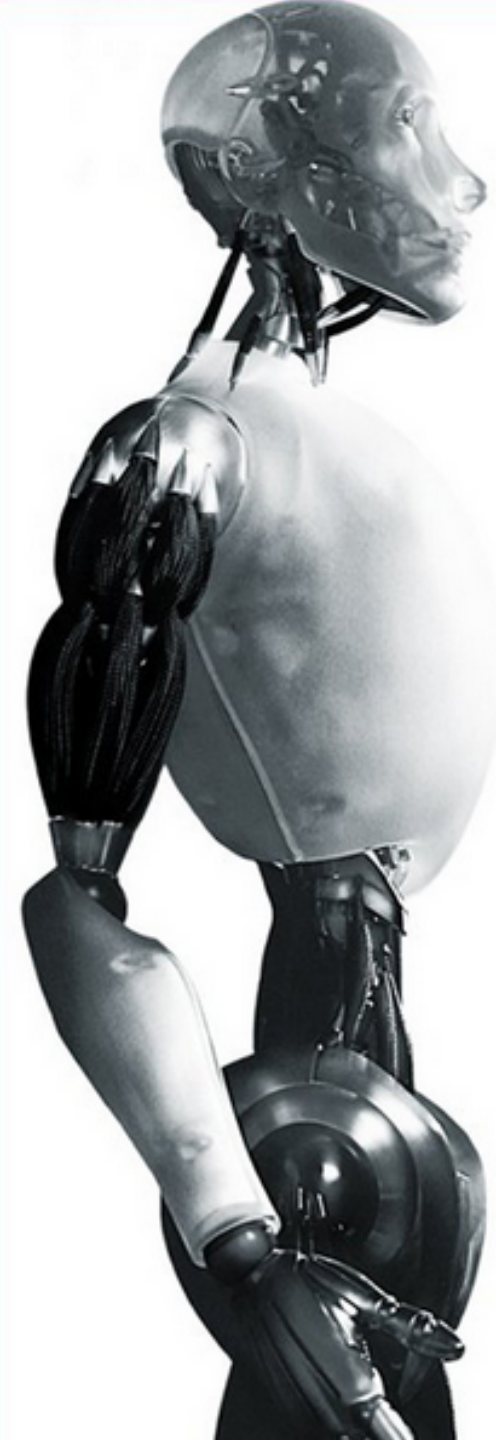
- 1848 by a Chess Player
- Place 8 chess queens
- 8 x 8 chess board
- So that no two queens attack each other
- Rules
  - Queens **cannot** share same row, column or diagonal

- n Queens problem is a variant

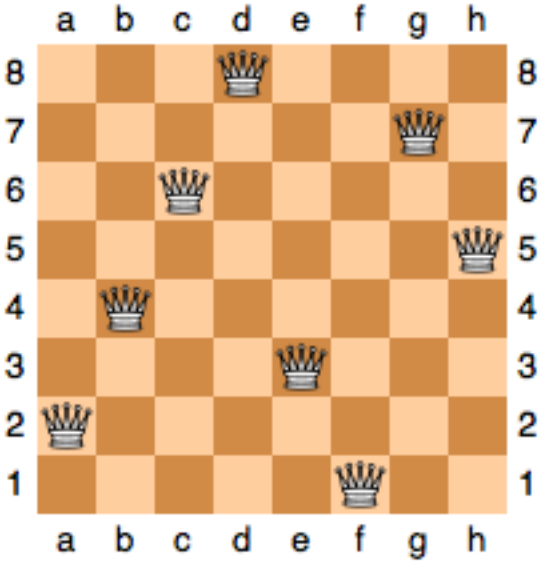


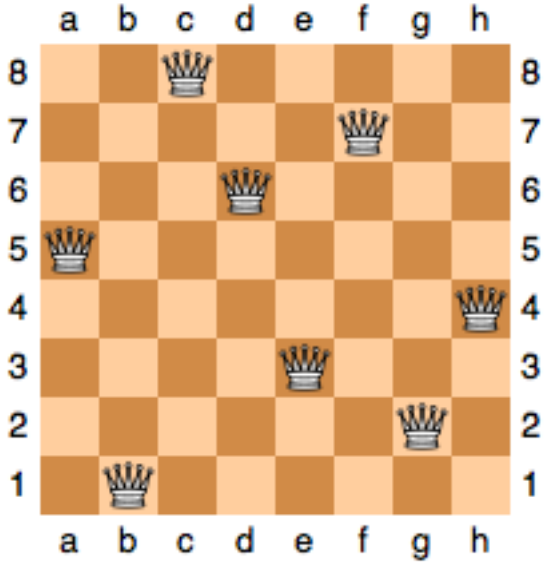
The 8-Queens Puzzle

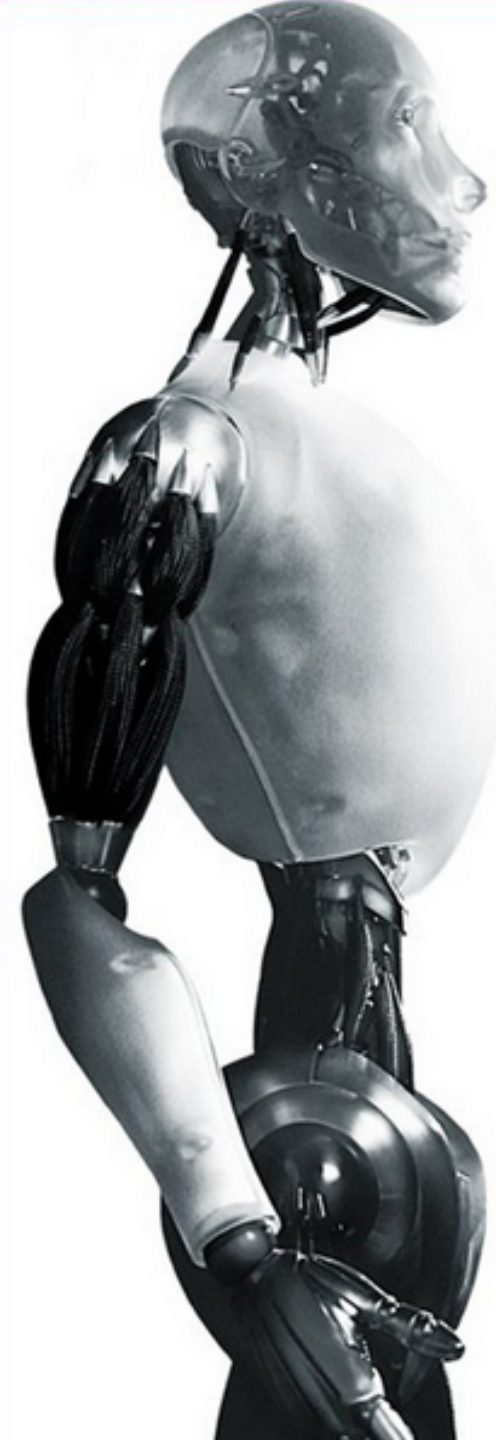
- How would you solve it using search trees?
  - How to formally define the problem state?
  - How to encode primitive actions?
  - What is the “generate successor state” function?



The 8-Queens Puzzle

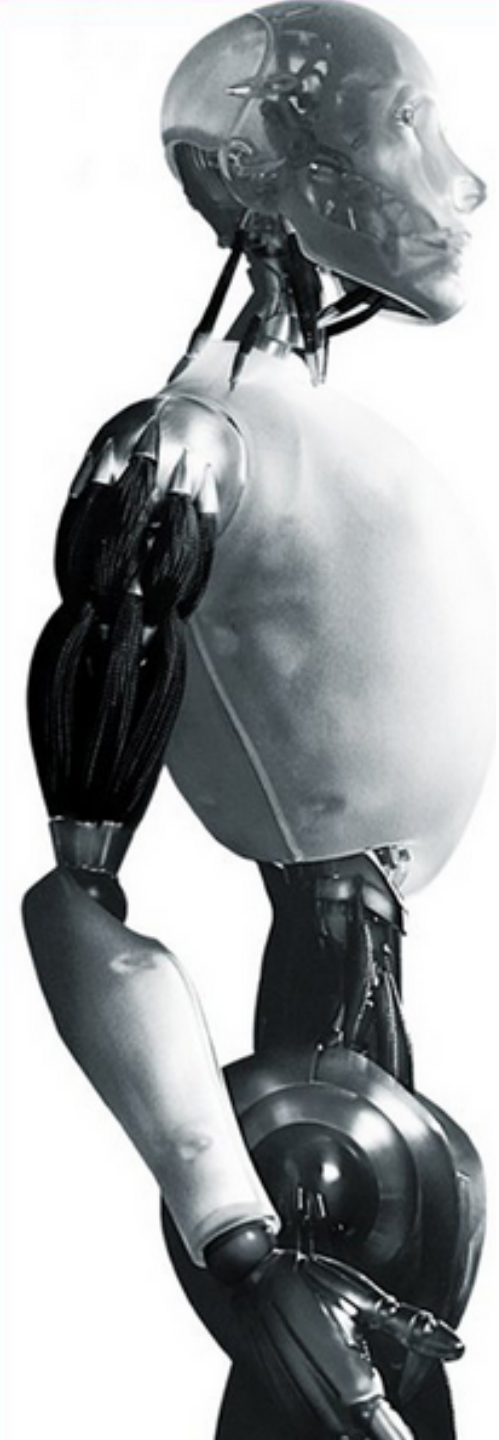






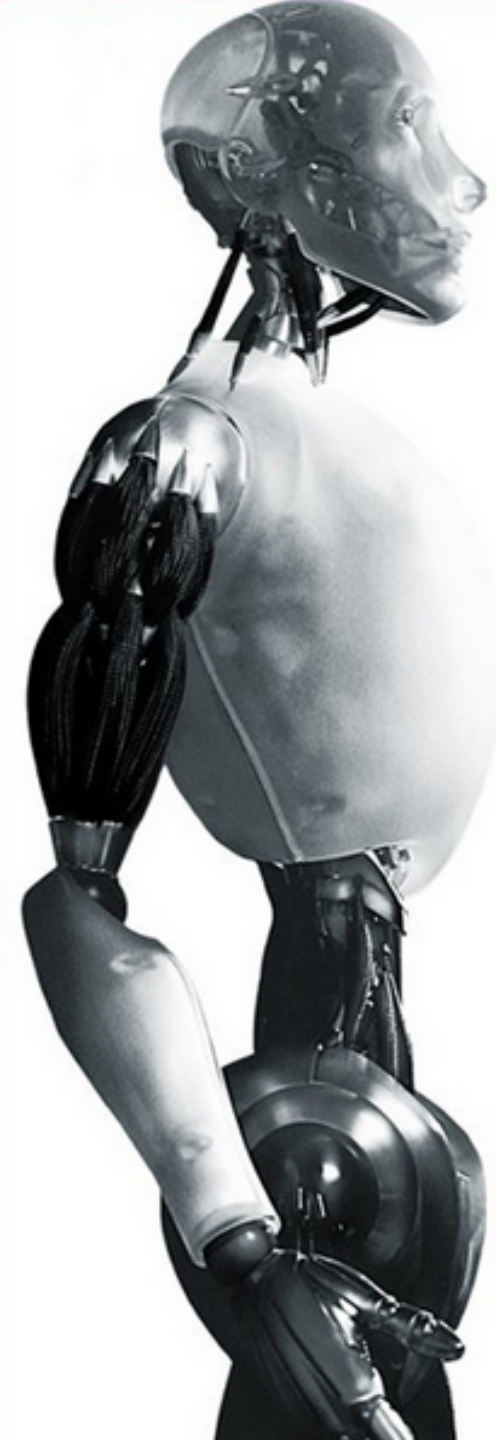
Missionaries and Cannibals problem!!

- Three missionaries and three cannibals are on one side of the river, along with a boat that can hold one or two people. Find a way to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. This problem is famous in AI because it was the subject of the first paper that approached problem formulation from an analytical viewpoint (Amarel 1968)
  - Formulate the problem precisely, making only those distinctions necessary to ensure a valid solution. Draw a diagram of the complete state space.
  - Solve this problem by using depth first search and breadth first search.



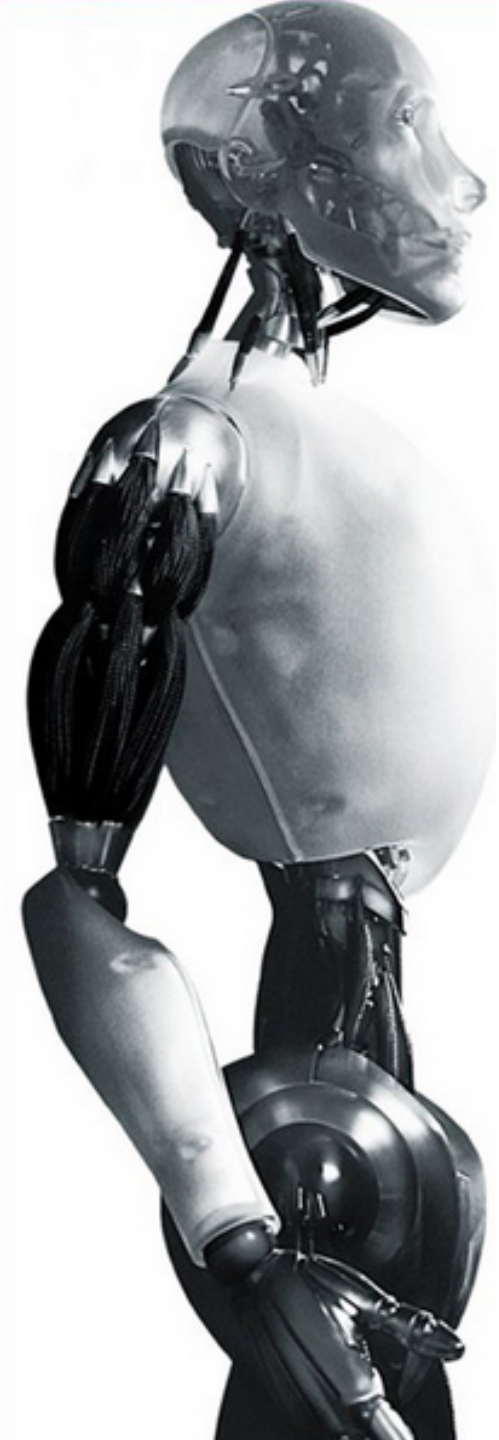
Heuristic search algorithms

- Depth first and breadth first search turn out to be too inefficient for really complex problems.
- Instead we turn to “heuristic search” methods, which don’t search the whole search space, but focus on promising areas.
- Simplest is best first search. We define some “heuristic evaluation function” to say roughly how close a node is to our target.
  - Eg Jug problem: How close to 2 gallons there are in 4 gallon jug.



Dijkstra's Algorithm

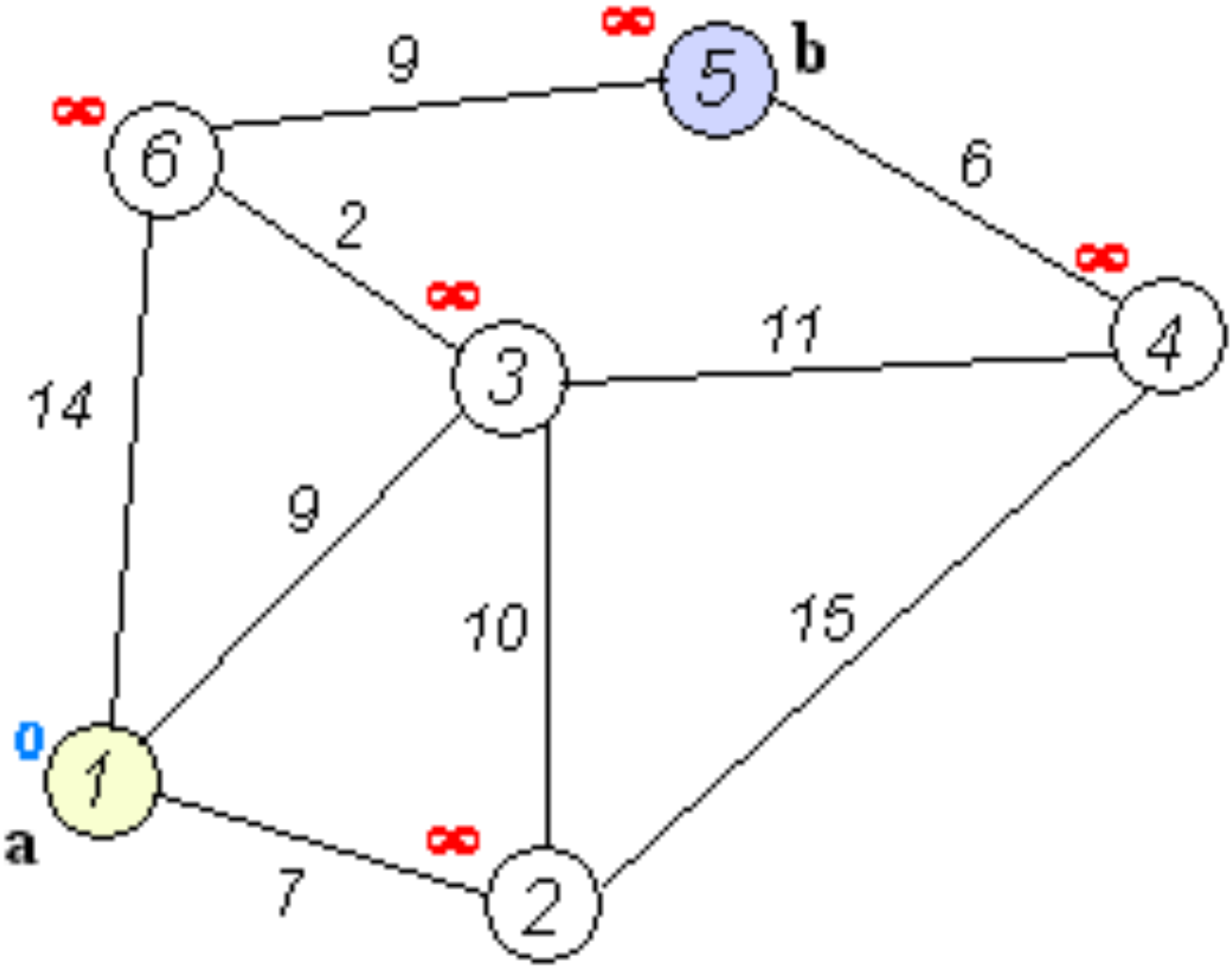
- Find the shortest distance between two points in a graph
- Imagine we have a weighted graph
- Start from A in order to reach Z
- At any point calculate L(Z) which is the least distance to reach Z from the current point
- Set L(Z) of every note to infinity as a starting point

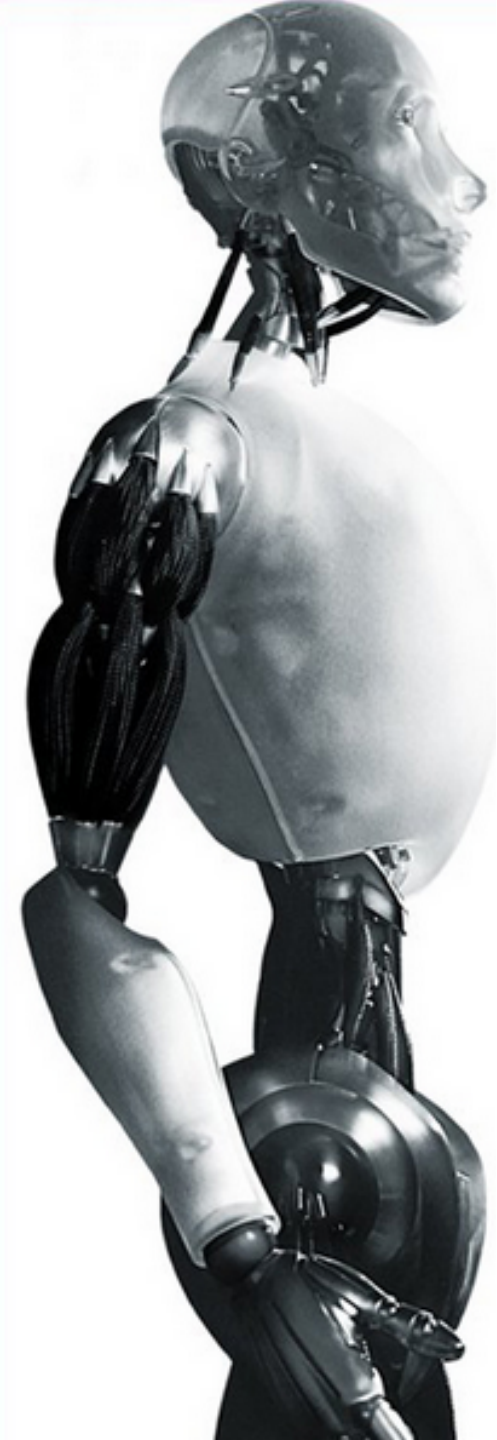


Dijkstra's Algorithm

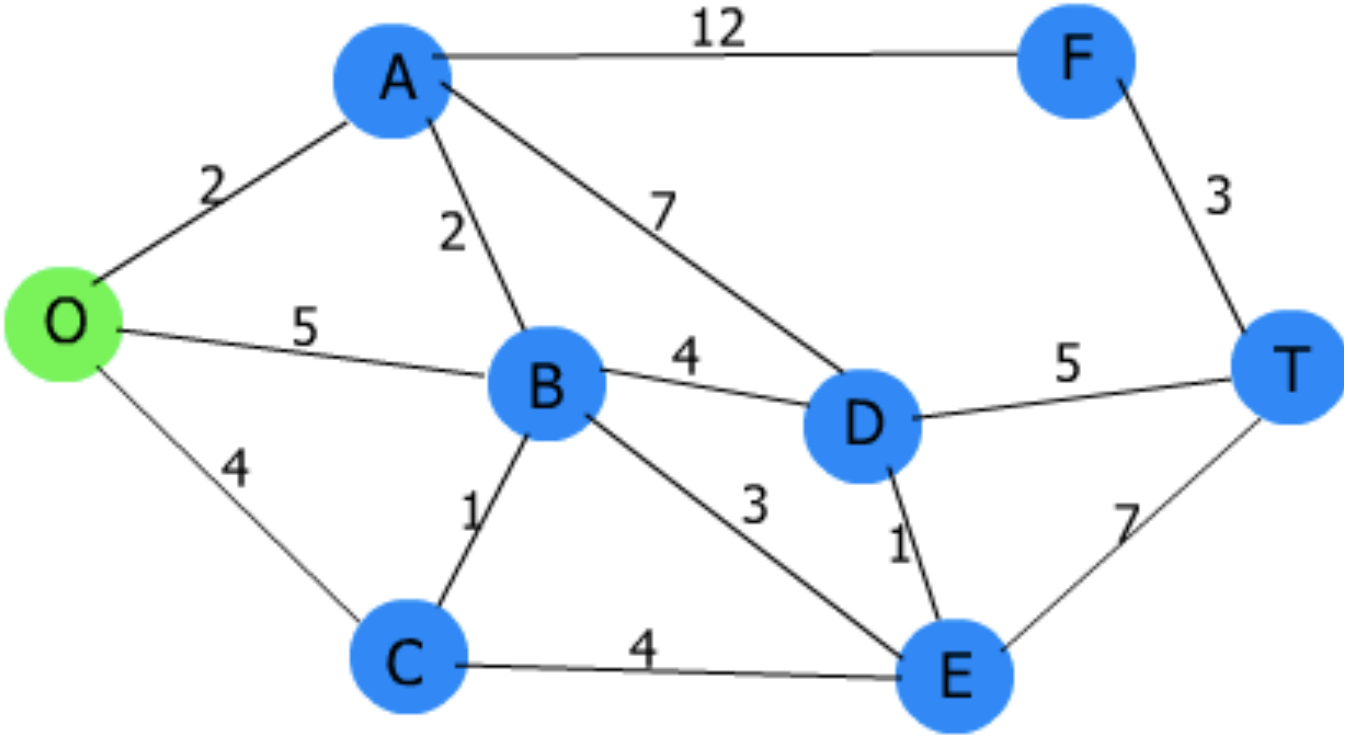
- L(A) = 0
- L(Z) = ∞

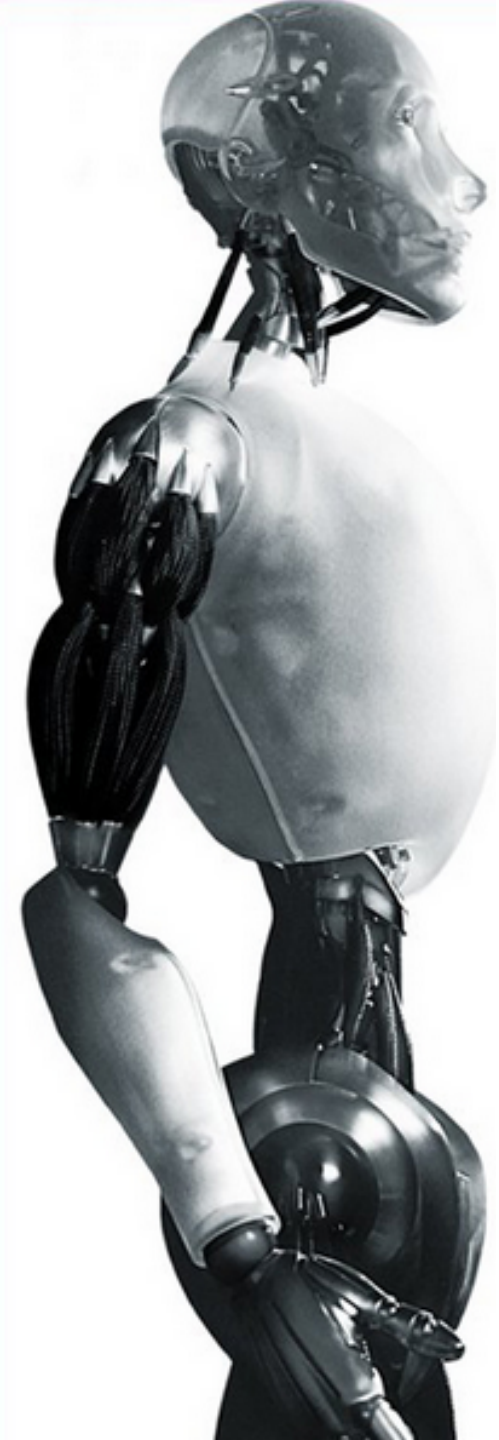
- While (Z is in the Graph)
  - Choose vertex with least distance
  - Remove it from Graph
  - Calculate distance between it and neighbors in Graph
  - Update distances


∞
9
5
b
∞
6
6
2
∞
∞
11
4
14
3
9
10
15
0
1
a
∞
7
2



Find the shortest path from O to T

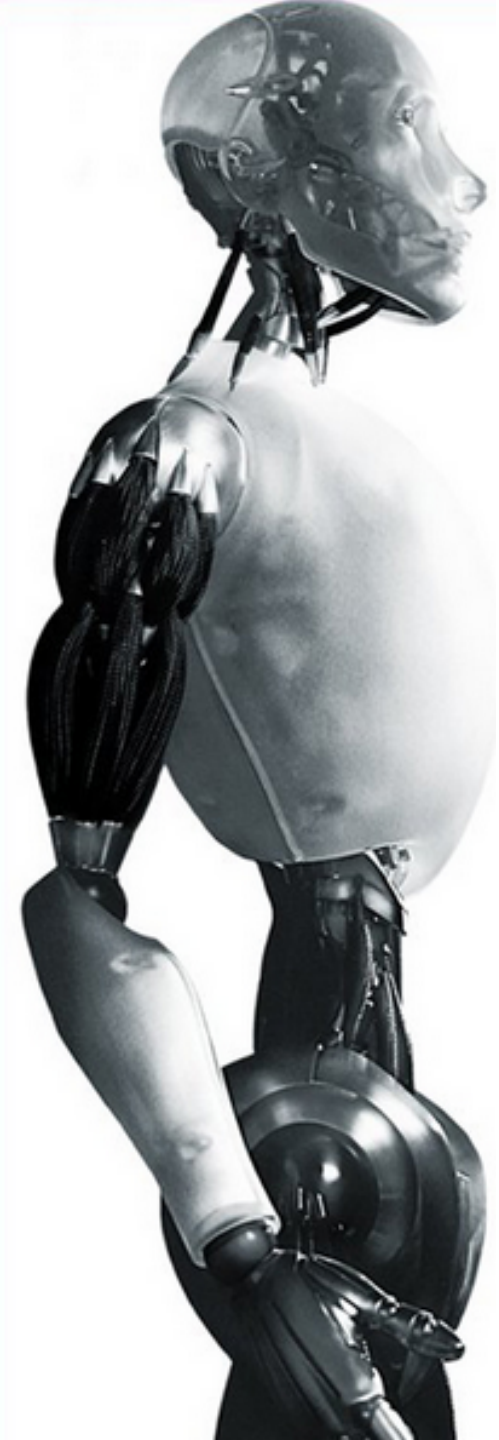




Best first search algorithm

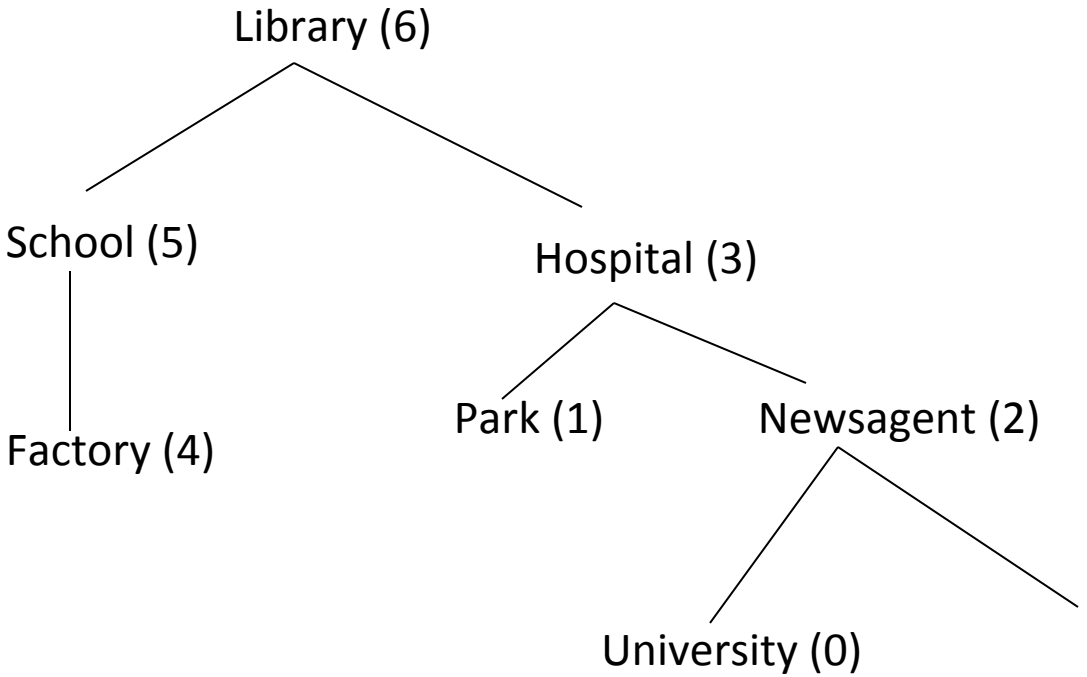
- Best first search algorithm almost same as depth/breadth. But we use a priority queue, where nodes with best scores are taken off the queue first.

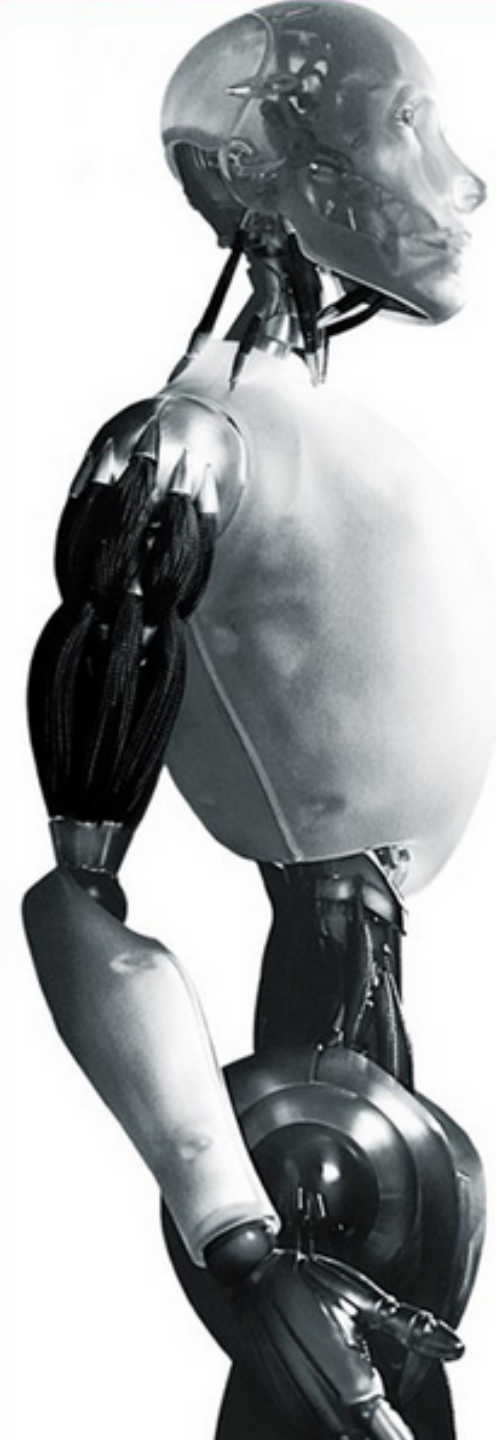
- While queue not empty and not found do:
  - Remove the BEST node N from queue.
  - If N is a goal state, then found = TRUE.
  - Find all the successor nodes of N, assign them a score, and put them on the queue..



Best first search

- Order nodes searched: Library, hospital, park, newsagent, university.

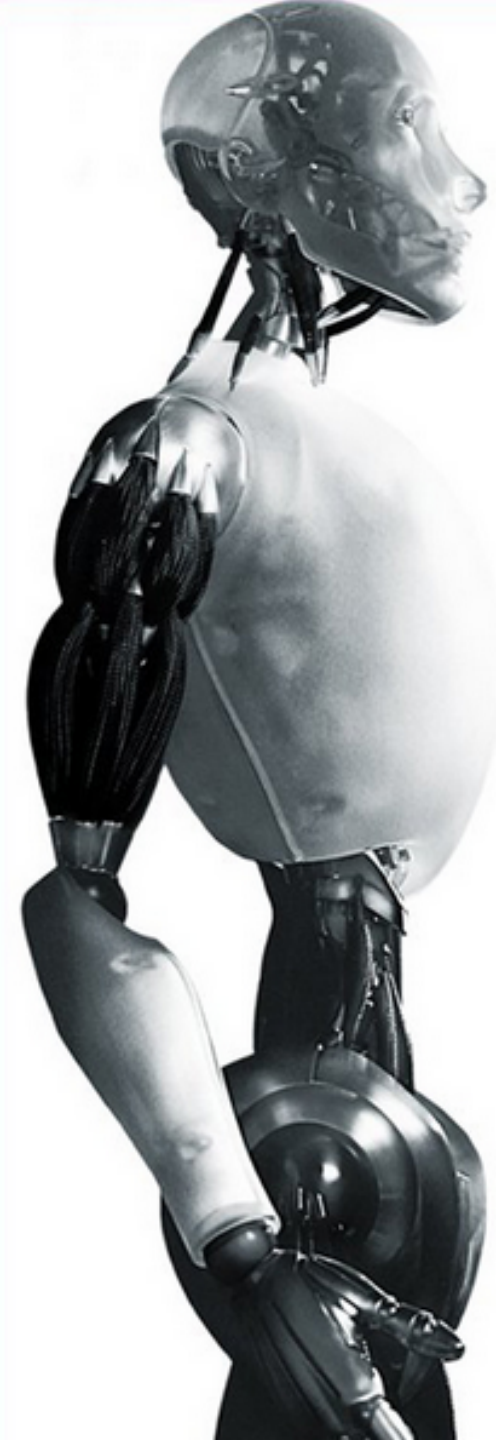




Other heuristic search methods

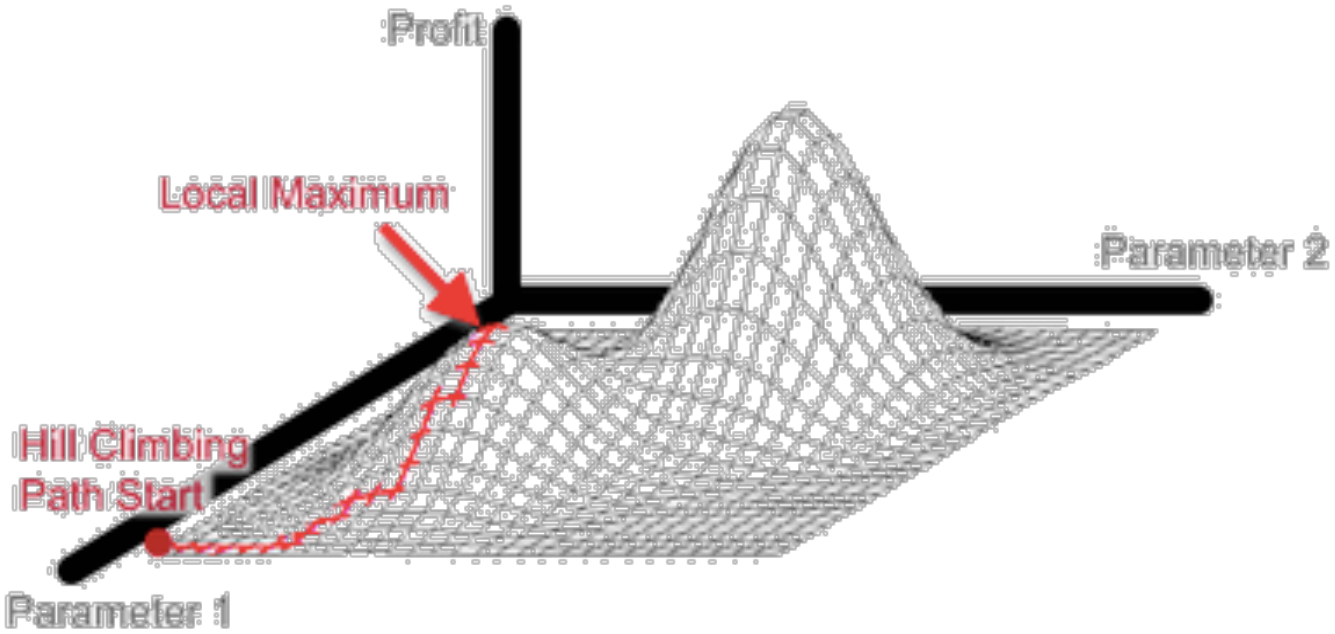
- Hill climbing: always choose successor node with highest score.

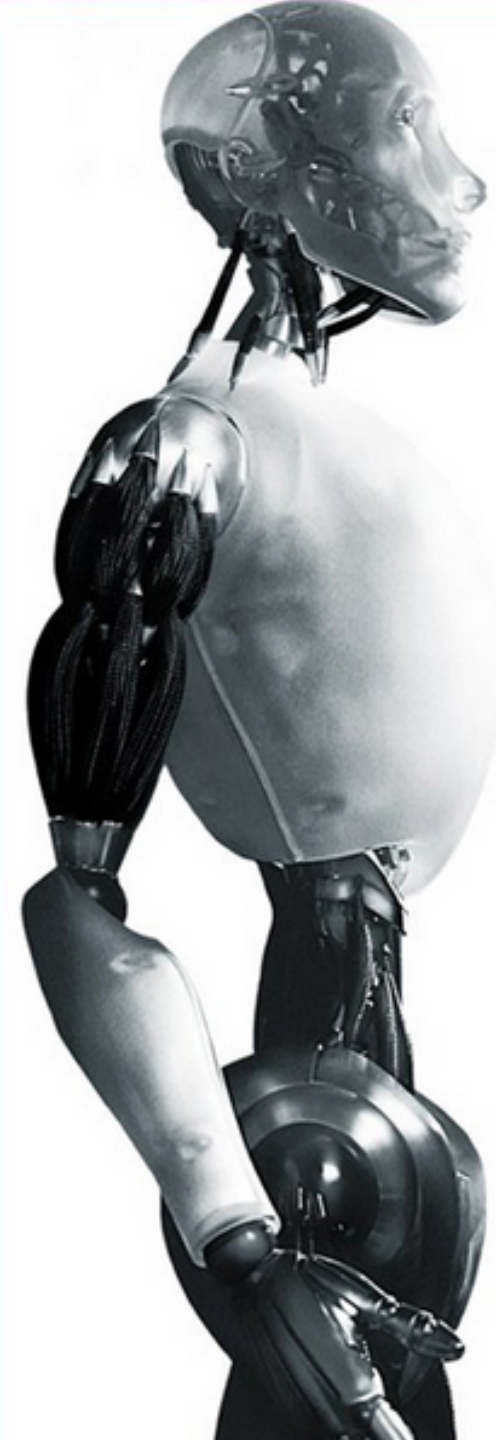
- A*: Score based on predicted total path "cost", so sum of
  - actual cost/distance from initial to current node,
  - predicted cost/distance to target node.



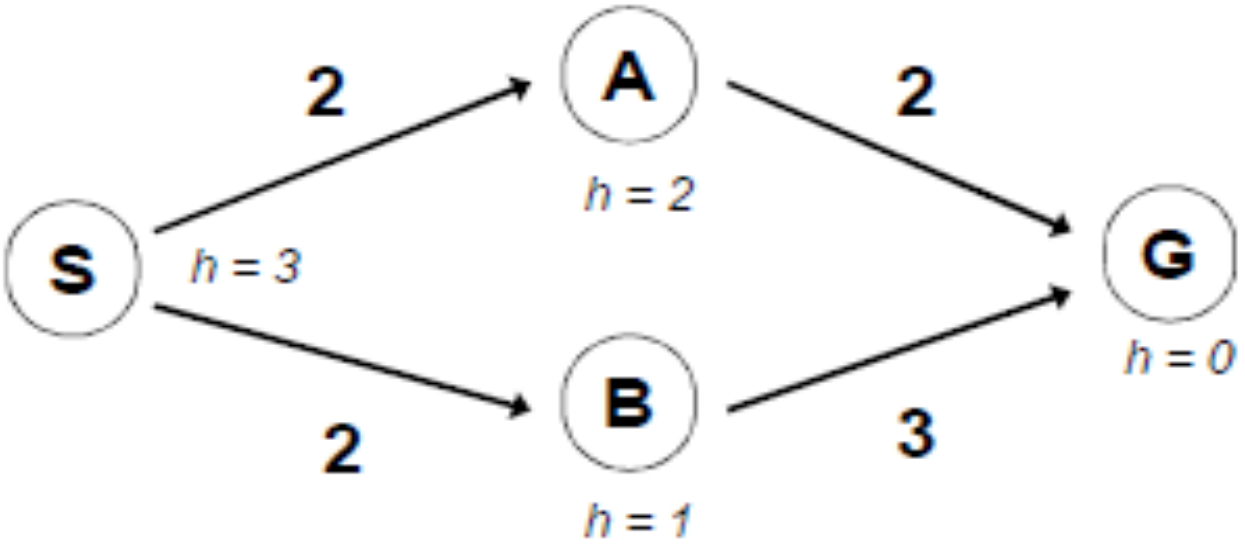
Hill Climbing

The problem with hill climbing is that it gets stuck on "local-maxima"





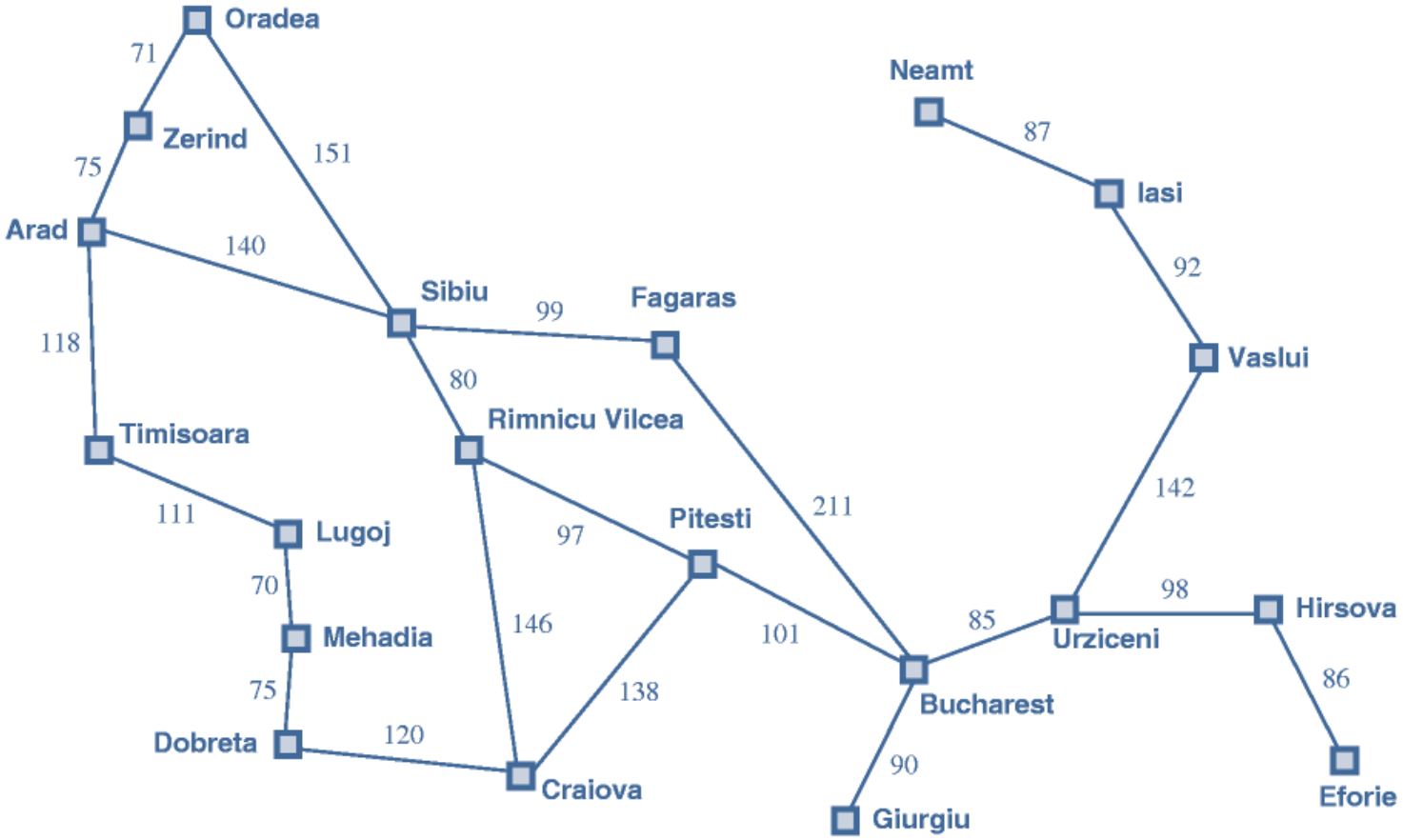
A*



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

Exercise

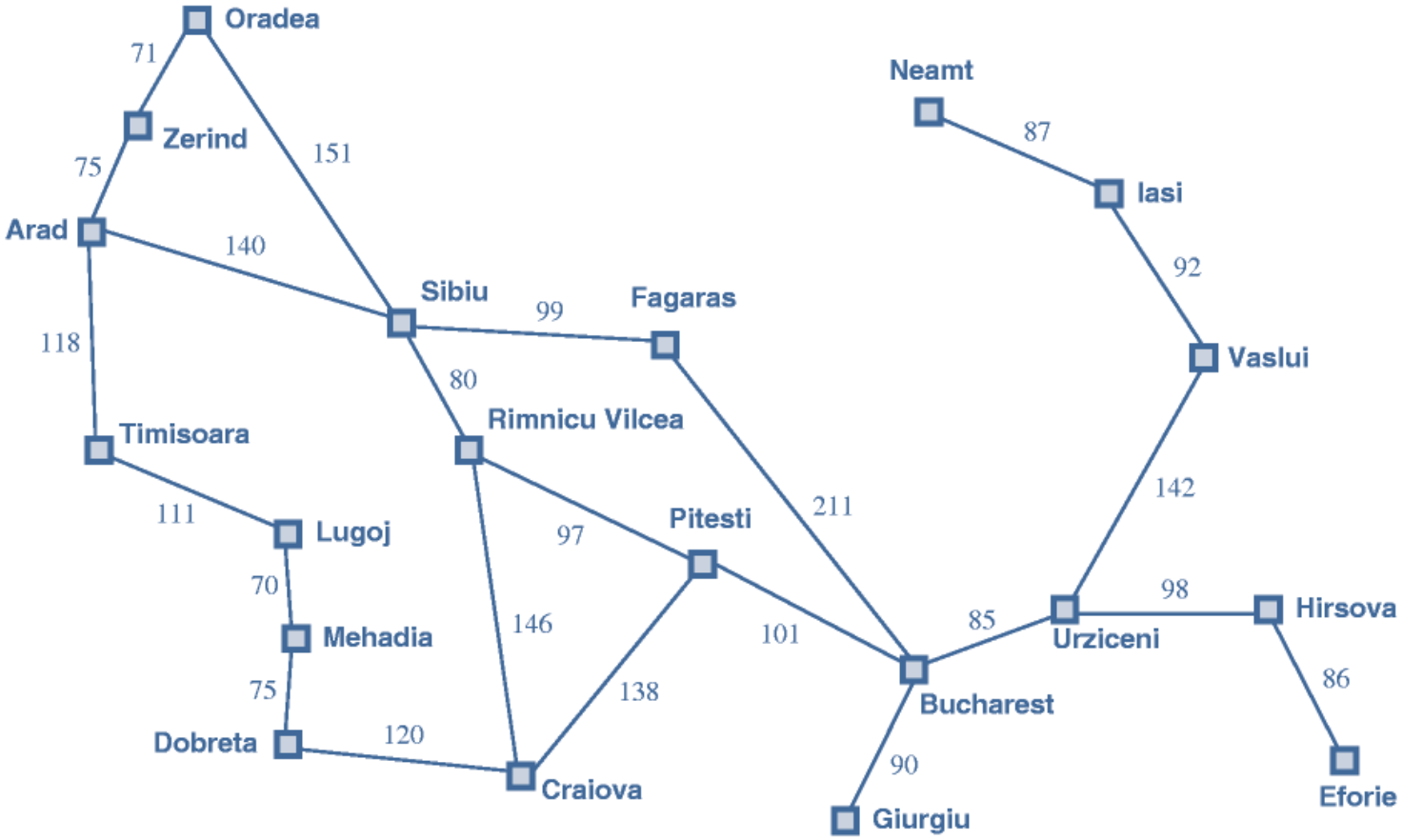
Use BFS to find path from Arad to Bucharest



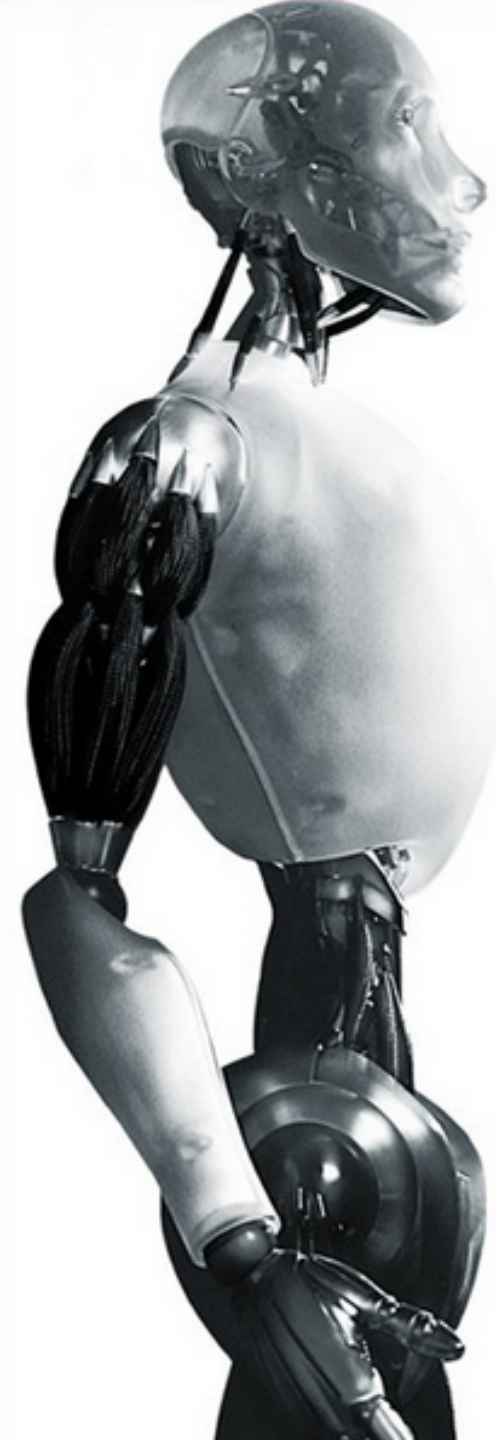
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Exercise

Use A* to find path from Arad to Bucharest



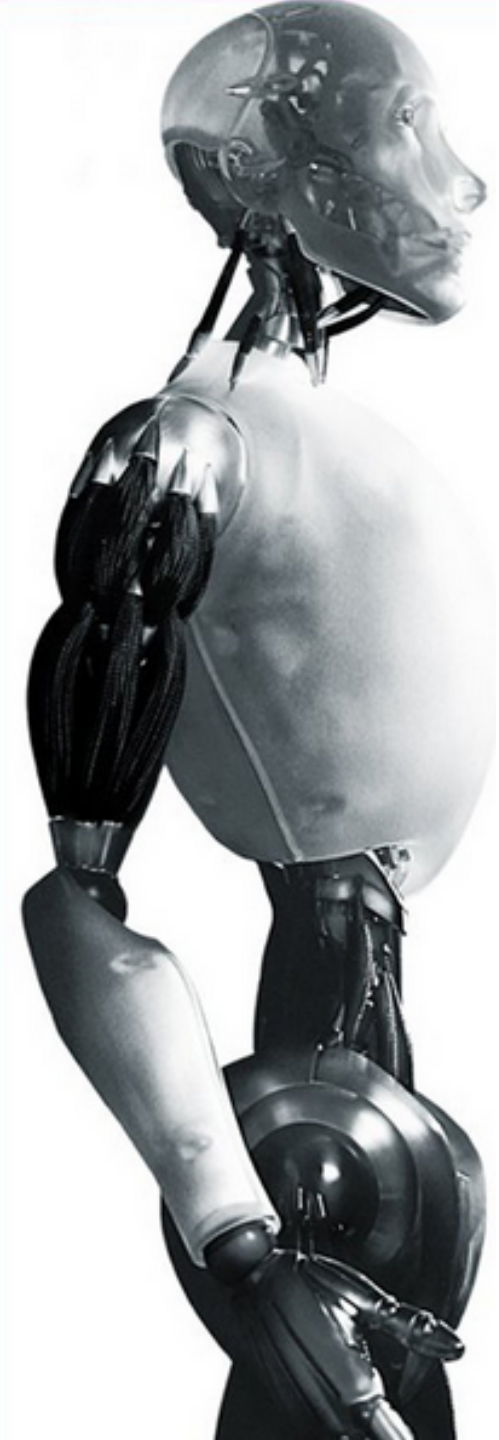
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



The 8-Puzzle problem

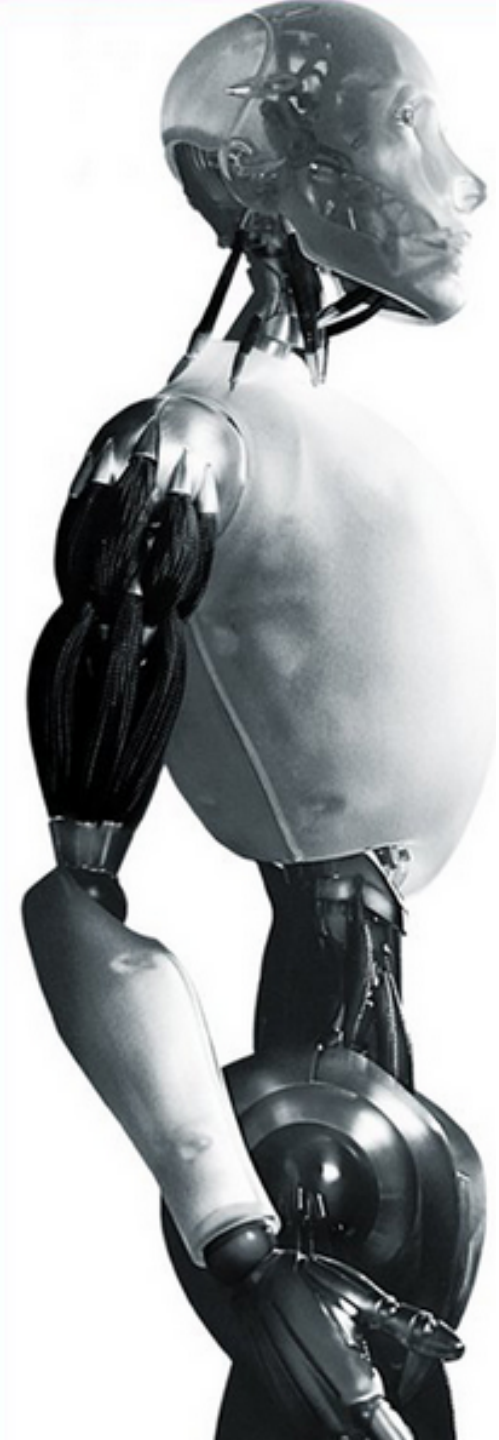
2	8	3
1	6	4
7		5

1	2	3
8		4
7	6	5



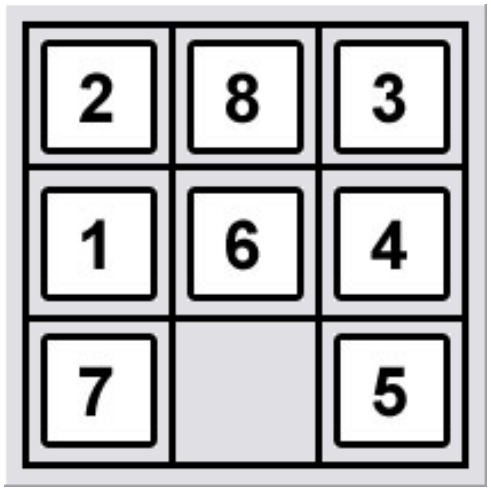
The 8-Puzzle problem

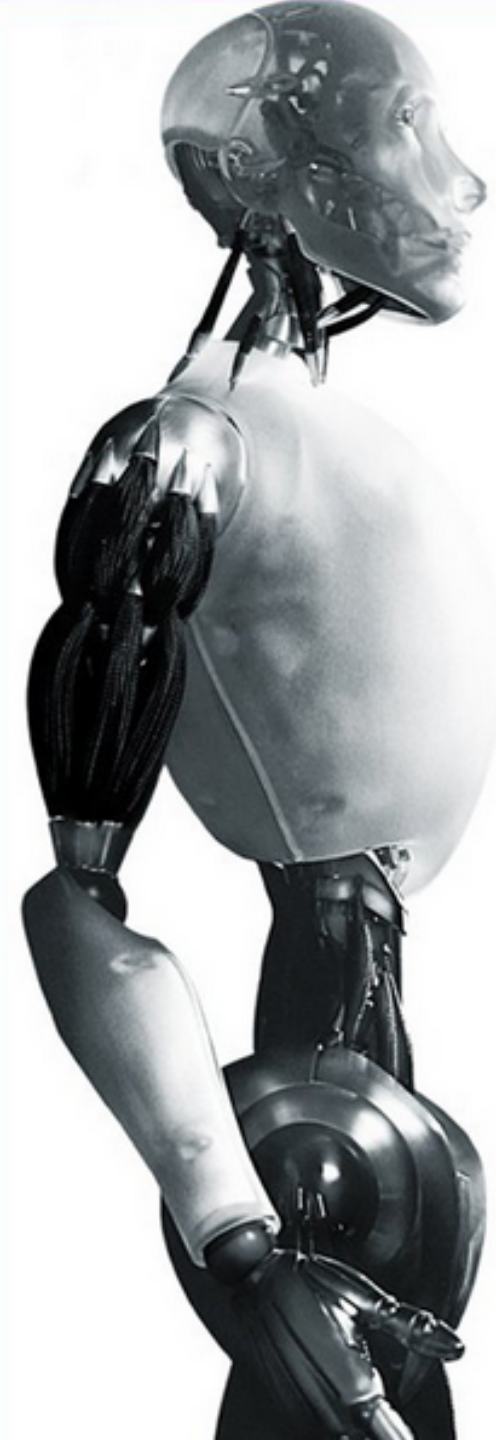
- How would you solve it using search trees?
  - How to formally define the problem state?
  - How to encode primitive actions?
  - What is the “generate successor state” function?



The 8-Puzzle problem

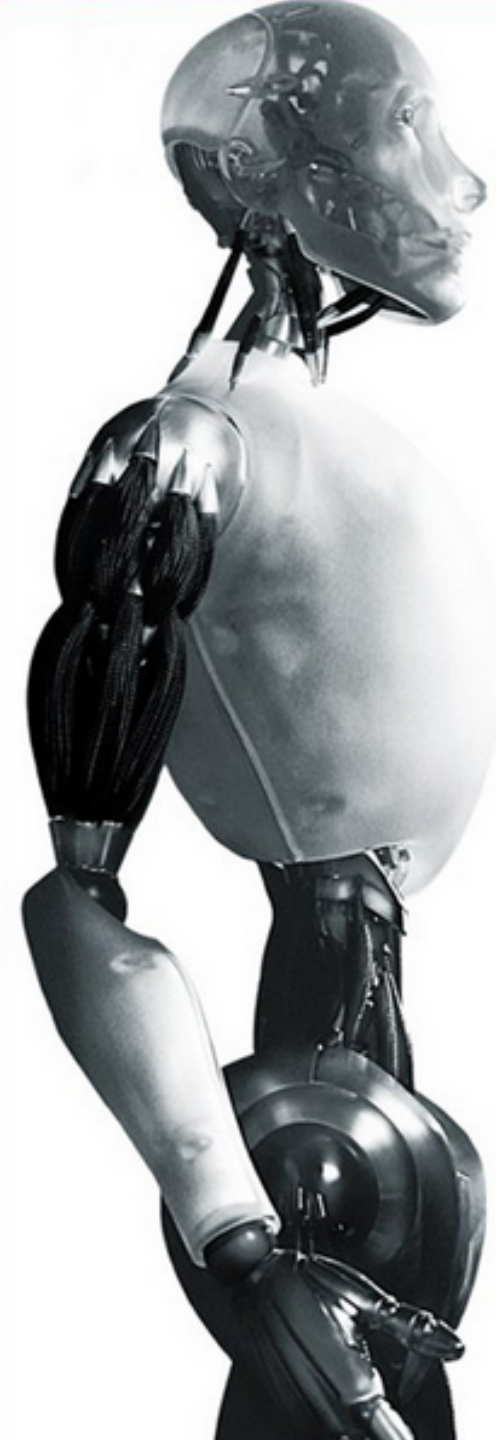
- Solve it using A*
  - Where the heuristic is the number of misplaced tiles E.g. h = 4





The 8-Puzzle problem

- Solve it using A*
  - Where the heuristic is the Manhattan distance i.e. the number of squares from the desired location of each tile



E.g.

Tile 1 = 1 (1 move up to reach its place)

Tile 2 = 1

Tile 3 = 0 (in its place)

Tile 4 = 0

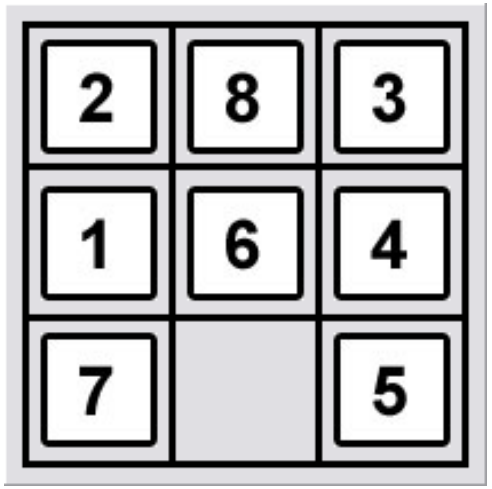
Tile 5 = 0

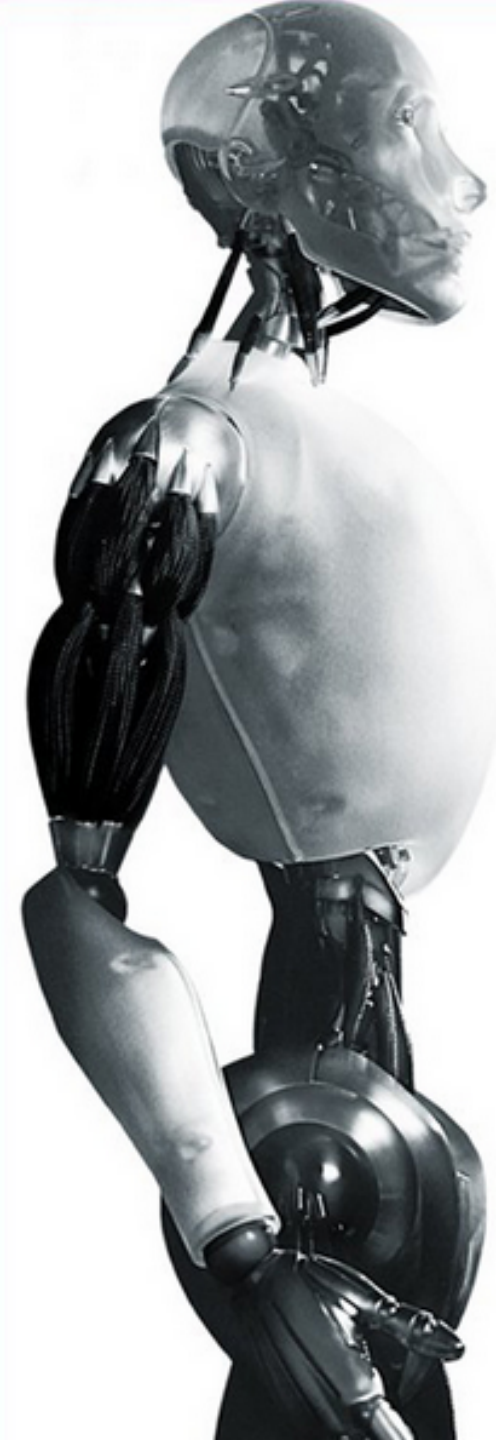
Tile 6 = 1

Tile 7 = 0

Tile 8 = 3

Total h = 6





Questions?