Artificial Intelligence

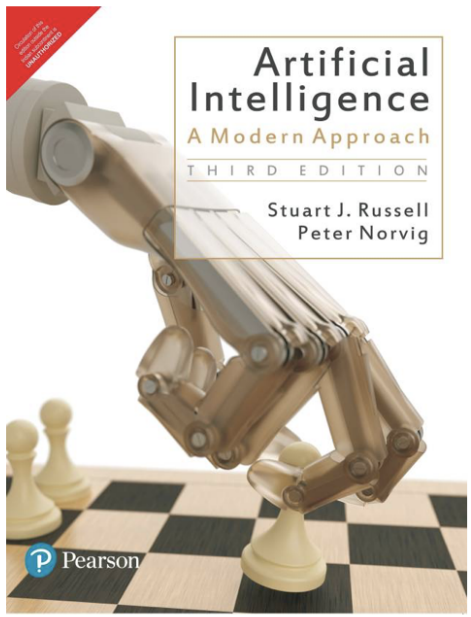
Augustin Cosse.

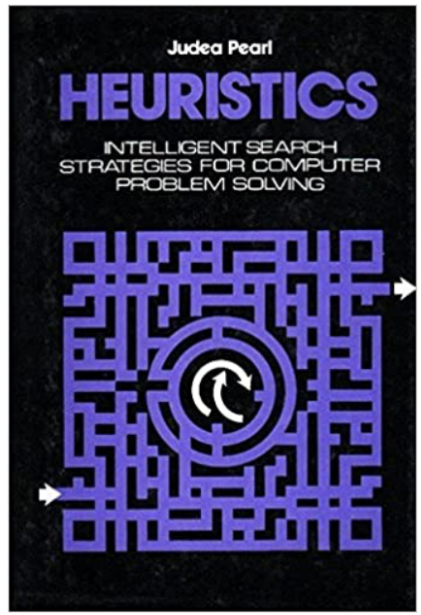
Fall 2020

September 22, 2021

References for this week

- Additional references for this week :
  - Pearl, *Heuristics, Intelligent Search Strategies for Computer Probem Solving*, Chap.
  - Russell and Norvig, *Artificial Intelligence, A modern Approach*, Chap. 1-3.





Knowledge Based agents (Recap I)

- Agent = entity that perceives and acts in a given environment.
- Rational Agent = agent that always chooses the action that is expected to maximize its performance measure
- An agent is autonomous when its actions only depend on its own experience (i.e the experience that it has accumulated through time) and not on any knowledge that was built in by the programmer.

Knowledge Based agents (Recap II)

- Different types of agents are usually classified based on how they take their decisions as well as on the information they use in the decision process. The design of an agent depends on 4 main aspects:
  - Percepts
  - Actions
  - Goals
  - Environment
- Reflex agents respond immediately to percepts. Goal-based agents act in order to maximize their goals, and utility based agents try to maximize their level of happiness.

Problem solving agents

- Recall that simple reflex agents are unable to plan ahead
- Such agents are limited in what they can achieve because their actions are determined only by the current percept
- Furthermore, they have no knowledge of what their actions do or what they are trying to achieve.

Problem solving agents

- Now that we have discussed the main families of agents, we will consider a special kind of goal-based agent called problem solving agent
- Problem solving agents rely on atomic representations of the world (each state of the world is indivisible, it has no internal structure). Goal based agents that use more advanced factored or structured representations are usually called planning agents
- A goal together with a set of means to achieve this goal will be called a problem.

Problem solving agents

- Goals help organize behavior by limiting the objectives that an agent is trying to achieve and hence the actions it needs to consider
- Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.
- We will consider a goal to be a set of world states and the agent task will be to find out how to act, now and in the future, so that it reaches the goal state.
- In order to make sure it can meet the goal, the agent needs to decide what sorts of actions and states it should consider.

Problem solving agents

- Problem formulation is the process of deciding what actions and states to consider, given a goal
- The agent will not always know which of the possible actions is best, because it does not yet know enough about the state that results from taking each action. If the agent has no information (i.e. if the environment is unknown), then the agent has no other option than to try one of the actions at random.
- But if we suppose that the agent has a map, the agent can use this information to determine the subsequent stages it has to consider to reach the goal state.

Problem solving agents

- A problem can be defined formally by five components:
  - The initial state that the agent starts in
  - A description of the possible actions available to the agent. Given a particular state s, $\textsc{action}(s)$ should return the set of possible actions that can be executed in s (we will say that each of those actions are applicable in s)
  - A description of what each action does. The formal name for this is a transition model specified by a function $\textsc{result}(s, a)$ that returns the state that results from doing action a in state s. We also use the term successor to refer to any state reachable from a given state by a single action. Together, the initial state, actions and transition model implicitely define the state space (set of all states reachable from the initial state by any sequence of actions).

Problem solving agents

- A problem can be defined formally by five components (continued):
  - The goal test which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states (this is the case with chess for example where the goal is to reach a state called "checkmate")
  - Finally, we sometimes consider a path cost function which assigns a numeric cost to each path. The problem solving agent chooses a cost function that reflects its own performance measure. The step cost of taking action a in state s to reach state s' is denoted by $c(s, a, s')$.

Problem solving agents

- All the preceding elements define a problem and can be gathered into a single data structure that is given as input to a problem solving algorithm.
- Note that the state space forms a directed network or graph in which the nodes are used to represent the states and the links between the nodes are used to represent the actions
- A path in the state space is a sequence of states connected by a sequence of actions
- A solution to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.

Problem solving agents

- Assume that the environment is observable (the agent always knows the current state), the environment is known and discrete (the agent knows which state (among a finite set) is reached by which action) as well as deterministic (each action has exactly one outcome).
- Under these assumptions, the solution to any problem is a fixed sequence of actions (if the agent knows the initial state and the environment is known and deterministic, it knows exactly where it will be after the first action and what it will perceive)
- The process of looking for a sequence of actions that reaches the goal is called a search

Problem solving agents

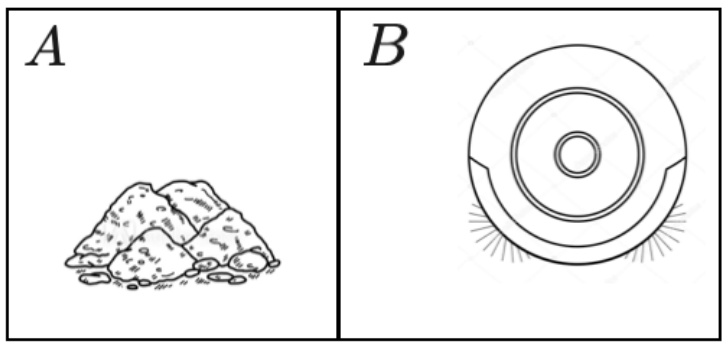
- A search algorithm takes a problem as input and returns a solution in the form of an actions sequence
- Once a solution has been found, the actions it recommends can be carried out. This is called the execution phase.
- This leads to simple formulate-search-execute design for the agent.

Problem solving agents

```
Agent Simple Problem Solving Agent(percept) returns action ;
persistent: seq (action sequence, init. empty) ;
            state (description of current world state);
            goal (init. empty);
            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
    end
end
action ← FIRST(seq);
seq ← REST(seq);
return action
```

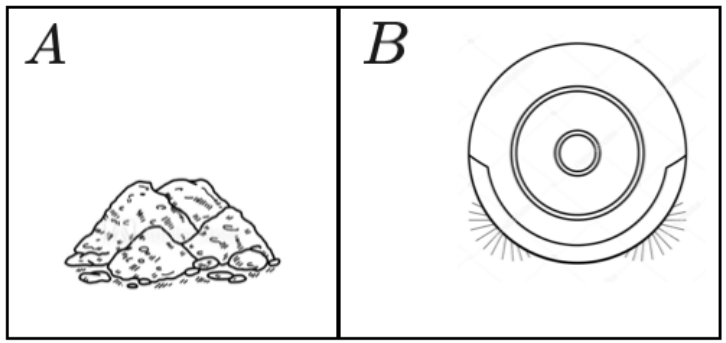
Problem Examples

- The first example we consider is the vacuum world below. The constitutive elements of this problem can be defined as follows:
  - **States** The state is determined by both the agent location and the dirt location. The agent can be in one of two locations, each of which might or might not contain dirt. There are thus a total of $2 \times 2^2 = 8$ possible world states
  - **Initial state**: Any state can be taken to be the initial state

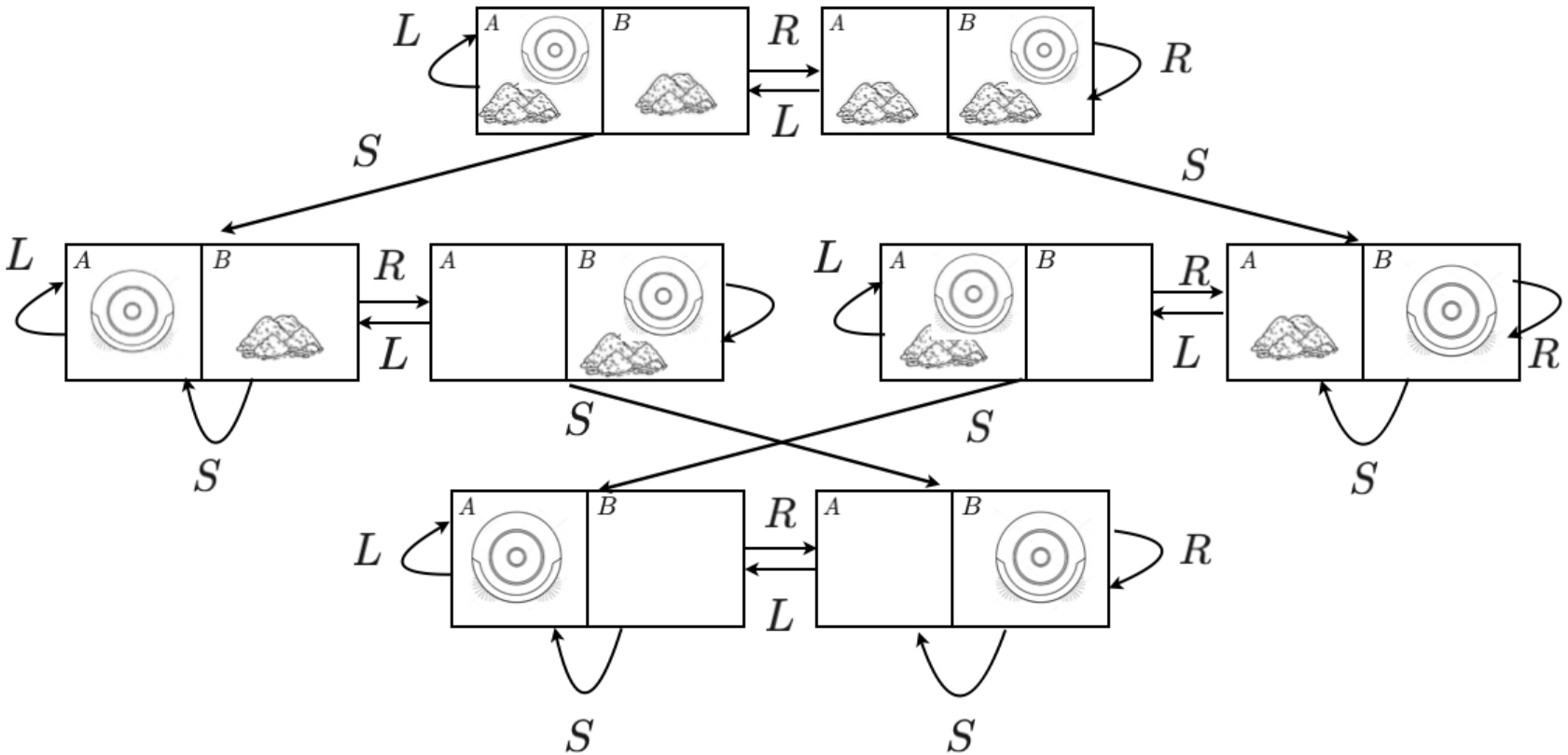


Problem Examples

- Vacuum world (continued)
  - **Actions**: In this simple environment, each state just has 3 actions: Left, Right and Suck
  - **Transition Model** The actions have their expected effects, except moving *Left* in the leftmost square, moving *Right* in the rightmost square and *Suck* have no effect
  - **Goal test**. The check is for all the cells to be clean
  - **Path cost**: Each step costs 1 hence the past cost is the total number of steps in the path.

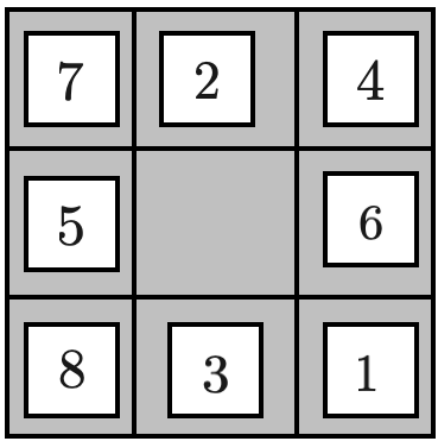


State Space (Vacuum cleaner)

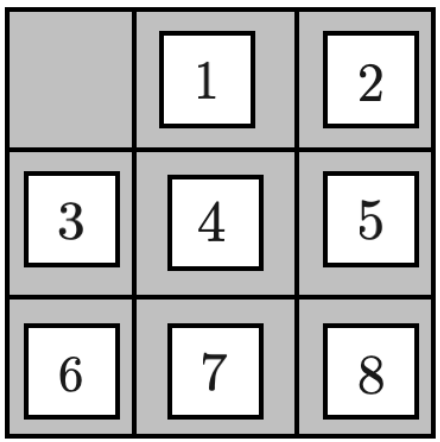


Problem Examples

- Another example of a simple problem is the so-called 8 puzzle
  - **States** A state description specifies the location of each of the 8 tiles and the blank in one of them
  - **Initial state**: Any state can be taken to be the initial state
  - **Actions**: The simplest formulation defines the actions as movements of the blank space (*Left*, *Right*, *Up* or *Down*) (Different subsets of these are possible depending on where the blank cell is)



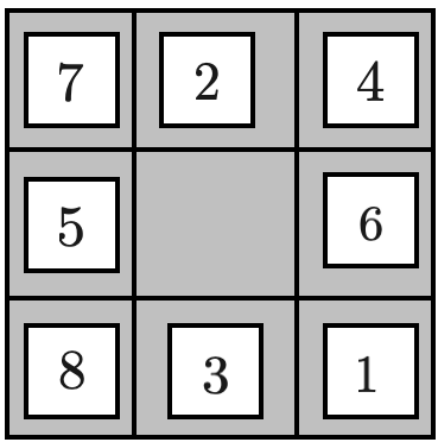
Initial State



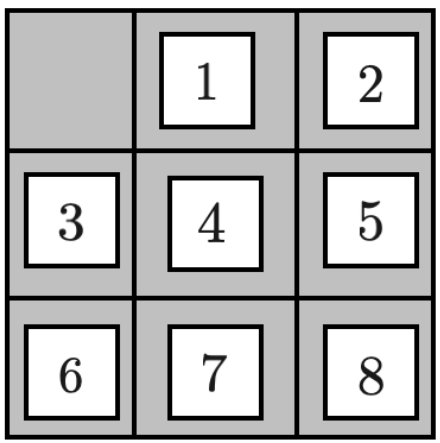
Goal State

Problem Examples

- 8 puzzle (continued)
  - **Transition Model**: Given a state and action, this should return the resulting state. For example, if we apply *Left* to the starting state, the resulting state has the 5 and the blank space swapped
  - **Goal test**: This check whether the current state matches the goal configuration
  - **Path cost**: Each step costs 1 so the path cost is the number of steps in the path



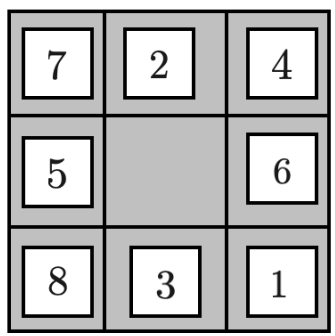
Initial State



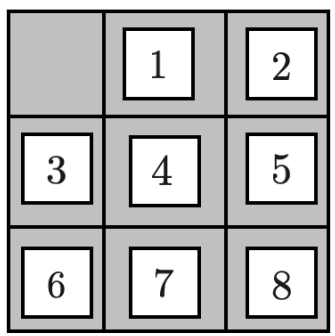
Goal State

Problem Examples

- The 8-puzzle belongs to the family of Sliding-block puzzles which are often used as test problems for new search algorithms in AI
- The family is known to be NP-complete so one does not expect to find methods significantly better in the worst case than the search algorithms that we will cover today.
- The 8-puzzle has $9!/2 = 181\,440$ reachable states while the 15-puzzle has around 1.3 trillion states (note however that random instances can be solved by the best search algorithms in a few milliseconds)



Initial State



Goal State

Problem Examples

- The route finding problem is defined in terms of locations and transitions along the links between them.
- Route finding algorithms are used in a variety of applications including websites, in-car systems, routing video streams, military operations planning and airline travel planning,...
- In route finding the goal is to reach the final destination
- Robot navigation is an example of a route-finding problem. However, rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states.

Problem Examples

- Touring problems are related to route finding problems but with the exception that the state space now not only includes the current location but also the set of cities that the agent has already visited.
- The travelling salesman problem (TSP) is an example of a touring problem in which every city must be visited exactly once. The aim is to find the shortest tour.
- The TSP problem is NP-hard but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms.

Search algorithms

- Given a precise definition of a problem it is relatively straighforward to construct a search process for finding solutions.

- We make the distinction between Blind or Uninformed Search Strategies and Informed Search strategies:

  - In **Uninformed Search**, there is no additional information about states beyond that provided in the problem definition. All the search strategy can do is generate successors and discriminate between a goal state from a non goal state. The uninformed search strategies are distinguished by the order in which nodes are expanded.

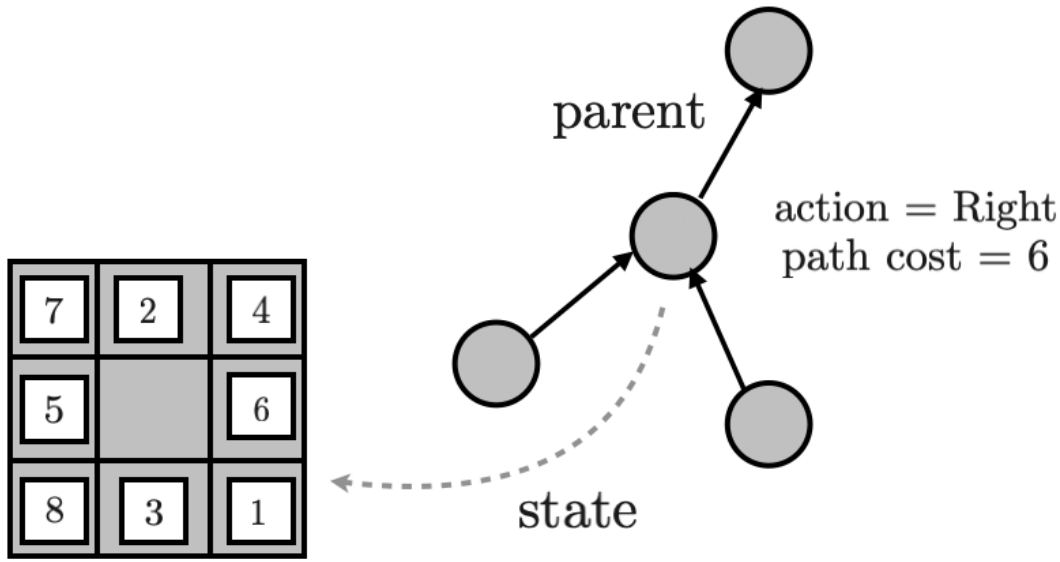
  - Strategies that know whether one non-goal state is "more promising" than another are called **informed search** or **heuristic search** strategies.

Infrastructure for search algorithms

- Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we should have a structure that contains the four components:
  - node.STATE: the state (in the state space) to which the node corresponds
  - node.PARENT: the node in the search tree that generated this node
  - node.ACTION: the action that was applied to the parent to generate the node
  - node.PATH-COST: the cost, traditionally denoted as $g(n)$ of the path from the initial state to the node, as indicated by the parent pointers.

Infrastructure for search algorithms

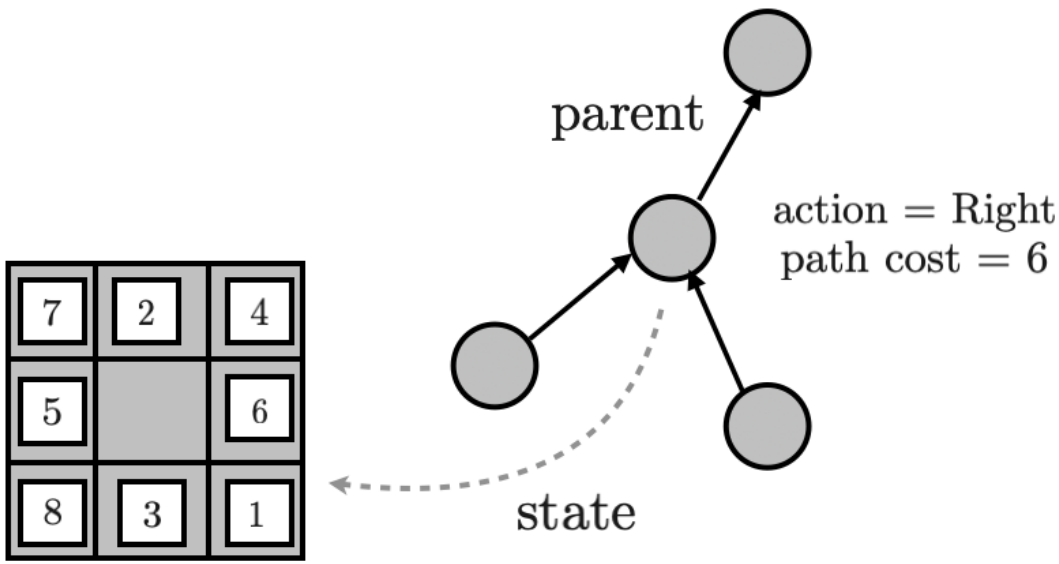
- The set of nodes available for expansion at any point is called the frontier (some authors call it the open list)
- An example of a node data structure is given below



Solving Problems by searching

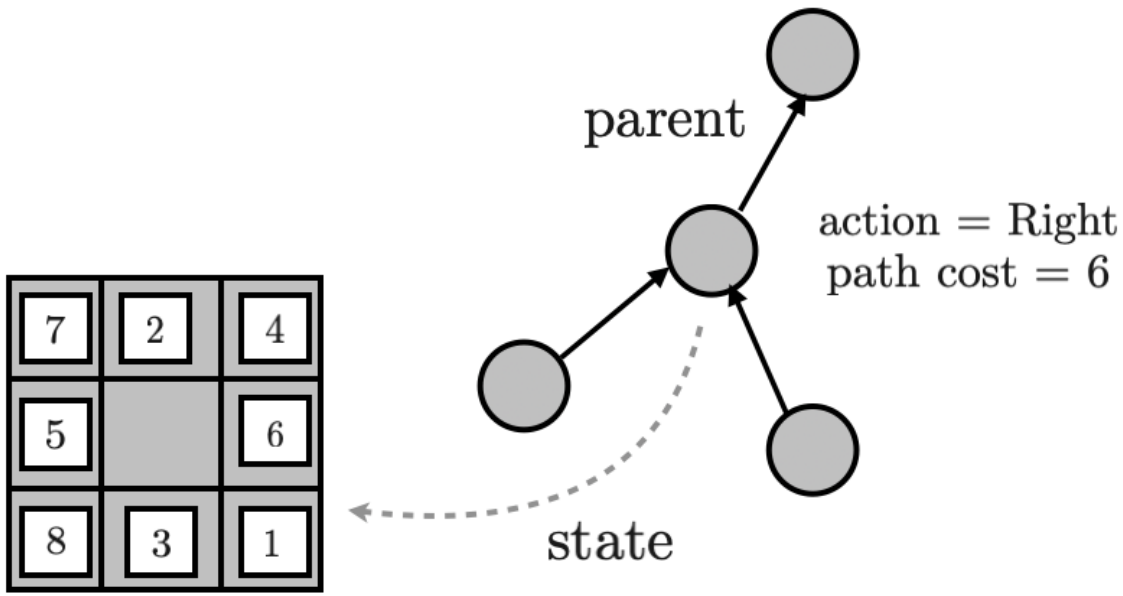
- Now that we have nodes, we need somewhere to put them. The frontier needs to be chosen in such a way that the search algorithm can easily select the next node to expand according to its preferred strategy. The appropriate structure for this is a queue.

- Nodes that have been generated but haven't yet been explored are sometimes called open. On the contrary, the nodes that have already been expanded are called closed.



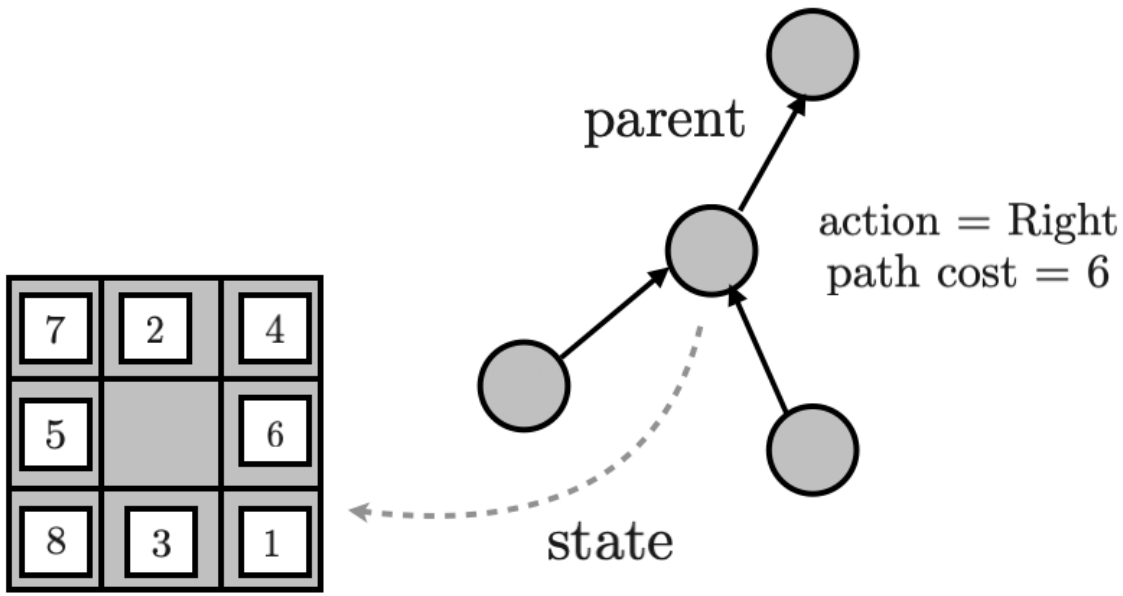
Infrastructure for search algorithms

- The operations on a queue are as follows:
  - EMPTY?(*queue*) returns true if there are no more elements in the queue
  - POP(*queue*) removes the first element of the queue and returns it
  - INSERT(*element*, *queue*) inserts an element in the queue and returns the resulting queue.



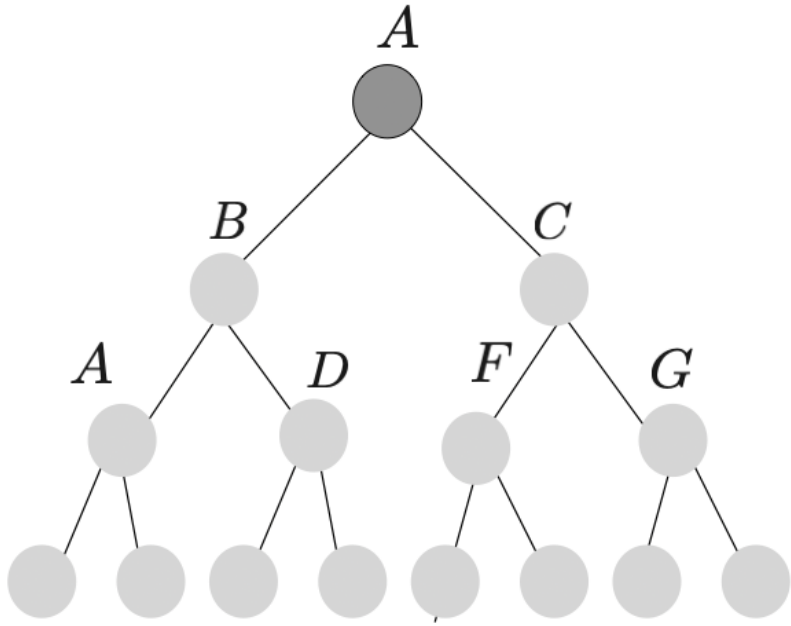
Infrastructure for search algorithms

- Queues are characterized by the order in which they store the inserted nodes. Three common variants are
  - FIFO queue which pops the oldest element in the queue
  - LIFO queue which pops the newest element of the queue
  - Priority queue which pops the element of the queue with the highest priority



Infrastructure for search algorithms

- Note that a tree can sometimes contain repeated nodes such as shown below. Such nodes are generated by a loopy path. When looopy paths are present in a tree, they also imply that the complete search tree will be infinite. because there is no limit on how often one can traverse a loop.



Infrastructure for search algorithms

- It is possible to remove those loopy paths by relying on some additional intuition. We note that path costs are additive and we assume that step costs are always non negative. A loopy path to any given state is therefore never better than the path with the loop removed.

- Loopy paths are a special case of the general concept of redundant paths which exists whenever there is more than one way to get from one state to another

- As indicated above, in some cases, it is possible to define the problem so as to eliminate redundant paths. In other cases however, redundant paths are unavoidable. This includes cases where the actions are reversible such as the route-finding problems and sliding block puzzles.

Infrastructure for search algorithms

- To avoid exploring redundant paths, we can augment the search algorithm with a data structure that stores previously explored nodes (sometimes known as the closed list)
- Newly generated nodes that match previously generated nodes are then discarded instead of being added to the frontier.
- The resulting two alternative implementations are respectively named Tree Search and Graph Search

Tree Search

Agent Tree-Search(*problem*) **returns** solution or failure ;
Initialize the frontier using the initial state of *problem*;
while *frontier is not empty* **do**
    choose a leaf node and remove it from the frontier;
    if *the leaf node contains a goal state* **then**
        return the corresponding solution
    end
    Expand the chosen node, adding the resulting nodes to the frontier
end

Graph Search

Agent Graph-Search(*problem*) **returns** solution or failure ;
Initialize the frontier using the initial state of *problem*;
initialize the explored set to be empty;
while *frontier is not empty* **do**
    choose a leaf node and remove it from the frontier;
    if *the node contains a goal state* **then**
        return the corresponding solution
    end
    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
end

Search Space and Problem representations

- Most problems can be posed either as Optimization tasks (e.g. Road map, travelling salesman,..) or Satisfaction tasks (e.g 8 queens, counterfeit problem)
  - In Optimization problems, the objective is not just to exhibit a formal object satisfying an established set of criteria but also to ascertain that this object possesses qualities unmatched by the other objects in the candidate space
  - In Satisfaction Problems, on the other hand, the only objective is to discover a qualified object with as little search effort as possible.

A first algorithm: Hill climbing

- Hill climbing is simply a loop that continually moves in the direction of increasing value (that is a hill). It terminates when it reaches a "peak", where no neighbor has a higher value.
- The algorithm does not maintain a search tree so the data structure for the current node needs only record the state and the value of the objective function.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state.

A first algorithm: Hill climbing

- Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next

- Hill climbing often makes rapid progress towards a solution because it is usually quite easy to improve a bad state. Unfortunately, it often gets stuck for the following reasons:

  - **Local Maxima**: a local maximum is a peak that is higher than each of its neighboring states but lower than the global minimum. Hill climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere to go.

  - **Ridges** : A Ridge results in a sequence of local maximas that are not directly connected to each other

  - **Plateaux**: a plateau is a flat area of the state space landscape for which no uphill exit exists.

A first algorithm: Hill climbing

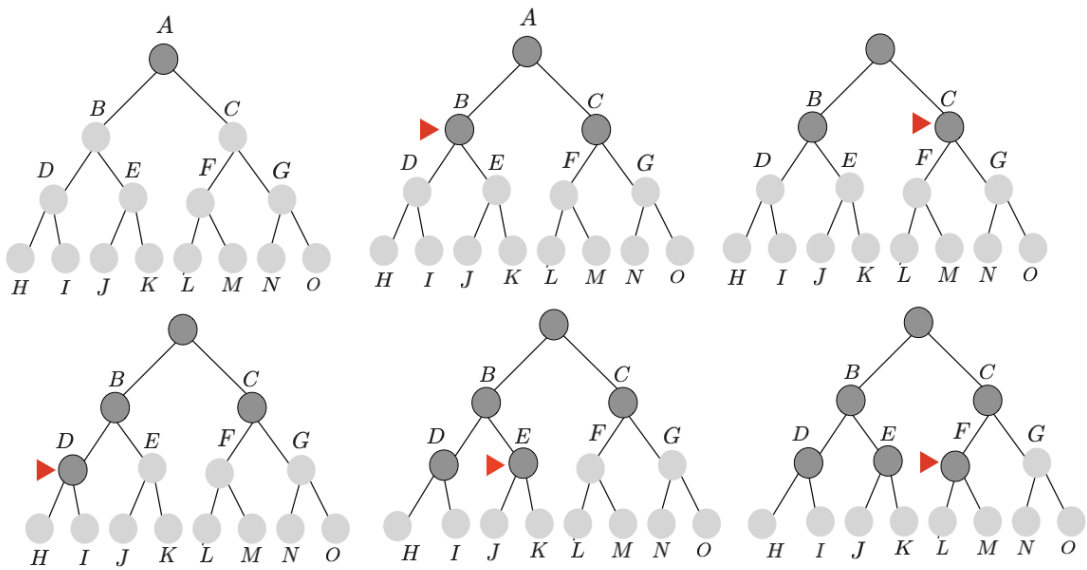
- In each case, the algorithm reaches a point at which no progress is being made.
- Many variants of hill climbing have been invented. Among those **Stochastic Hill Climbing** chooses at random from among the uphill moves (with a probability of selection that can vary with the steepness of the uphill moves), **First-choice hill climbing** imlements hill climbing by generating successors randomly until one is generated that is better than the current state. **Random restart hill climbing** adopts the well known adage: "If at first you don't succeed, try again". and conducts a series of hill-climbing strategies from randomly generated initial states until a goal is found.

A first algorithm: Hill climbing

- The success of Hill Climbing depends very much on the shape of the state space landscape. If there are few local maxima and plateaux, random restart hill climbing will find a good solution very quickly
- On the other hand, many real problems have a landscape that is highly non convex. NP hard problems typically have exponential number of local maxima to get stuck on.

Uninformed Search Methods: Breadth First Search

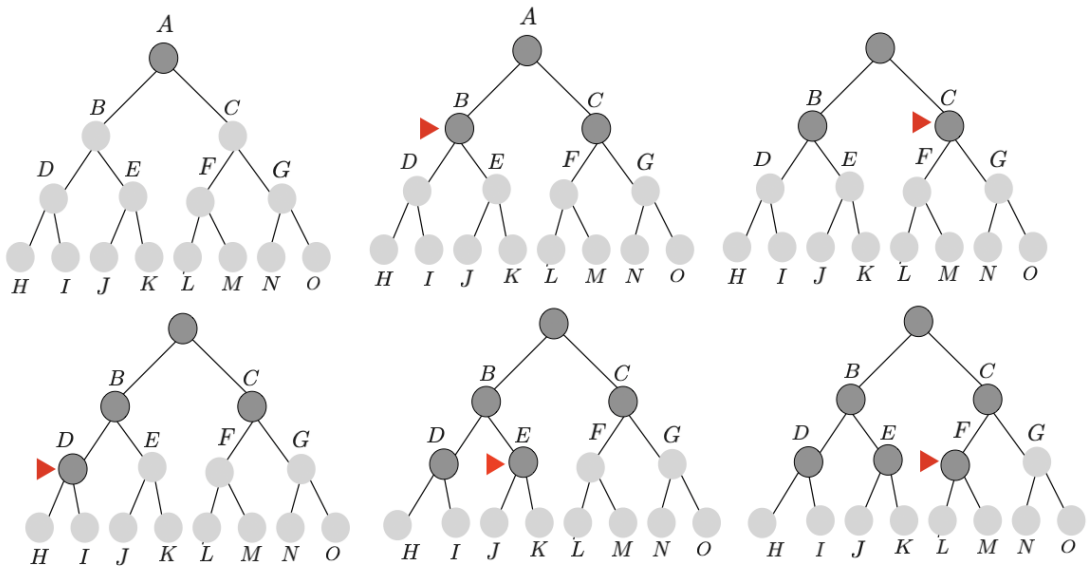
- **Breadth First search** is a simple strategy in which the root node is expanded first then all the successors of the root are expanded, then their successors and so on.
- In general, all the nodes at a given depth in the tree are expanded before any node at the next level can be expanded



Uninformed Search Methods: Breadth First Search

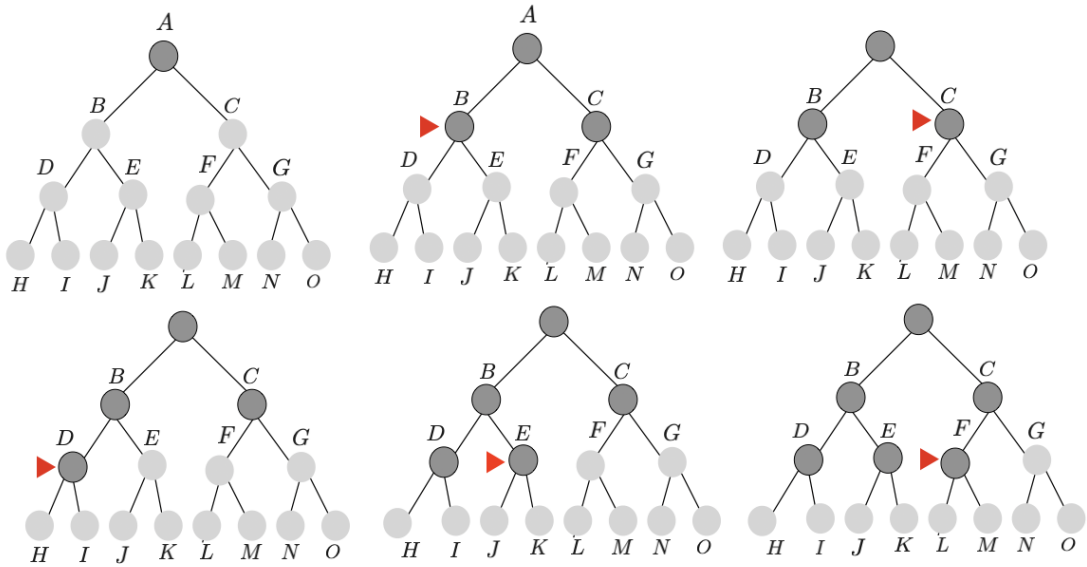
- BFS is carried out using a FIFO queue for the frontier. I.e. new nodes go to the back of the queue and older nodes get expanded first.

- In terms of complexity, if we search a uniform tree such that every state has b successors (the root of the tree thus generated b nodes at the first level, each of which generated b more nodes, for a total of b^2 at the second level, ...), the total number of nodes expanded by BFS to find a goal node located at depth d is thus $O(b^d)$.



Uninformed Search Methods: Breadth First Search

- Consequently, if we want to keep track of the explored set, this explored set contains $O(b^d)$ nodes when reaching the goal state.



Function Breadth-First-Search(*problem*) **returns** a solution or failure
node ← a node with STATE = *problem*.INITIAL-STATE,PATH-COST= 0
if *problem*.GOAL-TEST(*node*.STATE) **then**
  return SOLUTION(*node*)
end
fontier ← a FIFO queue with *node* as the only element
explored ← an empty set
(see next slide)

```
Function Breadth-First-Search(problem) returns a solution or failure
(continued)
while EMPTY?(frontier) do
    node ← POP(frontier) /*shallowest node in frontier*/
    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
            if problem.GOAL-TEST(child.STATE) then
                return SOLUTION(child)
            end
            frontier ← INSERT(child, frontier)
        end
    end
end
```

Uninformed Search Methods: DFS

- **Depth First Search** always expands the deepest node in the current frontier of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded they are droped from the frontier.
- Then the search "backs up" to the next deepest node that still has unexplored successors.
- Whereas Breadth First Search uses a FIFO queue, Depth First Search uses a LIFO stack (which means that it is the most recently generated node that is chosen for expansion)

Uninformed Search Methods: DFS

- It is common to implement Depth First with a recursive function that calls itself on each of its children in turn.
- The time complexity of depth first graph search is bounded by the size of the state space (which may be infinite)
- The main advantage of Depth First Search over Breadth First Search is space complexity. For a depth first graph search, there is no advantage but for a depth first tree search, the search only needs to store a single path from the root node to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
- once a node has been expanded and all its descendants have been explored, it can be removed from memory.

Uninformed Search Methods: DFS

Function Depth-Limited-Search(*problem*) **returns** solution or failure ;
return RECURSIVE-DLS(MAKE-NODE(*problem*, INITIAL-STATE), *problem*, *limit*)

Function Recursive-DLS(*node*, *problem*, *limit*) **returns** a solution or failure/cutoff;
if *problem*.GOAL-TEST(*node*.STATE) **then**
  return SOLUTION(*node*)
end
else if *limit* = *0* **then**
  return *cutoff*
end
else
  cutoff_occured? ← *false*
  (see next slide)
end

Function Recursive-DLS(*node*, *problem*, *limit*) **returns** a solution or failure/cutoff

(see previous slide)

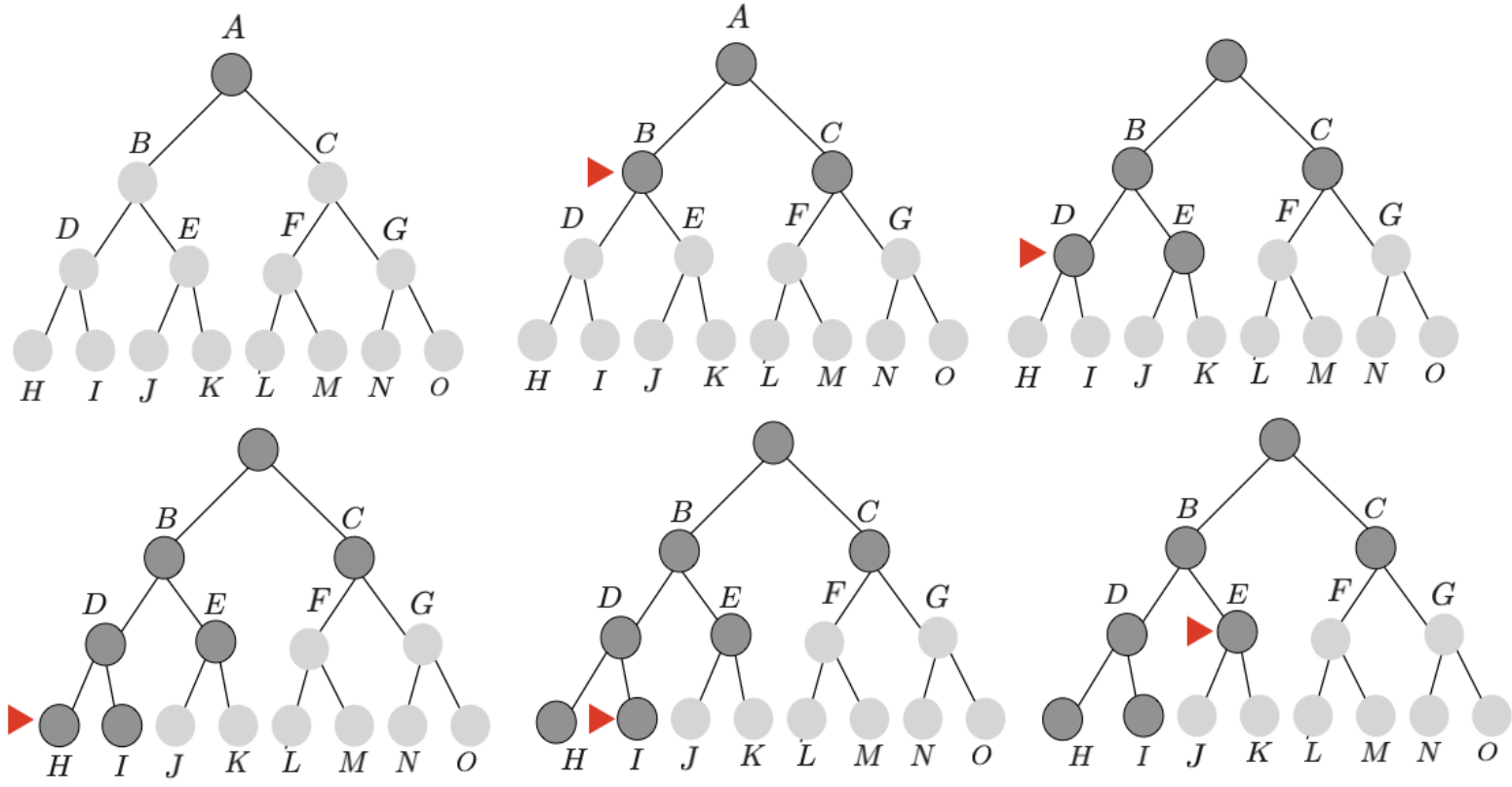
else

```
    cutoff_occured? ← false
    for each action in problem.ACTION(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        result ← RECURSIVE-DLS(child, problem, limit -1)
        if result = cutoff then
            cutoff_occured = true
        end
        else if result ≠ failure then
            return result
        end
    end
    if cutoff_occured? then
        return cutoff
    end
    else
        return failure
    end
end
```

Uninformed Search Methods: DFS

- The properties of Depth First Search depend strongly on whether the graph search or tree search version is used
- A variant of depth first search called **backtracking line search** uses still less memory. In backtracking, only one successor is generated at a time rather than all successors. Each partially expanded node remembers which successors to generate next. If m is the maximum depth, backtracking thus uses $O(m)$ memory instead of $O(bm)$
- If the generated node meets some stopping criterion, the program backtracks to the closest unexpanded ancestor, that is, an ancestor still having ungenerated successors.

Uninformed Search Methods: DFS

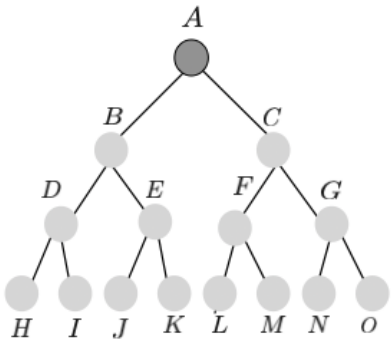


Iterative deepening DFS

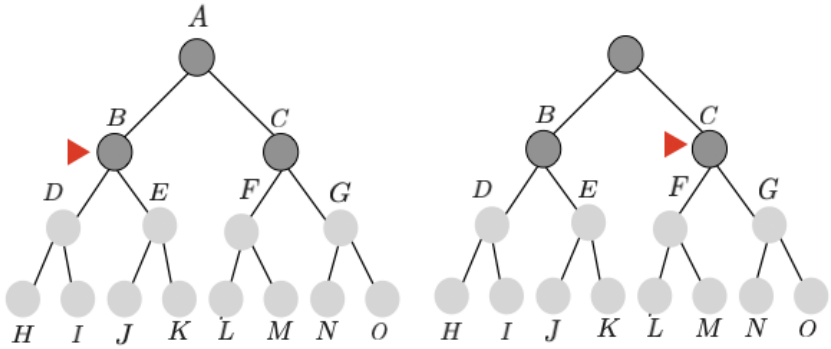
- Iterative deepening search (or iterative deepening DFS) is a general strategy often used in combination with depth first tree search, that finds the best depth
- It does this by gradually increasing the limit until a goal is found.
- Iterative deepening search combines the benefits of DFS and BFS. Like DFS, its memory requirements are modest ($O(bd)$) and like BFS it is complete when the branching factor is finite and optimal when the path cost is a non decreasing function of the depth of the nodes.

Iterative deepening DFS

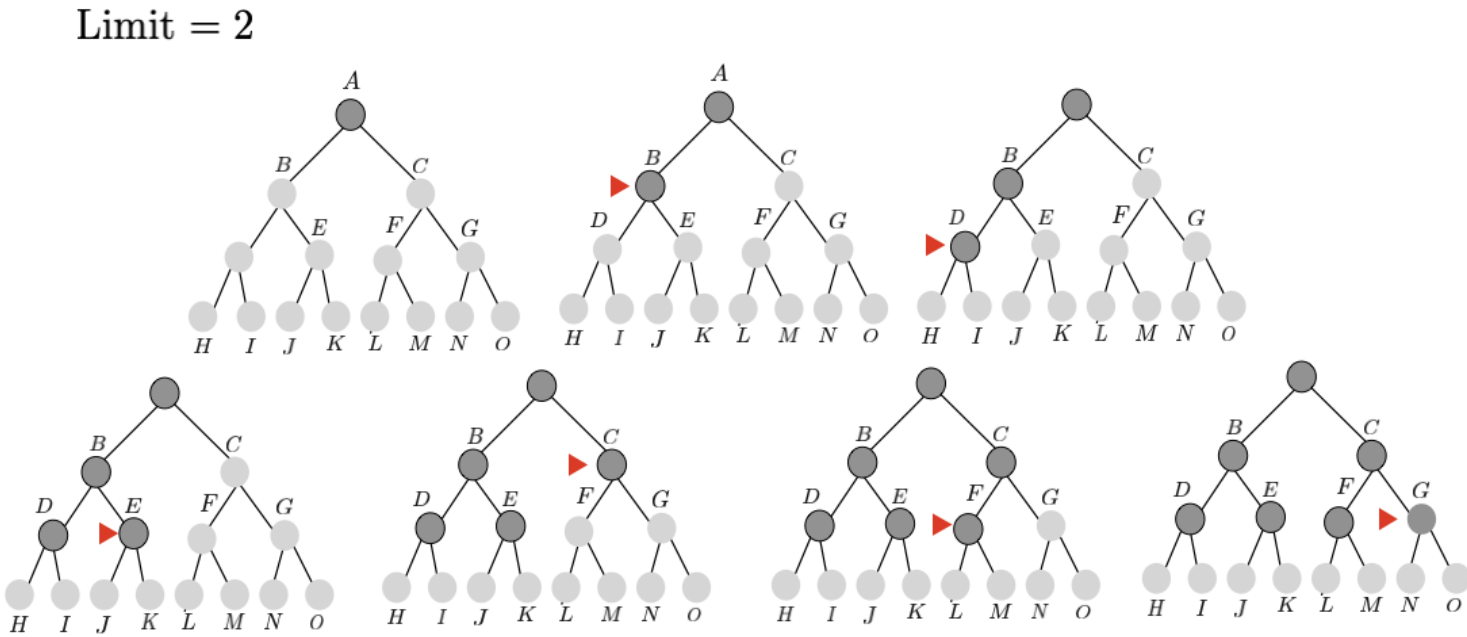
Limit $= 0$



Limit $= 1$

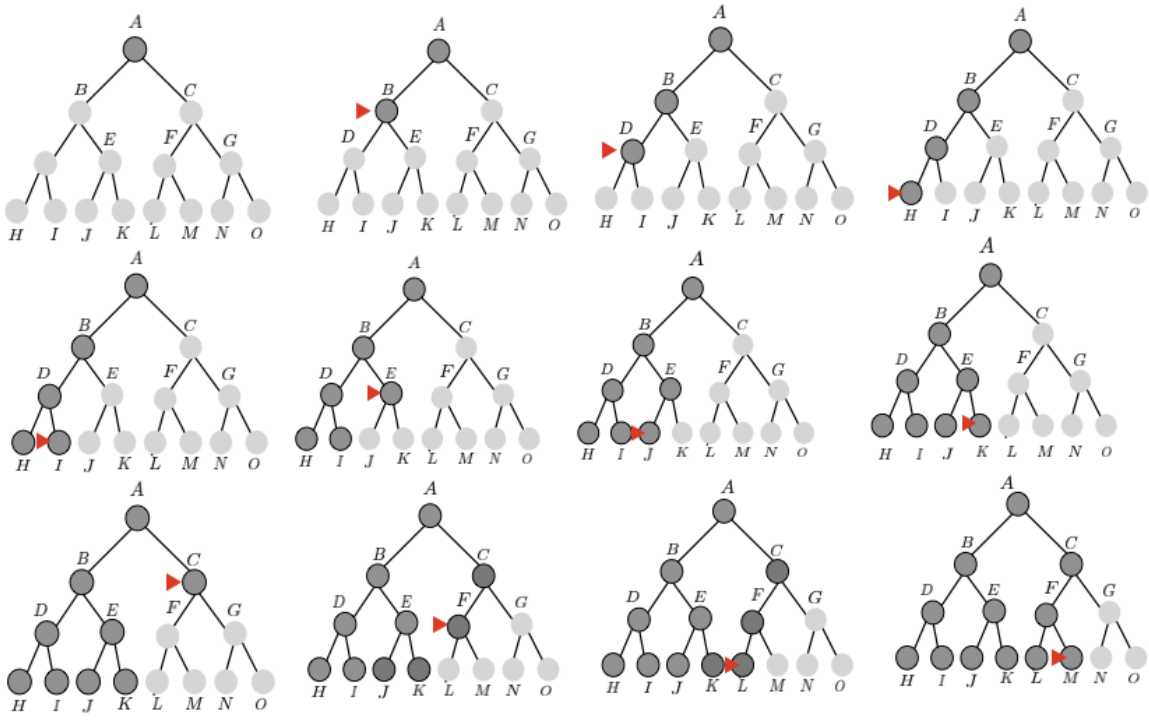


Iterative deepening DFS



Iterative deepening DFS

Limit $= 3$



etc..

Iterative deepening DFS

Function Iterative-Deepening-search(*problem*) **returns** a solution or failure
for *depth* = 0 *to* ∞ **do**
  result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
  if *result* $\neq$ *cutoff* **then**
    return *result*
  end
end

Informed Search Strategies

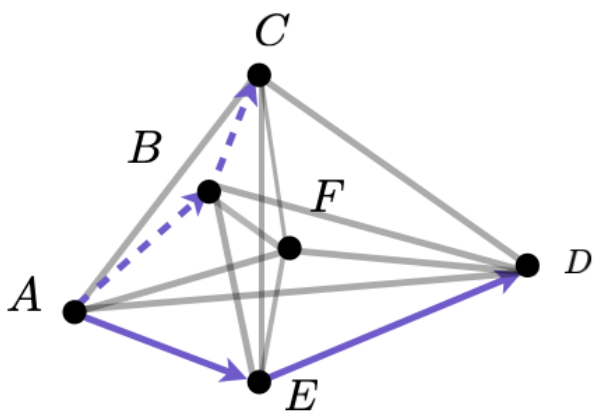
- Informed Search strategies use problem specific knowledge, beyond the definition of the problem itself, to find solutions more efficiently
- Best First Search is a particular instance of the general TREE SEARCH or GRAPH SEARCH algorithm in which a node is selected for expansion based on some evaluation function $f(n)$
- The evaluation $f(n)$ is construed as a cost estimate so that the node with the lowest cost estimate is expanded first.
- The choice of $f(n)$ determines the search strategy. Most informed search strategies include as a component of $f(n)$ a heuristic function denoted $h(n)$.

Informed Search Strategies

- The heuristic function $h(n)$ can be defined as the estimated cost of the cheapest path from the state at node n to a goal state.
- Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm
- Heuristics are usually non negative functions with the constraint that $h(n) = 0$ for any goal node n.

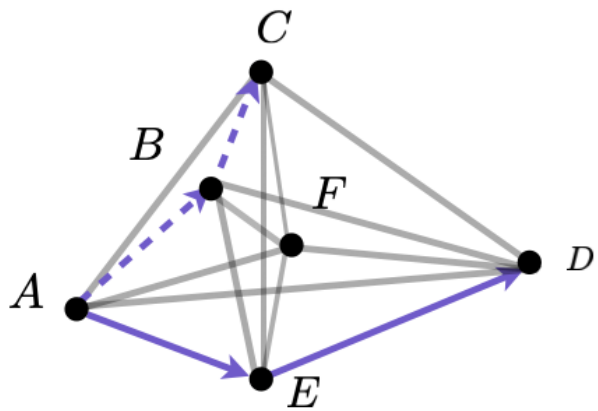
Examples of heuristics (I): Travelling salesman

- Consider the graph below, where the two marked paths ABC and AED represent two candidate subtours being considered by the search procedure.
- We would like to know which of these two if properly completed to form a circuit, is more likely to be part of the optimal solution.
- The overall cost is given by the cost of completing the tour added to the cost of the initial subtour



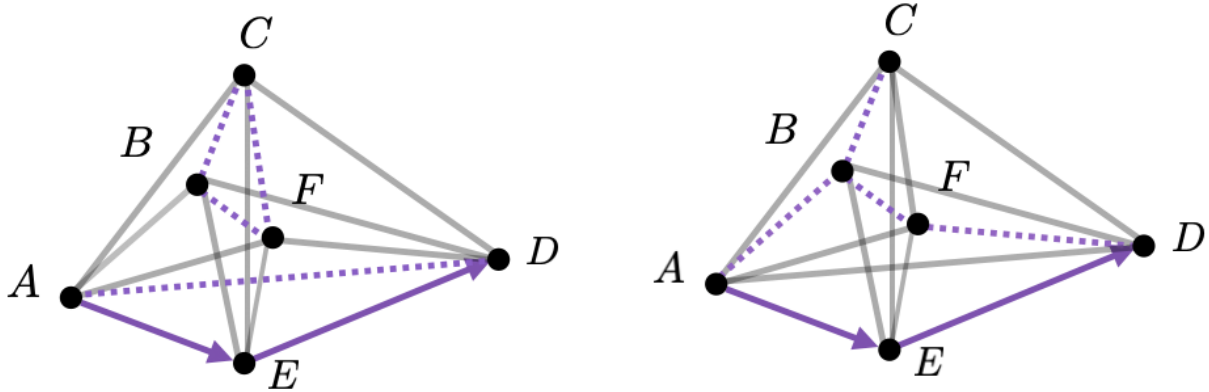
Examples of heuristics (I): Travelling salesman

- However, since the computational effort required to find the optimal completion is almost as hard as as that of finding the entire tour, we must settle for an estimate of the completion cost



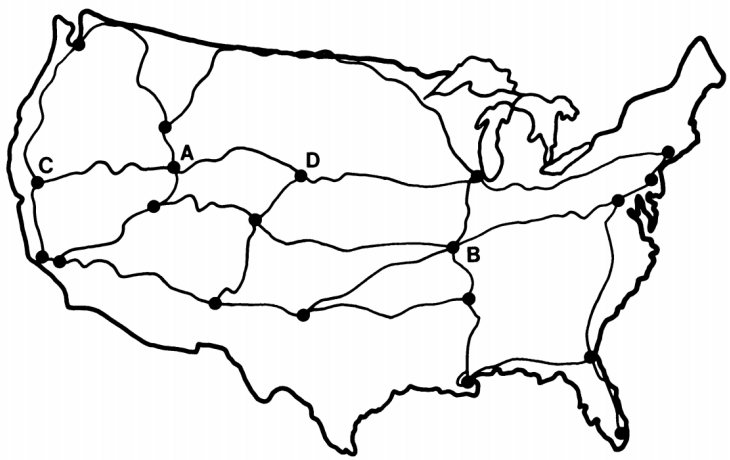
Examples of heuristics (I): Travelling salesman

- Possible heuristics include
  - The cheapest degree 2 graph going through all the remaining nodes ($O(n^3)$)
  - The minimum spanning tree (MST) through all remaining nodes ($O(n^2)$)
- Other simpler heuristics include taking the cost of the edge (or the two edges) from the end of the tour to the initial node.



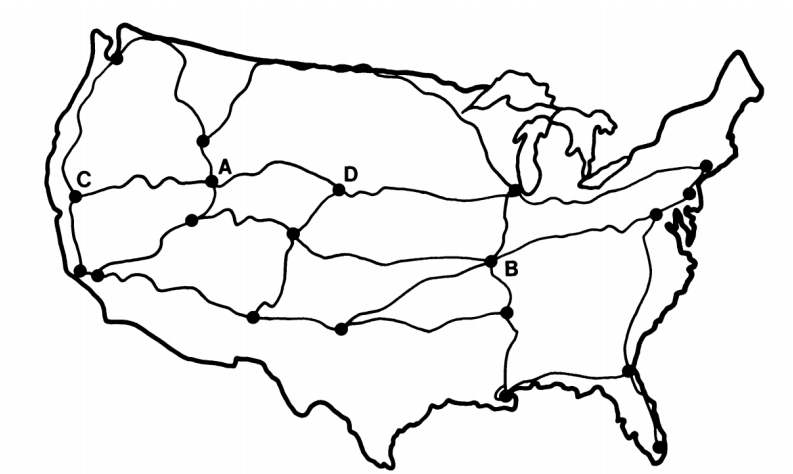
Examples of heuristics (II): Roadmap problem

- Given a map such as shown below, we want to find the shortest path between city A and city B
- When given an actual map, we would like to rule out the roads that lead away from the general direction of the destination.



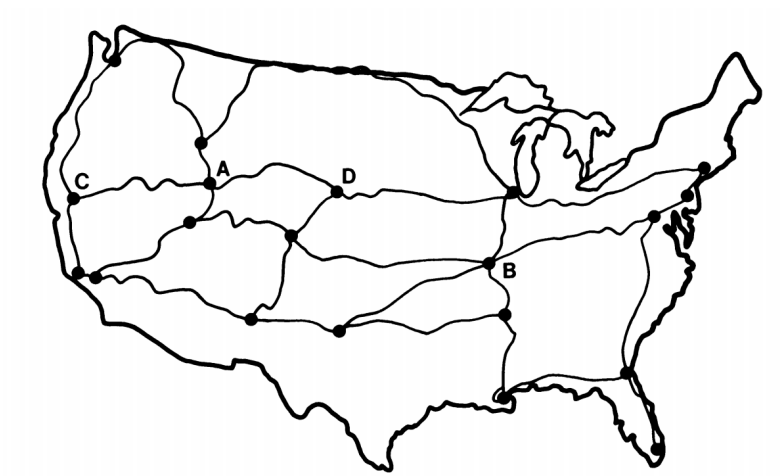
Examples of heuristics (II): Roadmap problem

- A human observer, located in A, wanting to reach B, and looking at the map exploits vision machinery to estimate the Euclidean distances on the map and since the distance from D to B is shorter than the distance from C to B, city D appears as a more promising candidate.



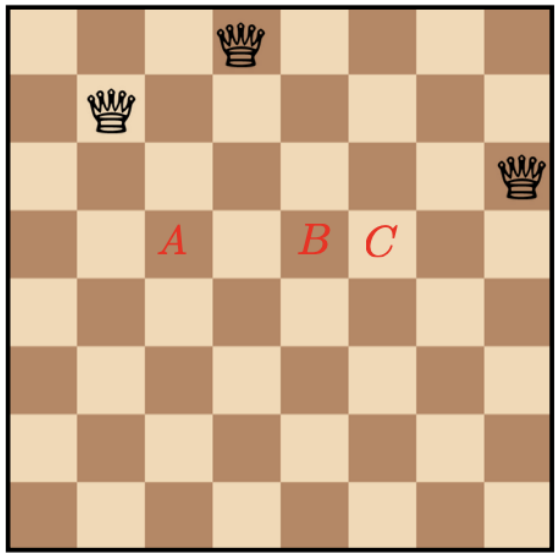
Examples of heuristics (II): Roadmap problem

- In the absence of a map (e.g. when given a table of pairwise distances between connected cities) we could attempt to simulate this extra information
- For example, as we can easily estimate air distances between cities from their coordinates, we can consider a heuristic function $h(i)$ which computes the air distance from city i to the goal city B.



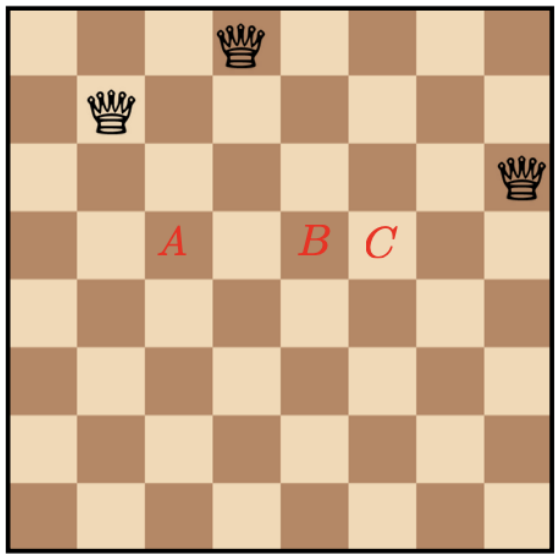
Examples of heuristics (III): 8 queens problem

- The goal of the 8 queens problem is to place eight queens on a chess board such that no queen can attack the others. This is equivalent to placing the queens so that no row, column or diagonal contains more than one queen



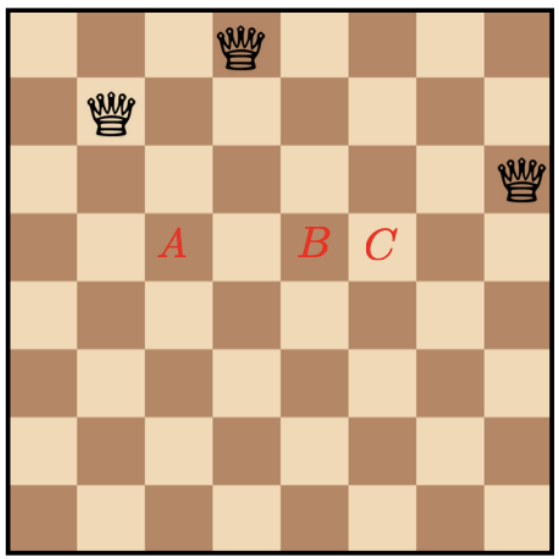
Examples of heuristics (III): 8 queens problem

- As for the other problems, we can forgo the hope of obtaining the solution in one step and proceed step by step, in an incremental manner.



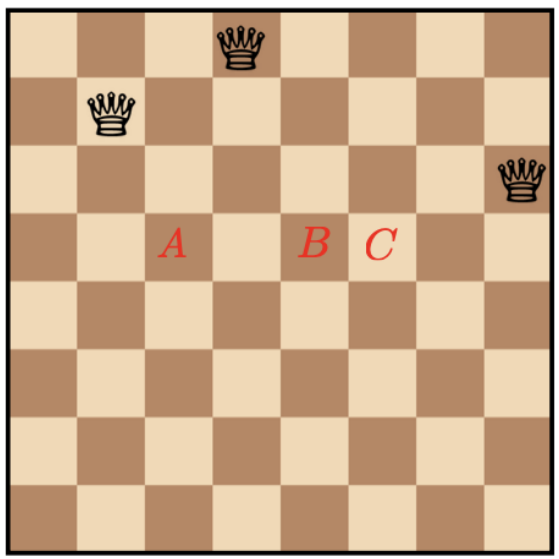
Examples of heuristics (III): 8 queens problem

- One approach could be to start from an arbitrary arrangement of the 8 queens that we would then transform iteratively, going from one board configuration to another, until the queens are adequately dispersed. The transformation should be systematic so that we do not apply the same transformation twice.



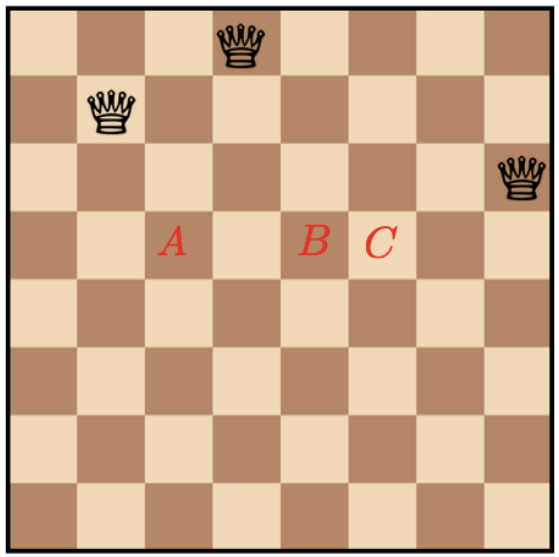
Examples of heuristics (III): 8 queens problem

- An alternative would be to start with an empty board, then attempt to place the queens one at a time. This way we already rule out violations of the problem constraints. Since there can be only one queen in each row, we can assign the first queen to the first row, the second queen to the second row and so on



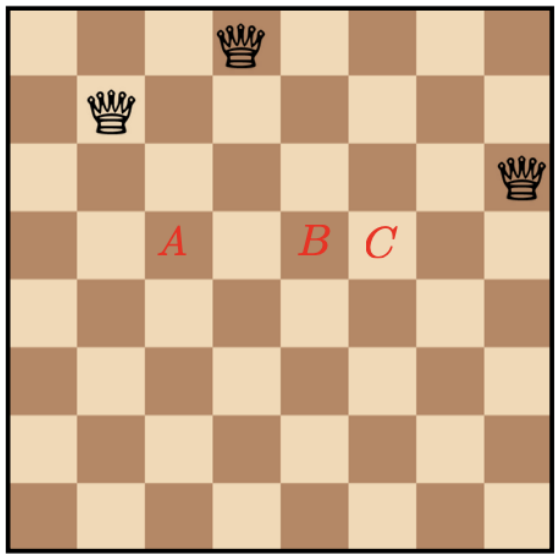
Examples of heuristics (III): 8 queens problem

- Assume that we have positioned three queens as below and wonder whether we should position the fourth on A, B or C. A heuristic in this case would have to determine, at least tentatively, which of the three positions appears to have the highest chance of leading to a satisfactory solution.



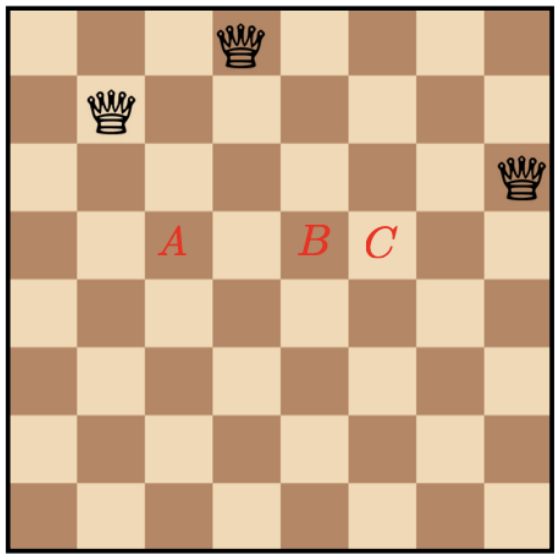
Examples of heuristics (III): 8 queens problem

- We could define a first heuristic by preferring candidate solutions that leaves the highest number of unattacked cells on the board (i.e. to be able to place the remaining queens, we want to leave as many options as possible for future additions). If we let $f(\cdot)$ denote the number of unattacked cells, we get $f(A) = 8$, $f(B) = 9$ and $f(C) = 10$.



Examples of heuristics (III): 8 queens problem

- A more sophisticated heuristic would focus on the rows with the smallest number of unattacked cells as those rows are more likely to be blocked quicker in the future. We should then focus our attention on number of cells left by each option on those rows. Using this as our heuristic, we get $f'(A) = f'(B) = 1$ and $f'(C) = 2$ (i.e C leaves two rows with 2 unattacked cells)



Properties of heuristic methods

- An algorithm is said to be **complete** if it terminates with a solution when one exists
- An algorithm is **admissible** if it is guaranteed to return an optimal solution whenever such a solution exists
- An algorithm A_1 is said to **dominate** another algorithm A_2 if every node expanded by A_1 is also expanded by A_2. Similarly, A_1 strictly dominates A_2 if A_1 dominates A_2 and A_2 does not dominate A_1. The expression "*more effcient than*" is sometimes used instead of "*dominates*".
- An algorithm is said to be **optimal** over a class of algorithms if it dominates all members of this class.

Bidirectional Search

- The idea behind Bidirectional Search is to run two simultaneous searches: one forward from the initial state and the other backward from the goal, hoping that the two searches meet in the middle
- Bidirectional Search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect. If they do, a solution has been found.

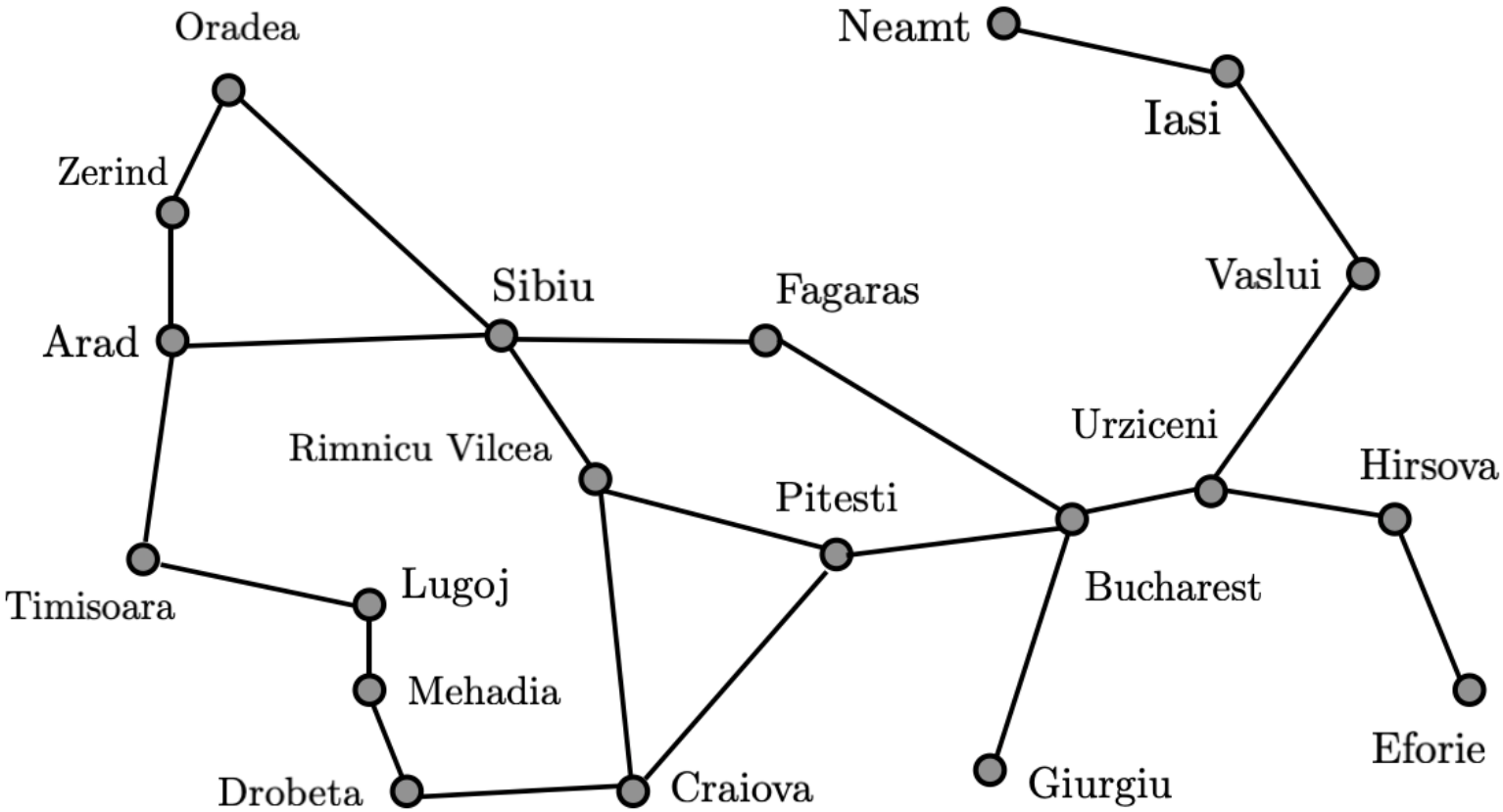
Informed Search Methods: (Greedy) Best First

- Greedy Best First Search tries to expand the node that is the closest to the goal, on the grounds that it is likely to lead to a solution quickly.
- Best First Search evaluates nodes by relying solely on the heuristic function $f(n) = h(n)$
- Greedy Best First Search is incomplete, even in a finite state space, just as DFS.

Informed Search Methods: (Greedy) Best First

- To illustrate the incompleteness of Best First Search, consider the problem of getting from **Iasi** to **Fagaras** on the Romanian map and to rely on the straight line distance as our heuristic.
- This heuristic suggests that Neamt should be expanded first because it is the closest to Faragas but it is a dead end.
- The next solution would be to go to **Vaslui**, a step that is actually farther from the goal according to the heuristic, and then continue to Urziceni, Bucharest and Fagaras.
- However, the algorithm will never find this solution because expanding **Neamt** puts **Iasi** back into the frontier and **Iasi** is closer to **Fagaras** than **Vaslui**. From this, **Iasi** will be expanded again, leading to an infinite loop.

Informed Search Methods: (Greedy) Best First



Informed Search Methods: (Greedy) Best First

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vasui	199
Lugoj	244	Zerind	374

Informed Search Methods: (Greedy) Best First

Function Recursive-best-first-search(*problem*) **returns** a solution or failure
return RBFS(*problem*, MAKE-NODE(*problem*, INITIAL-STATE), ∞)

Function RBFS(*problem*, *node*, f_{limit})
returns a solution or failure and a new f_{cost} limit
if *problem*.GOAL-TEST(*node*.STATE) **then**
  return SOLUTION(*node*)
end
successors $\leftarrow$ []
for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
  add CHILD-NODE(*problem*, *node*, *action*) into successors
end
if *successors is empty* **then**
  return *failure*, ∞
end
for each *s* **in** *successors* **do**
  /*update f with value from previous search, if any*/
  $s.f \leftarrow \max(s.g + s.h, node.f)$
end
(See next slide)

Informed Search Methods: (Greedy) Best First

Function RBFS(*problem*, *node*, f_{limit})
returns a solution or failure and a new f_{cost} limit
(See previous slide)
while ∞ **do**
  best $\leftarrow$ the lowest $f-$*value* node in *successors*
  if *best.f* $>$ f_{limit} **then**
    return *failure*, *best*.f
  end
  alternative $\leftarrow$ the second-lowest f-value among *successors*
  result, *best*.f $\leftarrow$ RBFS(*problem*, *best*, min(f_{lim}, *alternative*))
  if *result*$\neq$ *failure* **then**
    return result
  end
end

Informed Search Methods: A*

- The most widely known form of best first search strategies is called A^* **search** (A-star search). It evaluates nodes by combining $g(n)$, the cost to reach the node and $h(n)$, the cost to get from the node to the goal. I.e. $f(n) = g(n) + h(n)$
- Since $g(n)$ gives the path cost from the start node to node n and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have

$$f(n) = \text{estimated cost of the cheapest solution through } n$$

- Hence, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value for $g(n) + h(n)$.

Informed Search Methods: A^*

- It turns out that this strategy is more than just reasonable. Provided that the heuristic function $h(n)$ satisfies certain conditions, A^* search is both complete and optimal
- The algorithm is identical to UNIFORM-COST-SEARCH except that A^* uses $g + h$ instead of g.

Informed Search Methods: (Greedy) Best First

Function A^* SEARCH(*problem*) **returns** a solution or failure
Put first node *s* on OPEN
if OPEN *is empty* **then**
  exit with failure
end
Remove from OPEN and place on CLOSED a node *n* for which *f* is minimum
if *n is a goal node* **then**
  exit successfully with the solution obtained by
  tracing back the pointers from *n* to *s*
end
(see next slide)

Informed Search Methods: (Greedy) Best First

Function A^* SEARCH(*problem*) **returns** a solution or failure
(see previous slide)
else
  Expand n, generating all its successors, and attach to them pointers back to n.
  for every *successor* n' *of* n **do**
    if n' *is not already on* OPEN *or* CLOSED **then**
      estimate $h(n')$ (estimate of the cost of the best path from n' to some goal node) and calculate $f(n') = g(n') + h(n')$ where $g(n') = g(n) + c(n, n')$ and $g(s) = 0$
    end
    if n' *is already on* OPEN *or* CLOSED **then**
      direct its pointers along the path yielding the lowest $g(n')$
    end
    if n' *required pointer adjustment and was found on* CLOSED **then**
      reopen it
    end
  end
end
Go to step 2.

A^* search

- The first condition we require for optimality is that $h(n)$ is an admissible heuristic. An admissible heuristic is one that never overestimates the cost to reach the goal. Because $g(n)$ is the actual cost to reach n along the current path and $f(n) = g(n) + h(n)$, we have as an immediate consequence that $f(n)$ never overestimates the true cost of a solution along the current path through n

- Admissible heuristics are by nature optimistic as they assume that the cost of solving the problem is less than it actually is.

- An obvious example of admissible heuristic is the straight line distance that we used to navigate through Romania.

A^* search

- A second condition, called consistency (or sometimes monotonicity) is required only for applications of A^* to graph search. A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal node from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'

$$h(n) \leq c(n, a, n') + h(n')$$

- Note that his makes perfect sense for admissible heuristics as a violation of this inequality would imply that there is a route from n to the goal G_n via n' that is cheaper than $h(n)$ (this would in turn violate the property that $h(n)$ is a lower bound on the cost to reach G_n)

A^* search

- The tree-search version of A^* is optimal if $h(n)$ is admissible and the graph-search version of A^* is optimal if $h(n)$ is consistent