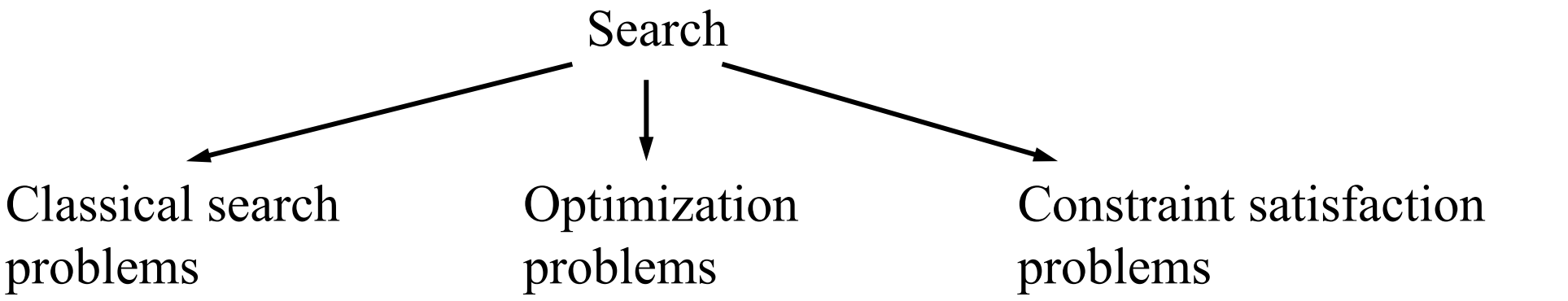
COSC 4368
Fundamentals of Artificial Intelligence

Search Review
October 4th, 2023

Classification of Search Problems

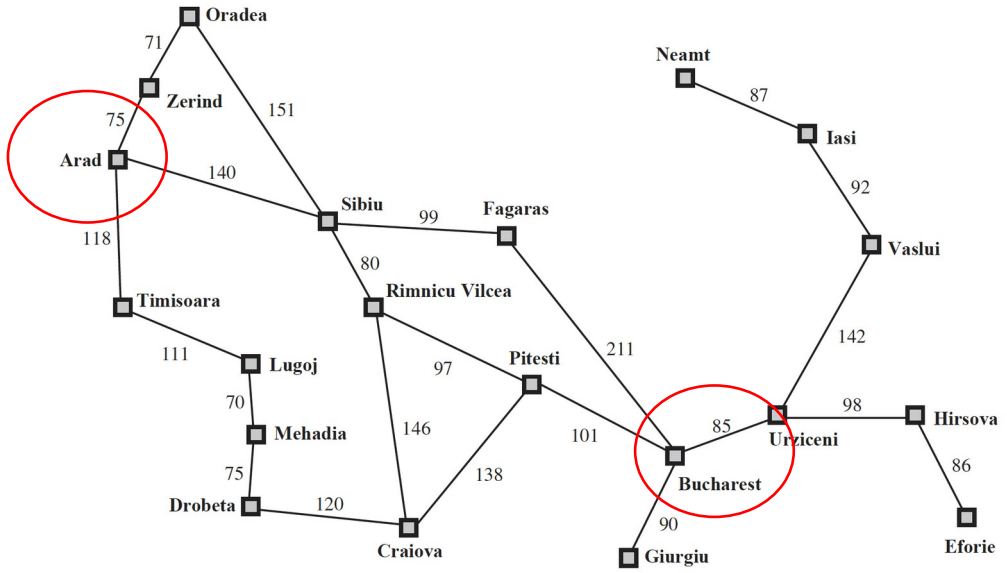
- Depending on the nature of problems, search problems can be divided into multiple categories:



When these assumptions do not hold, the problem becomes more difficult

Problem Formulation

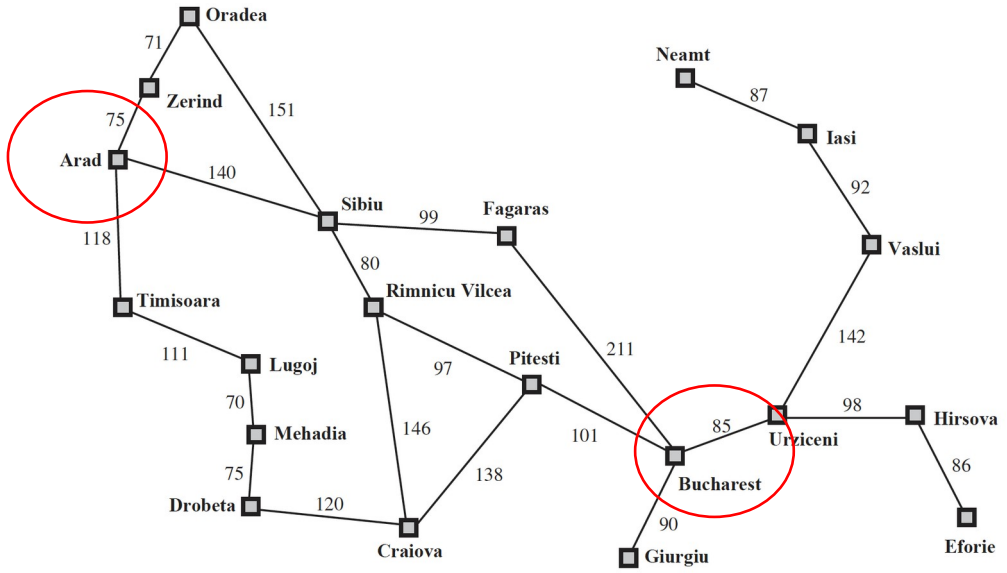
- Five components:
  1. Initial state and State space
  2. Action space
  3. Transition model
  4. Goal test
  5. Path cost



- Initial state: where the agent starts from
  - e.g., *In(Arad)*
- State space: the set of all reachable states from the initial state by any sequence of actions
  - e.g., {*In(Arad), In(Zerind), In(Sibiu),…* }

Problem Formulation

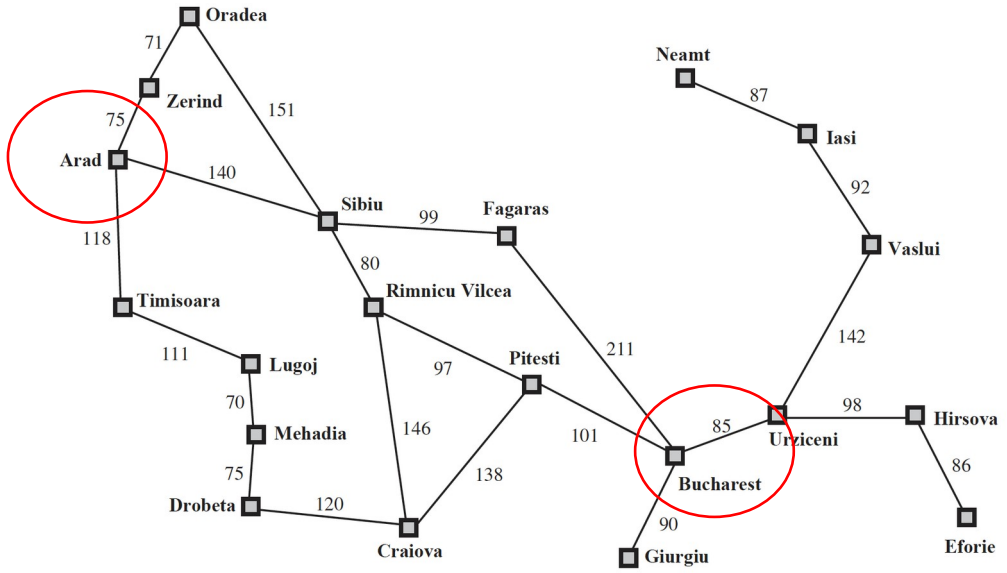
- Five components:
  1. State space and Initial state
  2. Action space
  3. Transition model
  4. Goal test
  5. Path cost



- Action space: the set of all possible actions the agent can take
  - e.g., action_space(state=*In(Arad)*)={*Go(Zerind), Go(Sibiu), Go(Timisoara)*}

Problem Formulation

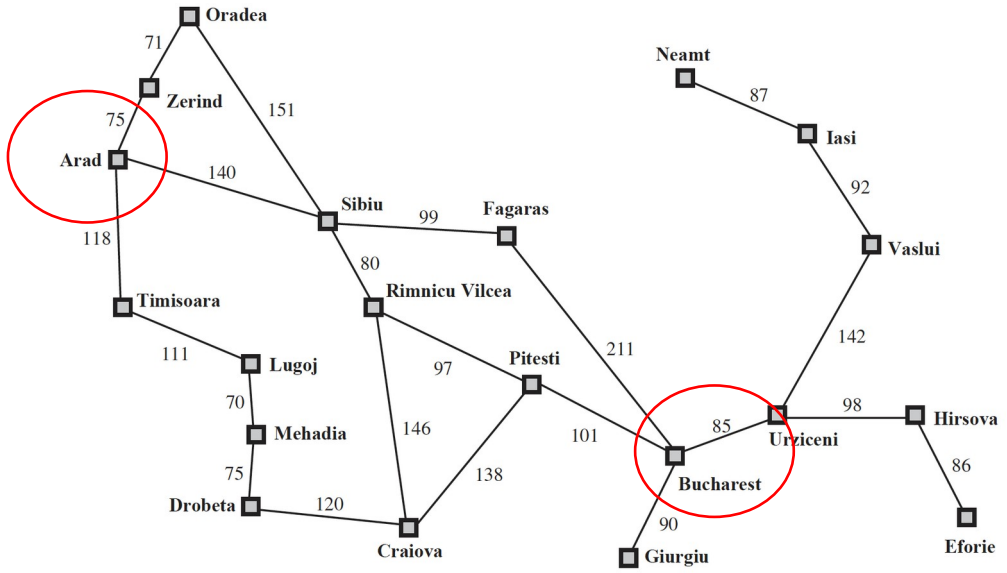
- Five components:
  1. State space and Initial state
  2. Action space
  3. Transition model
  4. Goal test
  5. Path cost



- Transition model: returns the next state *s'* that results from taking action *a* in current state *s*
  - e.g., RESULT(state=*In(Arad), Go(Zerind)*)=*In(Zerind)*

Problem Formulation

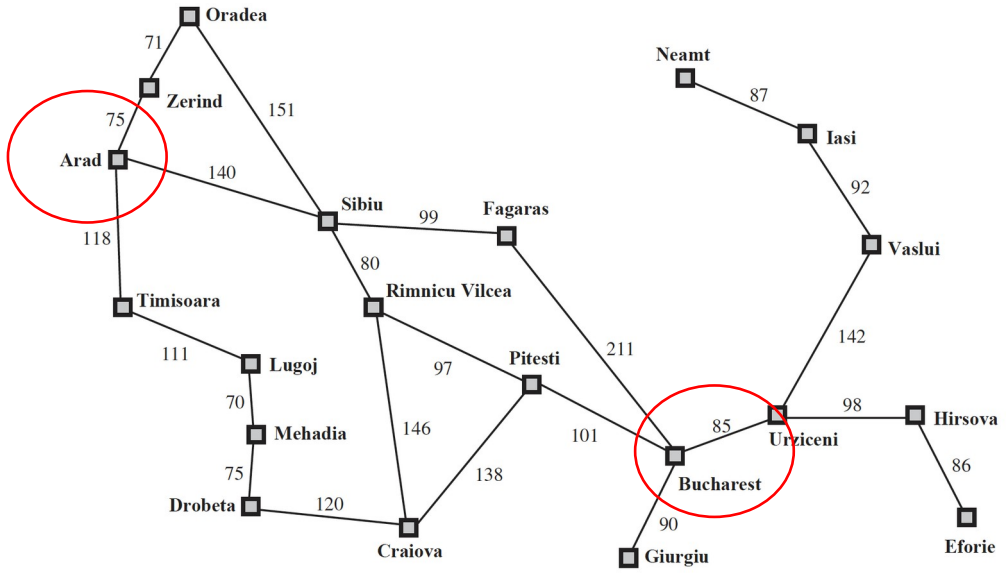
- Five components:
  1. State space and Initial state
  2. Action space
  3. Transition model
  4. Goal test
  5. Path cost



- Goal test: check if a given state is a goal state
  - e.g., check if current_state = *In(Bucharest)*

Problem Formulation

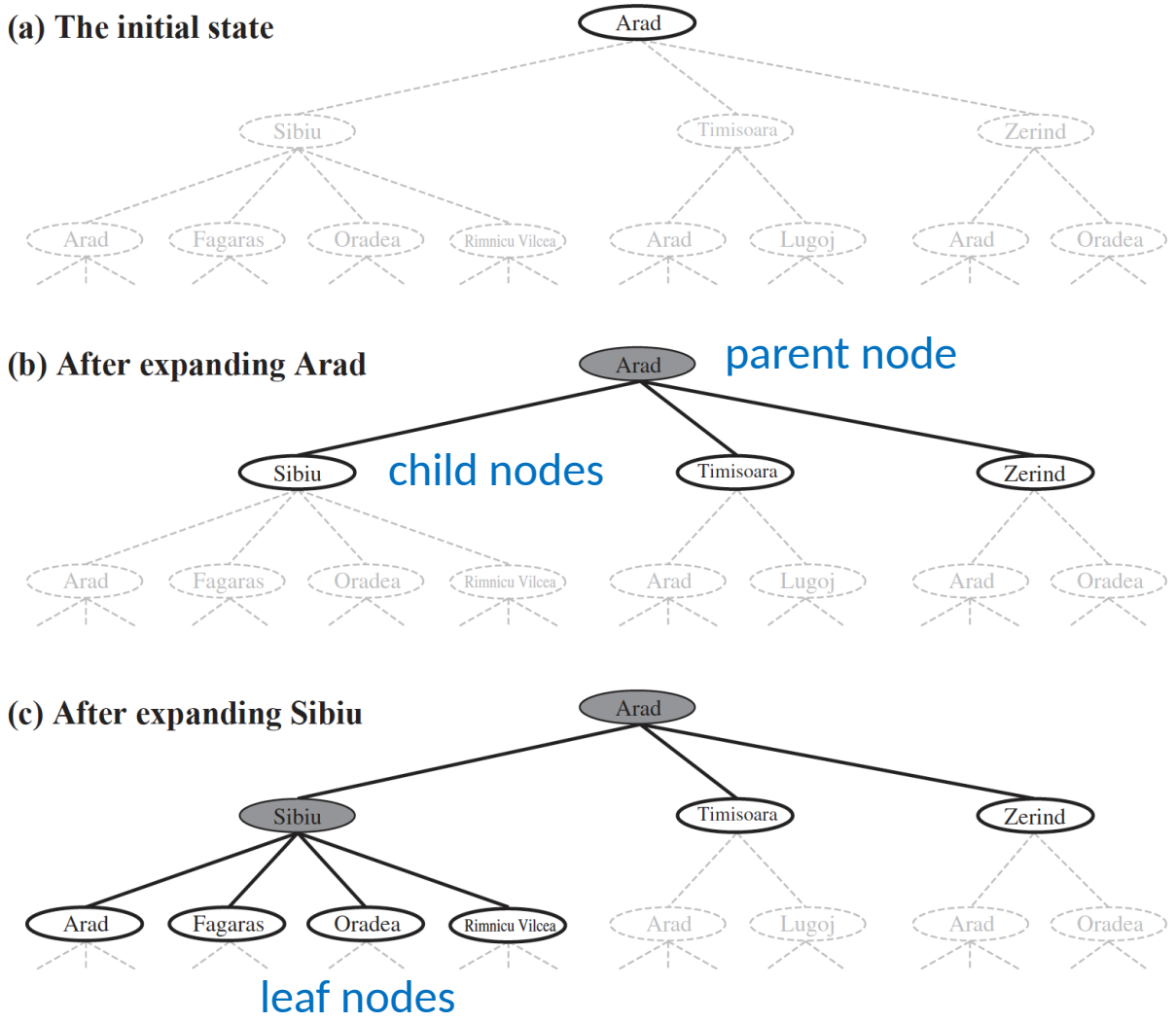
- Five components:
  1. State space and Initial state
  2. Action space
  3. Transition model
  4. Goal test
  5. Path cost



- Path cost: a numeric cost for a path (a sequence of states connected by a sequence of actions)
  - Step cost: cost(current state *s*, take action *a*, next state *s'*)
    - e.g., cost(*In(Arad), Go(Zerind), In(Zerind)*)=*75* (distance in km)
  - Path cost: sum of step costs along the path
    - E.g., path_cost({Arad, Sibiu, Fagaras})=*140+99=239*

Searching for Solutions

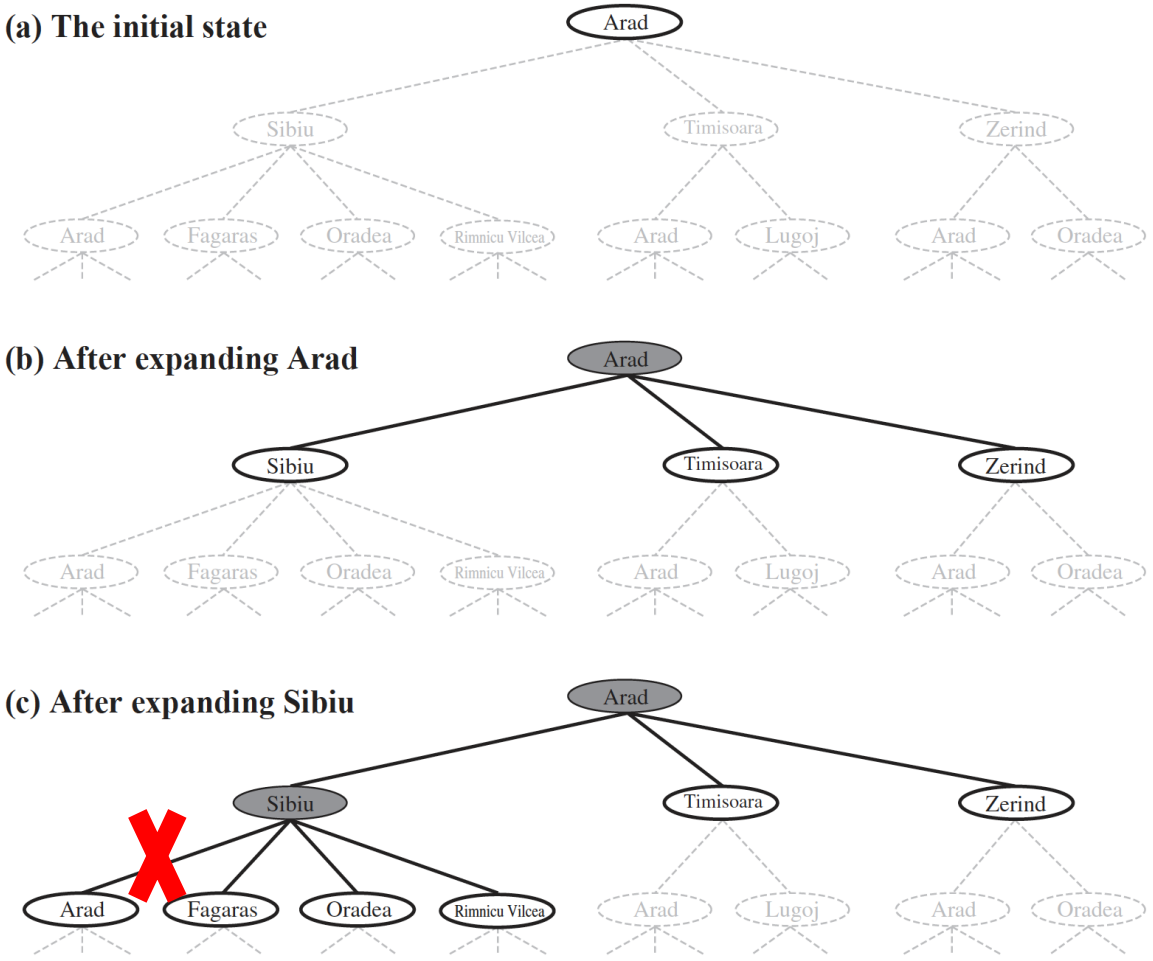
- Solution is a sequence of actions
- Search tree:
  - The possible action sequences starting from the initial state form a search tree with the initial state as the root
  - Branches correspond to actions, nodes correspond to states
  - Expand new nodes to grow the tree
- Search algorithms all have this basic structure; they vary in search strategy – choose which state to expand next

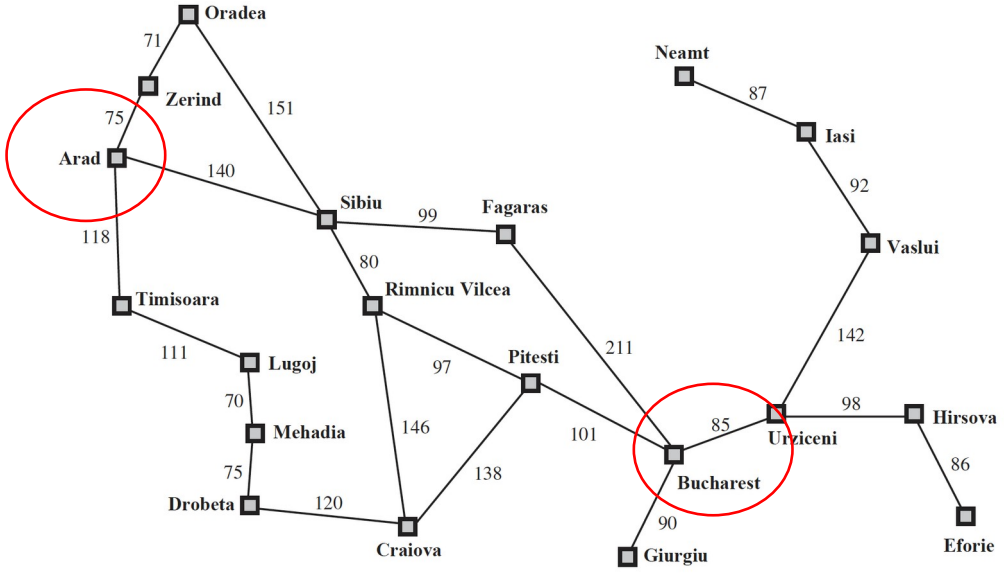


Frontier: set of lead nodes available for expansion

Avoid Looping & Repeated States

- Tree search vs Graph search
  - Graph search removes the redundant paths by introducing the explored set to remember the expanded nodes





Explored set={Arad, Sibiu}
Frontier={Fagaras, Oradea, Rimnicu Vilcea, Timisoara, Zerind}

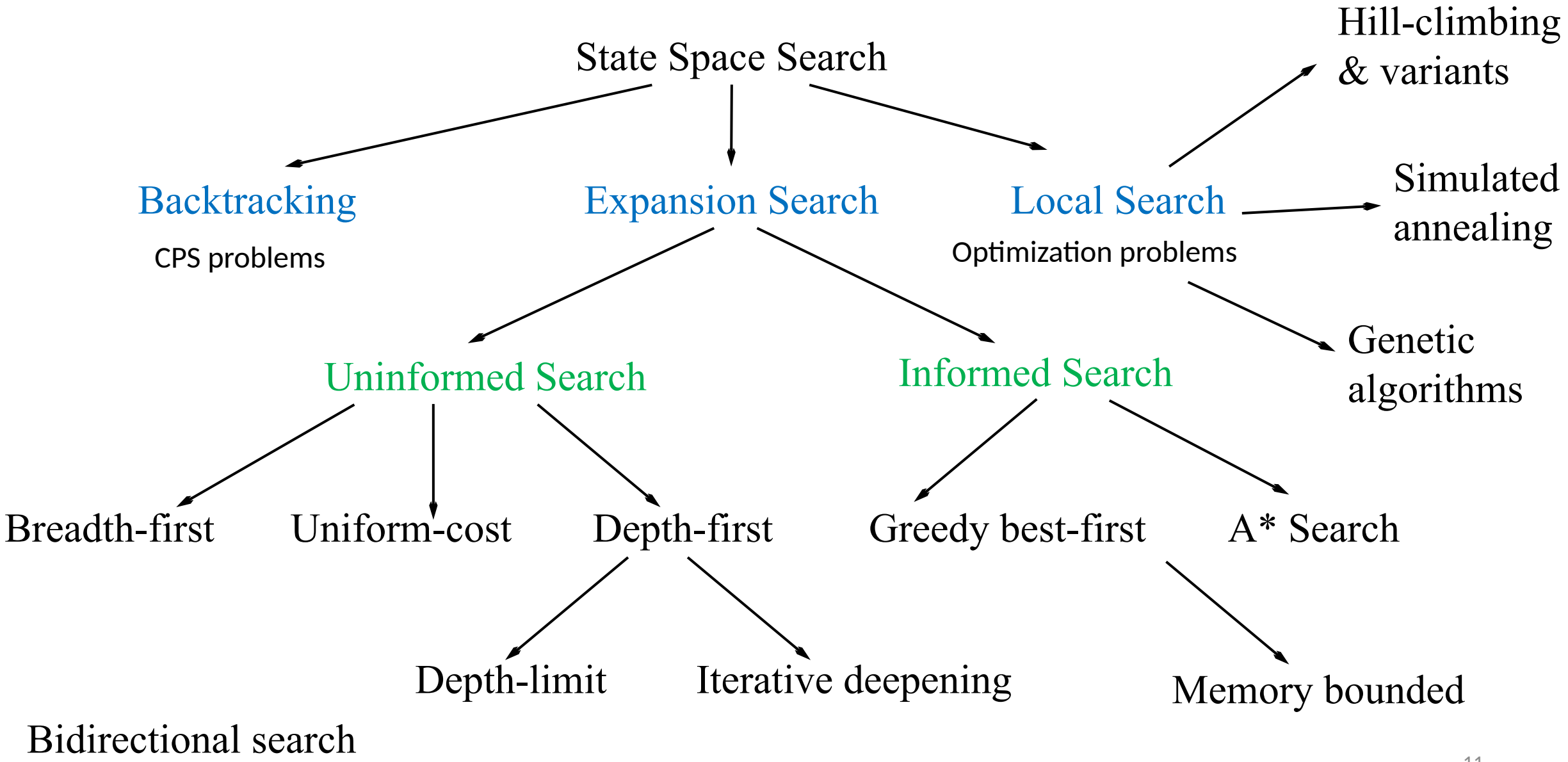
Performance Measure

- Four elements of performance:
  - Completeness
  - Optimality
  - Time Complexity
  - Space Complexity

- Parameters for the complexity
  - Branching factor:
  - Depth of the shallowest goal node:
  - Maximum length of any path in the state space:

- Search cost and total cost

Classification of Search Algorithms

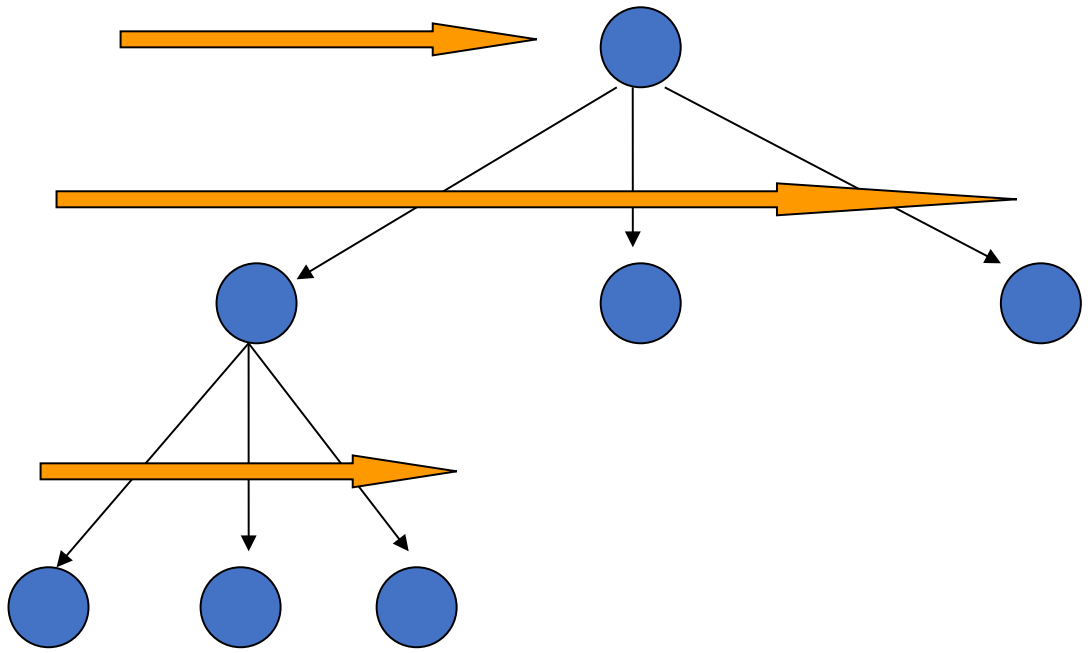


Uninformed Search

- Generate successors and distinguish a goal state from a non-goal state
- Only consider the problem definition itself

Breadth-first Search

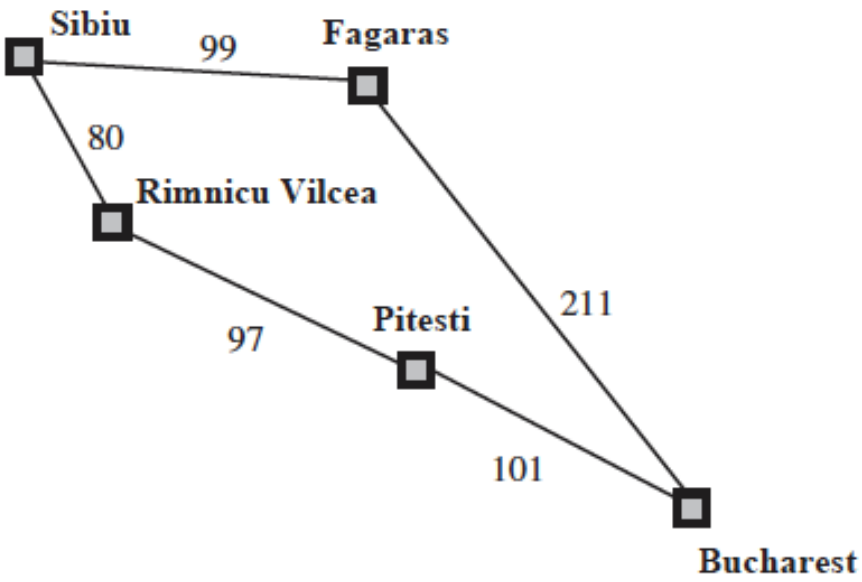
- Root is expanded first
- Then all successors at level 2
- Then all successors at level 3, etc.
- Goal test when a node is generated



- Properties:
  - Complete if and are finite
  - Optimal if path cost increases with depth
  - Time complexity and space complexity are both

Uniform-cost Search

- Path cost is important: always expand the node with lowest path cost
- Goal test when a node is selected for expansion

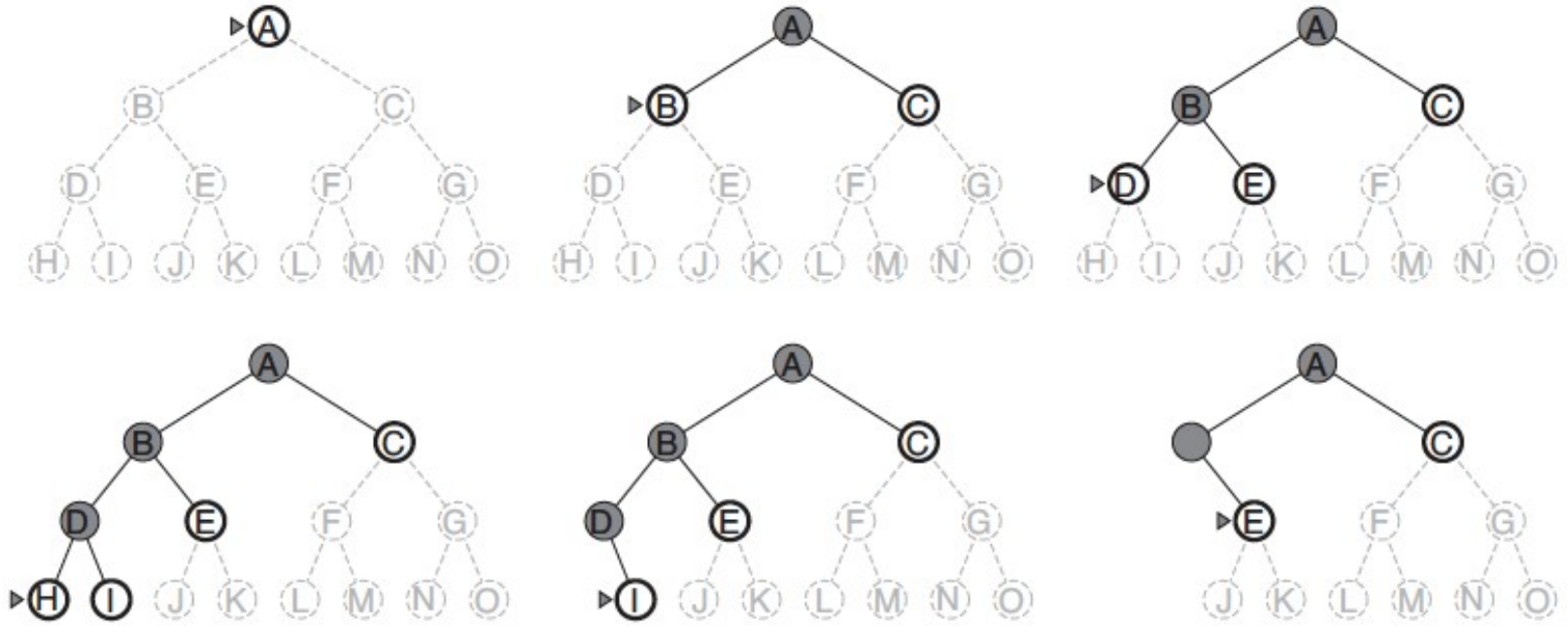


- Properties:
  - Complete if and are finite (step cost positive constant
  - Optimal if step cost is at least some positive constant
  - Cost:
    - Let denote the cost of the optimal solution
    - Assume that step cost is at least
    - Worst-case time and space complexity is
  - Could be worse than breadth-first search (similar cost when all step costs are equal)

Exponential complexity is always a big problem

Depth-first Search

- Always expand the deepest node at the bottom of the tree
- Search proceeds immediately to the deepest level
- Back up to the next deepest node that still has unexplored successors



Depth-first Search

- Properties:
  - Complete only for graph search in finite state spaces
  - Suboptimal
  - No clear advantage in terms of time complexity
  - Space complexity for tree-search version:
    - Only need to store a single path from the root to a leaf node and the remaining unexpanded sibling nodes for each node on the path
    - Backtracking even does better space-wise:

Depth-first search can fail embarrassingly in infinite state spaces

Depth-limited Search

- Depth-first search can fail in infinite state spaces
- Like depth-first search but with depth limit
  - Solve the infinite-path problem

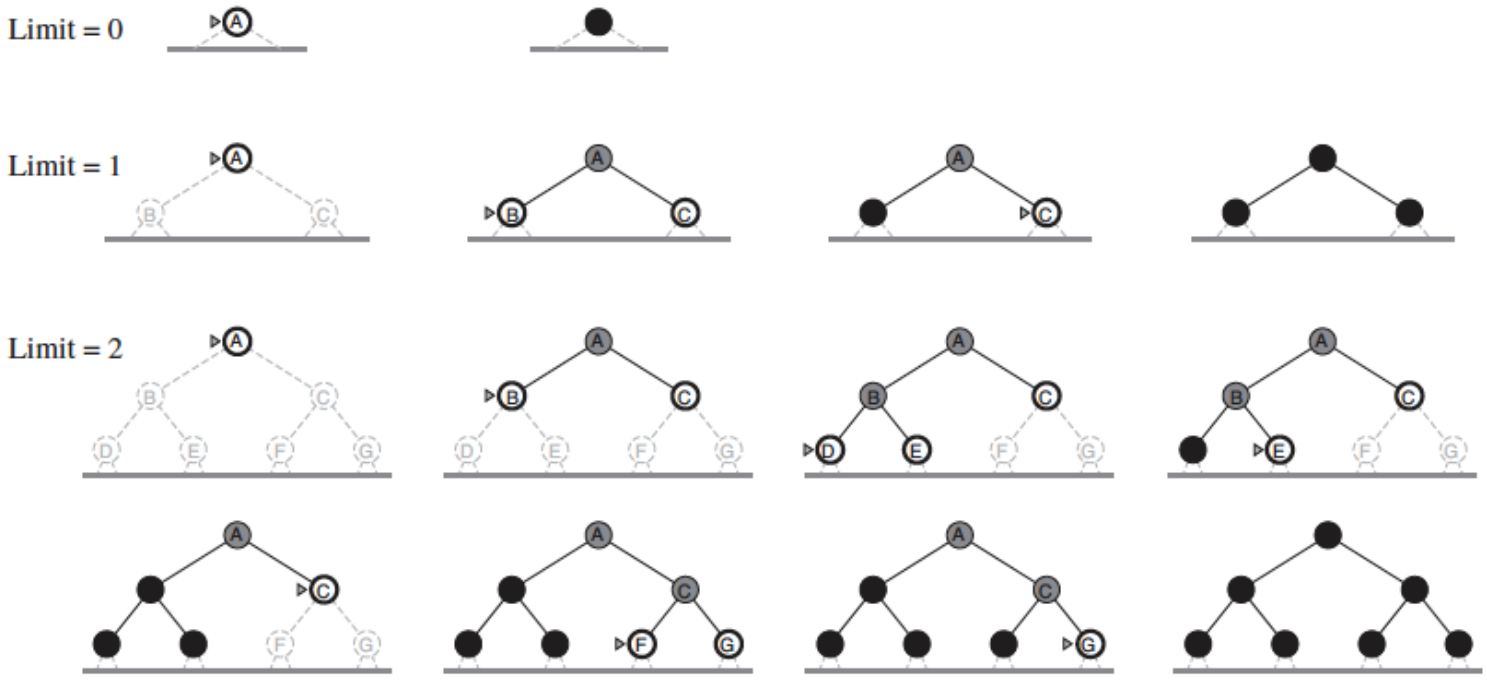
- Properties:
  - Incomplete if
  - Suboptimal if (similar to depth-first)
  - Time complexity is
  - Space complexity is

How to address the incompleteness and suboptimality?

Iterative Deepening Search

- A combination of depth and breadth-first search
- Gradually increase the limit until (a goal will be found)

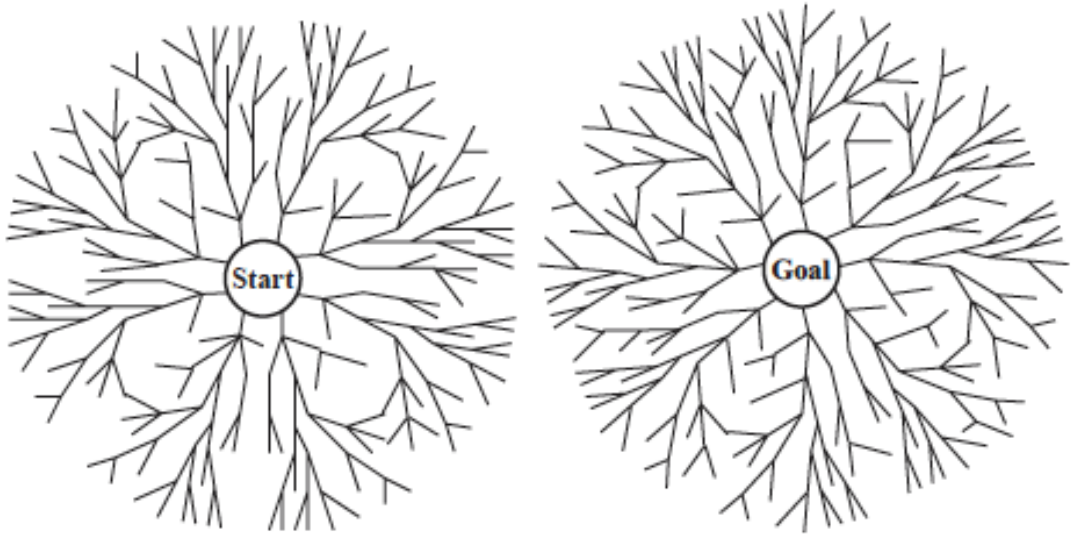
- Properties:
  - Complete if and are finite
  - Optimal if path cost increases with depth
  - Time complexity
  - Space complexity



In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

Bidirectional Search

- Previous methods still suffer from exponential time complexity
- Run two simultaneous searches
  - One forward from the initial state; the other backward from the goal
  - Check if the frontiers of the two searches intersect
- Properties (when both use breadth-first):
  - Complete if and are finite
  - Optimal if path cost increases with depth
  - Time complexity
  - Space complexity

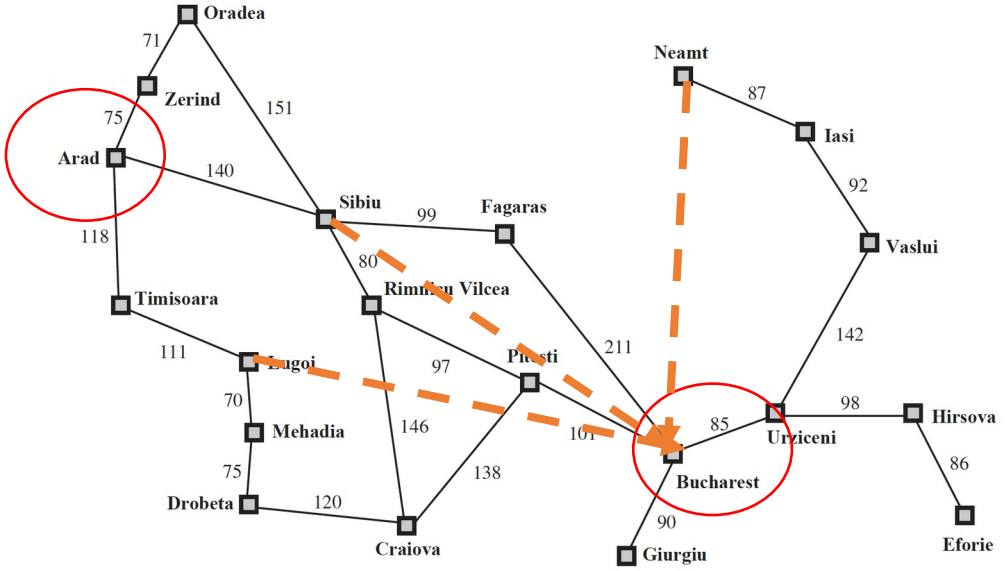


Informed Search

- Best-first search (BFS)
  - Select the node based on an **evaluation function** (cost estimate)
    - Depend on node , goal, search so far, domain
  - Identical to uniform-cost search when is the path cost
  - Usually include a heuristic function
    - estimated cost of the cheapest path from the state at node to a goal state
    - the most common form to incorporate problem-specific knowledge
    - Nonnegative, and if is a goal node
- Two popular algorithms:
  - Greedy best-first search
  - A* search

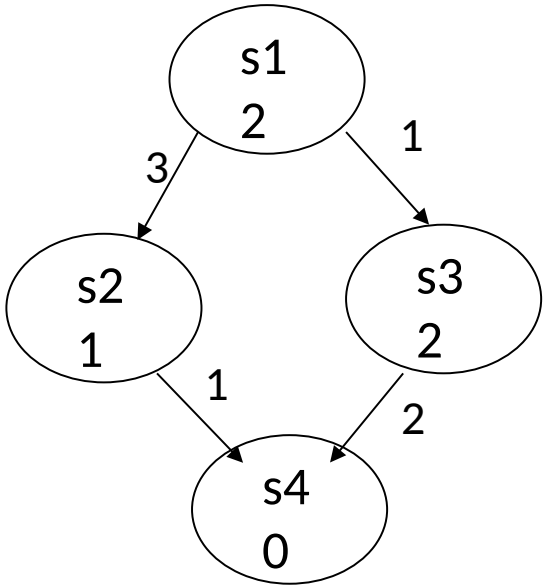
Greedy Best-first Search (Greedy BFS)

- Expand the node that appears to be closest to the goal (in some metrics)
- Ordering with
  - Estimate of cost from node *n* to the goal
- e.g., the **straight-line distance heuristic** (from node *n* to the Bucharest) in the route-finding problem



Properties of Greedy BFS

- Complete only for graph search in finite state spaces (like depth-first)
- Not optimal, and sometimes can get deceived and find really bad solutions
- Complexity depends on the problem and the quality of the heuristic
  - Worst case time/space complexity
- In practice, fast and therefore attractive to solve problems with high complexity



A* Search

- Minimize the total estimated solution cost
- Avoid expanding paths that are already expensive

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

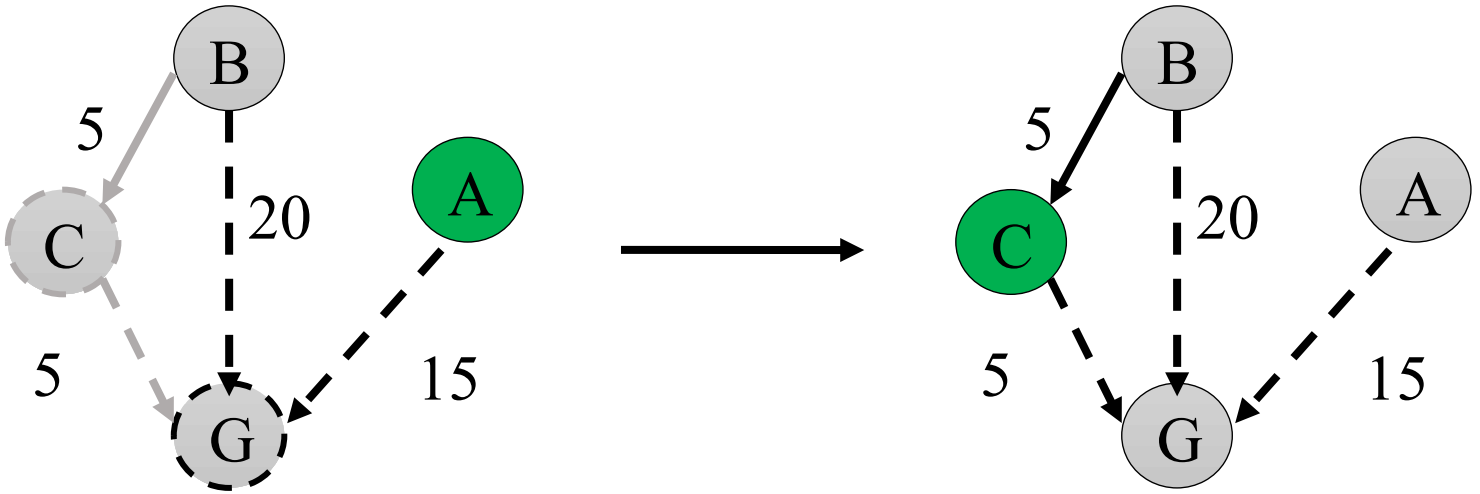
Estimated cost of cheapest path from node to goal

Path cost from the root to node

A* search is both complete and optimal given that satisfies certain conditions

Properties of A* Search

- The tree-search (graph-search) version of A* is optimal if *is admissible (consistent)*
  - Admissible: never overestimate the cost to reach the goal, $h(n) \leq h^*(n)$,
  - Consistent: triangle inequality, .

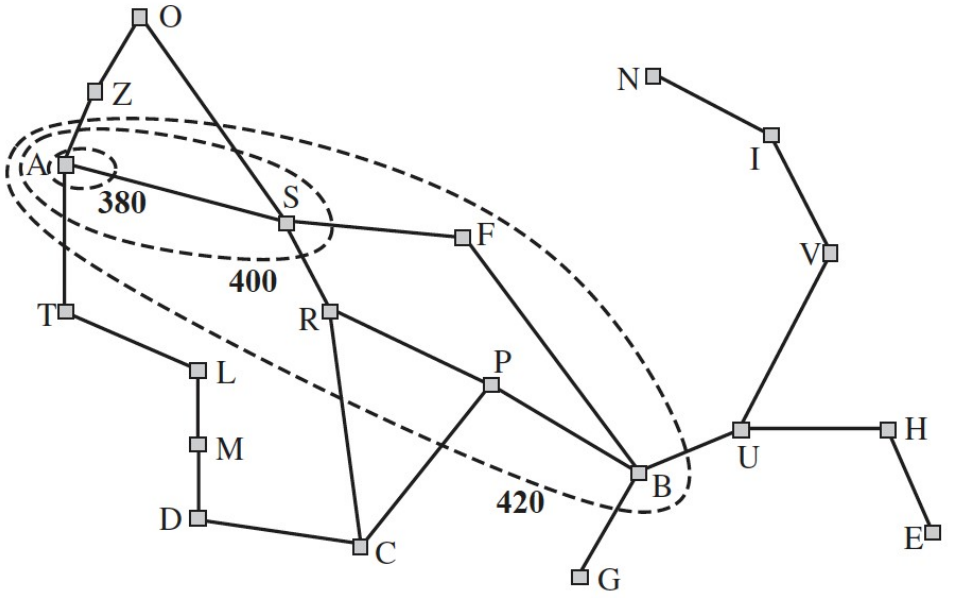


A is better than B

After discovering C, B is better, but can not be revisited, because it is in the explored set

Completeness of A* Search

- Complete for finite and step cost > some positive value
  - The fact that is non-decreasing along any path: draw contours in the state space
  - A* expands nodes in order of increasing value
  - Contour has all nodes with , where
  - Gradually adds “-contours” of nodes
  - A* expands all nodes with (cost of the optimal path), some nodes with , no nodes with



Other Properties of A* Search

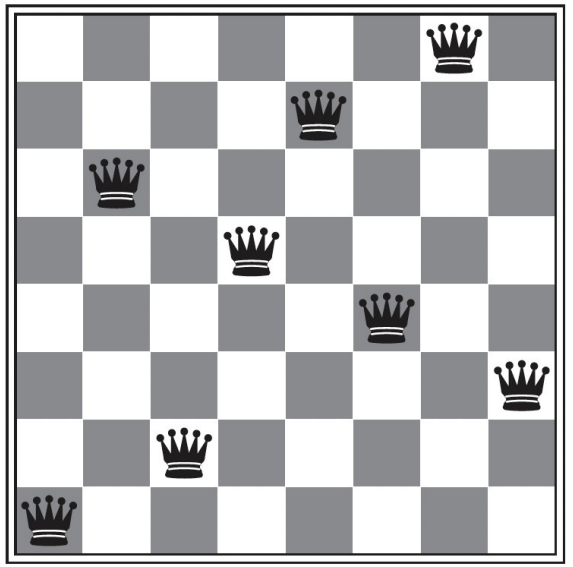
- A* is optimally efficient for any given consistent heuristics and suitable tie-breaking rules
  - No other optimal algorithms expand fewer nodes than A*

- Complexity
  - Exponential time complexity
  - Exponential space complexity

Local Search

- Previously: the path to the goal is also a solution to the problem
  - Systematic exploration of the search space

- Now: a state is a solution to the problem
  - The path to the goal is irrelevant in many problems, e.g., the 8-queens problem
  - State space = a set of "complete" configurations
  - Find a complete configuration satisfying constraints



- Local search algorithms can be used
  - Keep track of a single current node (rather than multiple paths)
  - Move only to neighbors of that node
  - Paths are not retained

Basic Idea of Local Search (many variations)

```
// initialize to something, usually a random initial state
// alternatively, might pass in a human-generated initial state
best_found ← current_state ← RandomState()
// now do local search
loop do
    if (tired of doing it) then return best_found
    else
        current_state ← MakeNeighbor( current_state )
        if ( Cost(current_state) < Cost(best_found) ) then
            // keep best result found so far
            best_found ← current_state
```

You, as algorithm designer, write the functions named in red.

Typically, "tired of doing it" means that some resource limit is exceeded, e.g., number of iterations, wall clock time, CPU time, etc.
It may also mean that result improvements are small and infrequent, e.g., less than 0.1% result improvement in the last week of run time.

Hill-climbing Search

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

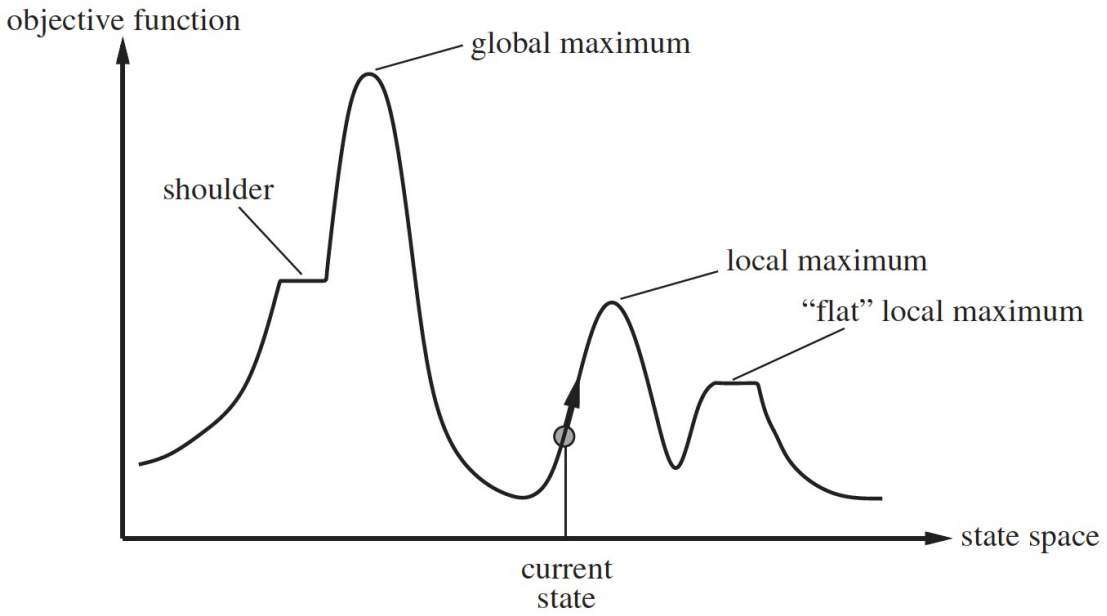
current ← MAKE-NODE(*problem*.INITIAL-STATE)
loop do
  neighbor ← a highest-valued successor of *current*
  if neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE
  current ← *neighbor*

maximize value vs minimize cost

- A loop that continuously moves in the direction of increasing value (steepest-ascent version)
  - Objective function value or heuristic function value
- Terminates when it reaches a "peak" where no neighbor has a higher value
- No search tree, only record the state and the value of the objective function
- Aka Greedy local search
  - Does not look beyond immediate neighbors of the current state

Local Search Difficulties

- Local optima
  - E.g., local maximum, a peak that is higher than each of its neighbors but lower than the global maximum
- Plateau: A flat area of the state-space landscape
  - E.g., flat local maximum, shoulder
- Ridges: a sequence of local maxima that algorithms to navigate
  - A sequence of local maxima that are not dir
  - Every neighbor appears to be downhill
  - But the search space has an uphill



How to Escape Local Optima

- **Sideways move:** if no uphill moves, allow sideways moves in hope that the algorithm can escape
  - Need to place a limit on the possible number of sideways moves to avoid infinite loops

- **Stochastic (randomized) hill climbing:**
  - Iterate the process of randomly selecting a neighbor for a candidate solution
  - Accept it only if it results in an improvement
    - Can generate a set of neighbors for the current state and pick the best one among the selected neighbors

- **Random restart:**
  - If at first you don't succeed, try, try again
  - Launch multiple hill-climbing searches from randomly generated initial states

Simulated Annealing

Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(Δ E/T)
```

Instead of picking the best move, pick one randomly

: change in objective function

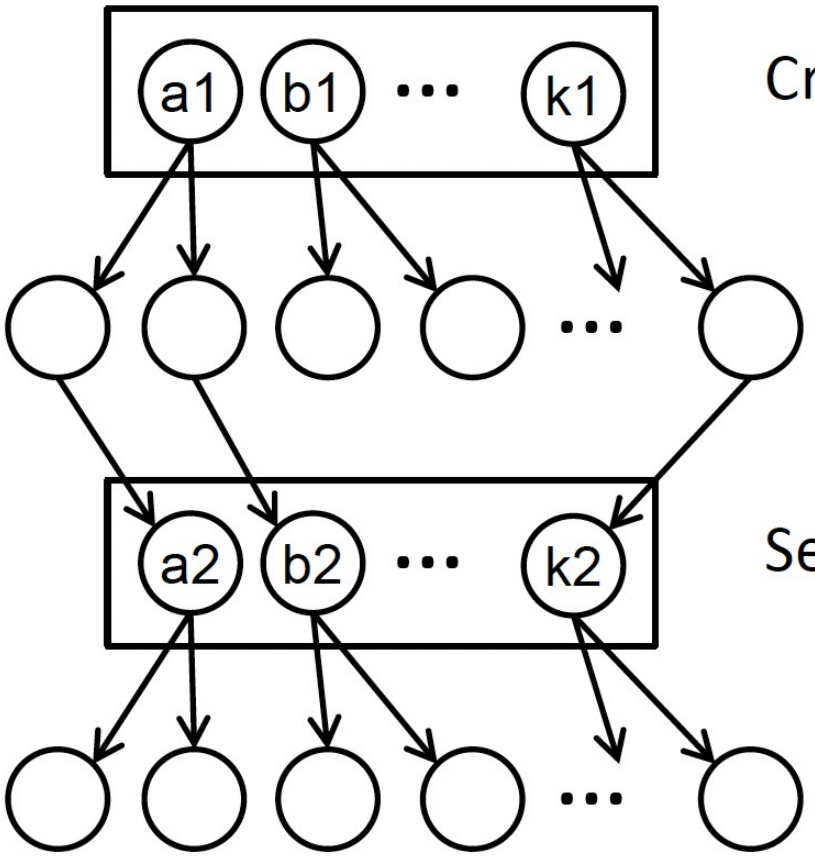
If is positive, move to that state (like hill-climbing)

Otherwise, move to this state with probability proportional to

Gradually decrease

Local Beam Search

- Idea: keeping only one node in memory is an extreme reaction to memory problems



Create *k* random initial states

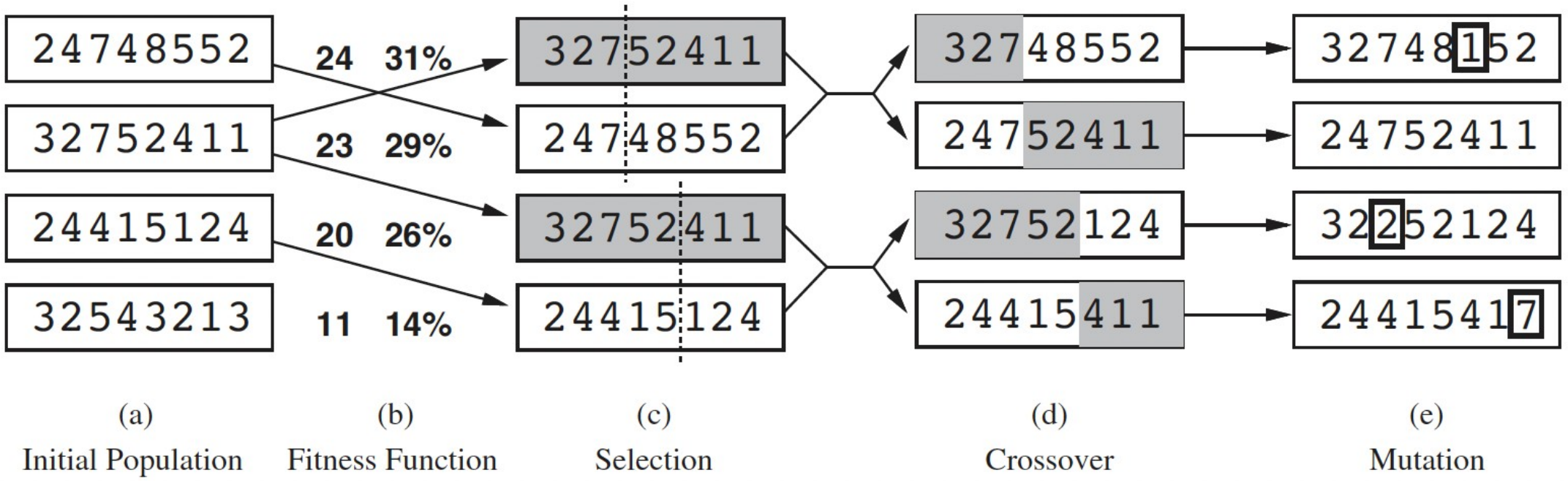
Generate their children

Select the *k* best children

Repeat indefinitely…

Genetic Algorithms

- Produce the next generation of states by "simulated evolution"
  - Crossover has the effect of "jumping" to a completely different new part of the search space (quite non-local)

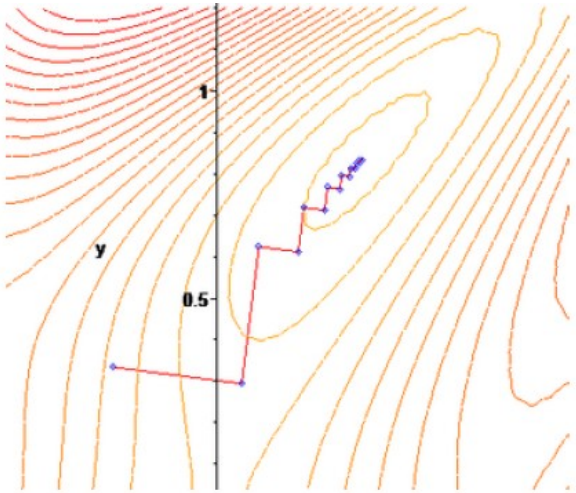


Local Search in Continuous Space

- Gradient descent:
  - Assume we have a continuous cost function
  - We want to minimize it over continuous variables

1. Compute the gradient with respect to every :
2. Take a small step downhill in the opposite direction of gradient with step size :

3. Repeat



Local Search Summary

- Hill climbing is a steady monotonous ascent to better nodes
- Stochastic hill climbing, simulated annealing, local beam search, and genetic algorithms are “random” searches with a bias towards better nodes
- Gradient descent works well for continuous space
- All need very little memory
- None guarantees to find the globally optimal solution

Constraint Satisfaction Problems

- Regular search problems:
  - Search in a state space
  - Each state is indivisible --- a black box with no internal structure

- Constraint satisfaction problem
  - Each state has a factored representation: a set of variables, each has a value
  - Goal condition consists of a set of sub-conditions: each variable should have a value that satisfies all the constraints on the variable

What is a Constraint Satisfaction Problem (CSP)

- A CSP consists of three components:
  - A finite set of variables:
  - A finite set of domains:
    - One for each variable; finite or infinite domain
  - A finite set of constraints:
    - Each constraint involves some subset of the variables
    - <scope, relation>
    - Specifies allowable combinations of values for that subset
    - E.g., <(,), >
- **State space:** each state is defined by an *assignment* of values to some or all of the variables, e.g.,
- **Goal:** find a complete and consistent assignment
  - Find an assignment of the variables from their domains such that none of the constraints are violated

Backtracking

- Basic uninformed search algorithm to solve CSPs
- Similar to depth-first search, exploring one node at a time
- Chooses values for one variable at a time
- Backtracks when a variable has no legal values left to assign

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

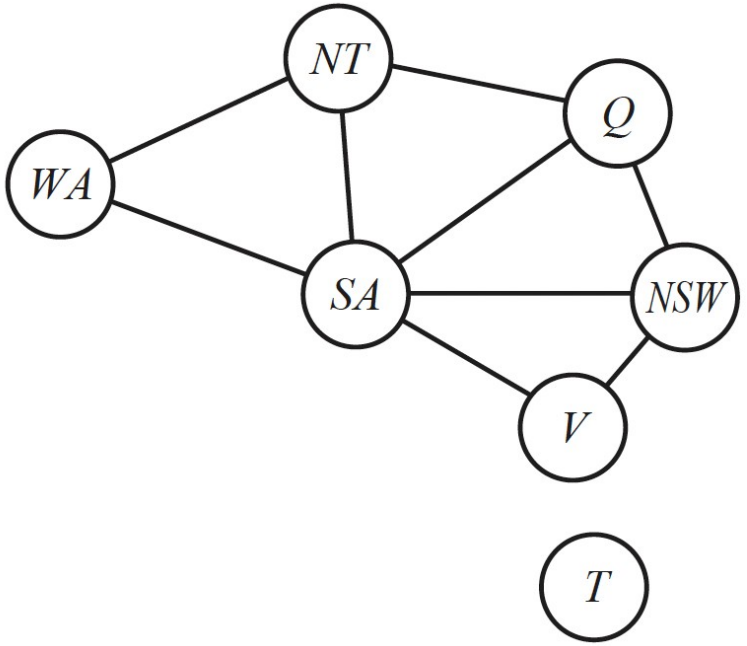
Improving Backtracking

- Which variable should be assigned next?
  - Minimum remaining values (MRV): Choose the variable with the fewest legal left values in its domain
  - Degree heuristics: select variable that is involved in the largest number of constraints on other unassigned variables
- In what order should its values be tried?
  - Least constraining value: Prefer the value that rules out the fewest choices for the neighboring variables in the constraint graph
- Can we detect inevitable failure early?
  - Forward checking: Keep track of remaining legal values for unassigned variables and cross off bad options
  - Constraint propagation: Using constraints to reduce the domain for a variable, which in turn can reduce the domain for another variable, and so on…

Improving Backtracking

- Which variable should be assigned next?
  - Minimum remaining values (MRV): Choose the variable with the fewest legal left values in its domain
  - Degree heuristics: select variable that is involved in the largest number of constraints on other unassigned variables
- In what order should its values be tried?
  - Least constraining value: Prefer the value that rules out the fewest choices for the neighboring variables in the constraint graph
- Can we detect inevitable failure early?
  - Forward checking: Keep track of remaining legal values for unassigned variables and cross off bad options
  - Constraint propagation: Using constraints to reduce the domain for a variable, which in turn can reduce the domain for another variable, and so on…

Variable Ordering: Minimum Remaining Values (MRV)

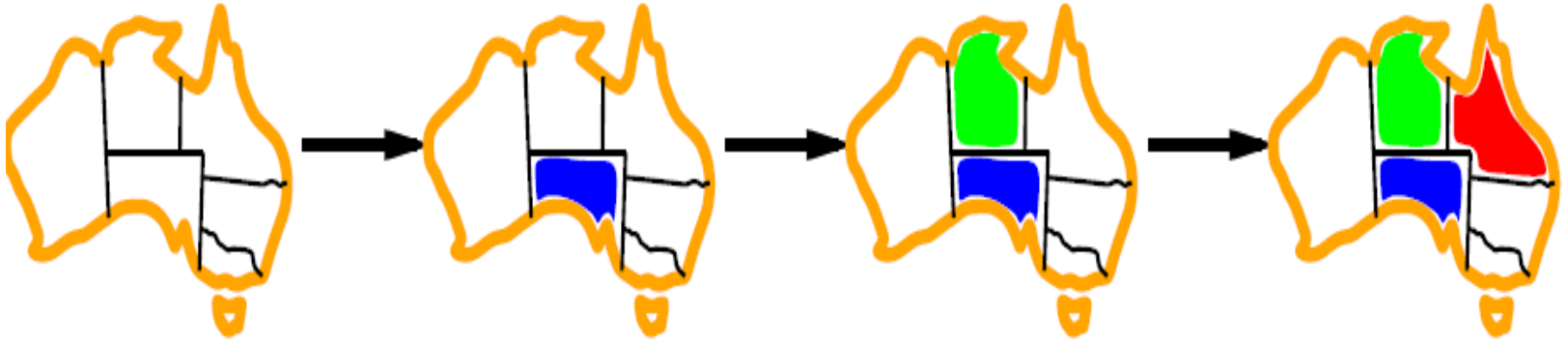


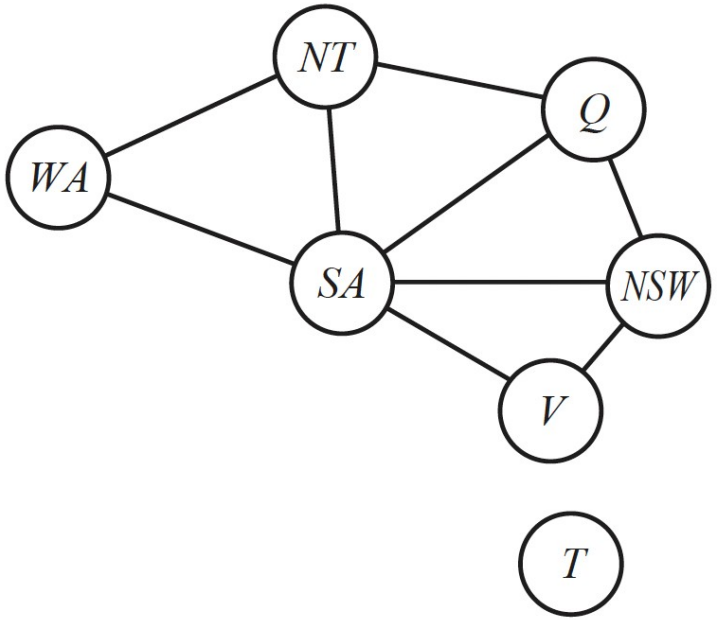
- Aka “most constrained variable” or “fail-first”:
  - Choose the variable with the fewest legal left values in its domain
  - If some variable X has no legal values left, select X and failure will be detected immediately --- avoiding pointless searches through other variables

Improving Backtracking

- Which variable should be assigned next?
  - Minimum remaining values (MRV): Choose the variable with the fewest legal left values in its domain
  - Degree heuristics: select variable that is involved in the largest number of constraints on other unassigned variables
- In what order should its values be tried?
  - Least constraining value: Prefer the value that rules out the fewest choices for the neighboring variables in the constraint graph
- Can we detect inevitable failure early?
  - Forward checking: Keep track of remaining legal values for unassigned variables and cross off bad options
  - Constraint propagation: Using constraints to reduce the domain for a variable, which in turn can reduce the domain for another variable, and so on…

Degree Heuristic



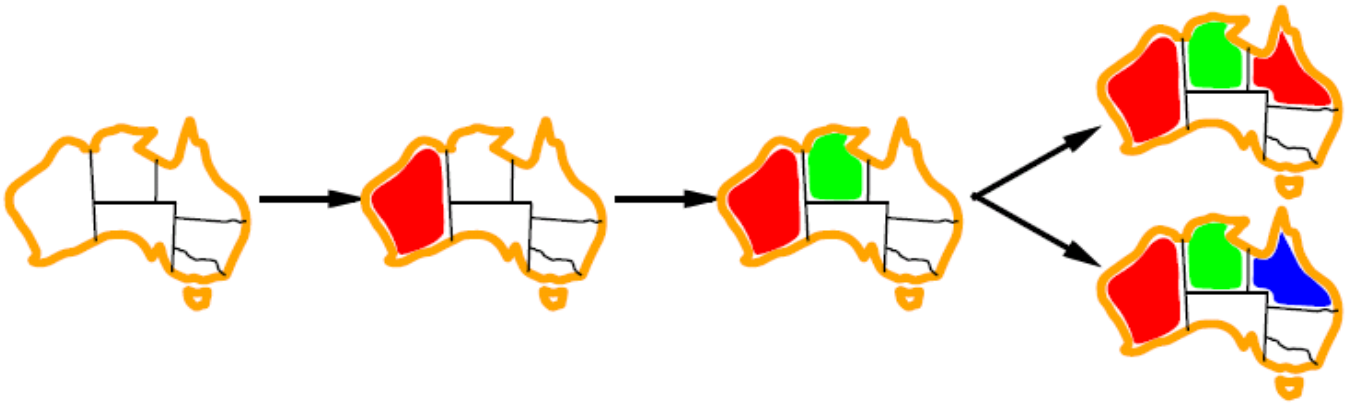


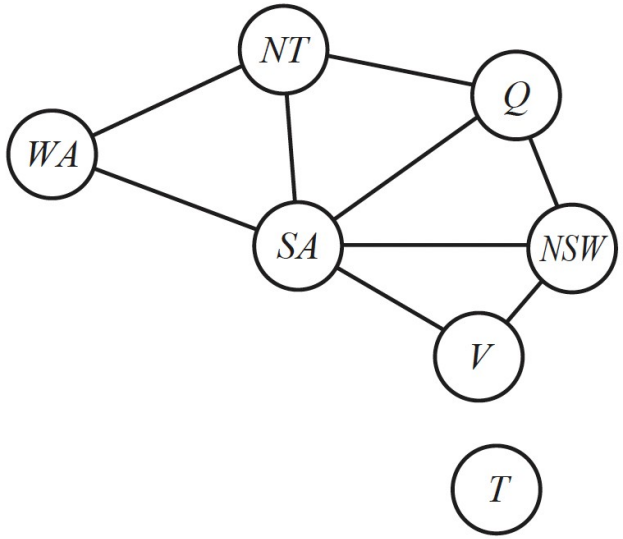
- MRV doesn't help in choosing the first region to color
- Degree heuristic: select variable that is involved in the largest number of constraints on other unassigned variables
- MRV is usually a more powerful guide, but degree heuristic can be useful as a tie breaker

Improving Backtracking

- Which variable should be assigned next?
  - Minimum remaining values (MRV): Choose the variable with the fewest legal left values in its domain
  - Degree heuristics: select variable that is involved in the largest number of constraints on other unassigned variables
- In what order should its values be tried?
  - Least constraining value: Prefer the value that rules out the fewest choices for the neighboring variables in the constraint graph
- Can we detect inevitable failure early?
  - Forward checking: Keep track of remaining legal values for unassigned variables and cross off bad options
  - Constraint propagation: Using constraints to reduce the domain for a variable, which in turn can reduce the domain for another variable, and so on…

Value Ordering: Least Constraining Value



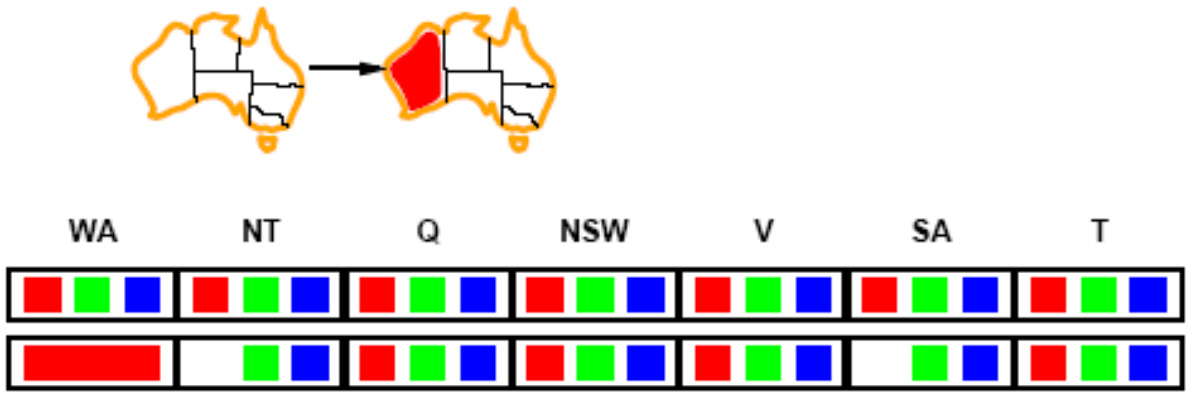


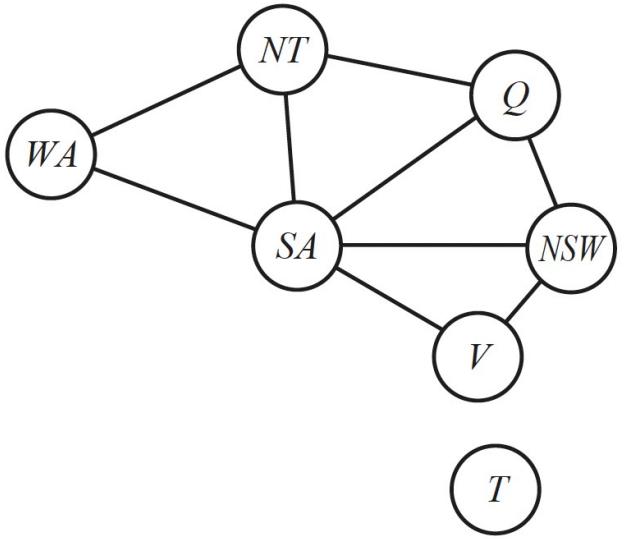
- Given a variable, choose the least constraining value
  - Prefer the value that rules out the fewest choices for the neighboring variables in the constraint graph
  - Leave the maximum flexibility for subsequent variable assignments
- Why least rather than most?
  - Only need one solution, look for the most likely values first

Improving Backtracking

- Which variable should be assigned next?
  - Minimum remaining values (MRV): Choose the variable with the fewest legal left values in its domain
  - Degree heuristics: select variable that is involved in the largest number of constraints on other unassigned variables
- In what order should its values be tried?
  - Least constraining value: Prefer the value that rules out the fewest choices for the neighboring variables in the constraint graph
- Can we detect inevitable failure early?
  - Forward checking: Keep track of remaining legal values for unassigned variables and cross off bad options
  - Constraint propagation: Using constraints to reduce the domain for a variable, which in turn can reduce the domain for another variable, and so on…

Filtering: Forward Checking





- Assign {WA=red}
- Effects on other variables connected by constraints with WA
  - NT can no longer be red
  - SA can no longer be red

Improving Backtracking

- Which variable should be assigned next?
  - Minimum remaining values (MRV): Choose the variable with the fewest legal left values in its domain
  - Degree heuristics: select variable that is involved in the largest number of constraints on other unassigned variables
- In what order should its values be tried?
  - Least constraining value: Prefer the value that rules out the fewest choices for the neighboring variables in the constraint graph
- Can we detect inevitable failure early?
  - Forward checking: Keep track of remaining legal values for unassigned variables and cross off bad options
  - Constraint propagation: Using constraints to reduce the domain for a variable, which in turn can reduce the domain for another variable, and so on…

Constraint Propagation

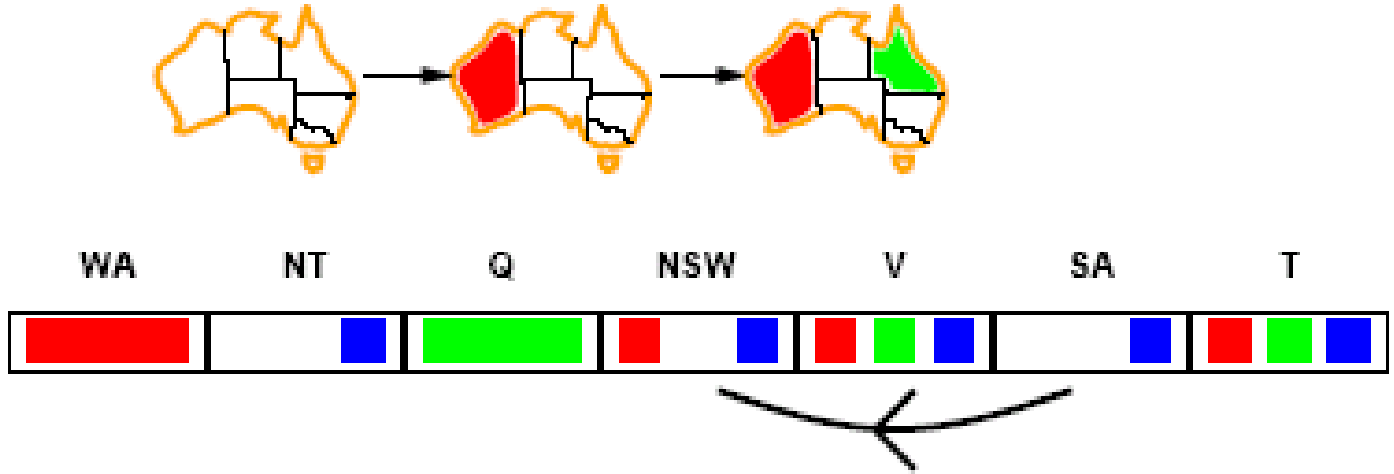
- Constraint propagation: a specific type of inference
  - Using constraints to reduce the domain for a variable, which in turn can reduce the domain for another variable, and so on…
  - Reason from constraint to constraint

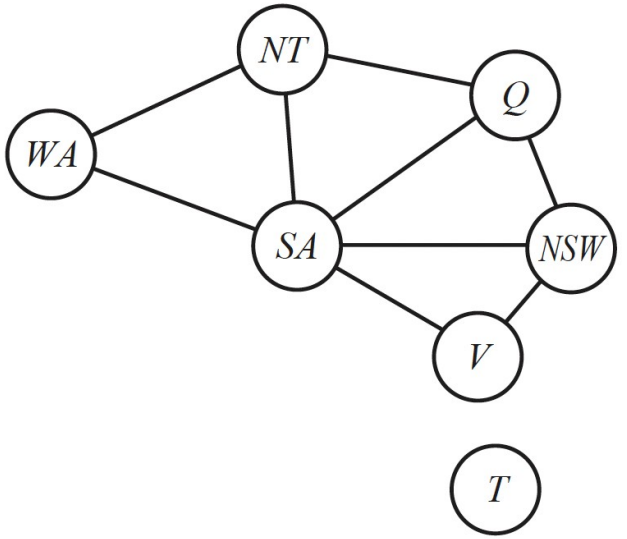
- Techniques like constraint propagation and forward checking are in effect eliminating parts of the search space
  - Somewhat complementary to search

- Constraint propagation goes further than FC by repeatedly enforcing constraints locally

- Arc-consistency (AC) is a systematic procedure for constraint propagation

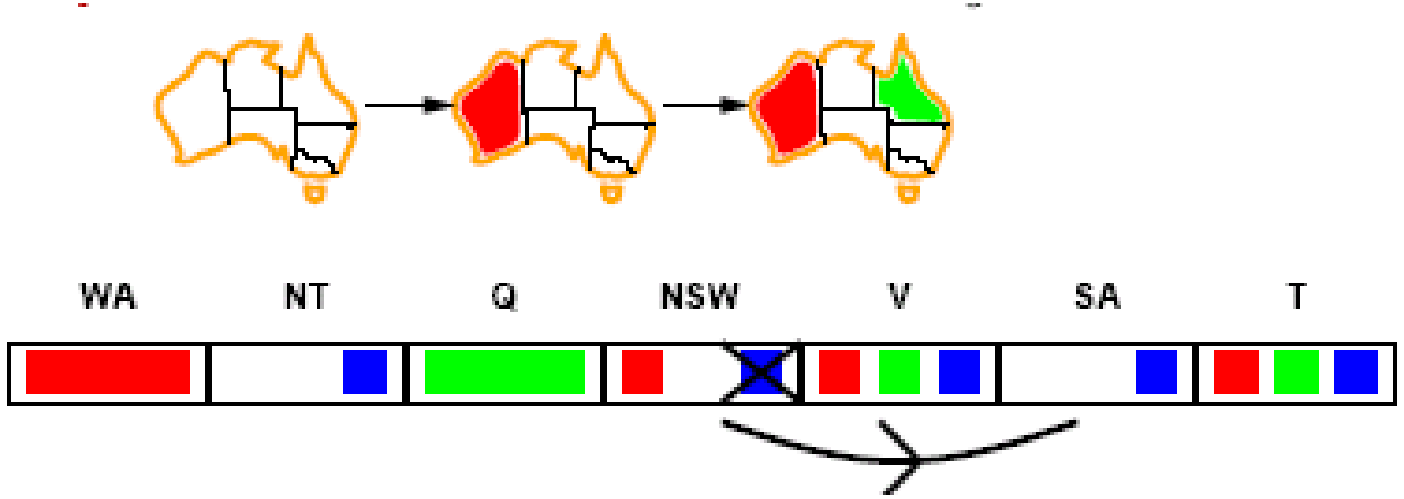
Arc Consistency

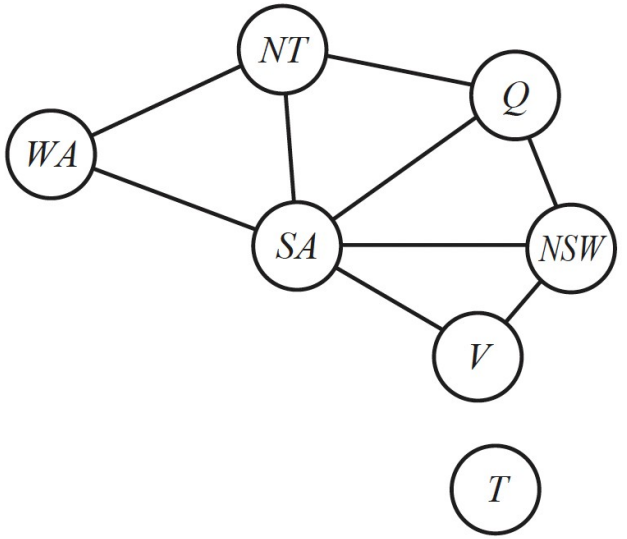




- An arc is consistent if for every value x of X there is some value y of Y consistent with x (without violating the constraint)
- Consider state of search after WA and Q are assigned
- SA NSW is consistent if
  - SA=blue and NSW=red

Arc Consistency





- NSW SA is consistent if
  - NSW=red and SA=blue
  - NSW=blue and SA=???
- Enforcing arc consistency removes blue from NSW

Local Search for CSPs

- Use complete-state representation
  - Initial state: assign a value to every variable
  - Successor function: change the value of one variable at a time
- Allow states with unsatisfied constraints (unlike backtracking)
- The point of local search is to eliminate the violated constraints
- Variable selection: randomly select any conflicted variable
- Value selection: min-conflicts heuristic
  - Select new value that results in the minimum number of conflicts with other variables

Local Search for CSPs

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
      if current is a solution for csp then return current
      var ← a randomly chosen conflicted variable from csp.VARIABLES
      value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
      set var = value in current
  return failure
```

- The runtime of min-conflicts is roughly independent of problem size
- Local search can be particularly useful in an online setting

Info about Midterm Exam (Oct. 9, Monday, 1-2:15pm)

- 75 minutes; open-book, open-note, electronic devices are not allowed
- Relevant topics:
  - Intro to AI
  - Search:
    - Basics, Uninformed and Informed Search, Local Search, CSP
  - Machine Learning:
    - Linear models

- Relevant material from the Russel textbook (Fourth Edition)
  - Chapter 3: page 82-124; chapter 4: page 128-140; chapter 5: page 164-188
  - Chapter 19: page 669-674, 694-697, 700-704