

COMP7015 Artificial Intelligence (S1, 2024-25)

Lecture 2: Solving Problems with Searching II

Instructor: Dr. Kejing Yin (cskjyin@hkbu.edu.hk)

Department of Computer Science
Hong Kong Baptist University

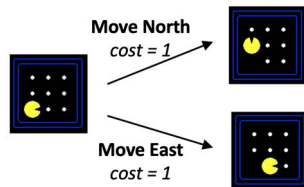
September 13, 2024

Recap: Elements of A Search Problem

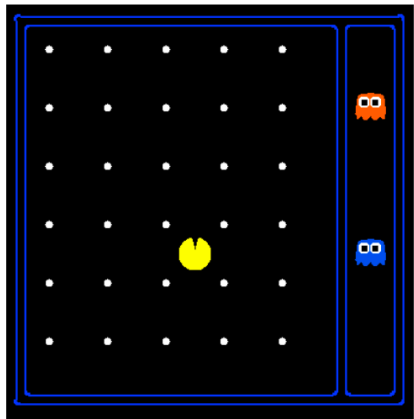
- A **state space**: a set of all possible configurations of the world.



- The **initial state**: the state that the agent starts in.
 - A set of **goal states**: can be one or many. (Need a *goal test*)
- A **successor function**: returns the **actions** available and the corresponding **costs**.
- A **solution**: a sequence of actions that starts from the initial state and reaches a goal state.



Recap: Level of Abstraction and Size of State Spaces



Example from CS188@UCB

- A 10×12 grid world: 120 possible agent positions
- Food count: 30
- Ghost position: 12
- Agent facing: N, S, E, W



How many world states in total?

$$120 \times 2^{30} \times 12^2 \times 4$$



State space size for path finding?

$$120$$



State space size for eating all dots?

$$120 \times 2^{30}$$

Recap: General Search Algorithms

- When an algorithm checks redundant paths, we call it “graph search”
- When we do not check redundant paths, it’s “tree-like search”

Graph Search

```

1 frontierList ← [initialState];
2 visitedList ← [];
3 while frontierList is not empty do
4     select a node  $S$  according to the search strategy;
5     if node  $S$  is a goal state then
6         return the corresponding solution;
7     else
8         expand node  $S$ ;
9         add children (not already in frontierList and
            visitedList) to frontierList;
10        add node  $S$  to visitedList;
11    end if
12 end
13 return failure;
  
```

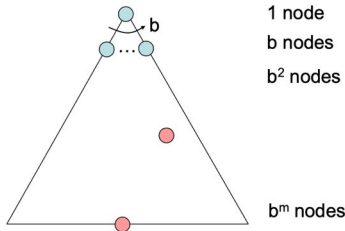
Tree-Like Search

```

1 frontierList ← [initialState];
2 while frontierList is not empty do
3     select a node  $S$  according to the search strategy;
4     if node  $S$  is a goal state then
5         return the corresponding solution;
6     else
7         expand node  $S$ ;
8         add all children to frontierList;
9     end if
10 end
11 return failure;
  
```

Properties of Search Algorithms (Performance Measure)

- **Completeness:** Is the algorithm **guaranteed to find a solution** if one exists?
- **Cost optimality:** Is the algorithm guaranteed to find a **solution with lowest cost**?
- **Time complexity:** How much time it takes? (“Big- O ” notation)
- **Space complexity:** How much memory is needed? (“Big- O ” notation)



- b : average number of children (*branching factor*)
- m : maximum depth
- Solutions at various depths; worst case at depth of m

🤔 How many nodes in the entire tree?

$1 + b + b^2 + \dots + b^m = O(b^m)$ \triangleleft “Big- O ”: how it grows as b and m grows.

Road Map of Search Algorithms

- Uninformed search (blind search; no clue about how close a state is to the goal state)
 - Breadth-first Search (BFS)
 - Uniform-cost Search (UCS)
 - Depth-first Search (DFS)
 - Depth-limited Search and Iterative deepening Search
- Informed search (heuristic search; we have some hints about the location of goals)
 - Greedy Search
 - A* Search

Outline for Today

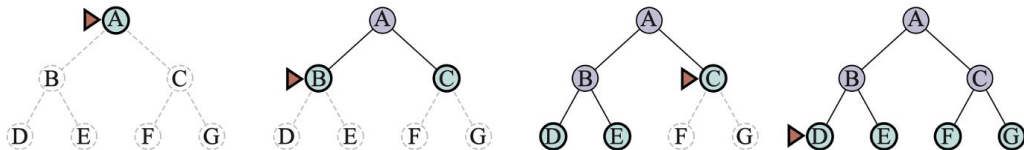
1. Uninformed Search Algorithms
2. Informed Search Algorithms
3. Constraint Satisfaction Problems

Uninformed Search Algorithms

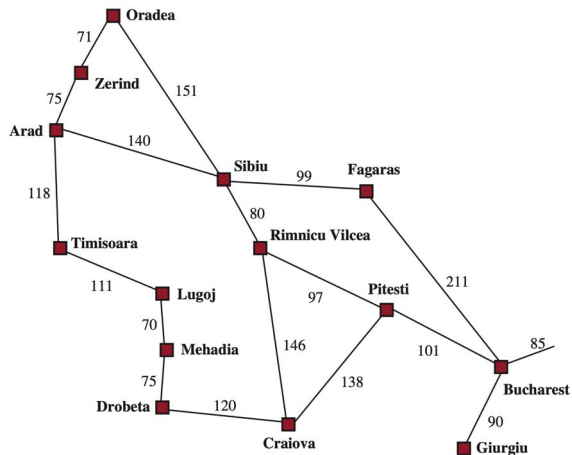
- Breadth-First Search
- Uniform-Cost Search
- Depth-First Search

Breadth-First Search (BFS)

- **Search strategy:** Expand a *shallowest* node first.
i.e., always expanding the nodes at depth $d - 1$ before expanding nodes at depth d .
- Often implemented as graph search (check redundant paths).



Example: Use BFS to Find A Route From Arad to Fagaras



Step 1: Initialization

Arad

```
frontiers = [Arad]
visited = []
```

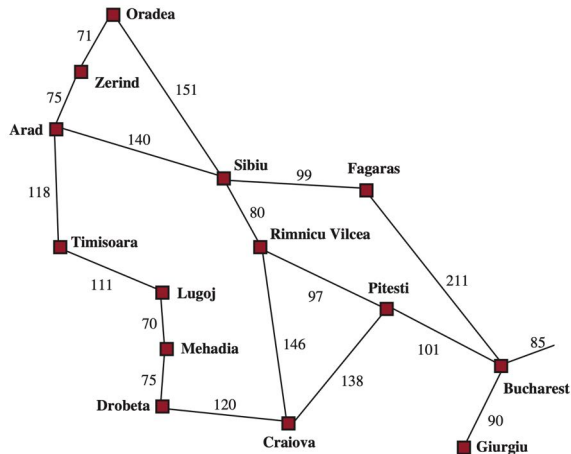
Step 2: Expand Arad

Arad

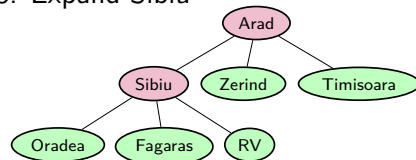
Sibiu Zerind Timisoara

```
frontiers = [Sibiu, Zerind, Timisoara]
visited = [Arad]
```

Example: Use BFS to Find A Route From Arad to Fagaras



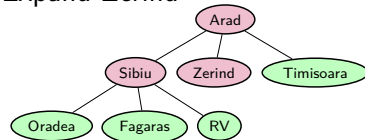
Step 3: Expand Sibiu



frontiers = [Zerind, Timisoara, Oradea, Fagaras, RV]

visited = [Arad, Sibiu]

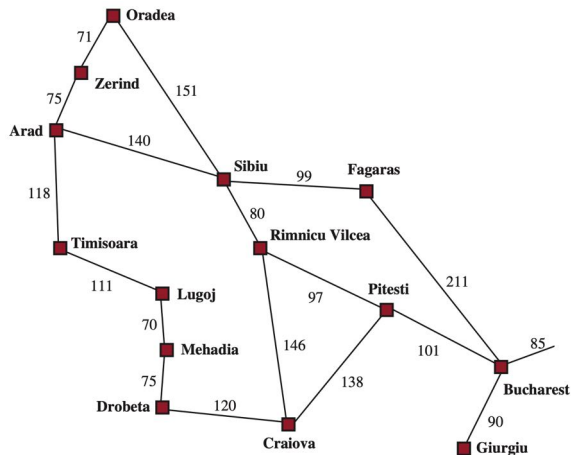
Step 4: Expand Zerind



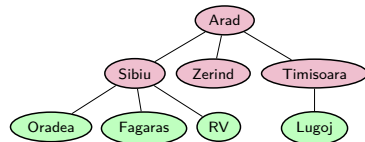
frontiers = [Timisoara, Oradea, Fagaras, RV]

visited = [Arad, Sibiu, Zerind]

Example: Use BFS to Find A Route From Arad to Fagaras

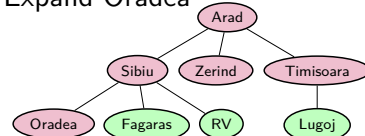


Step 5: Expand Timisoara



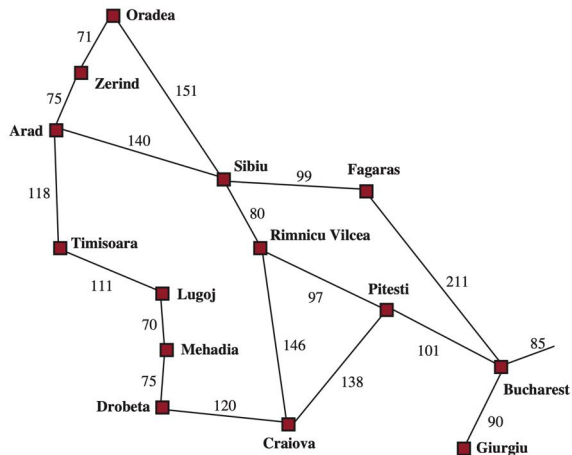
frontiers = [Oradea, Fagaras, RV, Lugoj]
 visited = [Arad, Sibiu, Zerind, Timisoara]

Step 6: Expand Oradea

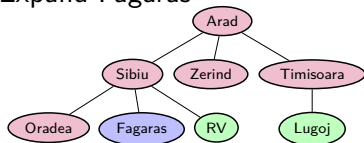


frontiers = [Fagaras, RV, Lugoj]
 visited = [Arad, Sibiu, Zerind, Timisoara, Oradea]

Example: Use BFS to Find A Route From Arad to Fagaras



Step 7: Expand Fagaras



Solution: Arad → Sibiu → Fagaras

Properties of Breadth-First Search

Suppose the depth of the shallowest solution is s .

- **Nodes expanded?** All nodes above the shallowest solution.
- **Is BFS Complete?** Yes, s is finite if a solution exists.

- **Is BFS Cost-Optimal?**

BFS finds the *shallowest* solution.

Cost-optimal only if all actions cost the same.

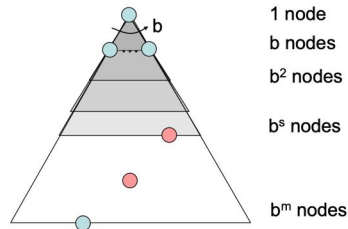
- **Time complexity?** Takes time $O(b^s)$

- **Space needed?**

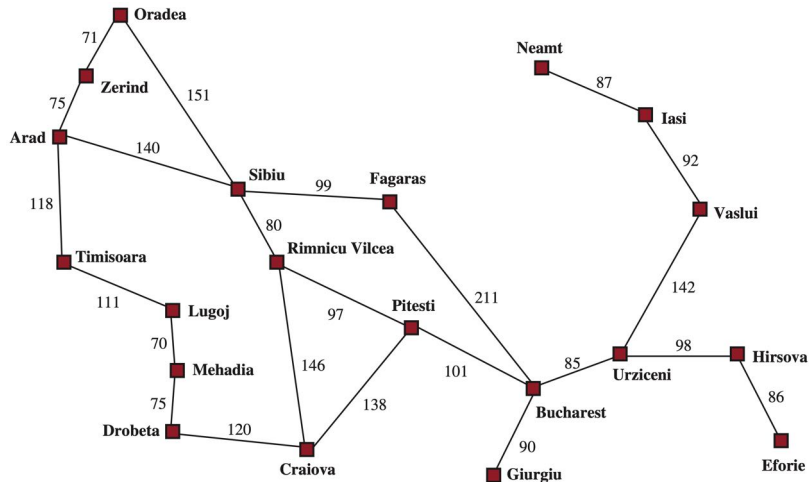
visitedList: roughly $1 + b + b^2 + \dots + b^{s-1}$

frontierList: roughly b^s

Total: $O(b^s)$

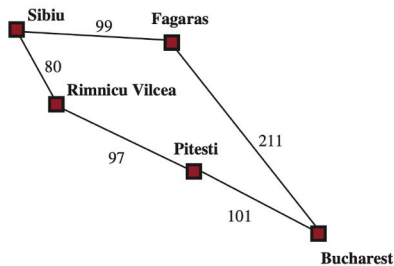


Exercise: Use BFS to Find A Route From Arad to Bucharest



Uniform-Cost Search (UCS)

- Consider the cost: also store the cost from root to nodes in frontierList.
- Search strategy*: Expand the node with the smallest cost from root to the node.



Route-finding from Sibiu to Bucharest

Numbers are distances

1. Expand Sibiu:

`frontiers=[Fagaras(99), RV(80)]` `visited=[Sibiu]`

2. Expand Fagaras or RV? (RV has the smallest cost)

`frontiers=[Fagaras(99), Pitesti(177)]` `visited=[Sibiu, RV]`

3. Expand Fagaras or Pitesti? (Fagaras now has the smallest cost)

`frontiers=[Pitesti(177), Bucharest(310)]`
`visited=[Sibiu, RV, Fagaras]`

4. Expand Pitesti or Bucharest? (Pitesti has the smallest cost)

`frontiers=[Bucharest(310 278)]`
`visited=[Sibiu, RV, Fagaras, Pitesti]`

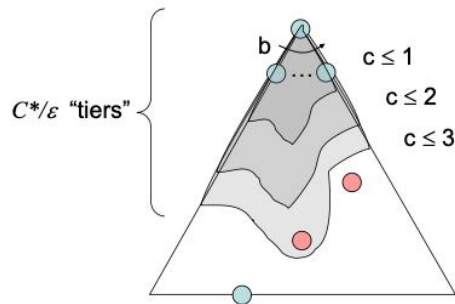
5. Expand Bucharest: *goal state reached*

Solution: Sibiu → RV → Pitesti → Bucharest (cost=278)

Properties of Uniform-Cost Search

Suppose the solution costs C^* and the lower bound of the cost is $\epsilon > 0$.

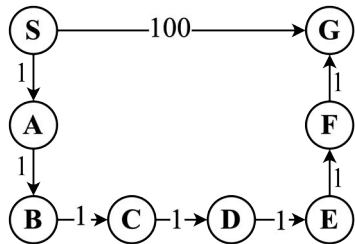
- **Nodes expanded?**
All nodes with cost less than the optimal solution.
- **Is UCS Complete?** Yes with finite cost.
- **Is UCS Cost-Optimal?** Yes!
- **Worse-Case Time complexity?** $O(b^{C^*/\epsilon})$
- **Worse-Case Space needed?** $O(b^{C^*/\epsilon})$



Complexity of BFS and UCS

- C^* : cost of the optimal solution
- $\varepsilon > 0$: lower bound on the cost of each action
- b : average number of descendants of each node
- d : depth of the goal (for BFS)

An extreme example (from S to G):

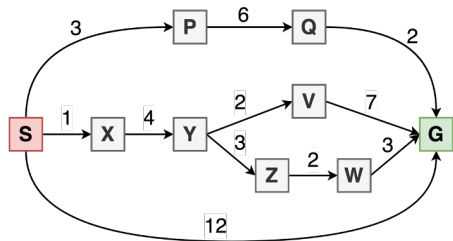


- $C^* = 7$, $\varepsilon = 1$, $b \approx 1$, $d = 1$ (for BFS)
- How many nodes expanded for BFS? **1**
- How many nodes expanded for UCS? **8**

BFS vs UCS: price of optimality

The worst-case time and space complexity of uniform-cost search could be much greater than BFS.

Exercise: BFS and UCS



Path finding from S to G

- What is the solution using BFS?
 - $S \rightarrow X \rightarrow Y \rightarrow V \rightarrow G$
 - $S \rightarrow X \rightarrow Y \rightarrow Z \rightarrow W \rightarrow G$
 - $S \rightarrow G$
 - $S \rightarrow P \rightarrow Q \rightarrow G$
- Is BFS cost-optimal in this example?
- What is the solution using uniform-cost search (UCS)?
 - $S \rightarrow X \rightarrow Y \rightarrow V \rightarrow G$
 - $S \rightarrow X \rightarrow Y \rightarrow Z \rightarrow W \rightarrow G$
 - $S \rightarrow G$
 - $S \rightarrow P \rightarrow Q \rightarrow G$
- Is uniform-cost search (UCS) cost-optimal?

Depth-First Search (DFS)

- Search strategy:
Expand a *deepest* node first.
- Often implemented as **tree-like search**
(does not record visited nodes).



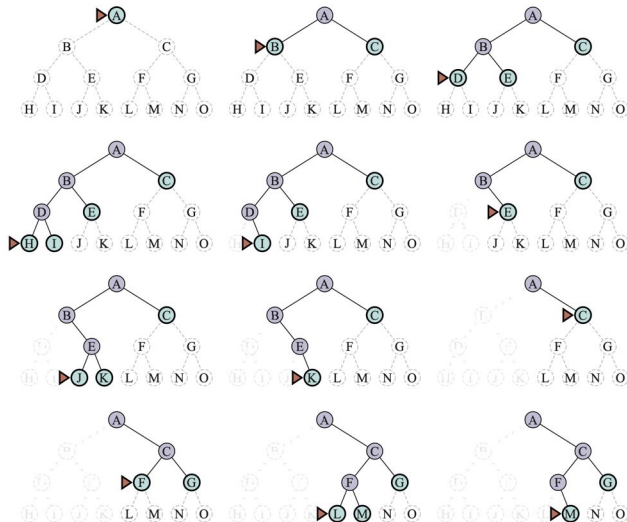
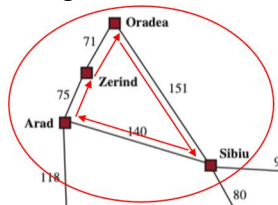
Memory efficient.



Expand some nodes multiple times.

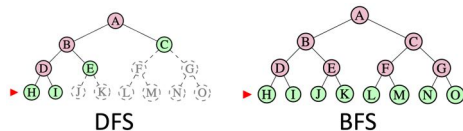
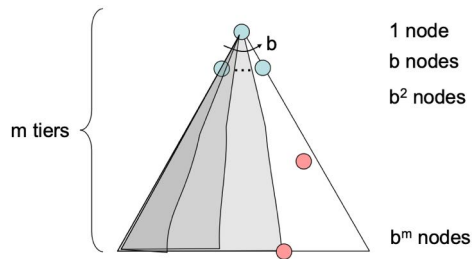


Can get stuck in infinite loops.



Properties of Depth-First Search

- **Nodes expanded?** Some left prefix of the tree.
Worst case: the whole tree
- **Is DFS Complete?** No! It can get stuck.
- **Is DFS Cost-Optimal?:** No, it returns the first solution it finds, be it optimal or not.
- **Time complexity?** $O(b^m)$
- **Space needed?**
frontiers: $b + b + \dots + b = O(bm)$
visited: 0 (not stored)

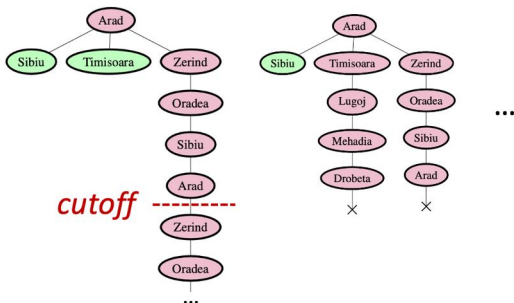


Depth-Limited Search

🤔 Can we avoid getting stuck in wrong path or infinite loops while same memory?

A simple idea: Set a depth limit l and ignore all nodes exceeding this depth.

E.g., if we set $l = 4$:



- If the goal is beyond depth l , we cannot find it.

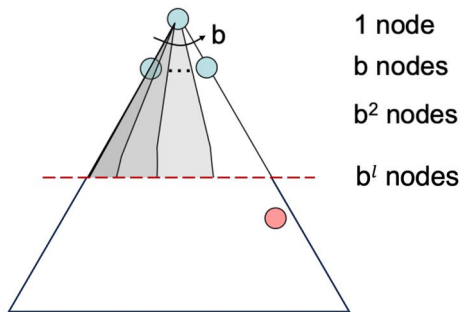
- How to set a good l ?

We need **knowledge** about the problem.

- There are only 20 cities in the Romania map:
 $l = 19$?
- A closer look at the map: from any city, we can reach any other city in at most 9 actions.
 $l = 9$?

Properties of Depth-Limited Search

- **Nodes expanded?** Some left prefix above depth l .
- **Is Depth-Limited Search Complete?**
No! Cannot find solution beyond depth l .
- **Is Depth-Limited Search Cost-Optimal?:**
No, it returns the first solution it finds.
- **Time complexity?** $O(b^l)$
- **Space needed?**
frontiers: $b + b + \dots + b = O(bl)$
visited: 0 (not stored)



Iterative Deepening Search

🤔 What if we cannot know a good l ?

Idea: We can try different values of l (from 0 up to infinity) iteratively.

- Completeness?**

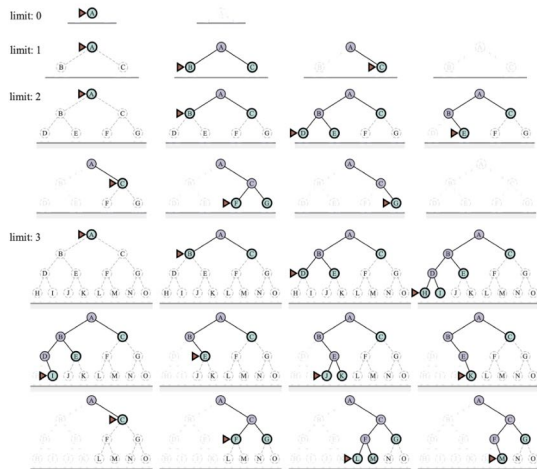
Yes, guaranteed to find a solution if one exists.

- Cost-optimality?** Finds the shallowest solution, cost-optimal if all costs are identical.

- Space complexity?** $O(bd)$ (d : depth of the shallowest solution).

- Time complexity?**

- Repeating upper levels takes more time.
- But upper levels do not have much nodes.
- Complexity: $O(b^d)$



Exercise: Comparing Uninformed Search Algorithms

Try to summarize and complete the following table:

Property	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?					
Cost-optimal?					
Time					
Space					

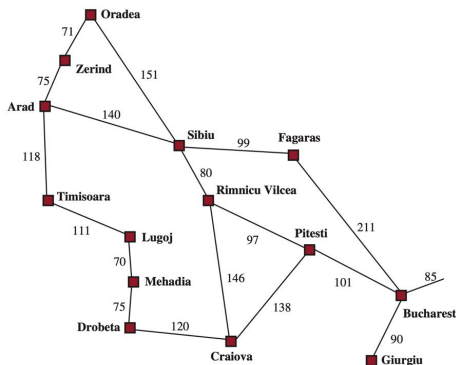
Informed Search Algorithms

- Heuristic Functions
- Greedy Search
- A* Search
- Optimality of A* Search
- Effectiveness of Heuristic Functions

Informed (Heuristic) Search

Sometimes, we have some domain-specific hints about the location of the goal.

For example, route-finding from Arad to Bucharest:



We do not know the path,
but we know the straight-line distance on the map

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	Vaslui	199
Lugoj	244	Zerind	374

e.g., straight-line distance from each city to Bucharest

Informed (Heuristic) Search

Can we use such *domain-specific hints* to make searching faster?

Heuristic Function

A heuristic function $h(n)$ of a state n is defined as the estimated cost of the *cheapest* path from the state n to a goal state.

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	Vaslui	199
Lugoj	244	Zerind	374

$$h_{\text{SLD}}(\text{Arad}) = 366$$

$$h_{\text{SLD}}(\text{Sibiu}) = 253$$

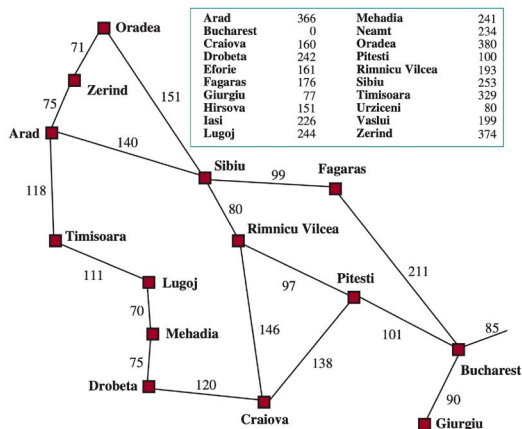
$$h_{\text{SLD}}(\text{Oradea}) = 380$$

e.g., straight-line distance from each city to Bucharest

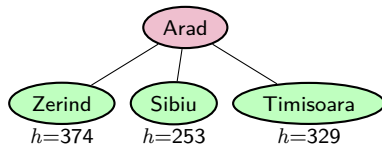
$h(n)$: an estimation of “how close” the state n is to a goal state.

Greedy Search

- Search strategy:** Expand the node with lowest $h(n)$ value, *i.e., expanding the nodes closer to the goal state first.*



Step 1: Expand Arad

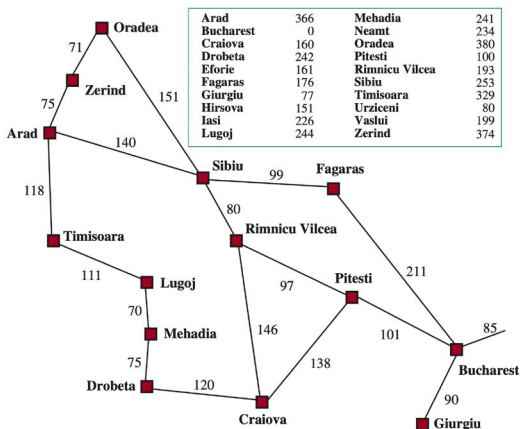


frontiers = [Zerind(374), Sibiu(253), Timisoara(329)]

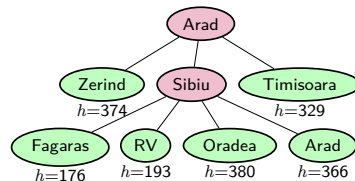
Which node to expand next? **Sibiu**

Greedy Search

- Search strategy:** Expand the node with lowest $h(n)$ value, i.e., expanding the nodes closer to the goal state first.



Step 2: Expand Sibiu

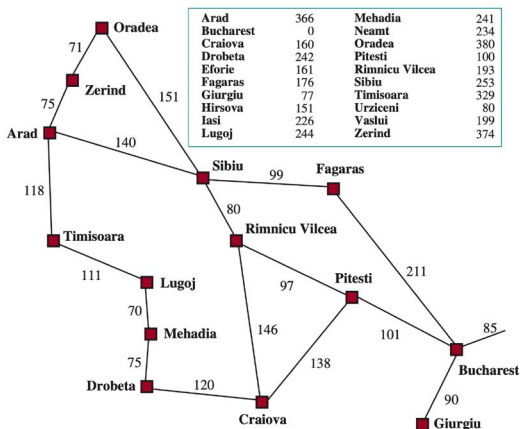


frontiers = [Zerind(374), Timisoara(329), Fagaras(176),
RV(193), Oradea(380)]

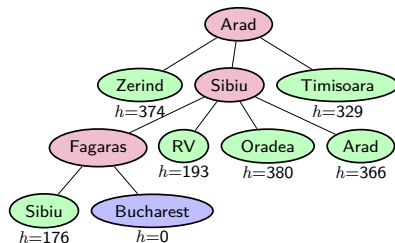
Which node to expand next? **Fagaras**

Greedy Search

- Search strategy:** Expand the node with lowest $h(n)$ value, *i.e.*, expanding the nodes closer to the goal state first.



Step 3: Expand Fagaras



Solution found: Arad \rightarrow Sibiu \rightarrow Fagaras \rightarrow Bucharest

Is it cost-optimal? **No in this example!**

Cost-optimal Solution: Arad \rightarrow Sibiu \rightarrow RV \rightarrow Pitesti \rightarrow Bucharest
(cost=140+80+97+101=418)

A* Search

- **Search strategy:** Expand the node with lowest $g(n) + h(n)$ value.
Combining greedy search and uniform-cost search.

Estimated total cost of the path

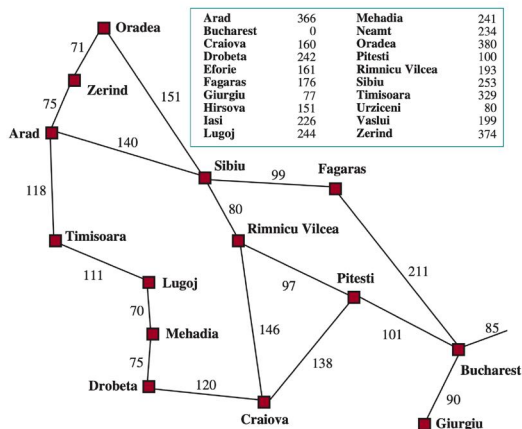
$$f(n) = g(n) + h(n)$$

Actual backward cost (used in UCS)


Estimated forward cost (used in greedy search)

A* Search

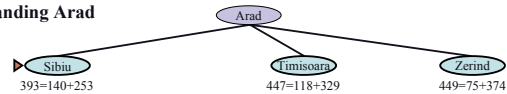
- Search strategy:** Expand the node with lowest $g(n) + h(n)$ value.
Combining greedy search and uniform-cost search.



(a) The initial state

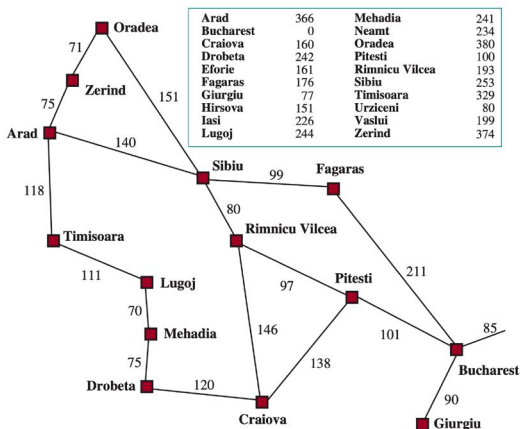

 $366 = 0 + 366$
 $g(\text{Arad}) = 0 \quad h(\text{Arad}) = 366$

(b) After expanding Arad

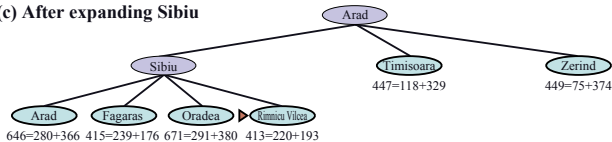


A* Search

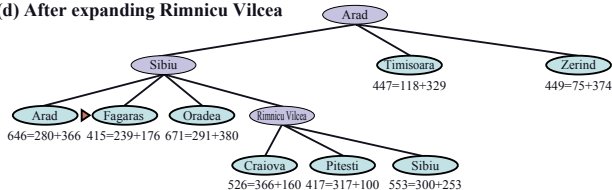
- Search strategy:** Expand the node with lowest $g(n) + h(n)$ value.
Combining greedy search and uniform-cost search.



(c) After expanding Sibiu



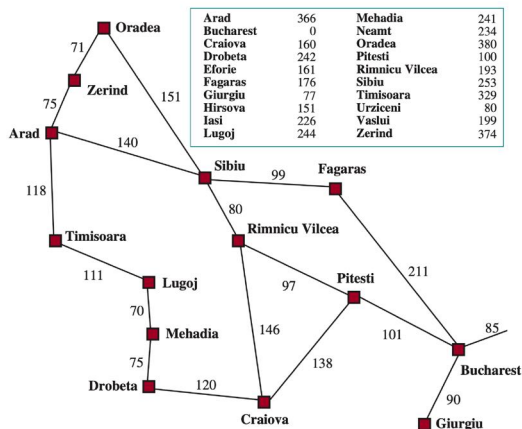
(d) After expanding Rimnicu Vilcea



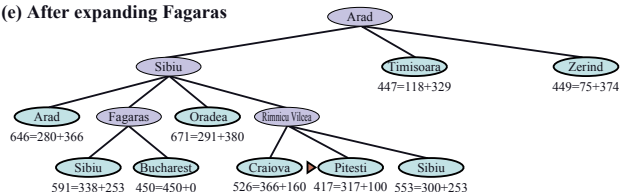
Can we stop and return the solution now? **No!**
 Other solutions with a smaller cost may exist!

A* Search

- Search strategy:** Expand the node with lowest $g(n) + h(n)$ value.
Combining greedy search and uniform-cost search.

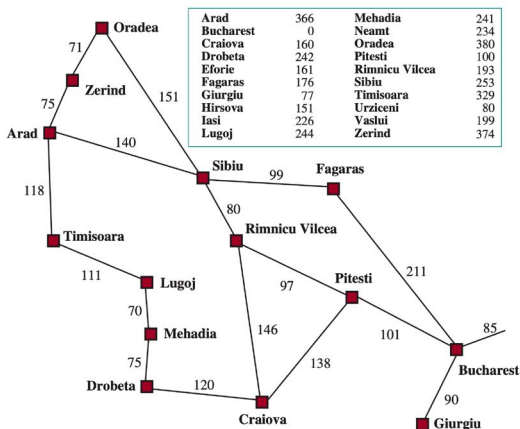


(e) After expanding Fagaras

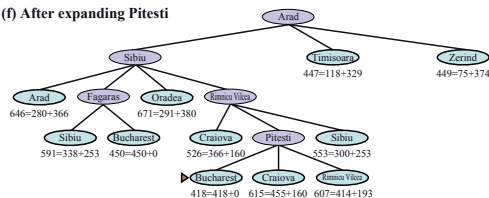


A* Search

- Search strategy:** Expand the node with lowest $g(n) + h(n)$ value.
Combining greedy search and uniform-cost search.



(f) After expanding Pitesti



Solution found: Arad→Sibiu→RV→Pitesti→Bucharest

Is it cost-optimal? **Yes, in this example.**

⚠ When we search for *cost-optimal solutions* (in UCS and A*), we check goal status when *expanding* nodes.

Is A* Search Always Cost-Optimal?

It depends on certain properties of the heuristic, e.g., admissibility.

Admissible Heuristic Function

An admissible heuristic is one that never overestimates the cost to reach a goal.

Optimality of A* Search

With an admissible heuristic, A* is cost-optimal.

Optimality of A* Search: A Proof by Contradiction

Suppose the optimal cost is C^* , but the algorithm returns a solution with cost of $C > C^*$:

- There must be some node n on the optimal path that are yet to be expanded.
- Let $g^*(n)$ be the cost of the optimal path from the start to n .
- Let $h^*(n)$ denote the cost of the optimal path from n to the nearest goal.

$$(1) \quad f(n) > C^* \quad (\text{otherwise } n \text{ would have been expanded})$$

$$(2) \quad f(n) = g(n) + h(n) \quad (\text{by definition})$$

$$(3) \quad f(n) = g^*(n) + h(n) \quad (\text{because } n \text{ is on an optimal path})$$

$$(4) \quad f(n) \leq g^*(n) + h^*(n) \quad (\text{because of admissibility, } h(n) \leq h^*(n))$$

$$(5) \quad f(n) \leq C^* \quad (\text{by definition, } C^* = g^*(n) + h^*(n))$$

(5) contradicts with (1) \Rightarrow There are no solutions with costs greater than C^* .

Optimality of Uniform-Cost Search

Can we also show that UCS is cost-optimal?

- UCS is a special case of A* search with $h(n) = 0 \ \forall n$.
- $h(n) = 0$ is an admissible heuristic function.
- Thus, UCS is cost-optimal.

How Much Do Heuristics Help?

Eight-puzzle: move the blank tile left, up, right, and down to match it with the goal state.

7	2	4
5		6
8	3	1

Initial State

	1	2
3	4	5
6	7	8

Goal State

- Number of reachable states for 8-puzzle: $9!/2 = 181,400$.
- How about 15-puzzle? Number of reachable states: $16!/2$ states – over 10 trillion.
- Demo: <https://deniz.co/8-puzzle-solver/> (*initial state: 724506831, iter limit: 1000*)

Another demo for path finding: <https://qiao.github.io/PathFinding.js/visual/>

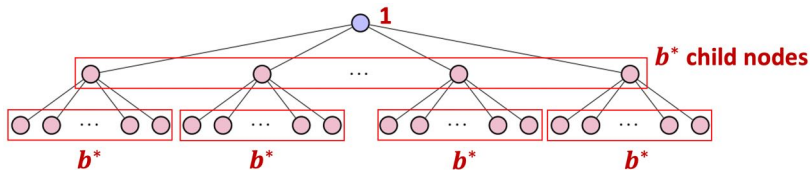
Characterizing the Quality of Heuristic Functions

Effective branching factor b^*

The effective branching factor b^* is the branching factor (number of child nodes for each node) that a uniform tree of depth d would have to have in order to contain $N + 1$ nodes.

- N : the total number of nodes generated by A* search.
- d : the depth of the solution found by A* search.
- A uniform tree: a tree where each node has the same number of child nodes.

$$N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$



Characterizing the Quality of Heuristic Functions

Example: A^* finds a solution at depth 5 with 52 nodes generated.

$$53 = 1 + b^* + (b^*)^2 + \dots + (b^*)^5 \Rightarrow b^* \approx 1.92$$

The closer b^* to one, the better the heuristic function. (less nodes need to be expanded)

Experimental measurements of b^* on a small set of problems can provide a good guide to the heuristic's overall usefulness.

- In Written Assignment 1, we will see two heuristic functions for the 8-puzzle and intuitively compare them.
- In Programming Assignment 1, we will implement the two heuristic functions and compare them by computing the b^* .

Constraint Satisfaction Problems

- Formulating CSPs: Exploiting The Structures
- Backtracking Search
- Constraint Propagation
- Variable and Value Ordering

Example: Coloring the Australian Map



- There are 7 provinces: WA, NT, SA, Q, NSW, V, T.
- Goal: to color each province either **red**, **green**, or **blue**.
- Rule: adjacent regions must have different colors.
Example: $\text{color}(\text{WA}) \neq \text{color}(\text{NT})$

Map Coloring: Can We Formulate It as An Ordinary Search Problem?

States: partial assignments of province colors.

- Initial state: No colors assigned.
- Goal state: an assignment that does not violate the rules.

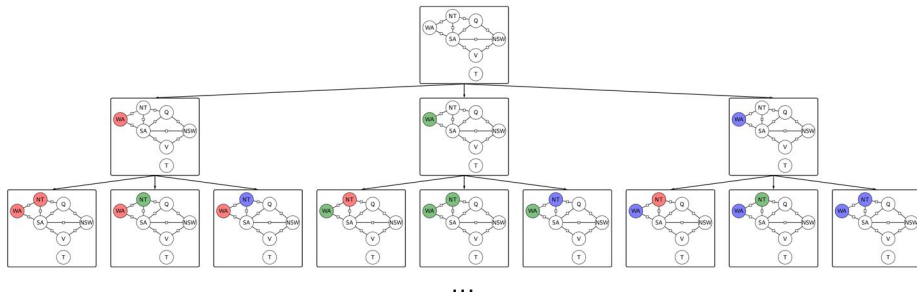


Figure: Part of the search tree (Figure from CS221@Stanford)

We can formulate it as a search problem, but it can be very inefficient!

What Goes Wrong? There Are Problem Structures



Why the atomic state-space searcher is not efficient?

- Suppose we color SA as **red**.
- How many partial assignments to consider for the five neighbors without using constraints? $3^5 = 243$.
- Constraints: none of its five neighbors can be **red**.
- How many partial assignments to consider for the five neighbors using constraints? **Only $2^5 = 32$.**

There are certain structures that we can use:

- We only need the assignments, but not the path!
(*The sequence is not important.*)
- We can use constraints to prune the search space!
e.g., $NT = \text{red} \Rightarrow WA \neq \text{red}$.

Formulating Constraint Satisfaction Problems (CSP)



Three components of a CSP:

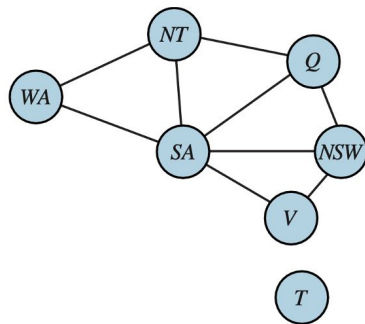
- A set of **variables**: $\mathcal{X} = \{\text{WA}, \text{NT}, \text{SA}, \text{Q}, \text{NSW}, \text{V}, \text{T}\}$.
- Each variable has a **domain** of possible values:
 $\mathcal{D}_i = \{\text{red}, \text{green}, \text{blue}\} \quad \forall i \in \mathcal{X}$
- A set of **constraints**:

$$\mathcal{C} = \{\text{WA} \neq \text{NT}, \text{WA} \neq \text{SA}, \text{NT} \neq \text{SA}, \text{NT} \neq \text{Q}, \\ \text{SA} \neq \text{Q}, \text{SA} \neq \text{NSW}, \text{SA} \neq \text{V}, \text{Q} \neq \text{NSW}, \\ \text{NSW} \neq \text{V}\}$$

A **solution** is an assignment to all variables such that no constraint is violated. For example:
 $\{\text{WA}=\text{red}, \text{NT}=\text{green}, \text{Q}=\text{red}, \text{NSW}=\text{green}, \text{V}=\text{red}, \text{SA}=\text{blue}, \text{T}=\text{red}\}$

Constraint Graphs

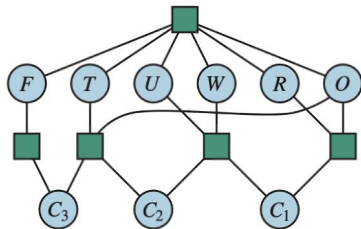
A *binary constraint graph* for the map coloring problem:



- Nodes are variables
- Arcs are constraints
- “*binary*” means each constraint involves two variables.

CSP Example: Cryptarithmic

$$\begin{array}{r}
 T \quad W \quad O \\
 + \quad T \quad W \quad O \\
 \hline
 F \quad O \quad U \quad R
 \end{array}$$



- Variables: $\mathcal{X} = \{T, W, O, F, U, R, C_1, C_2, C_3\}$
- Domains: $\mathcal{D}_i = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \quad \forall i \in \mathcal{X}$
- Constraints:
 - AllDiff(T, W, O, F, U, R)
 - $O + O = R + 10C_1$
 - $W + W + C_1 = U + 10C_2$
 - $T + T + C_2 = O + 10C_3$
 - $F = C_3$
- In a “*constraint hyper-graph*”, we use squares to represent constraints involving multiple variables.

CSP Example: Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

- Variables: $\mathcal{X} = \{A1, \dots, A9, \dots, I1, \dots, I9\}$
- Domains: $\mathcal{D}_i = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \quad \forall i \in \mathcal{X}$
- Constraints:
 - $A3 = 3, A5 = 2, \dots$
(existing numbers must match)
 - AllDiff($A1, A2, \dots, A9$), ...
(each row has all different values)
 - AllDiff($A1, B1, \dots, I1$), ...
(each column has all different values)
 - AllDiff($A1, A2, A3, B1, B2, B3, C1, C2, C3$), ...
(each 3x3 region has all different values)

Backtracking Search Algorithm

- Assign one variable at a time.
 - Variable assignments are commutative.
 - (WA=red then NT=blue) is the same as (NT=blue then WA=red)
 - We may choose between SA=red or SA=blue,
but we never choose between NSW=red or SA=blue. (consider one variable at a time!)
- Check constraints as we proceed.
 - We only consider values that do not conflict with previous assignments.
- Denote a partial assignment by $\mathcal{A} = \{X : v\}$ (like Python dictionary)
 - Example: $\mathcal{A} = \{\text{NT} : \text{red}, \text{SA} : \text{blue}\}$
- DFS with these two improvements is called *backtracking search*.

Backtracking Search Algorithm

The Naive Backtracking Search Algorithm (without Constraint Propagation)

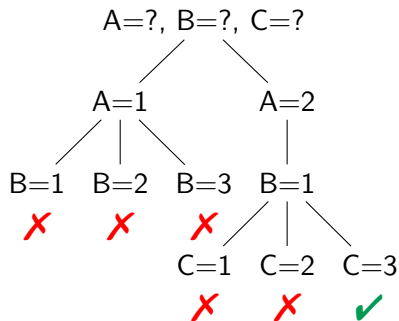
```
1 function Backtrack(assignment  $\mathcal{A}$ , domains  $\mathcal{D}$ )
2   if assignment  $\mathcal{A}$  is complete then return  $\mathcal{A}$ ;
3   select an unassigned variable  $X_i$ ;
4   order values for the variable  $X_i$ ;
5   for value  $v$  in that order do
6     if  $v$  is not consistent with  $\mathcal{A}$  then continue;
7     assign  $X_i = v$ ;
10    Backtrack( $\mathcal{A} \cup \{X_i : v\}$ ,  $\mathcal{D}'$ );
11  end
12 end
```

We will figure out how to select variable and order values later.

Backtracking Search: Example

Consider the following CSP problem:

- Variables:
 - $A \in \{1, 2, 3\}$
 - $B \in \{1, 2, 3\}$
 - $C \in \{1, 2, 3\}$
- Constraints:
 - $A > B$
 - $B \neq C$
 - $A \neq C$



One solution: $\{A = 2, B = 1, C = 3\}$

Can we improve the efficiency?

An observation: If $A = 1$, B cannot be 2 or 3.

Backtracking Search Algorithm with Constraint Propagation

Idea: Using constraints to prevent future failure and prune the search tree.

The Backtracking Search Algorithm (with Constraint Propagation)

```
1 function Backtrack(assignment  $\mathcal{A}$ , domains  $\mathcal{D}$ )
2   if assignment  $\mathcal{A}$  is complete then return  $\mathcal{A}$ ;
3   select an unassigned variable  $X_i$ ;
4   order values for the variable  $X_i$ ;
5   for value  $v$  in that order do
6     if  $v$  is not consistent with  $\mathcal{A}$  then continue;
7     assign  $X_i = v$ ;
8      $\mathcal{D}' \leftarrow$  propagate constraints ; // forward checking or arc consistency
9     if any variable has an empty domain in  $\mathcal{D}'$  then continue;
10    Backtrack( $\mathcal{A} \cup \{X_i : v\}$ ,  $\mathcal{D}'$ )
11  end
12 end
```

Constraint propagation: Using constraints to reduce the number of valid values for variables.

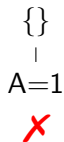
Backtracking with Forward Checking

Forward Checking:

- After selecting each assignment, remove any values of neighboring domains that are inconsistent with the new assignment.
- Terminate search when any variable has no legal values.

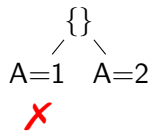
Example: Variables $A, B, C \in \{1, 2, 3\}$. Constraints: $A > B$, $B \neq C$, $A \neq C$.

step 1:



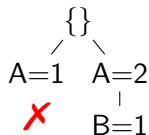
$$\mathcal{D}_B = \{1, 2, 3\}, \mathcal{D}_C = \{1, 2, 3\}$$

step 2:



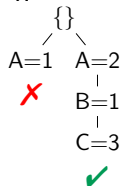
$$\mathcal{D}_B = \{1, 2, 3\}, \mathcal{D}_C = \{1, 2, 3\}$$

step 3:



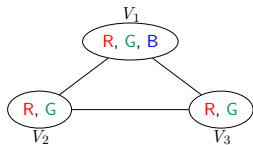
$$\mathcal{D}_B = \{1, 2, 3\}, \mathcal{D}_C = \{1, 2, 3\}$$

step 4:



$$\mathcal{D}_B = \{1, 2, 3\}, \mathcal{D}_C = \{1, 2, 3\}$$

Backtracking with Forward Checking: Exercise



- Variables: $\mathcal{X} = \{V_1, V_2, V_3\}$.
- Domains: $\mathcal{D}_1 = \{R, G, B\}$, $\mathcal{D}_2 = \{R, G\}$, $\mathcal{D}_3 = \{R, G\}$.
- Constraints: adjacent variables must have different colors.

Backtracking with Forward Checking: Exercise

In Australian map coloring problem, can we assign WA=**red**, Q=**green**, and V=**blue**?
Apply forward checking and show the steps.



Backtracking with Arc Consistency

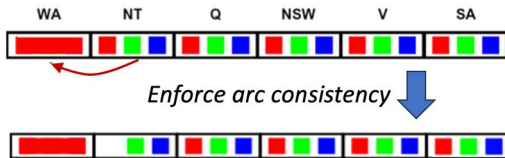
Arc Consistency

Variable X_i is *arc-consistent* with respect to another variable X_j if for every value in the current domain \mathcal{D}_i , there is some value in the domain \mathcal{D}_j that satisfies the binary constraint on the arc (X_i, X_j) .

Enforcing Arc Consistency

`EnforceArcConsistency(X_i, X_j)`: Remove values from \mathcal{D}_i to make X_i arc-consistent with respect to X_j .

Arc Consistency: Example



EnforceArcConsistency(NT, WA):

- If NT = blue or NT = green: binary constraint satisfied ($WA \neq NT$), consistent.
- If NT = red: constraint violated, delete red from the domain of NT to make NT arc consistent.

Forward checking is equivalent to enforcing arc consistency with each assigned variable as X_j .

Enforcing Arc Consistency of The Entire CSP

Suppose we assign $Q=\text{green}$ and we are left with the following domains:



Is arc (V, NSW) consistent?

Yes, for every value in the domain of V, there are some values in the domain of NSW which could be assigned without violating a constraint.



Is arc (SA, NSW) consistent? Yes!



Is arc (NSW, SA) consistent? No, when NSW=blue, constraint is violated.
 \Rightarrow Enforcing arc consistency: remove blue from \mathcal{D}_{NSW} .



Is arc (V, NSW) consistent? No, when V=red, constraint is violated.
 \Rightarrow Enforcing arc consistency: remove red from \mathcal{D}_V .

Important note: if X_i loses a value, recheck its neighbors!



Enforcing Arc Consistency of The Entire CSP

Remove **red** from \mathcal{D}_V :



Is arc (SA, NT) consistent? No, when SA=**blue**, constraint is violated.

⇒ Enforcing arc consistency: remove **blue** from \mathcal{D}_{SA} .



$\mathcal{D}_{SA} = \emptyset \Rightarrow$ failure, backtrack.

AC-3 Algorithm

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise

queue \leftarrow a queue of arcs, initially all the arcs in *csp*

while *queue* is not empty **do**

 (*X_i*, *X_j*) \leftarrow POP(*queue*)

if REVISE(*csp*, *X_i*, *X_j*) **then**

if size of *D_i* = 0 **then return** false

for each *X_k* **in** *X_i*.NEIGHBORS - {*X_j*} **do**

 add (*X_k*, *X_i*) to *queue*

return true

function REVISE(*csp*, *X_i*, *X_j*) **returns** true iff we revise the domain of *X_i*

revised \leftarrow false

for each *x* **in** *D_i* **do**

if no value *y* in *D_j* allows (*x*,*y*) to satisfy the constraint between *X_i* and *X_j* **then**

 delete *x* from *D_i*

revised \leftarrow true

return *revised*

Worst-case time complexity:

$$O(ed^3)$$

e: number of arcs;

d: size of the largest domain.

Bear in mind: CSPs in general are NP-complete!

Which Variable to Assign and How to Order Values?

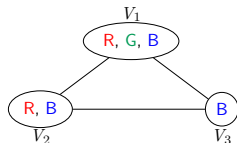
The Backtracking Search Algorithm (with Constraint Propagation)

```
1 function Backtrack(assignment  $\mathcal{A}$ , domains  $\mathcal{D}$ )
2   if assignment  $\mathcal{A}$  is complete then return  $\mathcal{A}$ ;
3   select an unassigned variable  $X_i$ ;
4   order values for the variable  $X_i$ ;
5   for value  $v$  in that order do
6     if  $v$  is not consistent with  $\mathcal{A}$  then continue;
7     assign  $X_i = v$ ;
8      $\mathcal{D}' \leftarrow$  propagate constraints ; // forward checking or arc consistency
9     if any variable has an empty domain in  $\mathcal{D}'$  then continue;
10    Backtrack( $\mathcal{A} \cup \{X_i : v\}$ ,  $\mathcal{D}'$ )
11  end
12 end
```

Remaining questions: 1) How to select variable? 2) How to order values?

Variable Ordering: Minimum-Remaining-Value (MRV) Heuristic

For the following problem, which variable will you assign first?



- Variables: $\mathcal{X} = \{V_1, V_2, V_3\}$.
- Domains: $\mathcal{D}_1 = \{R, G, B\}$, $\mathcal{D}_2 = \{R, B\}$, $\mathcal{D}_3 = \{B\}$.
- Constraints: adjacent variables must have different colors.

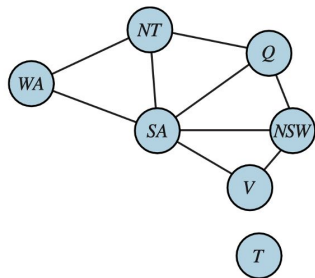


Minimum-Remaining-Value (MRV) Heuristic

- Choosing the variable with the fewest “legal” values.
- In this example, we assign V_3 first, then V_2 , and finally V_1 .
- Also called the “most constrained variable” or “fail-first” heuristic.
- Fewer “legal” values \Rightarrow more likely to cause a failure soon \Rightarrow prune the search tree.

Variable Ordering: Degree Heuristic

For the Australian map coloring problem, which variable will you assign first?



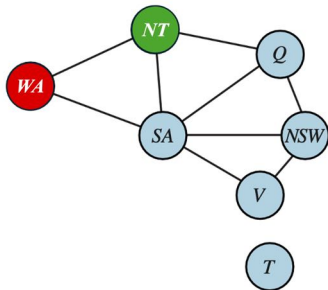
Degree Heuristic

- Choosing the variable with the highest degree.
- “Degree”: number of edges connecting to the node in a graph, e.g., $\text{deg}(\text{SA}) = 5$, $\text{deg}(\text{WA}) = 2$, $\text{deg}(\text{T}) = 0$.
- In this example, we assign SA first.

- The minimum-remaining-value heuristic is usually a more powerful guide.
- The degree heuristic can be useful as a tie-breaker.

Value Ordering: Least-Constraining-Value (LCV) Heuristic

Suppose we color WA as **red**, NT as **green**, and we choose Q to assign next. What value should we assign? **red** or **blue**?



Least-Constraining-Value (LCV) Heuristic

- Choosing the value that rules out the *fewest* choices for the neighboring variables in the constraint graph.
- In this example, we assign Q=**red** first. (*blue* eliminates the last legal value left for SA.)
- It attempts to leave the maximum flexibility for subsequent variable assignments.

Variable and Value Ordering: Summary

- For variable selection: “fail-first” minimum-remaining-value (MRV) heuristic.
- For value ordering: “fail-last” least-constraining-value (LCV) heuristic.

Why should variable selection be fail-first, but value selection be fail-last?

- **All variable has to be assigned eventually!** So, if an assignment is eventually going to fail, the sooner it fails, the more we prune the search tree.
- However, **each variable only need to take one value.** So, we choose the value that is most likely to lead to a solution.

Summary for Today

- Graph search vs. tree-like search: definitions? pros and cons?
- Measuring the performance of search algorithms:
Completeness, optimality, time complexity, and space complexity.
- Uninformed search: BFS, UCS, DFS, depth-limited, and iterative deepening search.
- Heuristic functions.
- Informed search algorithms: greedy search, A* search.
- Optimality of A* search and UCS.
- Effectiveness of heuristic functions.
- How to formulate a CSP: variables, domains, and constraints.
- Constraint graph and constraint hyper-graph for CSPs.
- Backtracking search algorithm for CSPs.
- Constraint propagation: forward checking and arc consistency.
- Variable and value ordering: MRV, LCV, and degree heuristic.

Let your voice be heard!



Thank you for your feedback! 🙌

Happy Mid-Autumn Festival! 🥮

Image generated by *StableDiffusionXL* with the prompt “draw a cartoon illustration of a lovely rabbit family eating mooncakes and admiring the moon to celebrate the mid-autumn festival”.