APPLIED ALGORITHMS

SOLVING PROBLEM BY SEARCHING

Lecture 2: Graphs and Basic Search Strategies

T.S Ha Minh Hoang
Th.S Nguyen Minh Anh

Phenikaa University

Last Update: 30th January 2023

TL;DR

Many practical problems can be represented using **graphs**. Today we recall the concept of graphs and practice representing many interesting problems in science and engineering.
Then we learn the most basic way of solving such problems (now in the graph form) using **uninformed search strategies**.

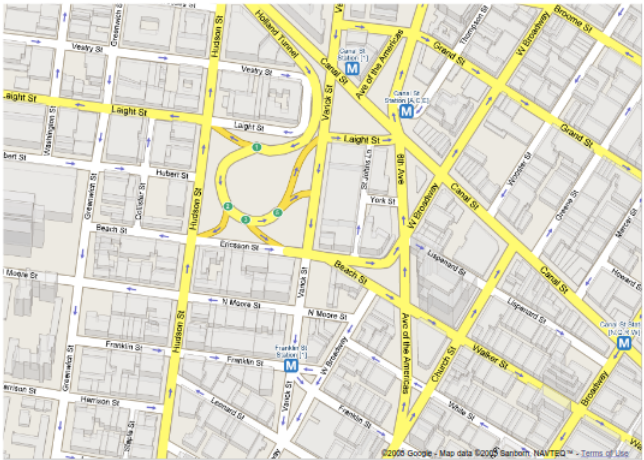
Outline

Motivations: Obvious examples

Social graph

Motivations: Obvious examples

Road network:



Motivations: Less obvious examples

Solving Sudoku puzzle:

	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		

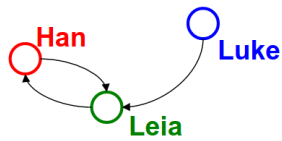
	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

This will be how we see graph throughout this course

Graph definition

- A *graph* is a formalism for representing relationships among items
  - Very general definition because very general concept

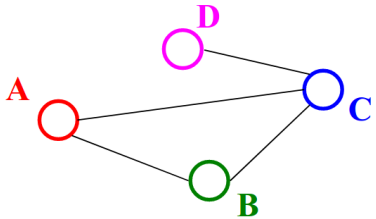
- A graph is a pair $G = (V, E)$
  - A set of vertices, also known as nodes $V = \{v_1, v_2, ..., v_n\}$
  - A set of edges $E = \{e_1, e_2, ..., e_m\}$
    - Each edge e_i is a pair of vertices (v_j, v_k)
    - An edge "connects" the vertices
- Graphs can be directed or undirected




```
V = {Han,Leia,Luke}
E = {(Luke,Leia),
     (Han,Leia),
     (Leia,Han)}
```

Undirected Graphs

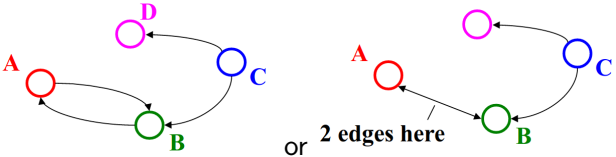
- In undirected graphs, edges have **no specific direction**
  - Edges are always "two-way"



- Thus, $(u, v) \in E$ implies $(v, u) \in E$
  - Only one of these edges needs to be in the set
- Degree of a vertex: number of edges containing that vertex
  - Put another way: the number of adjacent vertices

Directed Graphs

- In directed graphs (sometimes called digraphs), edges have a direction

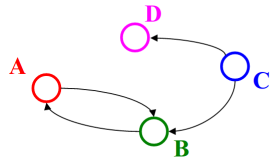


- Thus, $(u, v) \in E$ does not imply $(v, u) \in E$
  - Let $(u, v) \in E$ mean $u \to v$
  - Call u the source and v the destination
- In-degree of a vertex: number of in-bound edges,
  i.e., edges where the vertex is the destination
- Out-degree of a vertex: number of out-bound edges
  i.e., edges where the vertex is the source

More notations

For a graph $G = (V, E)$:

- $|V|$ is the number of vertices
- $|E|$ is the number of edges
  - Minimum?
  - Maximum for undirected?
  - Maximum for directed?
- If $(u, v) \in E$
  - Then v is a neighbor of u, i.e., v is adjacent to u
  - Order matters for directed edges - u is not adjacent to v unless $(v, u) \in E$




```
V = {A, B, C, D}
E = {(C, B),
     (A, B),
     (B, A)
     (C, D)}
```

Examples

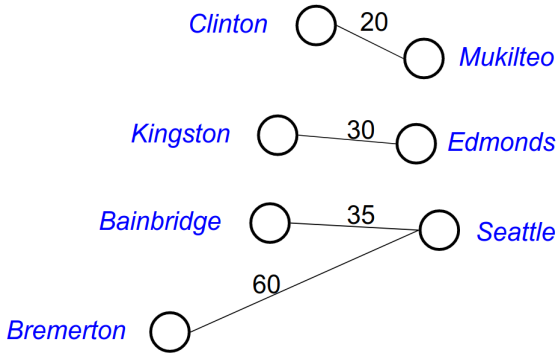
Which would use directed edges? Which would have self-edges?
Which would be connected? Which could have 0-degree nodes?

- Web pages with links
- Facebook friends
- Methods in a program that call each other
- Road maps (e.g., Google maps)
- Airline routes
- Family trees
- Course pre-requisites

Weighted Graphs

In a weighed graph, each edge might have a weight a.k.a. cost

- Typically numeric (most examples use ints)
- Some graphs allow *negative weights*; many do not



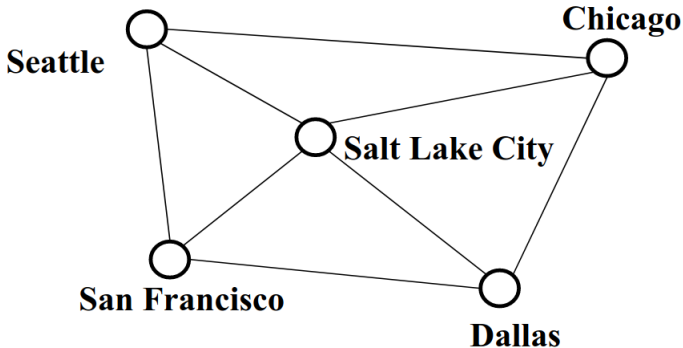
Examples

What, if anything, might weights represent for each of these?
Do negative weights make sense?

- Web pages with links
- Facebook friends
- Methods in a program that call each other
- Road maps (e.g., Google maps)
- Airline routes
- Family trees
- Course pre-requisites

Paths and Cycles

- A path is a list of vertices $[v_0, v_1, ..., v_n]$ such that $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$. Say "a path from v_0 to v_n"
- A cycle is a path that begins and ends at the same node ($v_0 == v_n$)

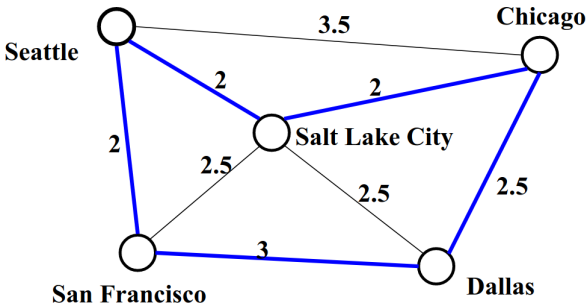


Example: [Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle]

Path Length and Cost

- Path length: Number of edges in a path
- Path cost: Sum of weights of edges in a path

Example where P= [Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle]

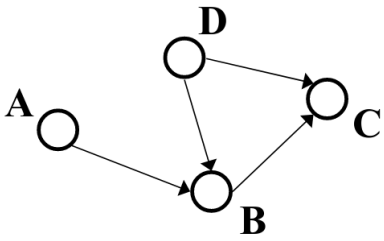


length(P) = 5

cost(P) = 11.5

Paths and Cycles in Directed Graphs

Example:



Is there a path from A to D?
Does the graph contain any cycles?

Trees as Graphs

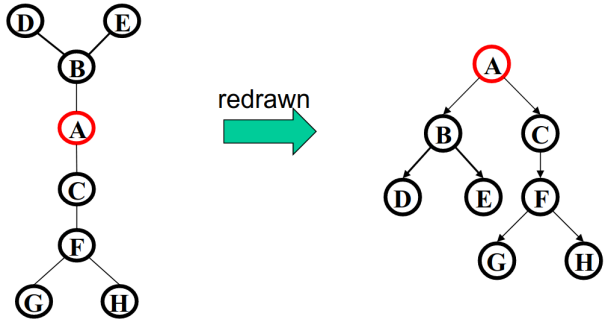
When talking about graphs, we say a tree is a graph that is:

- Undirected
- Acyclic
- Connected

So all trees are graphs, but not all graphs are trees

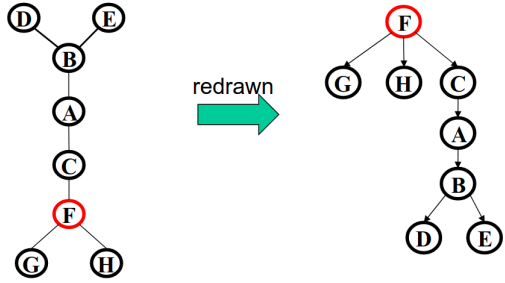
Rooted Trees

- We are more accustomed to rooted trees where:
  - We identify a unique root
  - We think of edges as directed: parent to children
- Given a tree, picking a root gives a unique rooted tree
  - The tree is just drawn differently



Rooted Trees

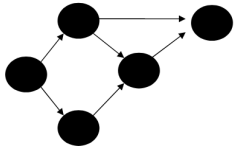
- We are more accustomed to rooted trees where:
  - We identify a unique root
  - We think of edges as directed: parent to children
- Given a tree, picking a root gives a unique rooted tree
  - The tree is just drawn differently



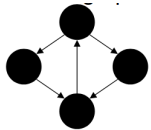
Directed Acyclic Graphs (DAGs)

A DAG is a directed graph with no (directed) cycles

- Every rooted directed tree is a DAG
- But not every DAG is a rooted directed tree



- Every DAG is a directed graph
- But not every directed graph is a DAG



Examples

Which of our directed-graph examples do you expect to be a DAG?

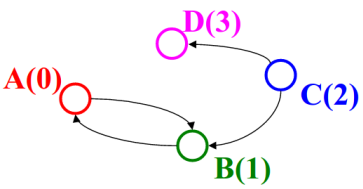
- Web pages with links
- Methods in a program that call each other
- Airline routes
- Family trees
- Course pre-requisites

Representations and Data Structures

- So graphs are really useful for lots of data and questions - For example, “what’s the lowest-cost path from x to y”
- But we need a data structure that represents graphs
- The ”best one” can depend on:
  - Properties of the graph (e.g., dense versus sparse)
  - The common queries (e.g., ”is (u,v) an edge?” versus “what are the neighbors of node u?”)
- So we’ll discuss the two standard graph representations
  - Adjacency Matrix and Adjacency List
  - Different trade-offs, particularly time versus space

Adjacency Matrix

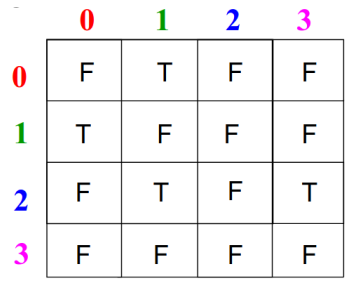
- A $|V||V|$ matrix (i.e., 2-D array) of Booleans (or 1 vs. 0)
- If M is the matrix, then M[u][v] being true means there is an edge from u to v



	0	1	2	3
0	F	T	F	F
1	T	F	F	F
2	F	T	F	T
3	F	F	F	F

Adjacency Matrix Properties

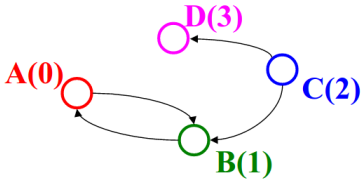
- Runtime complexity:
  - Get a vertex's out-edges: $O(|V|)$
  - Get a vertex's in-edges: $O(|V|)$
  - Decide if some edge exists: $O(1)$
  - Insert an edge: $O(1)$
  - Delete an edge: $O(1)$
- Space complexity: $O(|V|^2)$
- Best for sparse or dense graphs? Best for dense graphs

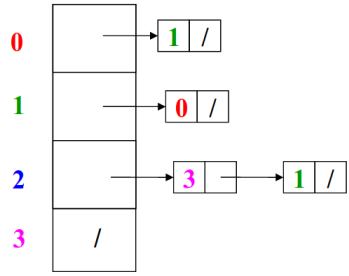


	0	1	2	3
0	F	T	F	F
1	T	F	F	F
2	F	T	F	T
3	F	F	F	F

Adjacency List

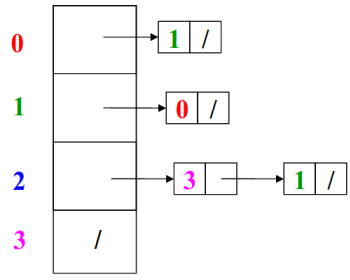
An array of length $|V|$ in which each entry stores a list of all adjacent vertices (e.g., linked list)





Adjacency List Properties

- Runtime complexity:
  - Get all of a vertex's out-edges: $O(|d|)$ where d is out-degree of vertex
  - Get all of a vertex's in-edges: $O(|E|)$ (but could keep a second adjacency list for this!)
  - Decide if some edge exists: $O(d)$ where d is out-degree of source
  - Insert an edge: $O(1)$ (unless you need to check if it's there)
  - Delete an edge: $O(d)$ where d is out-degree of source
- Space complexity: $O(|V| + |E|)$
- Best for sparse or dense graphs? Best for sparse graphs

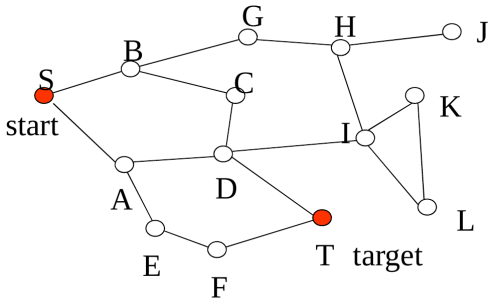


Outline

Graph representation of a search problem

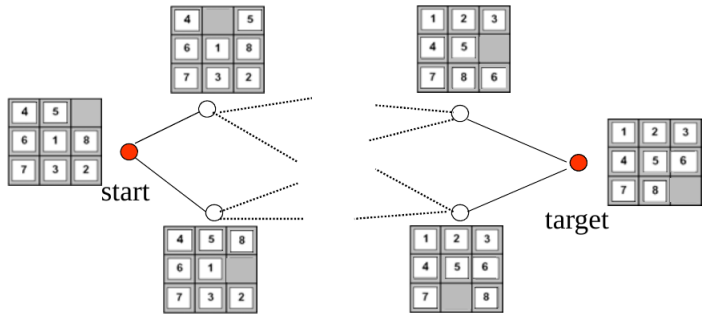
Why graphs?

- Search problems can be often represented using graphs
- Typical example: **Route finding**
  - Map corresponds to the graph, *nodes* to cities, *links* valid moves via available connections
  - Goal: find a route (sequence of moves) in the graph from S to T



Graph search

- Less obvious conversion: **Puzzle 8.**
  - *Nodes* corresponds to states of the game
  - *Links* to valid moves made by the player



Graph Search Problems

Search problems can be often represented as graph search problems:

- **Initial state**:
  - State (configuration) we start to search from (e.g. start city, initial game position)
- **Operators**:
  - Transform one state to another (e.g. valid connections between cities, valid moves in Puzzle 8)
- **Goal condition**:
  - Defines the target state (destination, winning position)

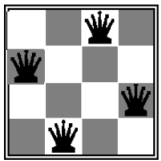
Search space is now defined indirectly through:

The initial state + Operators

N-queens

Some problems are easy to convert to the graph search problems

- But some problems are harder and less intuitive
  - Take e.g. N-queens problem



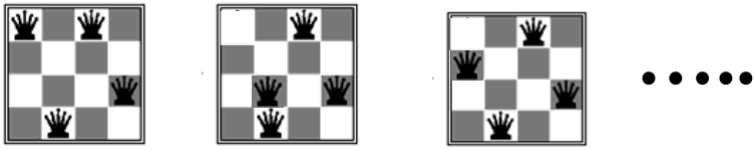
Goal configuration

- Problem:
  - We look for a configuration, not a sequence of moves
  - No distinguished initial state, no operators (moves)

N-queens

How to choose the search space for N-queens?

- Ideas?
  Search space:
  - all configurations of N queens on the board



- **Can we convert it to a graph search problem?**
- We need states, operators, initial state and goal condition.

N-queens

Search space: all configurations of N queens on the board

- To convert it to a graph search problem we need **states**, **operators**, **initial state** and **goal condition**.

Initial state: ?
Operators (moves): ?

N-queens

Search space: all configurations of N queens on the board

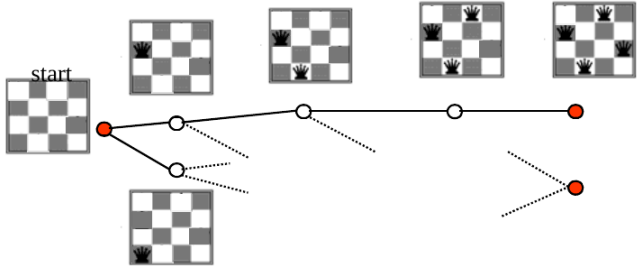
- To convert it to a graph search problem we need **states**, **operators**, **initial state** and **goal condition**.

Initial state: an arbitrary N-queen configuration
Operators (moves): change a position of one queen

N-queens

An alternative way to formulate the N-queens problem as a search problem:

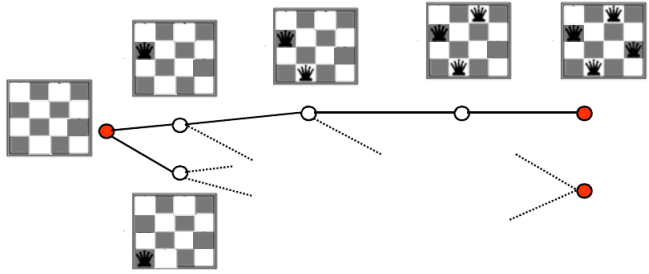
- **Search space**: configurations of 0,1,2, ... N queens
- **Graph search**:
  - States configurations of 0,1,2,...,N queens
  - **Operators**: additions of a queen to the board
  - **Initial state**: no queens on the board (empty board)



Graph search

N-queens problems

- This is a different graph search problem when compared to Puzzle 8 or Route planning: **We want to find only the target configuration, not a path**

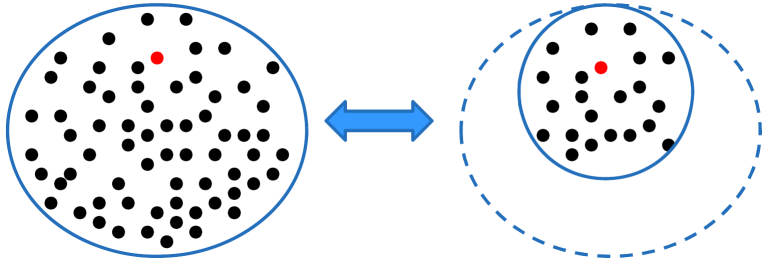


Two types of graph search problems

- **Path search**
  - Find a path between states S and T
  - Example: traveler problem, puzzle 8, maze puzzle, k-knights problem
  - Additional goal criterion: minimum length (cost) path
- **Configuration search (constraint satisfaction search)**
  - Find a state (configuration) satisfying the goal condition
  - Example: n-queens problem, sudoku
  - Additional goal criterion: "soft" preferences for configurations, e.g. minimum cost design

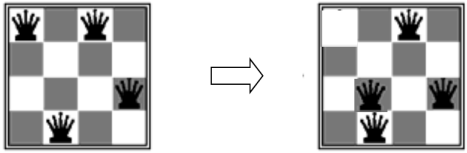
Search

- Search (process): The process of exploration of the search space
- The efficiency of the search depends on:
  - **The search space and its size**
  - Method used to explore (traverse) the search space
  - Condition to test the satisfaction of the search objective (what it takes to determine I found the desired goal object)



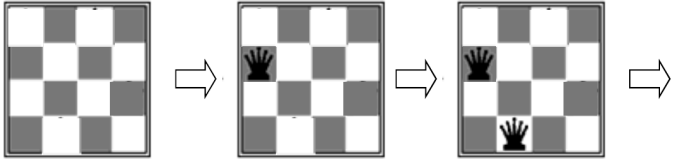
Comparison of two problem formulations

Solution 1:



Operators: switch one of the queens

Solution 2:

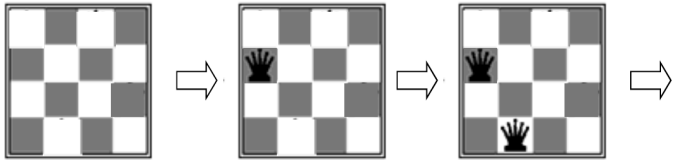


Operators: add a queen to the leftmost unoccupied column

Which one is better?

Even better solution to the N-queens

Solution 2:



Operators: add a queen to the leftmost unoccupied column

Improved solution with a smaller search space
Operators: add a queen to the leftmost unoccupied column such that it does not attack already placed queens
Question: the size of the search space now?
→ Think twice before solving the problem: Choose the search space wisely

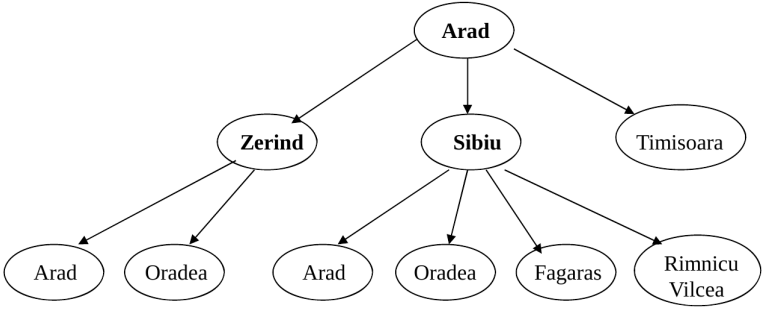
Search

- Search (process): The process of exploration of the search space
- The efficiency of the search depends on:
  - The search space and its size
  - **Method used to explore (traverse) the search space**
  - Condition to test the satisfaction of the search objective (what it takes to determine I found the desired goal object)

Search process

Exploration of the state space through successive application of operators from the initial state

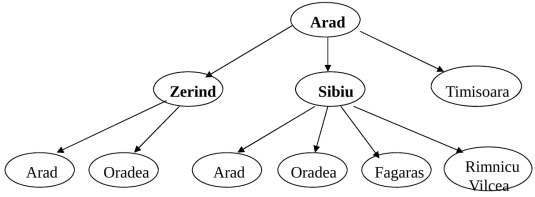
- Search tree = structure representing the exploration trace
  - Is built on-line during the search process
  - Branches correspond to explored paths, and leaf nodes to the exploration fringe



Search tree

- A search tree = (search) exploration trace
  - different from the graph representation of the problem
  - states can repeat in the search tree

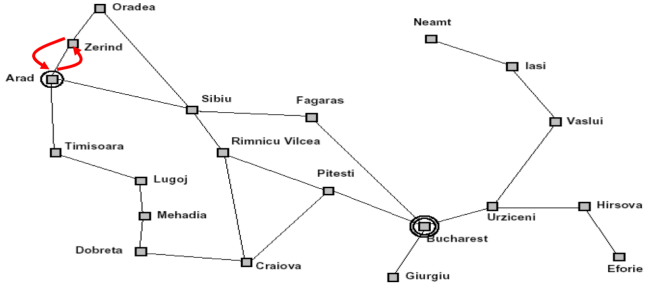
Graph



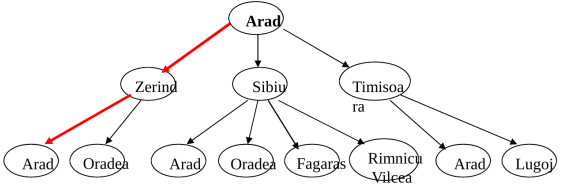
Search tree

built by exploring paths starting from city Arad

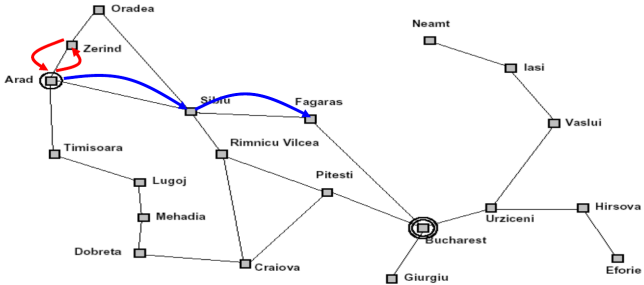
Search tree



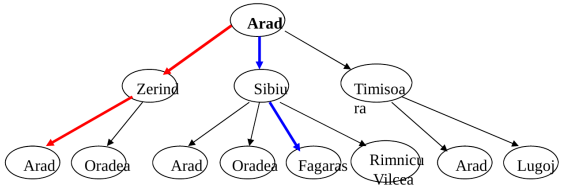
A branch in the search tree = a path in the graph



Search tree



A branch in the search tree = a path in the graph



General search algorithm

General-search (*problem*, *strategy*)
initialize the search tree with the initial state of *problem*
loop
  if there are no candidate states to explore **return** failure
  choose a leaf node of the tree to expand next according to *strategy*
  if the node satisfies the goal condition **return** the solution
  expand the node and add all of its successors to the tree
end loop

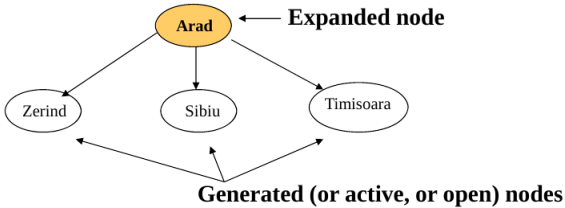
General search algorithm

General-search (*problem*, *strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **return** failure
choose a leaf node of the tree to expand next according to *strategy*
if the node satisfies the goal condition **return** the solution
expand the node and add all of its successors to the tree
end loop

Check if the node satisfied the goal

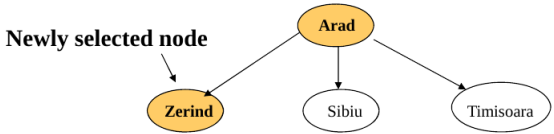
General search algorithm

General-search (*problem*, *strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **return** failure
choose a leaf node of the tree to expand next according to *strategy*
if the node satisfies the goal condition **return** the solution
expand the node and add all of its successors to the tree
end loop



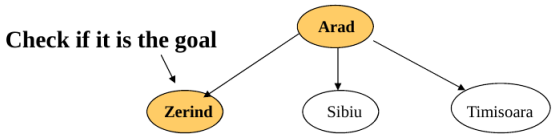
General search algorithm

General-search (*problem*, *strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **return** failure
choose a leaf node of the tree to expand next according to *strategy*
if the node satisfies the goal condition **return** the solution
expand the node and add all of its successors to the tree
end loop



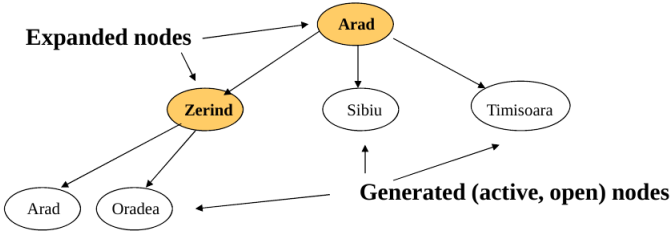
General search algorithm

General-search (*problem*, *strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **return** failure
choose a leaf node of the tree to expand next according to *strategy*
if the node satisfies the goal condition **return** the solution
expand the node and add all of its successors to the tree
end loop



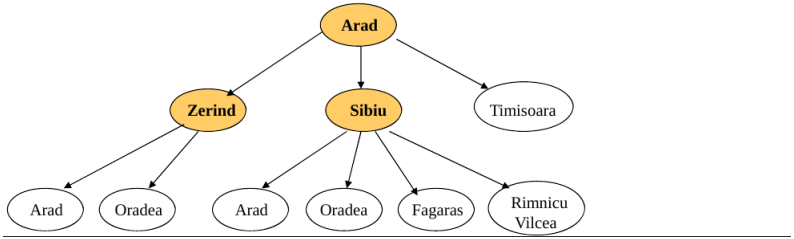
General search algorithm

General-search (*problem*, *strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **return** failure
choose a leaf node of the tree to expand next according to *strategy*
if the node satisfies the goal condition **return** the solution
expand the node and add all of its successors to the tree
end loop



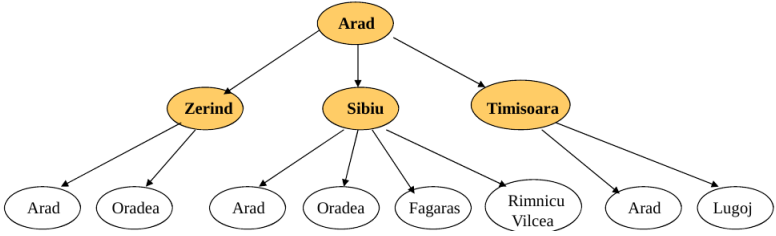
General search algorithm

General-search (*problem*, *strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **return** failure
choose a leaf node of the tree to expand next according to *strategy*
if the node satisfies the goal condition **return** the solution
expand the node and add all of its successors to the tree
end loop



General search algorithm

General-search (*problem*, *strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **return** failure
choose a leaf node of the tree to expand next according to *strategy*
if the node satisfies the goal condition **return** the solution
expand the node and add all of its successors to the tree
end loop



General search algorithm

General-search (*problem*, *strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **return** failure
choose a leaf node of the tree to expand next **according to a *strategy***
if the node satisfies the goal condition **return** the solution
expand the node and add all of its successors to the tree
end loop

Search methods differ in how they explore the space, that is how they choose the node to expand next !!!!!

Uninformed Search Methods

- Search techniques that rely only on the information available in the problem definition
  - Breadth first search
  - Depth first search
  - Iterative deepening
  - Bi-directional search
- For the minimum cost path problem:
  - Uniform cost search

Search methods

Properties of search methods:

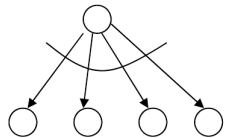
- Completeness
  - Does the method find the solution if it exists?
- Optimality
  - Is the solution returned by the algorithm optimal? Does it give a minimum length path?
- Space and time complexity
  - How much time it takes to find the solution?
  - How much memory is needed to do this?

Parameters to measure complexities

Complexity is measured in terms of the following tree parameters:

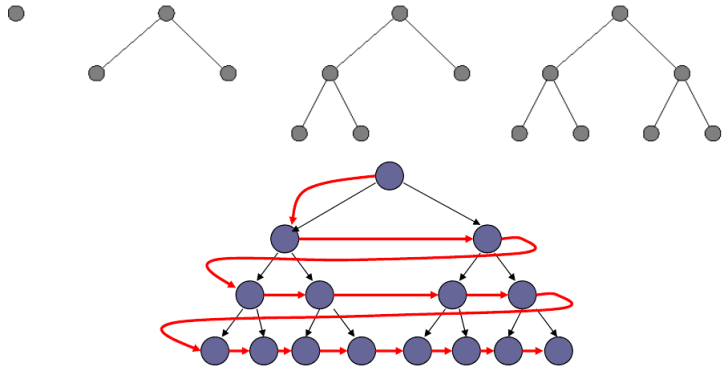
- b - maximum branching factor
- d - depth of the optimal solution
- m - maximum depth of the state space

Branching factor: The number of applicable operators



Breadth first search (BFS)

The shallowest node is expanded first



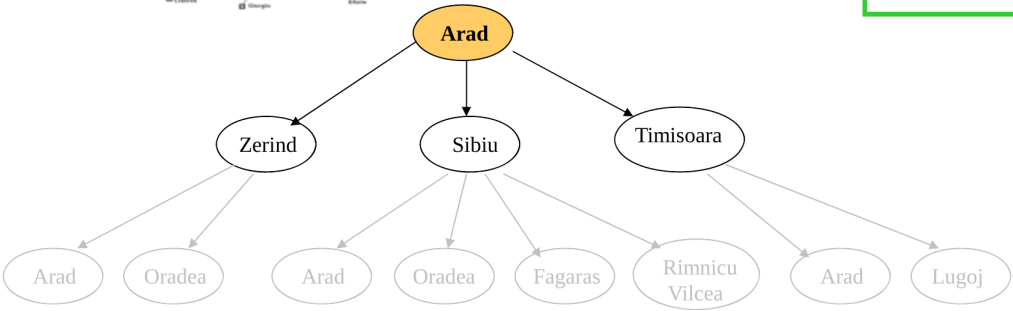
Breadth-first search

- Expand the shallowest node first
- Implementation: put successors to the end of the queue (FIFO)

Breadth-first search

queue ⇨

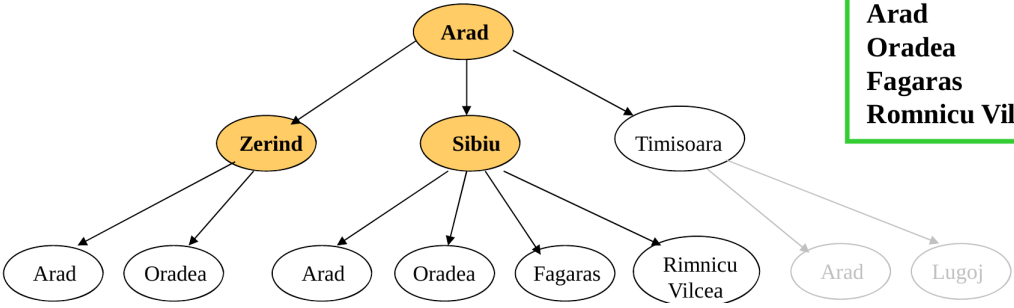
Zerind
Sibiu
Timisoara



Breadth-first search

queue ⇨

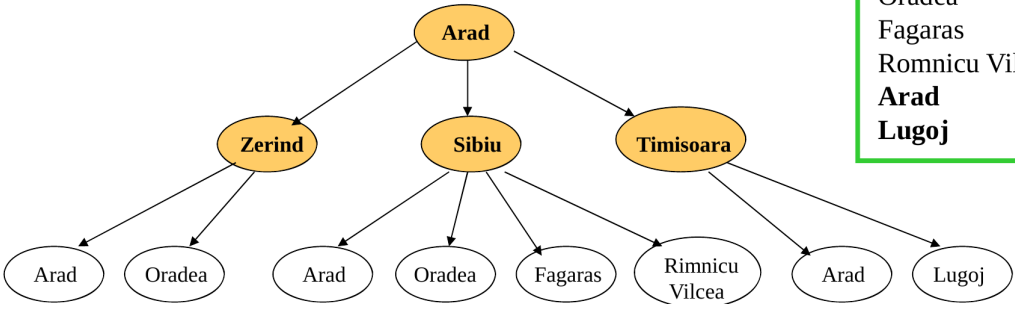
Timisoara
Arad
Oradea
Arad
Oradea
Fagaras
Romnicu Vilcea



Breadth-first search

queue

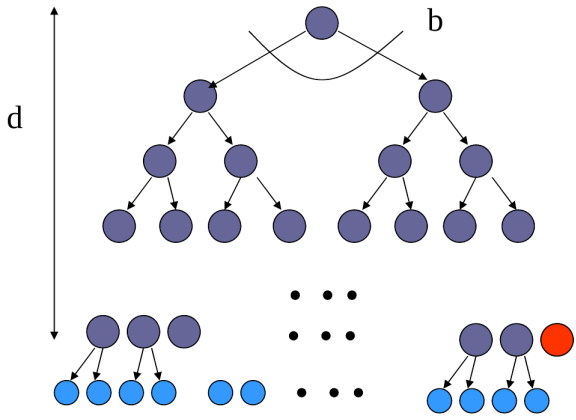
Arad
Oradea
Arad
Oradea
Fagaras
Romnicu Vilcea
Arad
Lugoj



Properties of breadth-first search

- **Completeness: Yes.** The solution is reached if it exists.
- **Optimality: Yes**, for the shortest path.
- **Time complexity: ?**
- **Memory (space) complexity: ?**

BFS - time complexity



depth	number of nodes
0	1
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
d	2^d (b^d)
d+1	2^{d+1} (b^{d+1})

Expanded nodes: $O(b^d)$

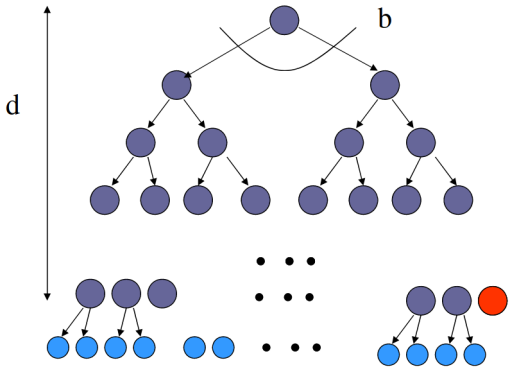
Total nodes: $O(b^{d+1})$

Properties of breadth-first search

- **Completeness: Yes.** The solution is reached if it exists.
- **Optimality: Yes**, for the shortest path.
- **Time complexity:**
  $1 + b + b^2 + ... + b^d = O(b^d)$
  exponential in the depth of the solution d
- **Memory (space) complexity: ?**

BFS - memory complexity

Count nodes kept in the tree structure or in the queue

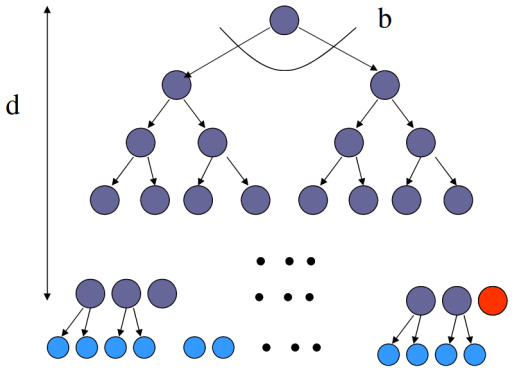


depth	number of nodes
0	1
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
d	2^d (b^d)
d+1	2^{d+1} (b^{d+1})

Total nodes: **?**

BFS - memory complexity

Count nodes kept in the tree structure or in the queue



depth	number of nodes
0	1
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
d	2^d (b^d)
d+1	2^{d+1} (b^{d+1})

Total nodes: **?**

Expanded nodes: $O(b^d)$

Total nodes: $O(d^{d+1})$

Properties of breadth-first search

- **Completeness: Yes.** The solution is reached if it exists.
- **Optimality: Yes**, for the shortest path.
- **Time complexity:**
  $1 + b + b^2 + ... + b^d = O(b^d)$
  exponential in the depth of the solution d
- **Memory (space) complexity:**
  $O(b^d)$
  nodes are kept in the memory

In the next lecture ...

DFS and others of uninformed search methods