COE206 – Principles of Artificial Intelligence

Mustafa MISIR

Istinye University, Department of Computer Engineering

mustafa.misir@istinye.edu.tr

http://mustafamisir.github.io
http://memoryrlab.github.io

L3: Problem Solving
Search Strategies

Outline

- Problem Solving Agents
- Example Problems
- Searching for Solutions

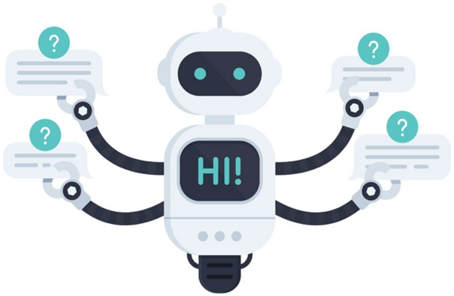
Outline

- Problem-Solving Agents
- Example Problems
- Searching for Solutions

Problem Solving Agents

The focus is a specific **goal-based agent** called a **problem-solving agent**.

- using **atomic** representations



Problem Solving Agents – Goal

Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.

- A goal is a set of world states—exactly those states in which the goal is satisfied.
- The agent's task is to find out how to act, now and in the future, so that it reaches a goal state.

Problem Solving Agents – Problem

Problem formulation is the process of deciding what actions and states to consider, given a goal.

- e.g. the agent will consider actions at the level of driving from one major town to another. Each state therefore corresponds to being in a particular town.

Problem Solving Agents – e.g. Trip to Bucharest

Our agent has now the **goal** of driving to Bucharest and is considering where to go from Arad.

- Three roads lead out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind.

Yet,

- the agent will not know which of its possible actions is best, because it does not yet know enough about the state that results from taking each action.
- If the agent has no additional information—i.e., if the environment is **unknown**, then it is has no choice but to try one of the actions at random.

Problem Solving Agents

- The process of looking for a sequence of actions that reaches the goal is called **search**.
- A **search** algorithm takes a problem as input and returns a **solution** in the form of an **action** sequence.
- Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase.

Problem Solving Agents

1. Formulate a **goal** and a **problem**
2. Search for a sequence of **actions** that would solve the problem
3. Execute the actions one at a time
4. When this is complete, formulate another goal and start over

   ...

Problem Solving Agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT($percept$) returns an action
  persistent: $seq$, an action sequence, initially empty
              $state$, some description of the current world state
              $goal$, a goal, initially null
              $problem$, a problem formulation

  $state \leftarrow$ UPDATE-STATE($state, percept$)
  if $seq$ is empty then
     $goal \leftarrow$ FORMULATE-GOAL($state$)
     $problem \leftarrow$ FORMULATE-PROBLEM($state, goal$)
     $seq \leftarrow$ SEARCH($problem$)
     if $seq$ = $failure$ then return a null action
  $action \leftarrow$ FIRST($seq$)
  $seq \leftarrow$ REST($seq$)
  return $action$
```

Problem

A problem can be defined formally by **five components**:

1. an initial state that the agent starts in
2. a description of the possible actions available to the agent
3. a description of what each action does – transition models

The initial state, actions, and transition model implicitly define the state space of the problem—the set of all states reachable from the initial state by any sequence of actions.

4. The goal test, which determines whether a given state is a goal state.
5. A path cost function that assigns a numeric cost to each path.

Problem – e.g. Trip to Bucharest

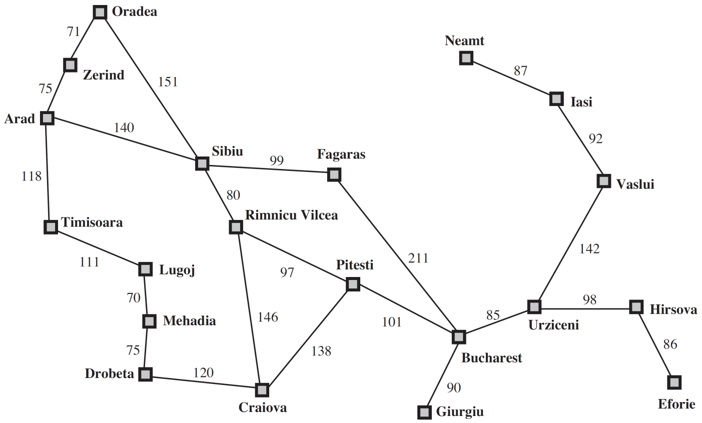
1. initial state: start from the city of Arad, $In(Arad)$
2. actions: from the state $In(Arad)$, the applicable actions are $\{$ $Go(Sibiu), Go(Timisoara), Go(Zerind)$ $\}$.
3. transition model: specified by a function RESULT(s, a) that returns the state that results from doing action a in state s, e.g. RESULT$(In(Arad), Go(Zerind)) = In(Zerind)$.
4. goal test: $\{$ $In(Bucharest)$ $\}$
5. path cost: distance traveled in *km*

A solution to a problem is an action sequence that leads from the initial state to a goal state.

- Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.

Problem Solving Agent

The path cost is the sum of the individual actions' costs, i.e. the step cost, taking action a in state s to reach state s', $c(s, a, s')$

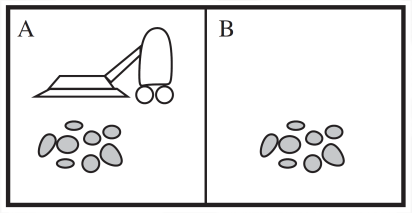


Outline

Toy Problems: e.g. Vacuum World

TASK: Determine the following aspects for a Vacuum Cleaner scenario:

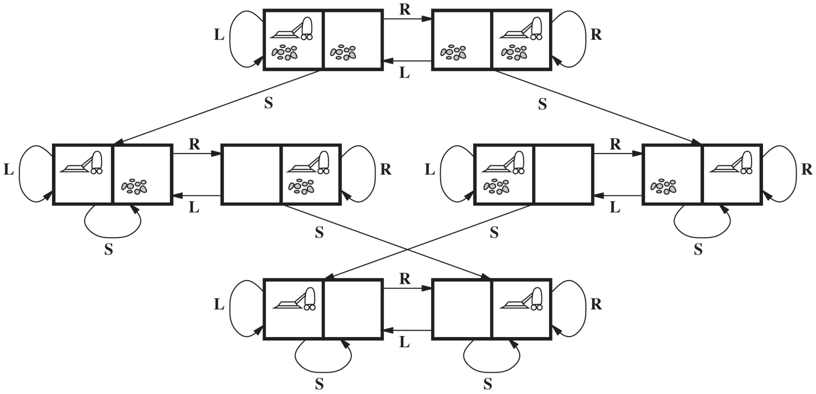
- All the **states**, **initial state**, **actions**, **transition model**, **goal test** and **path cost**.



Submit photo of your answer to **Piazza** as a *private message*. Also, deliver its hard copy.

Toy Problems: e.g. Vacuum World

Here is the state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.



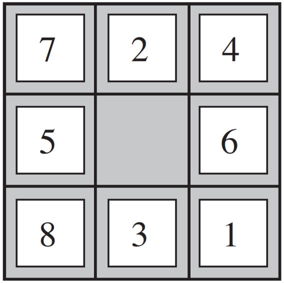
Toy Problems: e.g. Vacuum World

- **States**: The state is determined by both the agent and the dirt locations. There are $2 \times 2^2 = 8$ possible world states.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: Each state has 3 actions: Left, Right, and Suck. Larger environments might also include Up and Down.
- **Transition model**: The actions have their expected effects, except moving Left and Right in the leftmost and rightmost squares respectively, and Sucking in a clean square.
- **Goal test**: This checks whether all the squares are clean.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

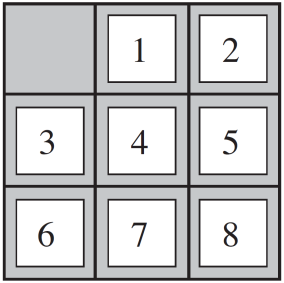
Toy Problems: e.g. 8-Puzzle[1]

A 3×3 board with 8 numbered tiles and a blank space, belonging to the family of **sliding-block puzzles**.

- A tile adjacent to the blank space can slide into the space.



Start State



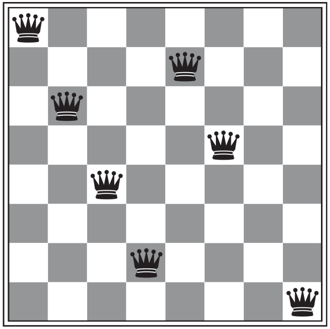
Goal State

[1] https://en.wikipedia.org/wiki/15_puzzle

Toy Problems: 8-Puzzle

- **States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down.
- **Transition model**: Given a state and action, this returns the resulting state.
- **Goal test**: This checks whether the state matches the goal configuration.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Toy Problems: 8-Queens



Place 8 queens on a chessboard such that no queen attacks any other. (A queen[2] attacks any piece in the same row, column or diagonal)

[2] In chess, the queen can be moved any number of unoccupied squares in a straight line vertically, horizontally, or diagonally: https://en.wikipedia.org/wiki/Queen_(chess)

Toy Problems: 8-Queens

- **States**: Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square.
- **Goal test**: 8 queens are on the board, none attacked.
- **Path cost**: Irrelevant

This formulations has a state space of size

$64 \times 63 \times \ldots \times 57 = 1.8 \times 10^{14}$ possible sequences.

Toy Problems: 8-Queens

A better formulation would prohibit placing a queen in any square that is already attacked:

- **States**: All possible arrangements of n queens $(0 \leqslant n \leqslant 8)$, one per column in the leftmost n columns, with no queen attacking another.
- **Actions**: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

This formulation reduces the state space.

Toy Problems: Knuth's Conjecture - Infinite State Spaces

Starting from number 4, any desired integer number can be reached via a sequence of factorial, square root, and floor operations.

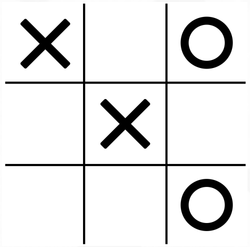
$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

- **States**: Positive numbers.
- **Initial state**: 4.
- **Actions**: Apply factorial, square root, or floor operation (factorial for integers only).
- **Transition model**: As given by the mathematical definitions of the operations.
- **Goal test**: State is the desired positive integer.

Toy Problems: Tic-Tac-Toe[3]

TASK : Determine the following aspects for a Tic Tac Toe game:

- All the **states**, **initial state**, **actions**, **transition model**, **goal test** and **path cost**.

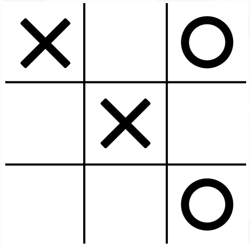


Submit photo of your answer to **Piazza** as a *private message*.
Also, deliver its hard copy.

[3] https://playtictactoe.org/ – Google Search: Tic Tac Toe

Toy Problems: Tic-Tac-Toe[4]

TASK: Draw / list the state space for the following game setting:



Submit photo of your answer to **Piazza** as a *private message*.
Also, deliver its hard copy.

[4] https://playtictactoe.org/ – Google Search: Tic Tac Toe

Real-world Problems: e.g. Route-Finding

Consider the airline travel problem that must be solved by a travel-planning Web site:

- **States**: Each state includes a location and the current time.
- **Initial state**: This is specified by the user's query.
- **Actions**: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
- **Transition model**: The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time.
- **Goal test**: Are we at the final destination?
- **Path cost**: This depends on monetary cost, waiting time, flight time, seat quality etc.

Real-world Problems: e.g. Touring

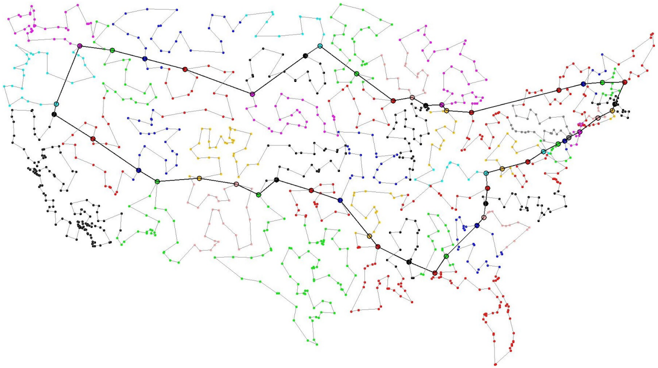
Closely related to route-finding problems, but with an important difference.

- Each state must include not just the current location but also the set of cities the agent has visited.
- So the initial state would be In(*Bucharest*), Visited($\{$*Bucharest*$\}$)
- A typical intermediate state would be In(*Vaslui*), Visited($\{$*Bucharest, Urziceni, Vaslui*$\}$)
- The goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

Real-world Problems: e.g. TSP

Traveling Salesperson Problem (TSP)[5]

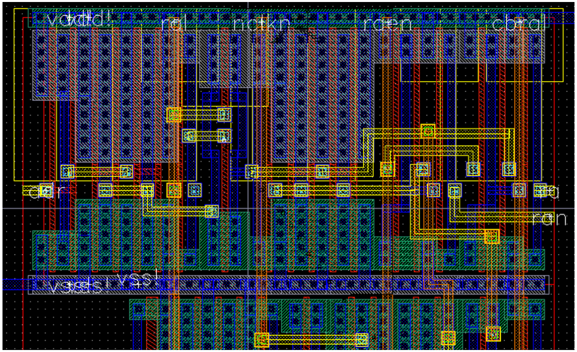
- Visit each city exactly once and return back to the starting city.



[5] image source: http://akira.ruc.dk/~keld/research/GLKH/ – example TSP solution while solving: https://commons.wikimedia.org/wiki/File:Travelling_salesman_problem_solved_with_simulated_annealing.gif

Real-world Problems: e.g. VLSI Layout

VLSI[6] layout requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield.

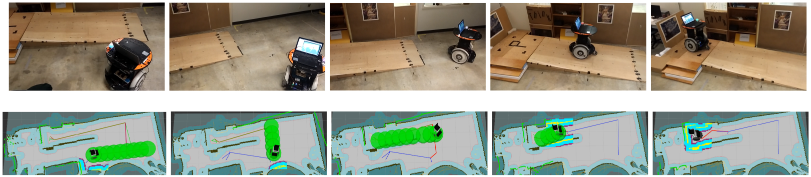


[6] https://en.wikipedia.org/wiki/Very_Large_Scale_Integration: Very large-scale integration is the process of creating an integrated circuit by combining millions of MOS transistors onto a single chip – image source: https://www.siliconsys.in/

Real-world Problems: e.g. Robot Navigation

Robot navigation[7]

- Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states.

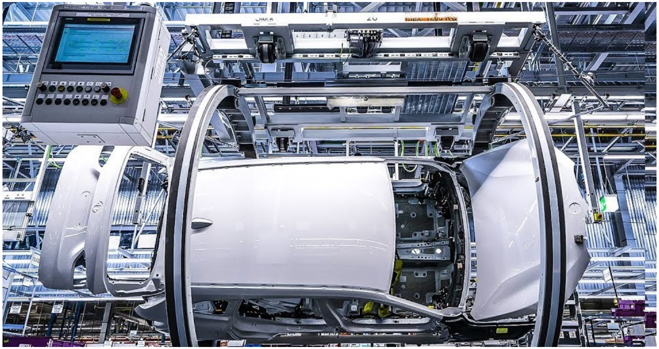

(a) (b) (c) (d) (e)

[7] image source: https://www.mdpi.com/1424-8220/19/13/2993/htm

Real-world Problems: e.g. Automatic Assembly Sequencing

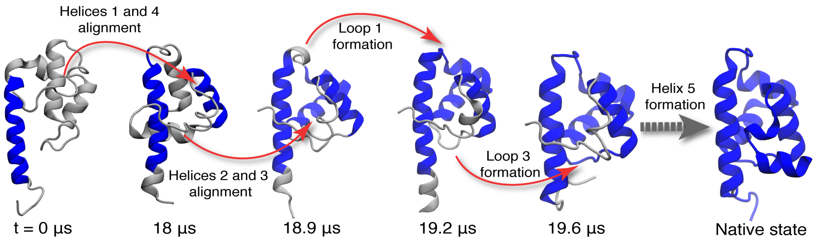
Specifying an order in which to assemble the parts of some object.

- If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done.



Real-world Problems: e.g. Protein Design[8]

The goal is to find a sequence of amino acids that will fold into a 3D protein with the right properties to cure some disease.



[8] image source: http://www.ks.uiuc.edu/Research/folding/

Outline

- Problem-Solving Agents
- Example Problems
- Searching for Solutions

Searching for Solutions

A solution is an action sequence, so search algorithms work by considering various possible action sequences.

- The possible **action** sequences starting at the **initial state** form a search tree with the initial state at the root.
- The branches are **actions** and the nodes correspond to states in the state space of the problem.
- The root node refers to the initial state.

The set of all leaf nodes available for expansion at any given point is called the **frontier** (open list) while already expanded nodes can be kept in the **explored set** (closed list).

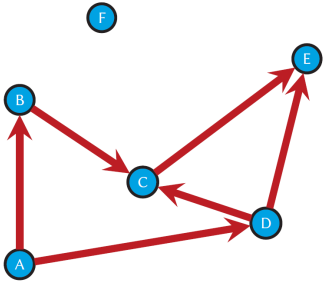
Graphs[9]

A **graph** G is a collection of (V, E) pairs where

- V: a set of vertices or nodes
- E: a set of edges connecting the vertices

[9] adapted from the slides of CS 5002: Discrete Math – Northeastern University: `https://course.ccs.neu.edu/cs5002f18-seattle/`

Graphs

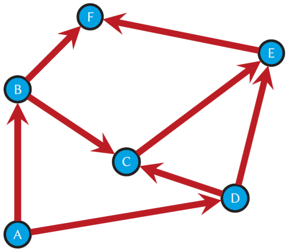


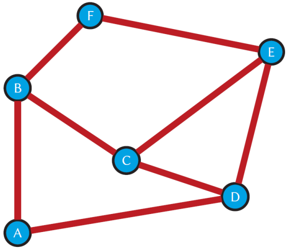
- Vertices: $V = \{A, B, C, D, E, F\}$
- Edges: $E = \{\ (A, B),$ (A, D), (B, C), (C, D), (C, E), $(D, E)\ \}$

Graphs – Types

- Directed vs. Undirected
- Labeled vs. Unlabeled
- Weighted vs. Unweighted
- Simple vs. Non-simple
- Sparse vs. Dense
- Cyclic vs. Acyclic

Graphs – Directed vs. Undirected

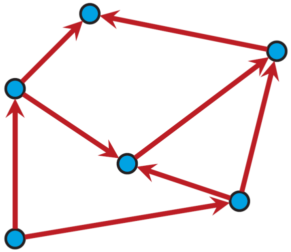




Undirected if edge (x, y) implies edge (y, x), otherwise Directed

- Roads between cities usually undirected (both ways)
- Streets in cities tend to be directed (one-way)

Graphs – Labeled[11] vs. Unlabeled



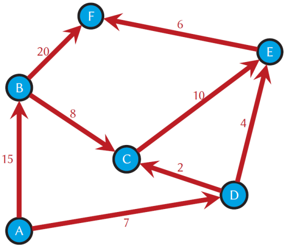
Each vertex is assigned a unique name or identifier in a Labeled graph, otherwise Unlabeled

- e.g. city names in a transportation network
- While labeled graphs mean vertex-labeled graphs, there are also edge-labeled graphs[10].

[10] https://en.wikipedia.org/wiki/Graph_labeling

[11] http://mathworld.wolfram.com/LabeledGraph.html

Graphs – Weighted vs. Unweighted

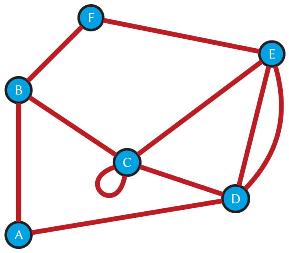


Weighted (a special type of labeled graphs – vertex-labeled) if each edge or vertex is assigned to a numerical value (weight), otherwise Unweighted

- A road network might be weighted with length, drive-time and speed-limit
- Streets in cities tend to be directed (one-way)

Traditionally, weighted graphs mean edge-weighted graphs. Yet, vertex-weighted graphs are also present.

Graphs – Simple[12] vs. Non-Simple

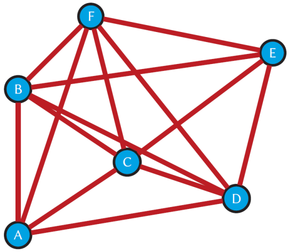


A simple (strict) graph is an unweighted, undirected without loops or multiple (parallel) edges

- A (self-)loop is an edge (x, x) on a vertex
- Multiple edges connect the same vertices (x, y)

[12] http://mathworld.wolfram.com/SimpleGraph.html

Graphs – Sparse vs. Dense

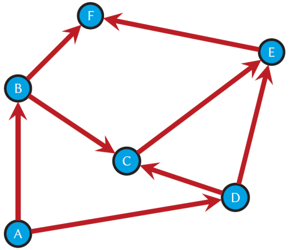


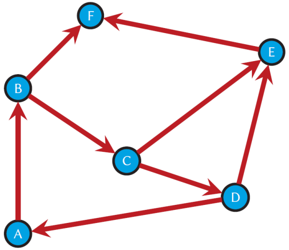
Graphs are Dense when a large fraction of vertex pairs have edges (close to the maximal number of edges), otherwise Sparse

- No formal distinction between two types, yet there is ratio of graph density[13] to quantify the level of density.

[13] the number of edges divided by the maximum number of edges: $\alpha|E|/(|V|(|V|-1))$ where $\alpha = 1$ for directed, $\alpha = 2$ for undirected, the maximum number of edges for an undirected graph is $|V|(|V|-1)/2$ – https://en.wikipedia.org/wiki/Dense_graph

Graphs – Cyclic[14] vs. Acyclic





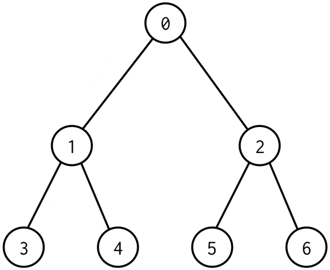
Graphs containing at least one cycle are Cylic, otherwise Acyclic

[14] http://mathworld.wolfram.com/CyclicGraph.html

Graphs vs. Trees[17]

Trees[15] are connected, acyclic and undirected graphs

- Although they are undirected, it is possible to see their directed variants[16]
- Directions can also be placed just to emphasize the parent-child relationships.



15 image source: http://mishadoff.com/blog/dfs-on-binary-tree-array/

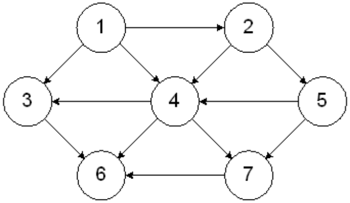
16 https://en.wikipedia.org/wiki/Tree_(graph_theory)

17 https://www.cs.cornell.edu/courses/cs312/2003fa/lectures/lec08.htm

Graph Representations[18] – Adjacency Matrix[19]

where maintain a V-by-V boolean / binary array, with the entry in row v and column w defined to be true if there is an edge in the graph that connects vertex v and vertex w, and to be false otherwise.

e.g. directed graph



	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[1]	0	1	1	1	0	0	0
[2]	0	0	0	1	1	0	0
[3]	0	0	0	0	0	1	0
[4]	0	0	1	0	0	1	1
[5]	0	0	0	1	0	0	1
[6]	0	0	0	0	0	0	0
[7]	0	0	0	0	0	1	0

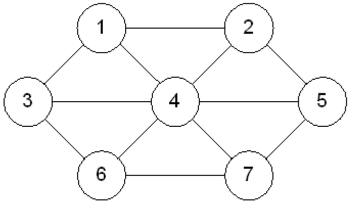
[18] Algorithms (4th Ed.) by Robert Sedgewick and Kevin Wayne, 2011 Addison-Wesley – https://algs4.cs.princeton.edu/home/

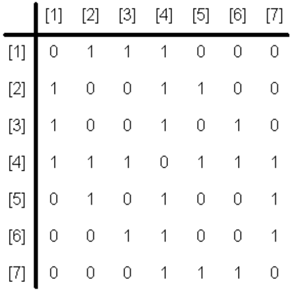
[19] example source: http://faculty.cs.niu.edu/~freedman/340/340notes/340graph.htm

Graph Representations – Adjacency Matrix[20]

where maintain a V-by-V boolean / binary array, with the entry in row v and column w defined to be true if there is an edge in the graph that connects vertex v and vertex w, and to be false otherwise.

e.g. undirected graph





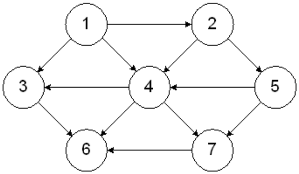
	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[1]	0	1	1	1	0	0	0
[2]	1	0	0	1	1	0	0
[3]	1	0	0	1	0	1	0
[4]	1	1	1	0	1	1	1
[5]	0	1	0	1	0	0	1
[6]	0	0	1	1	0	0	1
[7]	0	0	0	1	1	1	0

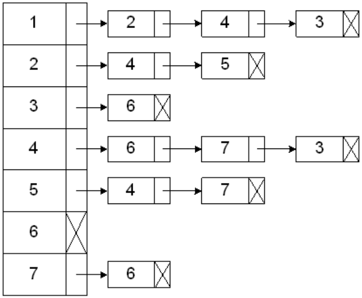
[20] example source: http://faculty.cs.niu.edu/~freedman/340/340notes/340graph.htm

Graph Representations – Adjacency List[21]

Maintain a vertex-indexed array of lists of the vertices adjacent to each vertex.

e.g. directed graph (only outgoing connections)



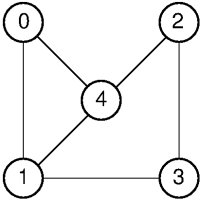


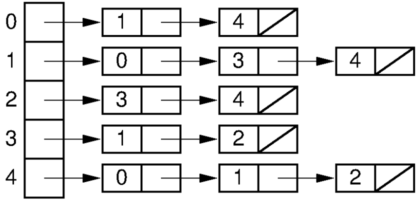
[21] example source: http://faculty.cs.niu.edu/~freedman/340/340notes/340graph.htm

Graph Representations – Adjacency List[22]

Maintain a vertex-indexed array of lists of the vertices adjacent to each vertex.

e.g. undirected graph

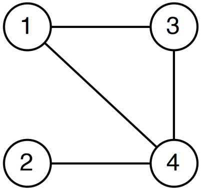


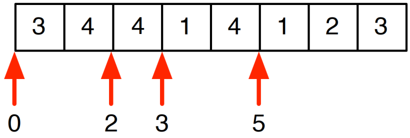


[22] example source: https://taylor.git-pages.mst.edu/index_files/Bioinformatics/Content/23-GraphTheory.html

Graph Representations – Others[23]

Edge Array: an adjacency array keeps the neighbors of all vertices, one after another; and separately, keeps an array of indices that tell us where in the adjarray to look for the neighbors of each vertex.





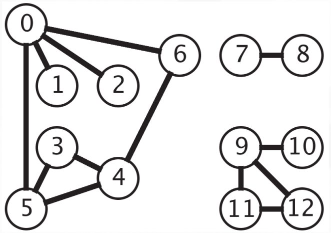
Edge List: A list of pairs $(i, j) \in E$.

[23] http://www.cs.cmu.edu/afs/cs/academic/class/15210-s12/www/lectures/lecture07.pdf

Graph Representation

TASK :

1. List the advantages and disadvantages between Adjacency Matrix and List for both Directed and Undirected graphs.
2. Show the Adjacency Matrix and List for the following graph.

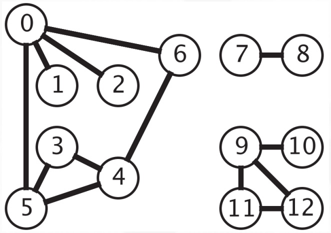


Submit photo of your answer to **Piazza** as a *private message*.
Also, deliver its hard copy.

Graph Representation

TASK : Implement two graph representations including Adjacency Matrix and Adjacency List.

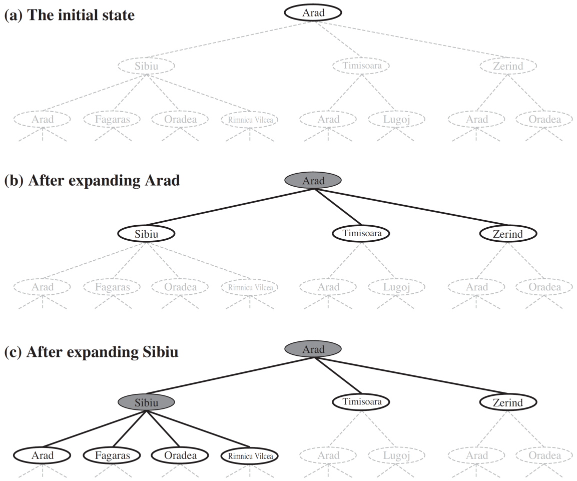
- With the functionalities of add and remove node



Submit your code to **Piazza** as a *private message*.

Searching

Partial search trees for finding a route from Arad to Bucharest.



Searching – Tree Search

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      expand the chosen node, adding the resulting nodes to the frontier
```

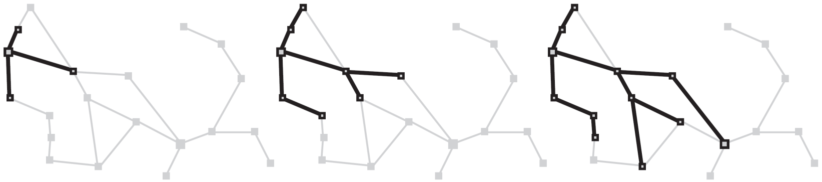
Searching – Graph Search

```
**function** GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  ***initialize the explored set to be empty***
  **loop do**
      **if** the frontier is empty **then return** failure
      choose a leaf node and remove it from the frontier
      **if** the node contains a goal state **then return** the corresponding solution
      ***add the node to the explored set***
      expand the chosen node, adding the resulting nodes to the frontier
        ***only if not in the frontier or explored set***
```

Searching – Graph Search

A sequence of search trees generated by a graph search on the Romania problem.

- At each stage, we have extended each path by one step.



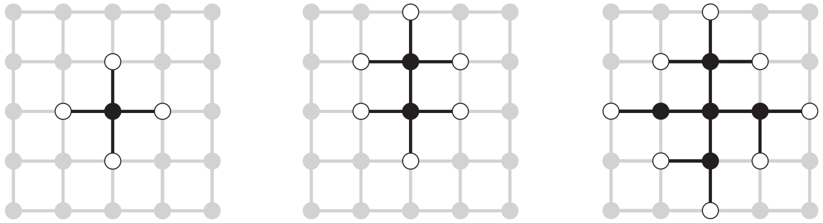
Searching – Graph Search

The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes).

(a) just the root has been expanded

(b) one leaf node has been expanded

(c) the remaining successors of the root have been expanded in clockwise order

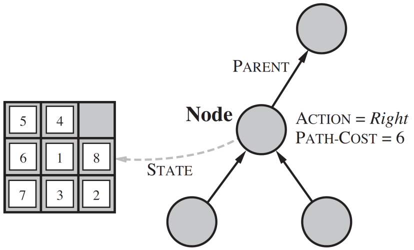


(a) (b) (c)

Search Algorithms – Infrastructure

For each node n of the tree, we have a structure that contains 4 components:

- n.STATE: the state in the state space the node corresponds;
- n.PARENT: the node in the tree generated this node;
- n.ACTION: the action applied to the parent to generate the node;
- n.PATH-COST: the cost, $g(n)$, of the path from the initial state to the node.



Use the SOLUTION function to return the sequence of actions obtained by following parent pointers back to the root.

Search Algorithms – Infrastructure

The function CHILD-NODE takes a **parent node** and an **action** and returns the resulting **child node**:

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Search Algorithms – Infrastructure

Now that we have nodes, we need somewhere to put them. The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy.

The appropriate data structure for this is a **queue**, here are its operations:

- EMPTY?(queue) returns true only if there are no more elements in the queue.
- POP(queue) removes the first element of the queue and returns it.
- INSERT(element, queue) inserts an element and returns the resulting queue.

Search Algorithms – Infrastructure

Queues are characterized by the order in which they store the inserted nodes. 3 common variants are:

- the first-in, first-out (**FIFO**) queue: pops the oldest element of the queue
- the last-in, first-out (**LIFO**) queue (a.k.a. stack): pops the newest element of the queue;
- the **priority** queue: pops the element of the queue with the highest priority according to some ordering function.

Search Algorithms – Performance Measure

We can evaluate an algorithm's performance in 4 ways:

- **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- **Optimality**: Does the strategy find the optimal solution?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

Graphs – Glossary[24]

- A self-loop is an edge that connects a vertex to itself.
- Two edges are parallel if they connect the same pair of vertices.
- When an edge connects two vertices, we say that the vertices are adjacent to one another and that the edge is incident on both vertices.
- The degree of a vertex is the number of edges incident on it.
- A subgraph is a subset of a graph's edges (and associated vertices) that constitutes a graph.
- A path in a graph is a sequence of vertices connected by edges, with no repeated edges.
- A simple path is a path with no repeated vertices.
- A cycle is a path (with at least one edge) whose first and last vertices are the same.
- A simple cycle is a cycle with no repeated vertices (other than the requisite repetition of the first and last vertices).
- The length of a path or a cycle is its number of edges.
- We say that one vertex is connected to another if there exists a path that contains both of them.
- A graph is connected if there is a path from every vertex to every other vertex.
- A graph that is not connected consists of a set of connected components, which are maximal connected subgraphs.
- An acyclic graph is a graph with no cycles.
- A tree is an acyclic connected graph.
- A forest is a disjoint set of trees.
- A spanning tree of a connected graph is a subgraph that contains all of that graph's vertices and is a single tree. A spanning forest of a graph is the union of the spanning trees of its connected components.
- A bipartite graph is a graph whose vertices we can divide into two sets such that all edges connect a vertex in one set with a vertex in the other set.

[24] https://algs4.cs.princeton.edu/41graph/