Graph ADT

Graphs are a formalism for representing relationships between objects:
- A graph \( G \) is represented as \( G = (V, E) \)
  - \( V \) is a set of vertices
  - \( E \) is a set of edges
- Operations include:
  - Iterating over vertices
  - Iterating over edges
  - Iterating over vertices adjacent to a specific vertex
  - Asking whether an edge exists connected two vertices

Graphs In Practice

- Web graph
  - Vertices are web pages
  - Edge from \( u \) to \( v \) is a link to \( v \) appears on \( u \)
- Call graph of a computer program
  - Vertices are functions
  - Edge from \( u \) to \( v \) is \( u \) calls \( v \)
- Task graph for a work flow
  - Vertices are tasks
  - Edge from \( u \) to \( v \) if \( u \) must be completed before \( v \) begins

Graph Representation 1: Adjacency Matrix

A \(|V| \times |V|\) array in which an element \((u, v)\) is true if and only if there is an edge from \( u \) to \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Han</th>
<th>Luke</th>
<th>Leia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Han</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Luke</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leia</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Runtime:
- Iterate over vertices \( O(|V|) \)
- Iterate over edges \( O(|V|^2) \)
- Ask edge exists? \( O(1) \)

Space required: \( O(|V|^2) \)

Graph Representation 2: Adjacency List

A \(|V|\)-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices

<table>
<thead>
<tr>
<th></th>
<th>Han</th>
<th>Luke</th>
<th>Leia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Han</td>
<td></td>
<td>Leia</td>
<td></td>
</tr>
<tr>
<td>Luke</td>
<td></td>
<td>Leia</td>
<td></td>
</tr>
<tr>
<td>Leia</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Runtime:
- Iterate over vertices \( O(|V|) \)
- Iterate over edges \( O(|E|) \)
- Iterate edges adj. to vertex \( O(|V|) \)
- Ask edge exists? \( O(d) \)

Space required: \( O(|V| + |E|) \)
### Terminology

- In **directed** graphs, edges have a specific direction.
- In **undirected** graphs, edges are two-way.
- Vertices $u$ and $v$ are **adjacent** if $(u, v) \in E$.
- A **sparse** graph has $O(|V|)$ edges (upper bound).
- A **dense** graph has $\Omega(|V|^2)$ edges (lower bound).
- A **complete** graph has an edge between every pair of vertices.
- An **undirected graph** is **connected** if there is a path between any two vertices.

### Weighted Graphs

Each edge has an associated weight or cost.

- **Clinton** to **Mukilteo**: 20
- **Kingston** to **Edmonds**: 30
- **Bainbridge** to **Seattle**: 35
- **Bremerton**

### Paths and Cycles

A **path** is a list of vertices $\{v_0, v_1, \ldots, v_n\}$ such that $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$.

A **cycle** is a path that begins and ends at the same node.

- **Seattle**
- **San Francisco**
- **Salt Lake City**
- **Chicago**
- **Dallas**

$p = \langle Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle \rangle$

### Path Length and Cost

- **Path length**: the number of edges in the path.
- **Path cost**: the sum of the costs of each edge.

- **Seattle** to **Chicago**:
  - **Seattle** to **Salt Lake City**: 2
  - **Salt Lake City** to **Chicago**: 3.5

\[ \text{length}(p) = 5 \]

\[ \text{cost}(p) = 11.5 \]

### Trees as Graphs

Every tree is a graph with some restrictions:

- The tree is **directed**.
- There are no cycles (directed or undirected).
- There is a directed path from the root to every node.

### Directed Acyclic Graphs (DAGs)

**DAGs** are directed graphs with no cycles.

\[ \text{program call graph} \]

Trees $\subset$ DAGs $\subset$ Graphs
Topological Sort

Given a directed graph, $G = (V, E)$, output all the vertices in $V$ such that no vertex is output before any other vertex with an edge to it.

**Example**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Queue = 1 2 5
Output = 5

**Example**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Queue = 1 2
Output = 5

**Example**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Queue = 6
Output = 5 2 1

**Topological Sort**

Label each vertex’s in-degree
Initialize a queue to contain all in-degree zero vertices
While there are vertices remaining in the queue
Remove a vertex $v$ with in-degree of zero and output it
Reduce the in-degree of all vertices adjacent to $v$
Put any of these with new in-degree zero on the queue

Runtime: $O(|V| + |E|)$
Example

Queue = 3 7
Output = 5 2 1 6

In-degree

Example

Queue = 3
Output = 5 2 1 6 7

In-degree

Example

Queue = 4
Output = 5 2 1 6 3

In-degree

Example

Queue = 3 7
Output = 5 2 1 6

In-degree

Exercise

Design the algorithm to initialize the in-degree array. Assume the adjacency list representation.

Graph Search

Many problems in computer science correspond to searching for a path in a graph, given a start node and goal criteria
- Route planning – Mapquest
- Packet-switching
- VLSI layout
- 6-degrees of Kevin Bacon
- Program synthesis
- Speech recognition
  - We’ll discuss these last two later...

General Graph Search Algorithm

Open – some data structure (e.g., stack, queue, heap)
Criteria – some method for removing an element from Open
Search( Start, Goal_test, Criteria)
insert(Start, Open);
repeat
  if (empty(Open)) then return fail;
  select Node from Open using Criteria;
  if (Goal_test(Node)) then return Node;
  for each Child of node do
    if (Child not already visited) then insert( Child, Open );
    Mark Node as visited;
end
**Depth-First Graph Search**

- **Open – Stack**
- **Criteria – Pop**

```
DFS( Start, Goal_test)
push(Start, Open);
repeat
  if (empty(Open)) then return fail;
  Node := pop(Open);
  if (Goal_test(Node)) then return Node;
  for each Child of node do
    if (Child not already visited) then push(Child, Open);
    Mark Node as visited;
end
```

**Breadth-First Graph Search**

- **Open – Queue**
- **Criteria – Dequeue (FIFO)**

```
BFS( Start, Goal_test)
enqueue(Start, Open);
repeat
  if (empty(Open)) then return fail;
  Node := dequeue(Open);
  if (Goal_test(Node)) then return Node;
  for each Child of node do
    if (Child not already visited) then enqueue(Child, Open);
    Mark Node as visited;
end
```

**BFS - Example**

```
1 2 3 4
6 5
7
```

QUEUE = 1

**DFS - Example**

```
1 2 3 4
6 5
7
```

STACK = 1

**Two Models**

1. **Standard Model:** Graph given explicitly with n vertices and e edges.
   - Search is O(n + e) time in adjacency list representation
2. **AI Model:** Graph generated on the fly.
   - Time for search need not visit every vertex.

**Planning Example**

A huge graph may be implicitly specified by rules for generating it on-the-fly

Blocks world:
- vertex = relative positions of all blocks
- edge = robot arm stacks one block
**AI Comparison: DFS versus BFS**

Depth-first search
- Does not always find shortest paths
- Must be careful to mark visited vertices, or you could go into an infinite loop if there is a cycle

Breadth-first search
- Always finds shortest paths – optimal solutions
- Marking visited nodes can improve efficiency, but even without doing so search is guaranteed to terminate

*Is BFS always preferable?*

---

**DFS Space Requirements**

Assume:
- Longest path in graph is length $d$
- Highest number of out-edges is $k$

DFS stack grows at most to size $dk$
- For $k=10$, $d=15$, size is 150

---

**BFS Space Requirements**

Assume
- Distance from start to a goal is $d$
- Highest number of out edges is $k$ BFS

Queue could grow to size $k^d$
- For $k=10$, $d=15$, size is $1,000,000,000,000,000$

---

**Conclusion**

In the AI Model, DFS is hugely more memory efficient, *if we can limit the maximum path length to some fixed $d$.*

- If we knew the distance from the start to the goal in advance, we can just not add any children to stack after level $d$
- But what if we don’t know $d$ in advance?

---

**Recursive Depth-First Search**

DFS(v: vertex)
mark v;
for each vertex w adjacent to v do
if w is unmarked then DFS(w)

Note: the recursion has the same effect as a stack

---

**Finding Connected Components**

For each vertex v do mark(v) = 0;
$C := 1$.
For each vertex v do
if mark(v) = 0 then
    dfs(v); $C \leftarrow C+1$;
    dfs(v: vertex)
    mark[v] := C;
    for each vertex w adjacent to v do
    if mark[w] = 0 then dfs(w)

All those vertices with the same mark are in the same connected component
For undirected graphs, each edge appears twice on the adjacency lists.

Our pseudocode returns the goal node found, but not the path to it.
How can we remember the path?
- Add a field to each node, that points to the previous node along the path
- Follow pointers from goal back to start to recover path

Example

Example (Unweighted Graph)

Example (Unweighted Graph)

Graph Search, Saving Path

Search( Start, Goal_test, Criteria)
insert(Start, Open);
repeat
  if (empty(Open)) then return fail;
  select Node from Open using Criteria;
  if (Goal_test(Node)) then return Node;
  for each Child of node do
    if (Child not already visited) then
      Child.previous := Node;
      Insert( Child, Open );
      Mark Node as visited;
  end
end
Shortest Path for Weighted Graphs

Given a graph $G = (V, E)$ with edge costs $c(e)$, and a vertex $s \in V$, find the shortest (lowest cost) path from $s$ to every vertex in $V$

Assume: only positive edge costs

---

Edsger Wybe Dijkstra
(1930-2002)

- Invented concepts of structured programming, synchronization, weakest precondition, and "semaphores" for controlling computer processes. The Oxford English Dictionary cites his use of the words "vector" and "stack" in a computing context.
- Believed programming should be taught without computers
- 1972 Turing Award
- "In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind."

---

Dijkstra's Algorithm for Single Source Shortest Path

Similar to breadth-first search, but uses a heap instead of a queue:
- Always select (expand) the vertex that has a lowest-cost path to the start vertex
- Correctly handles the case where the lowest-cost (shortest) path to a vertex is not the one with fewest edges

---

Pseudocode for Dijkstra

```
Initialize the cost of each node to $\infty$
insert(s, 0, heap);
While (! empty(heap))
    n := deleteMin(heap);
    For each edge e=(n,a) do
        if (n.cost + e.cost < a.cost) then
            a.cost = n.cost + e.cost;
            a.previous = n;
            if (a is in the heap) then
                decreaseKey(a, a.cost, heap)
            else
                insert(a, a.cost, heap)
    end
```

---

Important Features

Once a vertex is removed from the head, the cost of the shortest path to that node is known
While a vertex is still in the heap, another shorter path to it might still be found
The shortest path itself can be found by following the backward pointers stored in node.previous

---

Dijkstra's Algorithm in Action

---
Dijkstra's Algorithm in Action

Data Structures for Dijkstra's Algorithm

Problem: Large Graphs

Example

Best-First Search

The Manhattan distance ($\Delta x + \Delta y$) is an estimate of the distance to the goal

- It is a search heuristic
- Best-First Search
  - Order nodes in priority to minimize estimated distance to the goal
- Compare: BFS / Dijkstra
  - Order nodes in priority to minimize distance from the start
Best-First Search

Open – Heap (priority queue)
Criteria – Smallest key (highest priority)
h(n) – heuristic estimate of distance from n to closest goal

Best_First_Search(Start, Goal_test)
insert(Start, h(Start), heap);
repeat
  if (empty(heap)) then return fail;
  Node := deleteMin(heap);
  if (Goal_test(Node)) then return Node;
  for each Child of node do
    if (Child not already visited) then
      insert(Child, h(Child), heap);
  end
  Mark Node as visited;
end

Obstacles

Best-FS eventually will expand vertex to get back on the right track

Non-Optimality of Best-First

Path found by Best-first

Improving Best-First

Best-first is often tremendously faster than BFS/Dijkstra, but might stop with a non-optimal solution
How can it be modified to be (almost) as fast, but guaranteed to find optimal solutions?
• One of the first significant algorithms developed in AI
• Widely used in many applications

A*

Exactly like Best-first search, but using a different criteria for the priority queue:
minimize (distance from start) + (estimated distance to goal)

priority f(n) = g(n) + h(n)
  f(n) = priority of a node
  g(n) = true distance from start
  h(n) = heuristic distance to goal

Optimality of A*

Suppose the estimated distance is always less than or equal to the true distance to the goal
  • heuristic is a lower bound

Then: when the goal is removed from the priority queue, we are guaranteed to have found a shortest path!
**A* in Action**

A huge graph may be implicitly specified by rules for generating it on-the-fly.

**Blocks World**

Blocks world:
- distance = number of stacks to perform
- heuristic lower bound = number of blocks out of place

# out of place = 2, true distance to goal = 3

**Applications of A*: Planning**

(Simplified) Problem:
- System hears a sequence of 3 words
- It is unsure about what it heard
  - For each word, it has a set of possible “guesses”
  - E.g.: Word 1 is one of {“hi”, “high”, “I”}
- What is the most likely sentence it heard?

**Speech Recognition as Shortest Path**

Convert to a shortest-path problem:
- Utterance is a “layered” DAG
- Begins with a special dummy “start” node
- Next: A layer of nodes for each word position, one node for each word choice
- Edges between every node in layer i to every node in layer i+1
  - Cost of an edge is smaller if the pair of words frequently occur together in real speech
    - Technically: log probability of co-occurrence
- Finally: a dummy “end” node
- Find shortest path from start to end node
<table>
<thead>
<tr>
<th>Summary: Graph Search</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Depth First</strong></td>
</tr>
<tr>
<td>• Little memory required</td>
</tr>
<tr>
<td>• Might find non-optimal path</td>
</tr>
<tr>
<td><strong>Breadth First</strong></td>
</tr>
<tr>
<td>• Much memory required</td>
</tr>
<tr>
<td>• Always finds optimal path</td>
</tr>
<tr>
<td><strong>Dijkstra’s Short Path Algorithm</strong></td>
</tr>
<tr>
<td>• Like BFS for weighted graphs</td>
</tr>
<tr>
<td><strong>Best First</strong></td>
</tr>
<tr>
<td>• Can visit fewer nodes</td>
</tr>
<tr>
<td>• Might find non-optimal path</td>
</tr>
<tr>
<td><strong>A</strong></td>
</tr>
<tr>
<td>• Can visit fewer nodes than BFS or Dijkstra</td>
</tr>
<tr>
<td>• Optimal if heuristic estimate is a lower-bound</td>
</tr>
</tbody>
</table>