CSE 326: Data Structures

Graph Algorithms
Graph Search

Lecture 13

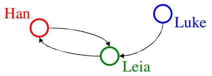
Reading

Chapter 9.1, 9.2, 9.3

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



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

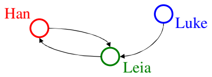
- operations include:
  - iterating over vertices
  - iterating over edges
  - iterating over vertices adjacent to a specific vertex
  - asking whether an edge exists connected two vertices

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



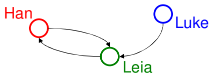
	Han 0	Luke 1	Leia 2
Han 0	0	0	1
Luke 1	0	0	1
Leia 2	1	0	0

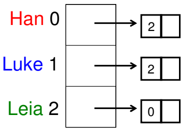
Runtime:
iterate over vertices $O(|v|)$
iterate ever edges $O(|v|^2)$
iterate edges adj. to vertex $O(|v|)$
edge exists? $O(1)$

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





Runtime:
iterate over vertices $O(v)$
iterate ever edges $O(|e|)$
iterate edges adj. to vertex $O(d)$ (d is number of adj. vertices)
edge exists? $O(d)$

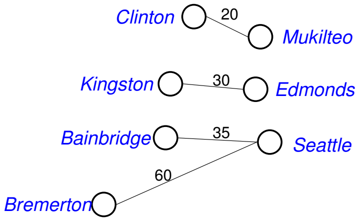
Space required: $O(|v| + |e|)$

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

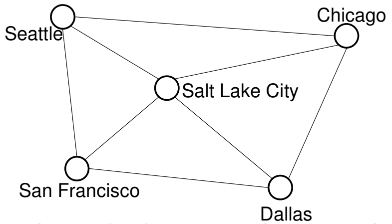
Each edge has an associated weight or cost.



Paths and Cycles

A *path* is a list of vertices $\{v_1, v_2, \ldots, v_n\}$ 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.

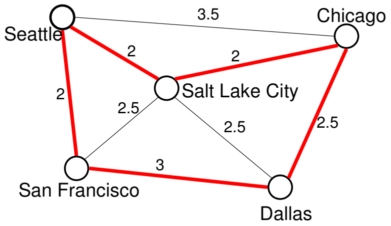


p = {Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle}

Path Length and Cost

Path length: the number of edges in the path

Path cost: the sum of the costs of each edge



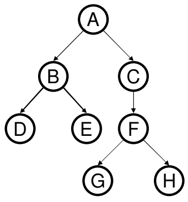
length(p) = 5

cost(p) = 11.5

Trees as Graphs

Every tree is a graph with some restrictions:

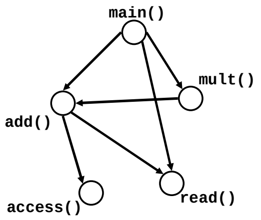
- the tree is *directed*
- there are *no cycles* (directed or undirected)
- there is a *directed path from the root to every node*



Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no cycles.

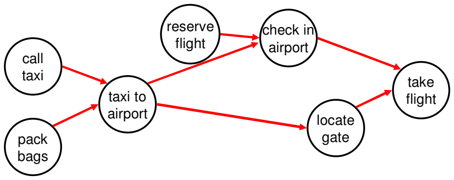
program call graph



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.

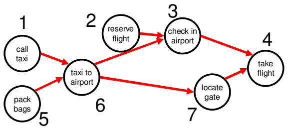


Topological Sort

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

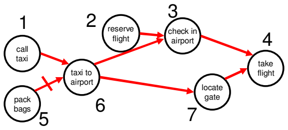


In-degree

1 0 → 6
2 0 → 3
3 2 → 4
4 2
5 0 → 6
6 2 → 3 → 7
7 1 → 4

Queue = 1 2 5
Output =

Example

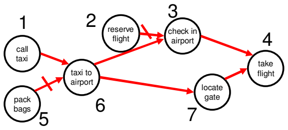


In-degree

1 0 → 6
2 0 → 3
3 2 → 4
4 2
5 0 → 6
6 1 → 3 → 7
7 1 → 4

Queue = 1 2
Output = 5

Example

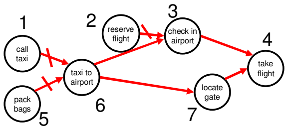


In-degree

1 0 → 6
2 0 → 3
3 1 → 4
4 2
5 0 → 6
6 1 → 3 → 7
7 1 → 4

Queue = 1
Output = 5 2

Example

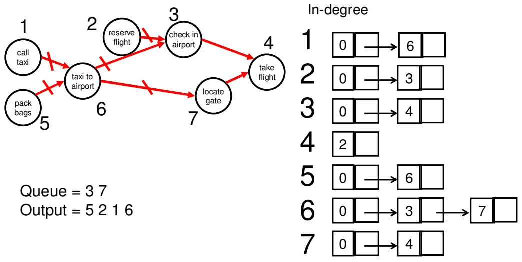


In-degree

1 0 → 6
2 0 → 3
3 1 → 4
4 2
5 0 → 6
6 0 → 3 → 7
7 1 → 4

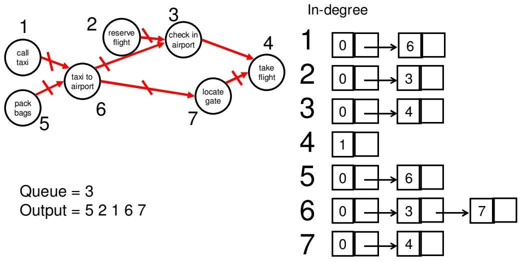
Queue = 6
Output = 5 2 1

Example



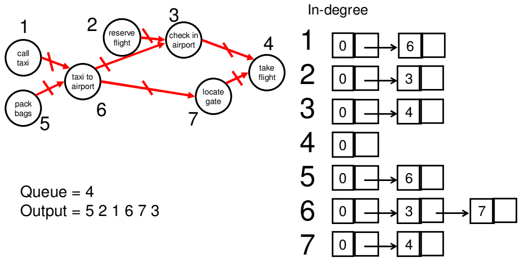
Queue = 3 7
Output = 5 2 1 6

Example



Queue = 3
Output = 5 2 1 6 7

Example



Queue = 4
Output = 5 2 1 6 7 3

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;
    Mark Node as visited;
    if (Goal_test(Node)) then return Node;
```

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);
      Mark Node as visited;
      if (Goal_test(Node)) then return Node;
```

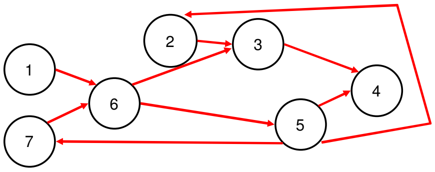
Breadth-First Graph Search

Open – Queue

Criteria – Dequeue (FIFO)

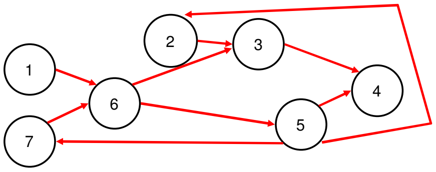
```
BFS( Start, Goal_test)
   enqueue(Start, Open);
   repeat
       if (empty(Open)) then return fail;
       Node := dequeue(Open);
       Mark Node as visited;
       if (Goal_test(Node)) then return Node;
       for each Child of node do
           if (Child not already visited) then enqueue(Child,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

BFS - Example



QUEUE = 1

DFS - Example



STACK = 1

Two Models

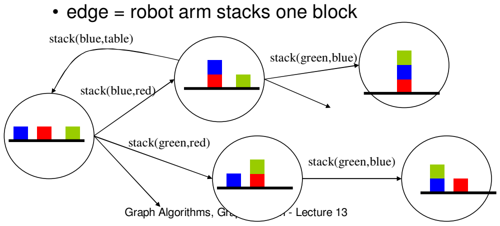
1. Standard Model: Graph given explicitly with n vertices and e edges.
   - q Search is O(n + e) time in adjacency list representation
2. AI Model: Graph generated on the fly.
   - q Time for search need not visit every vertex.

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

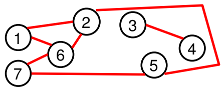
Finding Connected Components

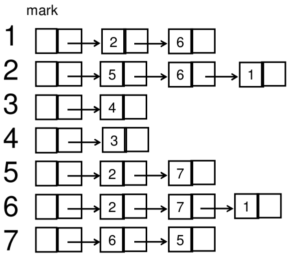
```
For each vertex v do mark[v]= 0;
C := 1.
For each vertex v do
  if mark[v] = 0 then
    dfs(v); C := C+1;

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

Example





For undirected graphs, each edge appears twice on the adjacency lists.

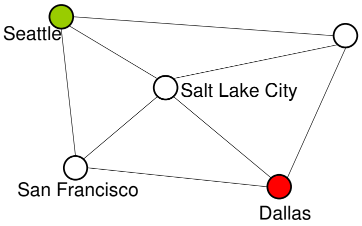
Saving the Path

Our pseudocode returns the goal node found, but not the path to it

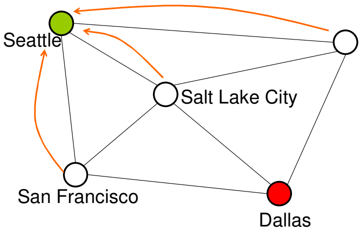
How can we remember the path?

- 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

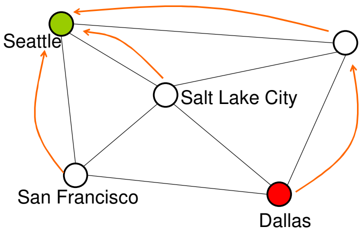
Example



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;
      Mark Node as visited;
      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 );
    end
```

Shortest Path for Weighted Graphs

Given a graph **`G = (V, E)`** with edge costs c(e), and a vertex **`s ∈ 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 ∞
s.cost := 0
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
end
```

Important Features

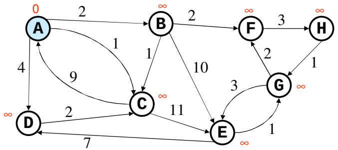
Once a vertex is removed from the heap, the cost of the shortest path to that node is known

While a vertex is still in the heap, another shorter path to it might still be found

The shortest path itself can found by following the backward pointers stored in node.previous

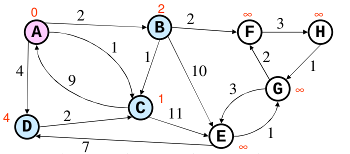
---

Dijkstra’s Algorithm in Action



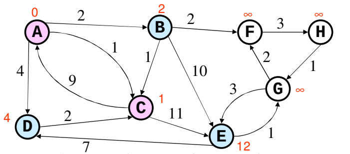
vertex	visited	cost
A		0
B		∞
C		∞
D		∞
E		∞
F		∞
G		∞
H		∞

Dijkstra's Algorithm in Action



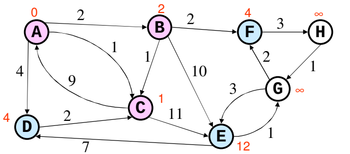
vertex	visited	cost
A	x	0
B		2
C		1
D		4
E		∞
F		∞
G		∞
H		∞

Dijkstra's Algorithm in Action



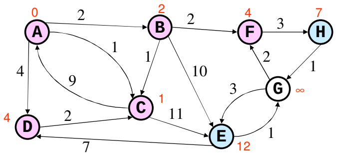
vertex	visited	cost
A	x	0
B		2
C	x	1
D		4
E		12
F		∞
G		∞
H		∞

Dijkstra's Algorithm in Action



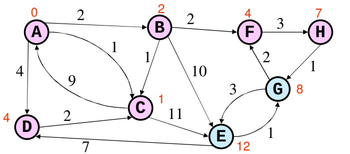
vertex	visited	cost
A	x	0
B	x	2
C	x	1
D		4
E		12
F		4
G		∞
H		∞

Dijkstra's Algorithm in Action



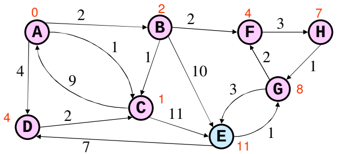
vertex	visited	cost
A	x	0
B	x	2
C	x	1
D	x	4
E		12
F	x	4
G		∞
H		7

Dijkstra's Algorithm in Action



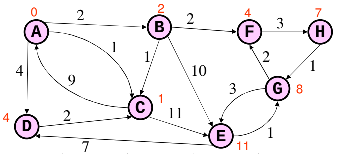
vertex	visited	cost
A	x	0
B	x	2
C	x	1
D	x	4
E		12
F	x	4
G		8
H	x	7

Dijkstra's Algorithm in Action



vertex	visited	cost
A	x	0
B	x	2
C	x	1
D	x	4
E		11
F	x	4
G	x	8
H	x	7

Dijkstra’s Algorithm in Action



vertex	visited	cost
A	x	0
B	x	2
C	x	1
D	x	4
E	x	11
F	x	4
G	x	8
H	x	7

Data Structures for Dijkstra’s Algorithm

|**V**| times:

Select the unknown node with the lowest cost

findMin/deleteMin O(log |V|)

|**E**| times:

a’s cost = min(*a*’s old cost, …)

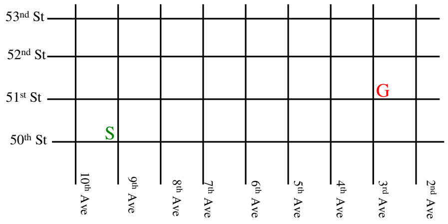
decreaseKey or insert O(log |V|)

runtime: O(|E| log |V|)

Problem: Large Graphs

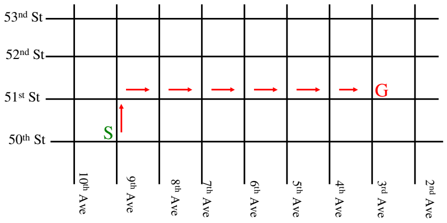
- It is expensive to find optimal paths in large graphs, using BFS or Dijkstra’s algorithm (for weighted graphs)
- How can we search large graphs efficiently by using “commonsense” about which direction looks most promising?

Example



Plan a route from 9th & 50th to 3rd & 51st

Example



Plan a route from 9th & 50th to 3rd & 51st

Best-First Search

The *Manhattan distance* (Δ x+ Δ 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

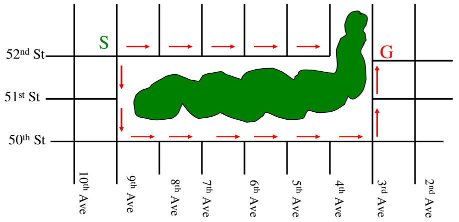
Best-First Search

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

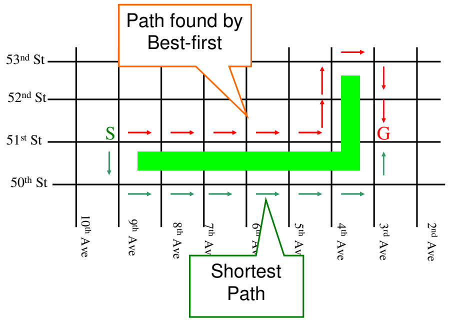
```
Best_First_Search( Start,
  Goal_test)
  insert(Start, h(Start), heap);
  repeat
    if (empty(heap)) then return fail;
    Node := deleteMin(heap);
    Mark Node as visited;
```

Obstacles

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



Non-Optimality of 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?
- A* - Hart, Nilsson, Raphael 1968
  - 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

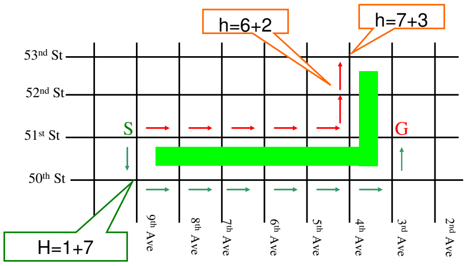
Optimality of A*

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

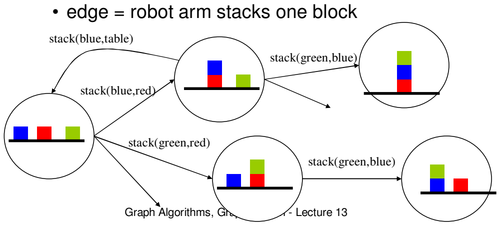


Applications of A*: Planning

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

Blocks world:

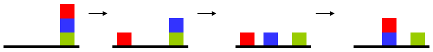
- vertex = relative positions of all blocks
- edge = robot arm stacks one block



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

Application of A*: Speech Recognition

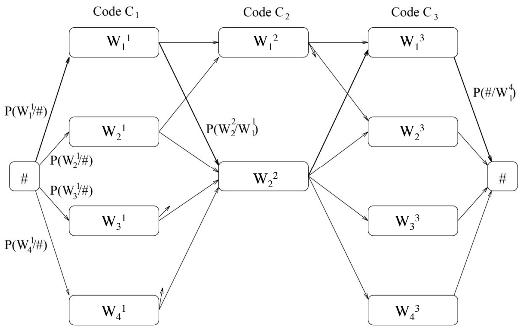
(Simplified) Problem:

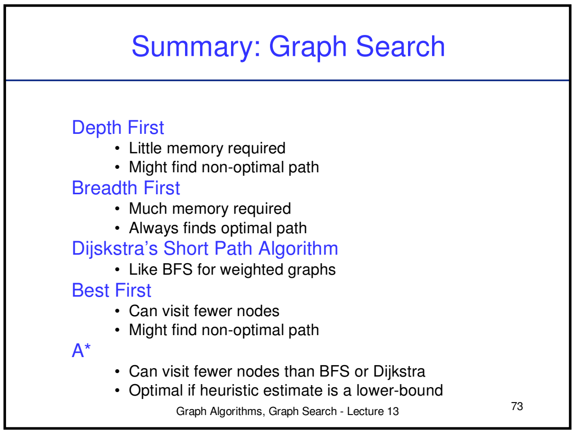
- System hears a sequence of 3 words
- It is unsure about what it heard
  - For each word, it has a set of possible "guesses"
  - E.g.: Word 1 is one of { "hi", "high", "I" }
- What is the most likely sentence it heard?

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





Summary: Graph Search

Depth First
- Little memory required
- Might find non-optimal path

Breadth First
- Much memory required
- Always finds optimal path

Dijskstra’s Short Path Algorithm
- Like BFS for weighted graphs

Best First
- Can visit fewer nodes
- Might find non-optimal path

A*
- Can visit fewer nodes than BFS or Dijkstra
- Optimal if heuristic estimate is a lower-bound