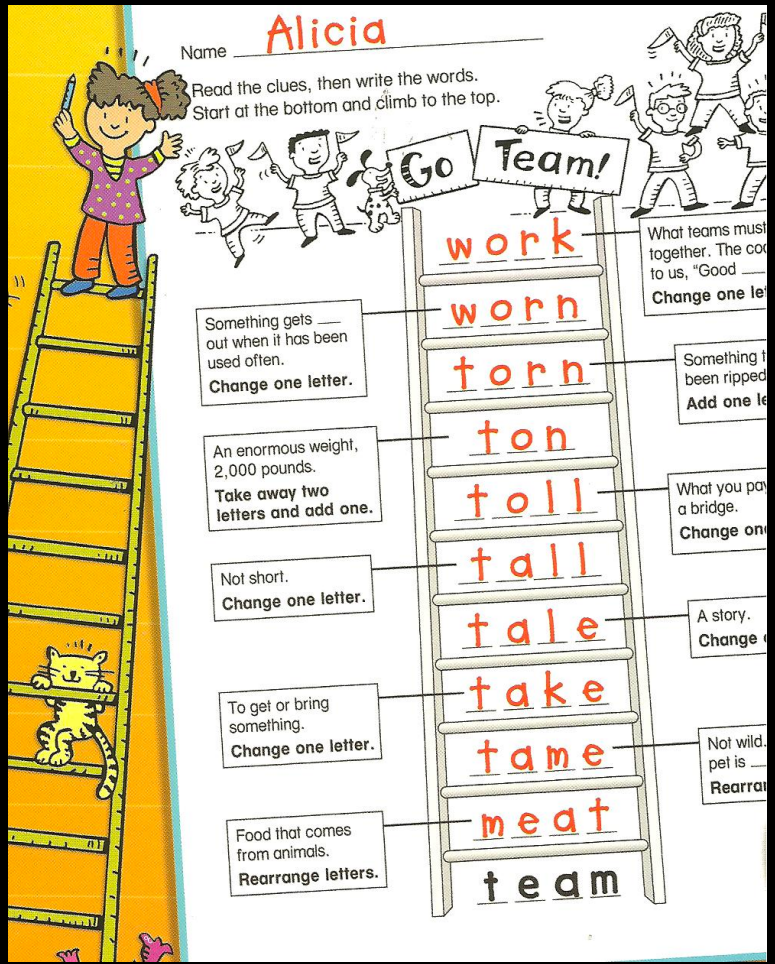
MODULE 02 – ALGORITHMS FOR STATISTICAL INFERENCE

GRAPH ALGORITHMS

HYUN MIN KANG @ UNIVERSITY OF MICHIGAN

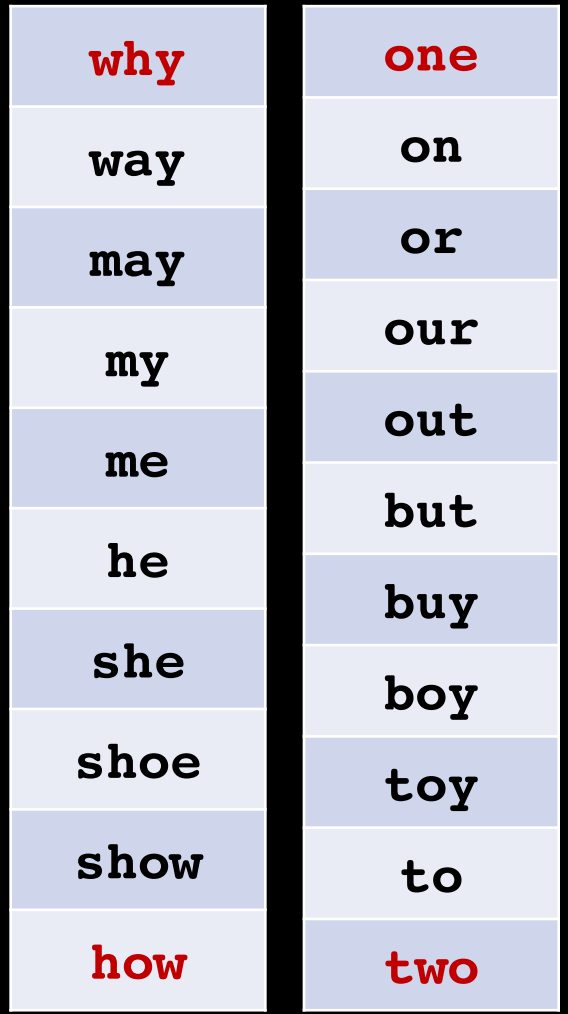
“Word Ladder”



- **A children’s puzzle to train vocabulary with fun.**
- **Slightly modify words from bottom to top.**
  - Substitute a letter
  - Add/remove a letter
  - Others (e.g. reverse)
- **Can we automatically generate word ladder problems?**

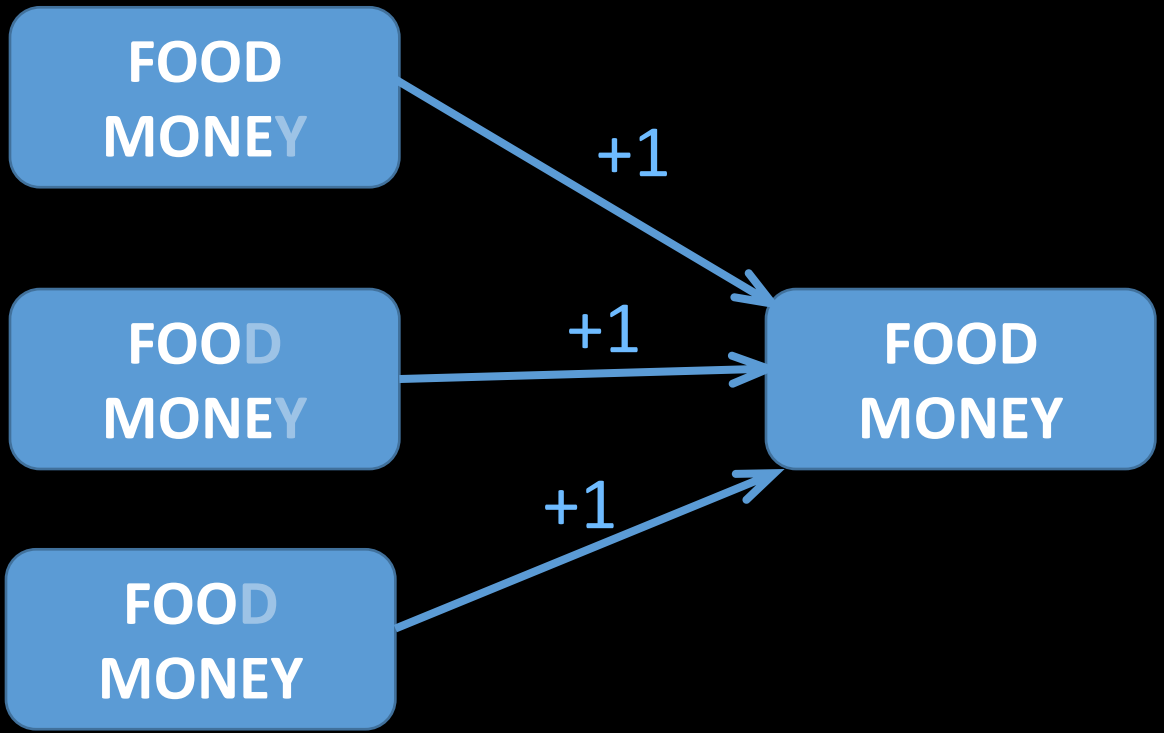
Creating "Word Ladders"

- **Start from a designated word list**
  - Dolch sight words (n=341)
  - **`/usr/share/dict/words`**
- **Allow 1-letter inserttion, deletion, substitution, and reversal.**
- **Find the ladder with shortest height.**
- **Can it be done in a similar way to the edit distance problem?**

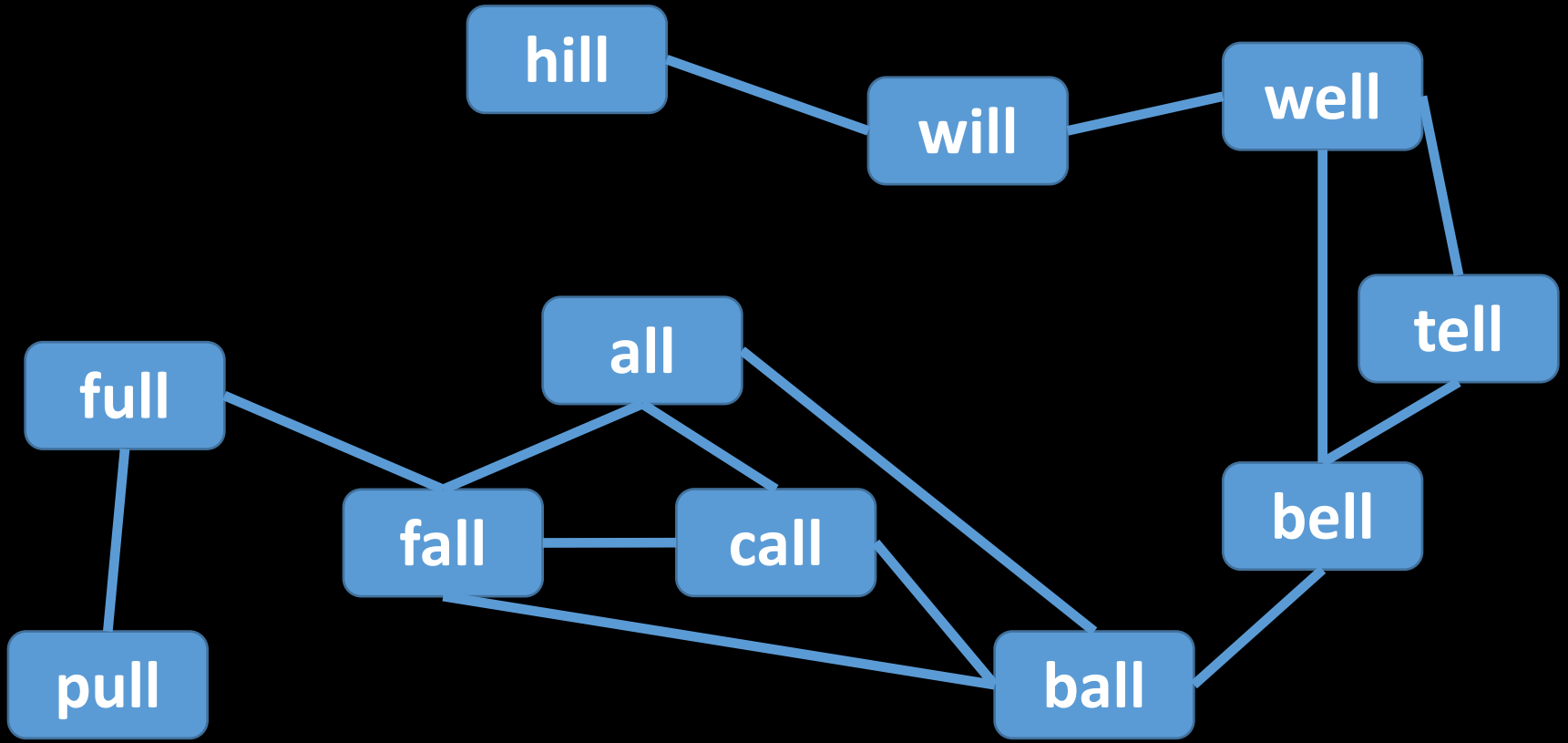


“Edit Distance” vs. “Word Ladder”

- What makes the two problems different?



A different approach : Use Graph



Example of 341 sight words

a	about	after	again	all	always	am	an	and
any	apple	are	around	as	ask	at	ate	away
baby	back	ball	be	bear	because	bed	been	before
bell	best	better	big	bird	birthday	black	blue	boat
both	box	boy	bread	bring	brother	brown	but	buy
by	cake	call	came	can	car	carry	cat	chair
chicken	children	Christmas	clean	coat	cold	come	corn	could
cow	cut	day	did	do	does	dog	doll	done
don't	door	down	draw	drink	duck	eat	egg	eight
every	eye	fall	far	farm	farmer	fast	father	feet
find	fire	first	fish	five	floor	flower	fly	for
found	four	from	full	funny	game	garden	gave	get
girl	give	go	goes	going	good	goodbye	got	grass
green	ground	grow	had	hand	has	have	he	head
help	her	here	hill	him	his	hold	home	horse
hot	house	how	hurt	I	if	in	into	is
it	its	jump	just	keep	kind	kitty	know	laugh
leg	let	letter	light	like	little	live	long	look
made	make	man	many	may	me	men	milk	money
morning	mother	much	must	my	myself	name	nest	never
new	night	no	not	now	of	off	old	on
once	one	only	open	or	our	out	over	own
paper	party	pick	picture	pig	play	please	pretty	pull
put	rabbit	rain	ran	read	red	ride	right	ring
robin	round	run	said	saw	say	school	see	seed
seven	shall	she	sheep	shoe	show	sing	sister	sit
six	sleep	small	snow	so	some	song	soon	squirrel
start	stick	stop	street	sun	table	take	tell	ten
thank	that	the	their	them	then	there	these	they
thing	think	this	those	three	time	to	today	together
too	top	toy	tree	try	two	under	up	upon
us	use	very	walk	want	warm	was	wash	watch
water	way	we	well	went	were	what	when	where
which	white	who	why	will	wind	window	wish	with
wood	work	would	write	yellow	yes	you	your	

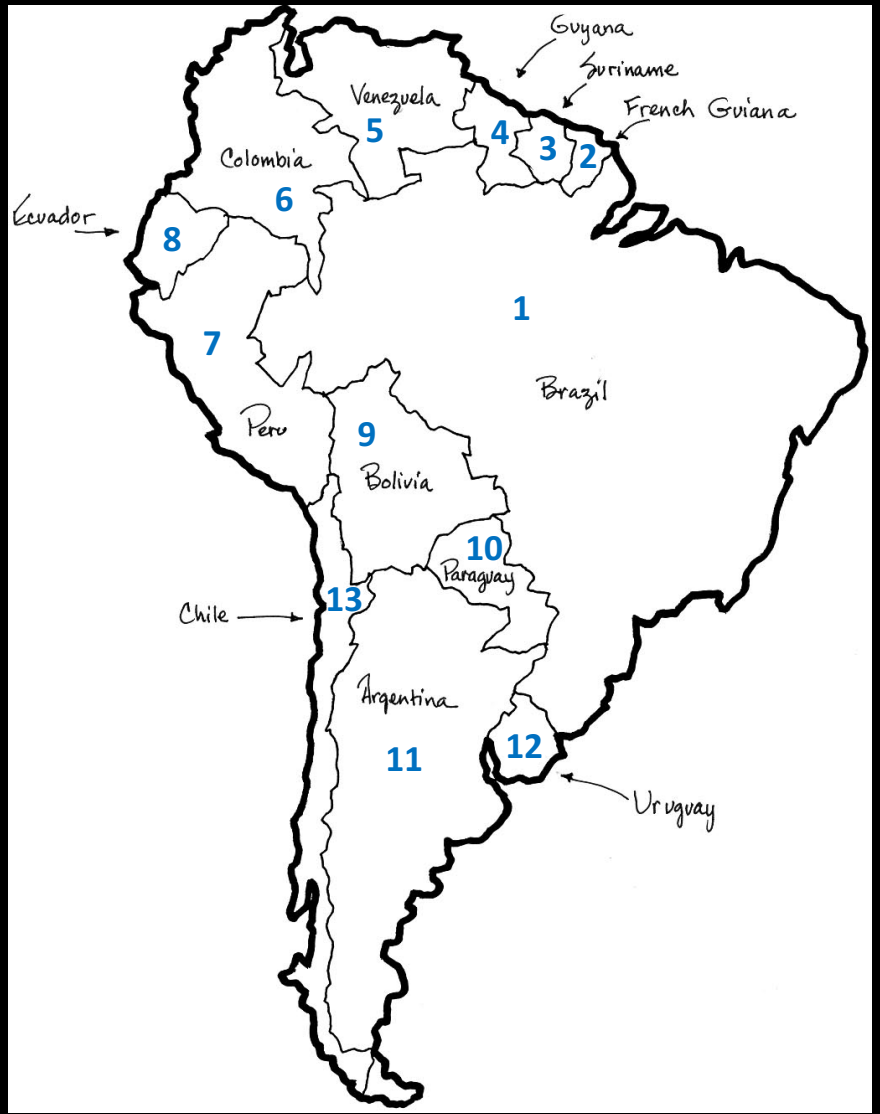
Why graphs?

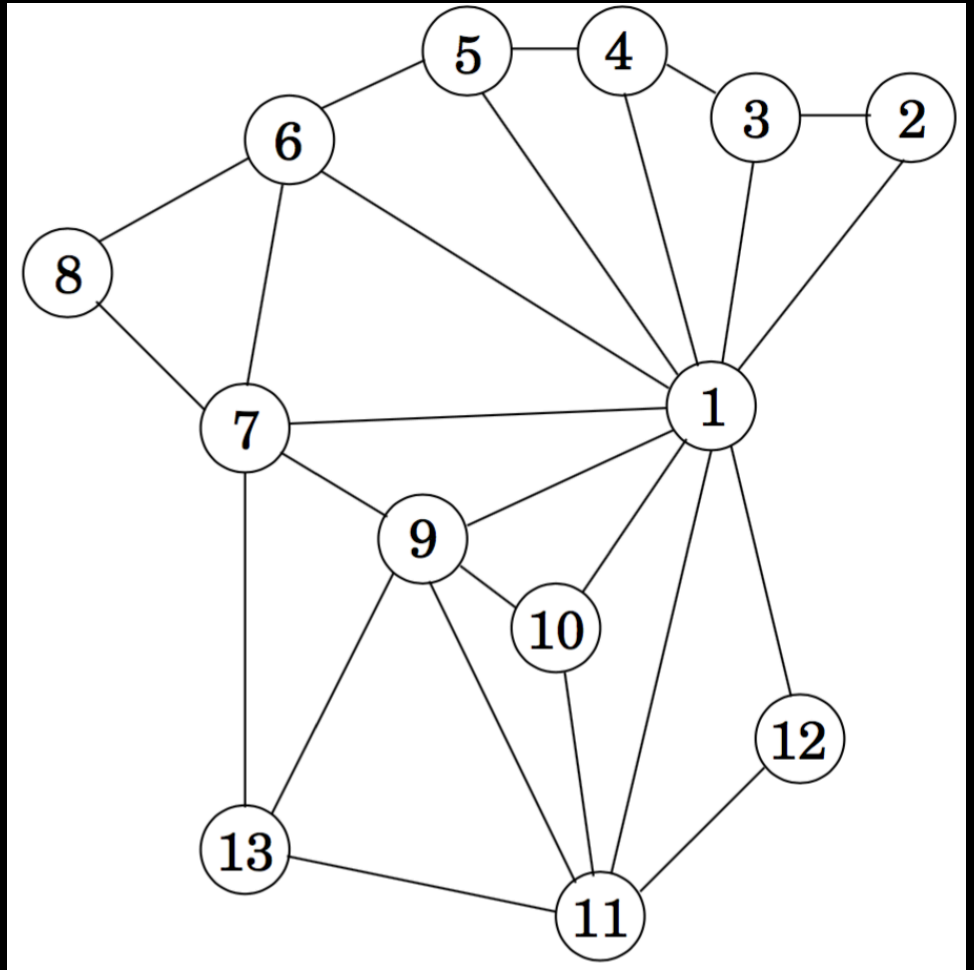
- **A wide range of problems can be formulated in terms of graphs.**
- **Once the problem is formulated in terms of graph, there is often a general solution.**
- **Graph is also a useful mean to represent the complex distribution of random variables**
  - Hidden Markov Models
  - Conditional Random Fields
  - Graphical models / Bayesian net

Map coloring problem

To avoid coloring adjacent countries with a same color, how many different colors are needed?

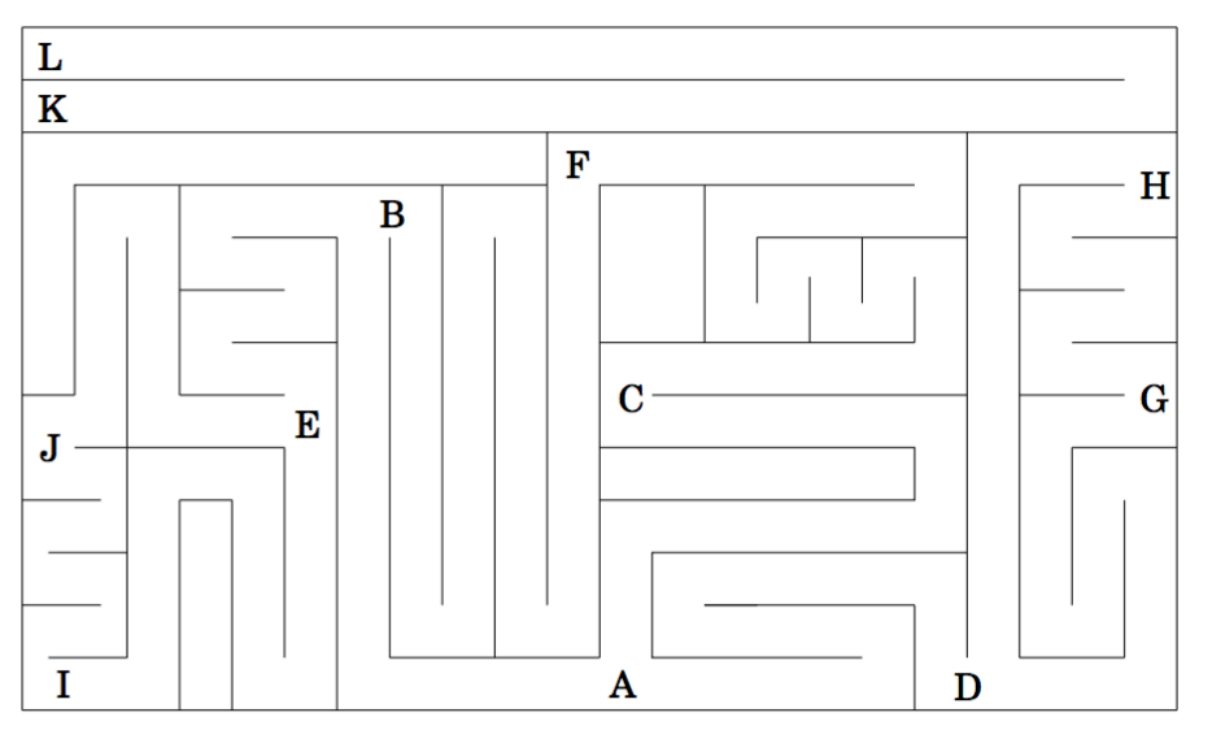
Map ⇔ Graph

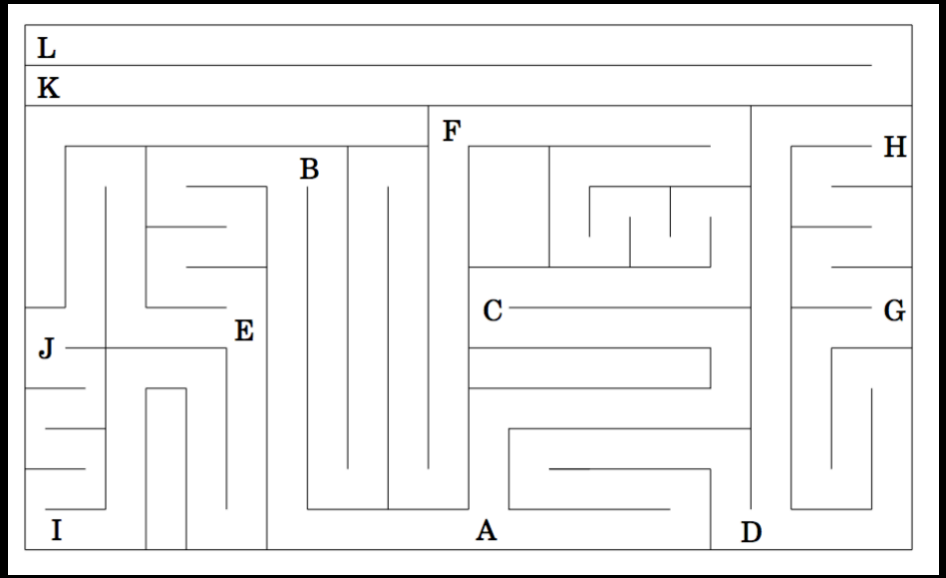

Guyana
Suriname
French Guiana
Venezuela
Colombia
Ecuador
Peru
Brazil
Bolivia
Paraguay
Chile
Argentina
Uruguay
1
2
3
4
5
6
7
8
9
10
11
12
13


1
2
3
4
5
6
7
8
9
10
11
12
13

Navigating a Maze

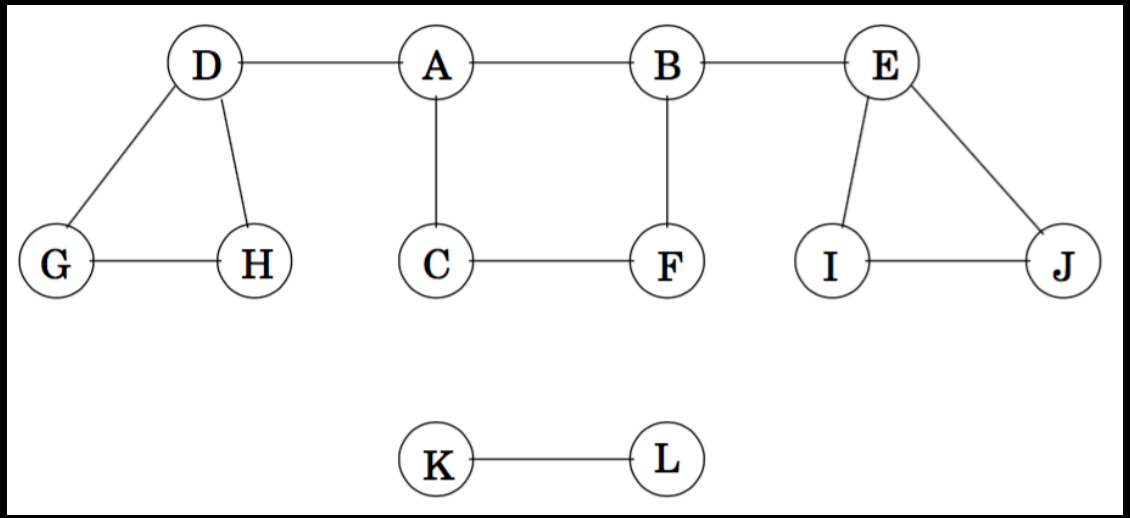
What are the list of reachable points from a given point?





An alternative representation

What parts of graphs are reachable from a given vertex?



How can we store a graph?

- **Vertices**

$$n = |V|$$

$$V = (v_1, \ldots, v_n)$$

- **Edges**

$$E = \{0,1\}^{n \times n}$$

$$E_{ij} = \begin{cases} 1 & v_i \text{ and } v_j \text{ are connected} \\ 0 & \text{otherwise} \end{cases}$$

- **Advantage : Fast edge checking**
  - Presence of a particular edge can be checked in one memory access : *O(1)*
- **Disadvantage : Large space requirements**
  - Storing matrix E requires $O(n^2)$ space.

Space vs. Time tradeoff

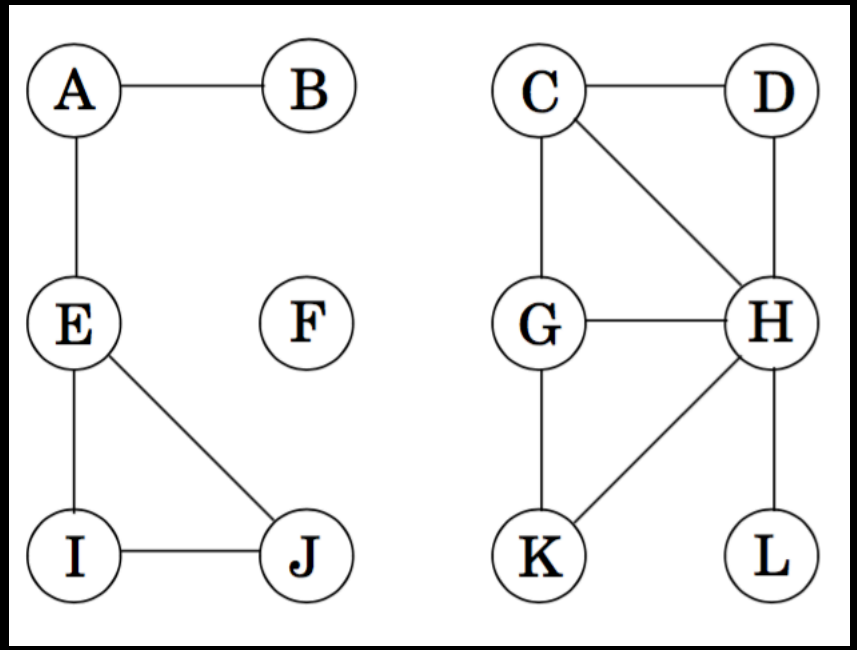
- **Store 'adjacency list'**
  - Make a list (or vector) of pair of connected vertices

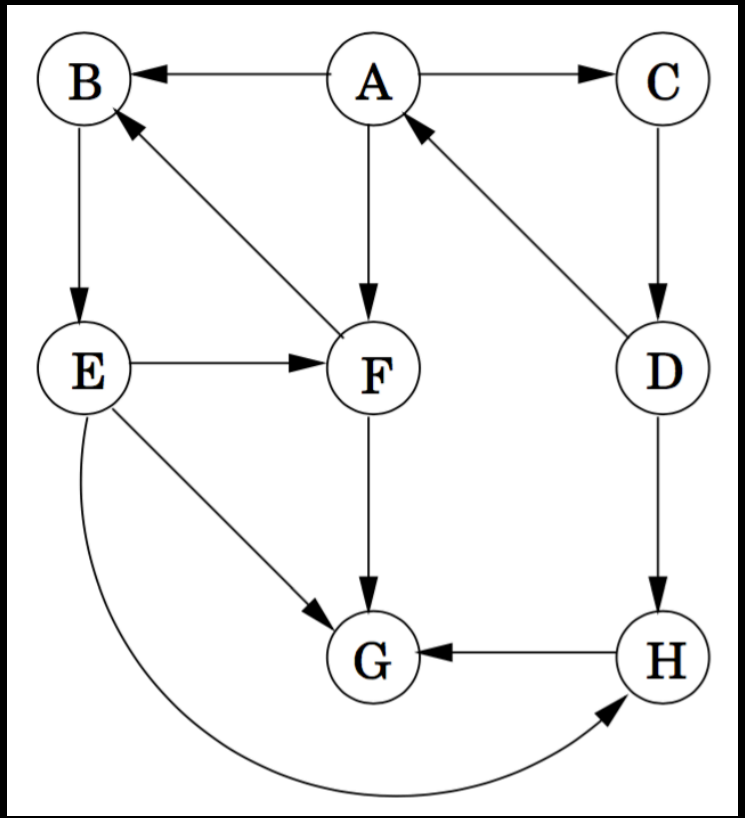
    $(u, v) \in E$

  - The total space to store edge information is $O(|E|)$

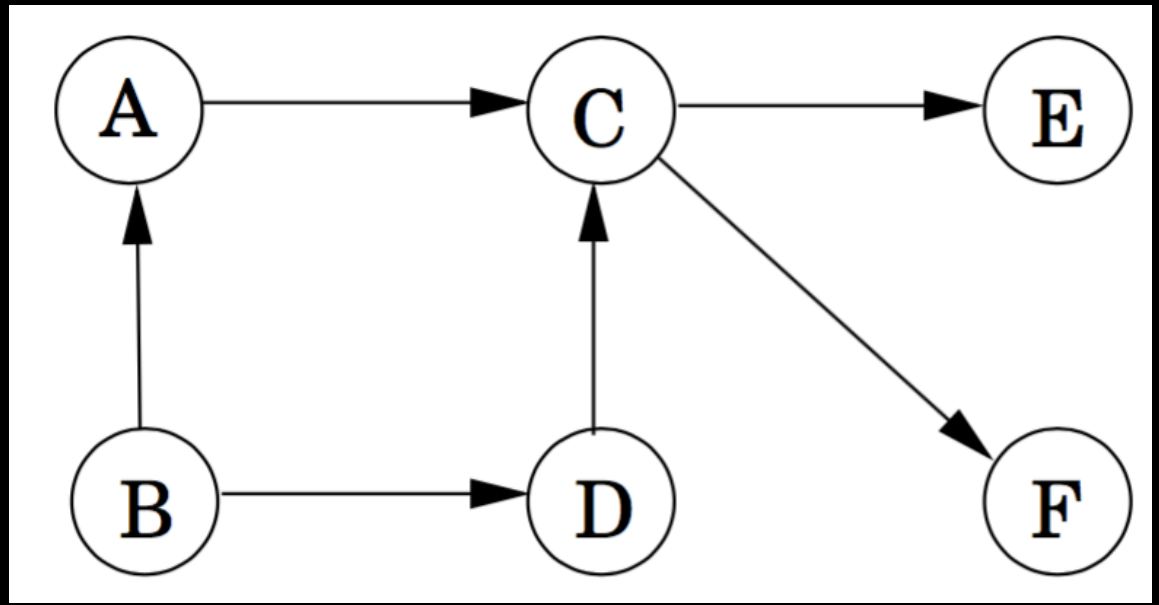
- **Time cost to checking for a particular edge is no longer constant**
  - What would be the time complexity to check for an edge?

Directed vs. Undirected Graph

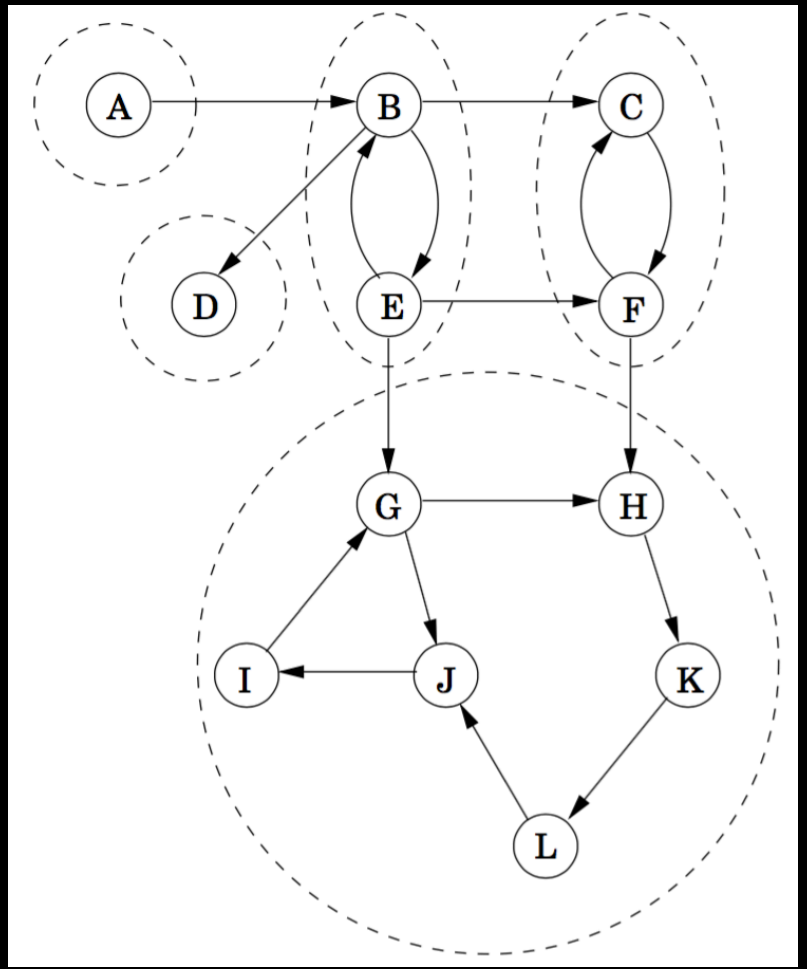


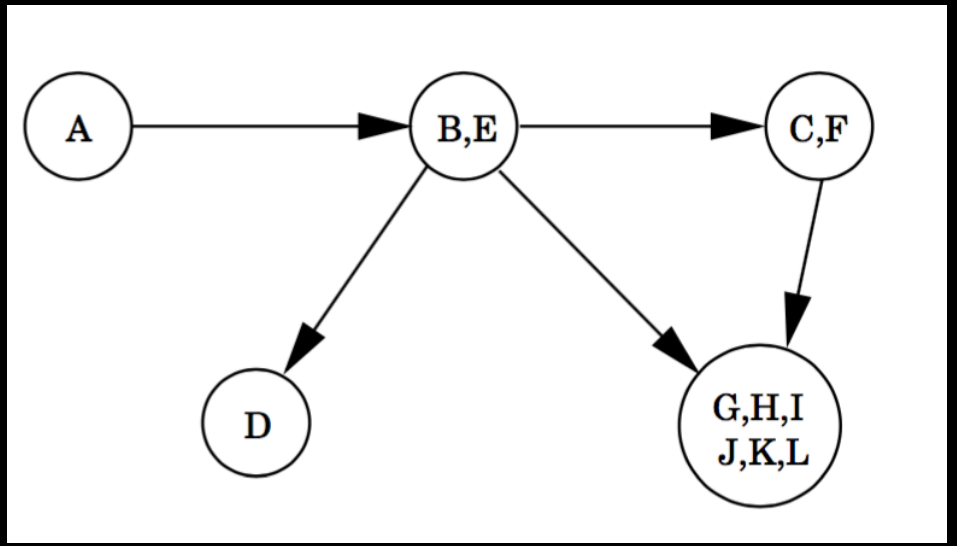


Directed acyclic graph (DAG)

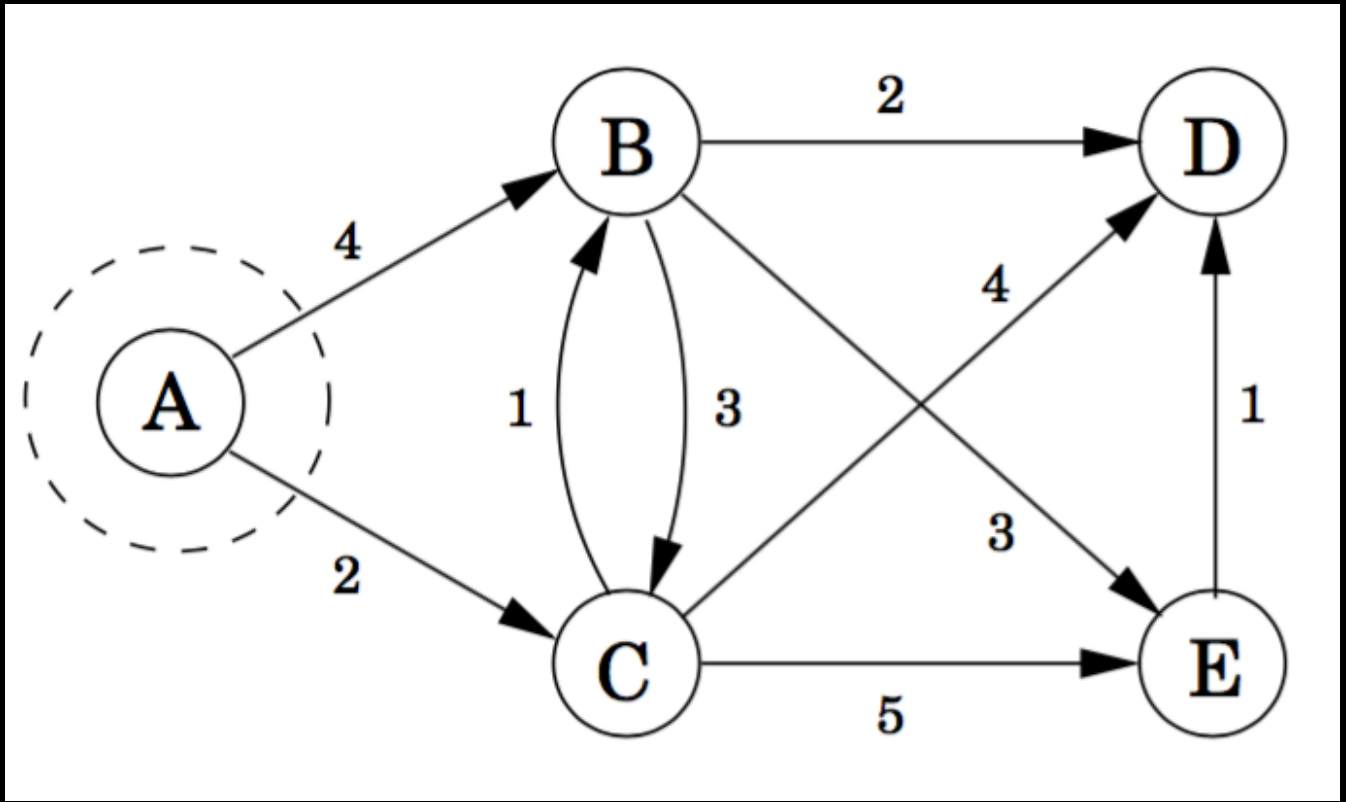


Strongly connected components

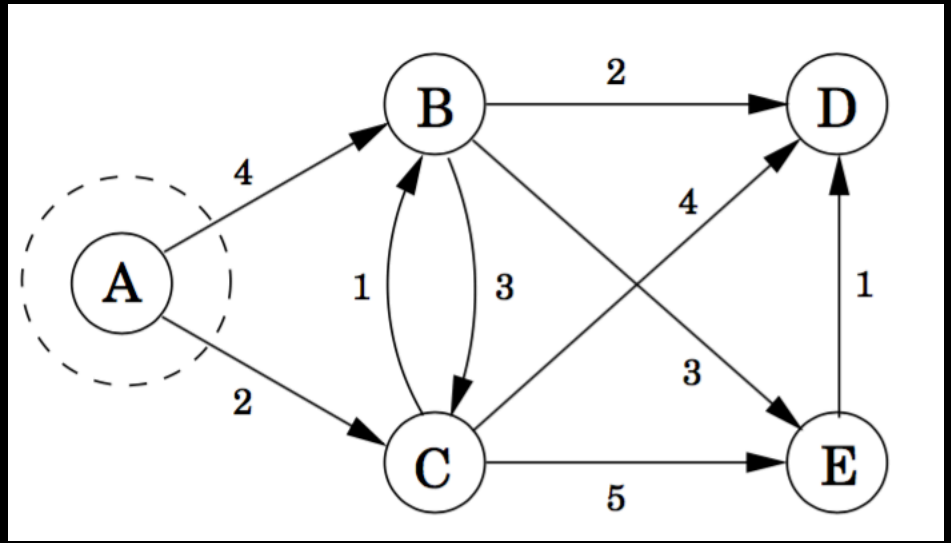




Finding shortest path from a graph

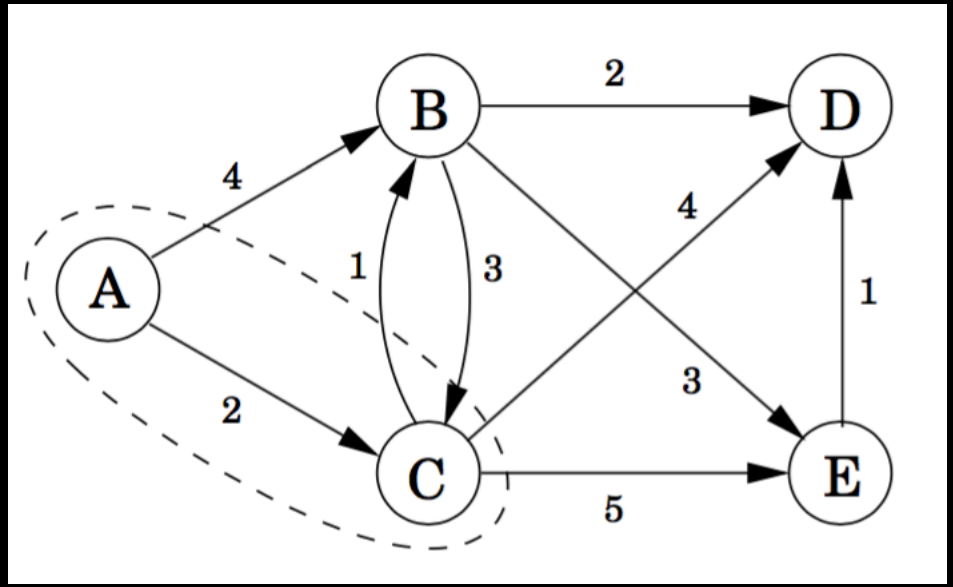


Dijkstra's algorithm



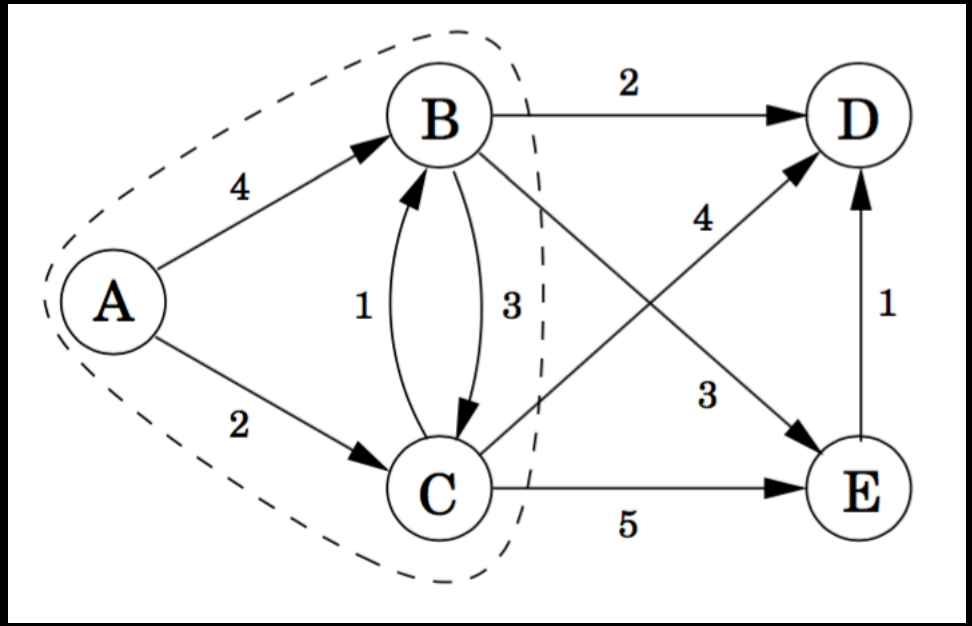
A: 0 B: 4 C: 2	D: ∞ E: ∞

Dijkstra's algorithm



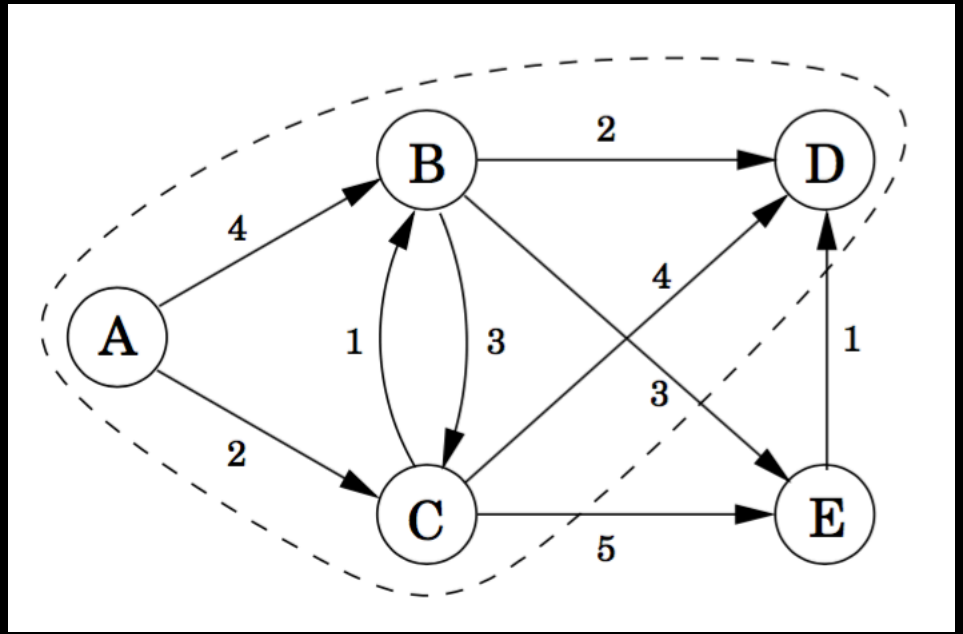
A: 0	D: 6
B: 3	E: 7
C: 2	

Dijkstra's algorithm



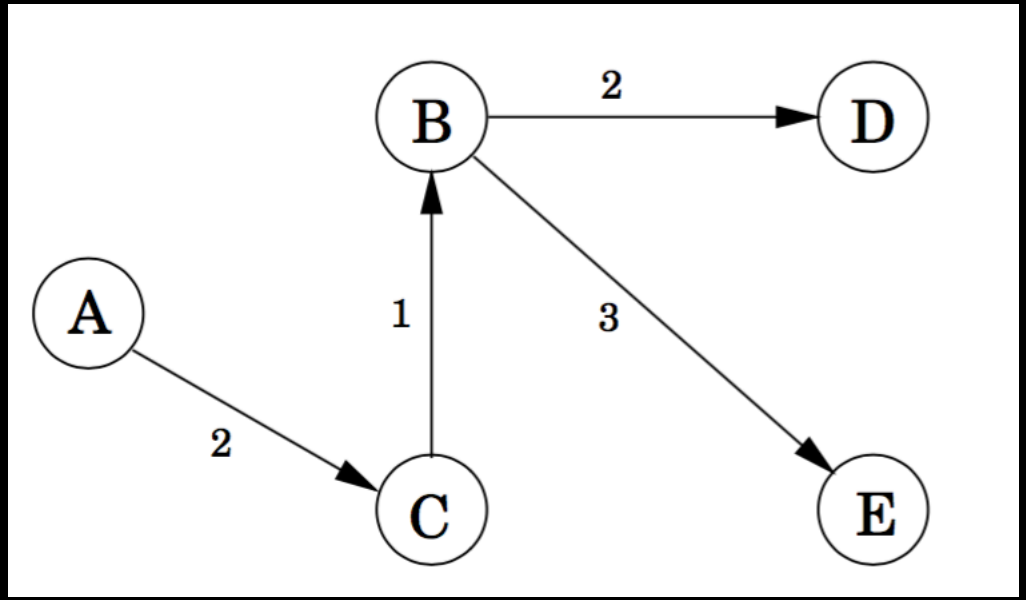
A: 0	D: 5
B: 3	E: 6
C: 2	

Dijkstra's algorithm



A: 0	D: 5
B: 3	E: 6
C: 2	

Final outcome of Dijkstra's algorithm



A: 0	D: 5
B: 3	E: 6
C: 2	

Description of the Dijkstra's Algorithm

Input: Graph *G(V,E)* with *l(I,j) ⩾ 0*, starting vertex *s ∈ V*

1. **Initialize:**
   - For all u $\in V$, $dist(u) = \infty$, $prev(u) = NULL$
   - $dist(s) = 0$
2. ***H = build_heap(V)*** *(using dist as key)*
3. ***while H is not empty:***
   - *u = extractMin(H)*
   - *For all edges (u,v)* $\in E$
     - *If* $dist(v) > dist(u) + l(u,v)$:
       - $dist(v) = dist(u) + l(u,v)$
       - *prev(v) = u*
       - *decreaseKey(H, v)*

Designing a Graph Data Structure

- **What kind of ADTs do we need?**
  - Vertex
  - Edge
  - Graph

- **For each ADT, what kind of behaviors are needed?**
  - For Graph…?
  - For Edge…?
  - For Vertex…?

What we need to store?

- **For each vertex `v`**
  - **`dist(v):`** Minimum cost to the vertex from the source
  - **`prev(v):`** Previous node to visit from to attain minimal cost.

- **We also need to be able to access adjacent nodes from each vertex easily**

- **We also need to build a heap that users the "current shortest distance" to each vertex from the source as a key.**

The heap data structure we need

- **It needs to have `build_heap()` function**
  - To initialize a heap using dist as key
- **It also needs `extractMin()` function**
  - To extract the vertex that is closest among the ”unresolved” vertices
- **It also needs to update `dist(v)` for an arbitrary vertex**
  - And call `decreaseKey(H, v)` after the update.
  - This is NOT a standard feature of heap or priority queue
  - STL’s `std::priority_queue` cannot be used.

Defining the graph data structure

```
class vertex_value_t; // need to declare first

typedef map<string, vertex_value_t> word_graph_t;
typedef map<string, vertex_value_t>::iterator
                                    word_graph_itr_t;

class vertex_value_t {
public:
  int32_t cost; // used for dijkstra's current cost
  string prev;  // used for dijkstra's backtracking
  vector<word_graph_itr_t> adjacent; // adjacency nodes
  vertex_value_t(int32_t c = INT_MAX) : cost(c) {} r
};
```

Reading the word list

```
int32_t main(int32_t argc, char** argv) {
  ifstream ifs(argv[1]); // word list file
  string src(argv[2]);   // string to start
  string s;
  word_graph_t graph;

  // Need to insert vertices first without adding edges yet
  cout << "Reading dictionaries..." << endl;
  while( ifs >> s ) {
    graph[s].cost = INT_MAX; // adding a vertex
  }
  cout << "Finished reading " << graph.size() << " words"
<< endl;
```

How should we connect edges?

- **Assume that the graph is bidirectional / undirected.**

- **A possible approach**
  - Select every pair of vertices
  - Check whether they have an edit distance of 1.
  - What would be the time complexity?

- **Another possible approach**
  - Start with each vertex
  - Make every possible insertion, deletion, substitution, and reversal, and see if the modified word also exists in the dictionary
  - What would the time complexity in the case?

Counting the number of possible modifications

For string with length *L*

- Deletions : L
- Substitutions : L x 51 (why?)
- Insertions : (L+1) x 52 (why?)
- Reverse : 1

However, because this is bidirectional, we only need to count deletion, half of substitutions and insertions (why?)

Connecting edges - implementation

```
cout << "Connecting edges..." << endl;
int32_t counter = 1, n_edges = 0;
for(word_graph_itr_t it = graph.begin();
                     it != graph.end(); ++it, ++counter) {
  if ( counter % 10000 == 0 )
    cout << counter << " words.. " << n_edges << " edges" << endl;
  // create a temporary vertex to check connection
  string tmp(it->first);
  uint32_t l = tmp.size();
  // First, remove one letter at a time
  for(uint32_t i=0; i < l; ++i) {
    tmp.erase(i,1);
    n_edges += connect_vertices(graph, it, tmp);
    tmp = it->first;
  }
```

```
 // Second, modify one letter at a time
  for(uint32_t i=0; i < l; ++i) {
    for(char c='a'; c <= 'z'; ++c) {
      for(int32_t upper = 0; upper < 2; ++upper) {
         tmp.at(i) = c + ( upper == 1 ? 'A' - 'a' : 0 );
         if ( tmp < it->first ) // to avoid redundancy
           n_edges += connect_vertices(graph, it, tmp);
       tmp = it->first;
       }
    }
  }
  // Third, reverse the string
  tmp = it->first;
  reverse(tmp.begin(), tmp.end());
  if ( tmp < it->first )
      n_edges += connect_vertices(graph, it, tmp);
}
```

After connecting edges..

```
dijkstra(src, graph);
// Print the shortest paths reachable from the source
for(word_graph_itr_t it = graph.begin();
                     it != graph.end(); ++it) {
  if ( it->second.cost < INT_MAX ) {
    cout << it->first << "\t" << it->second.cost;
    word_graph_itr_t it2 = it;
    while( !it2->second.prev.empty() ) {
       cout << "\t" << it2->second.prev;
       it2 = graph.find(it2->second.prev);
    }
    cout << endl;
  }
}
return 0;
```

Implementing Dijkstra's algorithm

```
// heap data structure needed for Dijstra's algorithm
typedef map< int32_t, set<string> > cost_heap_t;
typedef map< int32_t, set<string> >::iterator
                                    cost_heap_itr_t;

// Implementation of Dijstra's algorithm
void dijkstra(const string& src, word_graph_t& G) {
  // Check if the source string exist in the graph
  word_graph_itr_t itsrc = G.find(src);
  if ( itsrc == G.end() ) {
    cerr << "Cannot find word " << src << endl;
    exit(-1);
  }
```

```
// Fill heap with INT_MAX, except for the source node
cost_heap_t Q;
G[src].cost = 0;
for(word_graph_itr_t it = G.begin(); it != G.end(); ++it)
  Q[it->second.cost].insert(it->first);
cout << "Running Dijkstra's algorithm from " << src << endl;
// Vertex on the top of the queue has optimal cost
while( !Q.empty() ) {
  // The following several lines perform extractMin() of heap
  cost_heap_itr_t itQ = Q.begin();
  if ( itQ->first == INT_MAX )
    break;
  // copy all vertices with minimum cost
  set<string> min_cost_keys = itQ->second;
  Q.erase(itQ->first);
```

```
// For each vertex with minimum cost
 for(set<string>::iterator itK = min_cost_keys.begin();
      itK != min_cost_keys.end(); ++itK) {
   word_graph_itr_t itM = G.find(*itK);
   // Iterate through its adjacent vertices
   vector<word_graph_itr_t>& adj = itM->second.adjacent;
   for(int32_t i=0; i < (int)adj.size(); ++i) {
      const string& key = adj[i]->first;
      vertex_value_t& val = adj[i]->second;
      if ( val.cost > itM->second.cost + 1 ) {
        Q[val.cost].erase(key);
        Q[itM->second.cost+1].insert(key);
        val.cost = itM->second.cost + 1;
        val.prev = *itK;
  } } } } }
```

Running word ladder program

```
$ ./word_ladder_graph dolch.txt all
Reading dictionaries...
Finished reading 314 words
Number of edges are 278
all   0
ball  1     all
bell  2     ball  all
call  1     all
fall  1     all
full  2     fall  all
hill  5     will  well  bell  ball  all
pull  3     full  fall  all
tell  3     bell  ball  all
well  3     bell  ball  all
will  4     well  bell  ball  all
```

Homework Questions

- If we are interested in making optimal word ladder only ”between” two words, how should we change the Dijkstra’s algorithm and its implementation?

- Implement the changes needed

Summary : Graph Algorithms

- Graph is a useful tool to represent many problems into a similar kind.
- Implementing graph is more sophisticated than other data structure, and has multiple options of implementation depending on the characteristics of the graph, and the behavior required.
- Dijkstra's algorithm is not only useful for shortest path search, but also for apparently unrelated problems like word ladder problems.