CIS 330
C++ and Unix

Lecture 7

GDB and Valgrind

Logistics

We will go back to in-person teaching starting 2/8

The department will continue to monitor the number of absences, and may recommend going back to remote teaching

Grading

I will have the midterm and the assignments graded by next Tuesday

Please continue providing feedback

Switching to remote from in-person (and possibly back to remote)

TA office hours and labs

Creating a Library

Static library

```
gcc -c add.c -o add.o
gcc -c sub.c -o sub.o
ar rcs libmymath.a add.o sub.o
```

Dynamic library

```
gcc -fPIC -c *.c
gcc *.o -shared -o liball.so
```

Dynamic Libraries

- Difference between static and dynamic
  - Static – everything is include in the executable (you don’t need to go searching for your library)
  - Dynamic – it exists as a separate file (you DO need to go searching for your library e.g., LD_LIBRARY_PATH)
  - Pros and cons – ?

Dynamic Libraries

- Difference between static and dynamic
  - Static – everything is include in the executable (you don't need to go searching for your library)
  - Dynamic – it exists as a separate file (you DO need to go searching for your library e.g., LD_LIBRARY_PATH)
  - Pros and cons – dynamic library reduces the file size, and allows you to just recompile the library if changes are made. On the other hand, static libraries are faster at run-time, and less susceptible to breaking.

Questions?

Makefiles

- Convenient way to compile your code without having to type out everything every time
- Create a file named 'Makefile'
- Type 'make'
- <executable/target name> : pre-requisites (required files)

  recipe (rules for making the target)

Makefiles

- Defines a set of commands and rules for compiling your code
- Useful commands and rules
  - $(wildcard *pattern*) – space separated list of file names that matches the *pattern*
  - $(addsuffix *suffix*, *names*…) – names is a series of file names, and *suffix* is appended to the end of each name
  - $(basename *names*…) – extracts all but the suffix of each file name in *names*
  - %.c – Pattern rule % refers to exactly one (i.e., exactly one file with extension.c
    - %.o : %.c – pre-requisite for a .o file is the corresponding .c file
  - %.o : %.c $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
    - %@ and $< are used to substitute the names of the **target** and **source** file in each case.
    - Generate a.c from a.o using gcc –c <some flags> a.c –o a.o

Live Coding

Debugging

- use printf
- The end

Debugging

- GNU debugger (gdb)
- Compile your code with –g (and -Wall option is recommended)
- Start gdb with `gdb ./a.out’
- gdb provides an interactive shell
  - get help by typing `help <command>’

Debugging

- (gdb) run <arguments>
- Runs to completion if there are no problems with your code
- (gdb) run
  - Starting program: /home/users/jeec/lecture06/ex01/prog
  - 9
  - [Inferior 1 (process 10178) exited normally]

Debugging

- (gdb) run <arguments>
- Runs to completion if there are no problems with your code
- If there are problems, gdb takes control after it terminates and displays some useful information
  - line number where it terminated
  - what type of problem (e.g., seg fault)
  - enclosing function
  - etc.

Debugging

- (gdb) a.out
- Starting program: /home/users/jeec/lecture07/a.out
-
- Program received signal SIGSEGV, Segmentation fault.
- 0x000055555555513d in add_numbers (
- a=<error reading variable: Cannot access memory at address 0x7fffff7fefec>, b=<error reading variable: Cannot access memory at address 0x7fffff7fefe8>)
- at add.c:4
- 4 {

Useful commands

- gdb allows you to step through the code and print the contents of the memory, variables, etc.
- (gdb) bt
  - backtrace – traces the steps to see what happened
- breakpoint
  - break <location>
  - Location could be function name, or line number (add.c:8)
  - You can backtrace from the breakpoint
  - use `clear' to clear all breakpoints
- step
  - step through your code, including function invocation
- next
  - step through your code, but not into other functions
- continue
  - resume execution after gdb pauses (e.g., at a breakpoint)

Useful commands

- print – print the content of variables
- watch – you can `watch’ a variable and gdb will tell you when it has been modified
- info <args/locals/reg> - print information about these resources

valgrind

- Program execution monitoring framework
- memcheck
  - Use of uninitialized memory
  - Reading/writing to heap memory after it has been freed
  - Reading/writing to end of malloc space
  - Heap allocated memory leaks
  - Mismatched use of malloc and free
  - etc.

valgrind

```
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int x, y;
    if (x < 3)
        y = 4;
    else
        y = 5;
    return 0;
}
```

valgrind

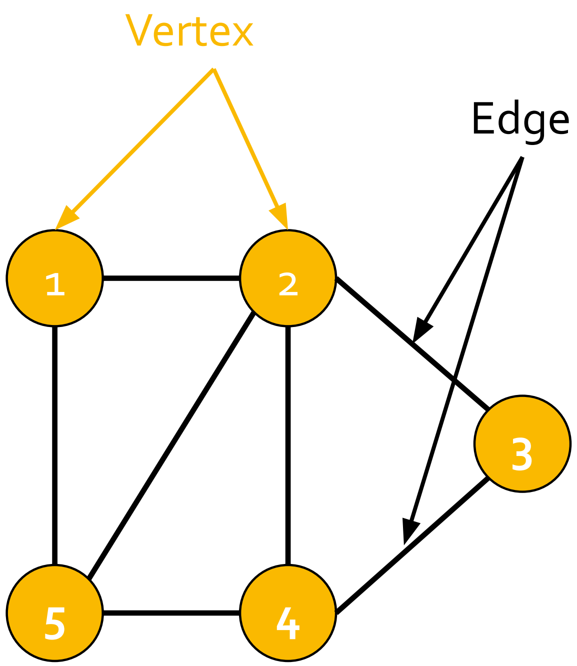
- valgrind a.out
- ==16716== Memcheck, a memory error detector
- ==16716==
- ==16716== **Conditional jump or move depends on uninitialised value(s)**
- ==16716==   **at 0x109134: main (main.c:7)**
- ==16716==
- ==16716==
- ==16716== HEAP SUMMARY:
- ==16716==   in use at exit: 0 bytes in 0 blocks
- ==16716==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
- ==16716==
- ==16716== All heap blocks were freed -- no leaks are possible
- ==16716==
- ==16716== For counts of detected and suppressed errors, rerun with: -v
- ==16716== Use **--track-origins=yes** to see where uninitialised values come from
- ==16716== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)

Live Coding

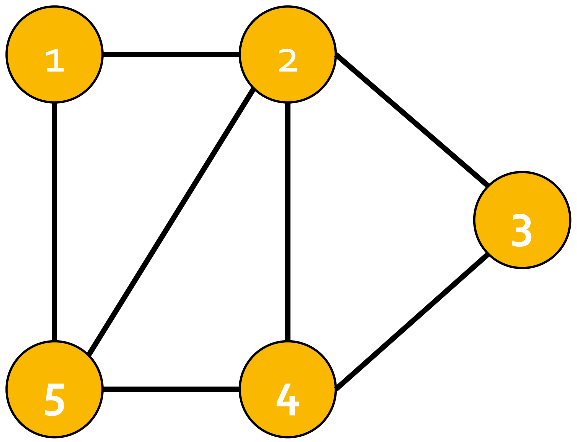
Graphs

- $G = (V, E)$
- V – a set of vertices
- E – a set of edges

Vertex and Edge



Undirected Graphs

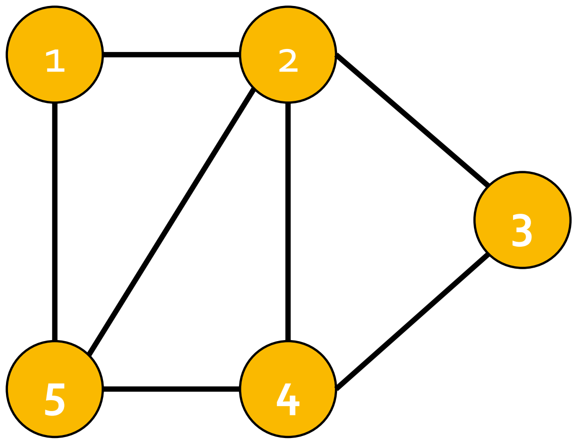


Undirected Graph

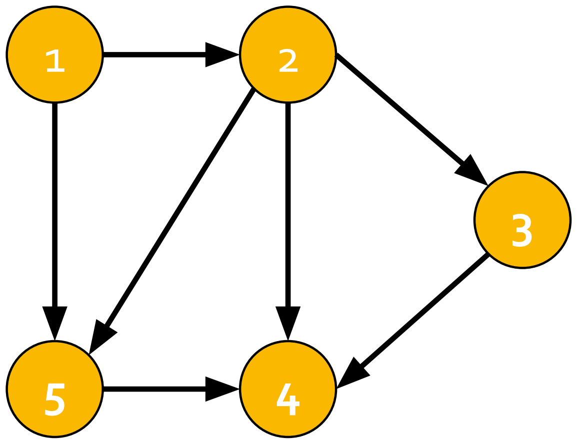
- ? vertices
- ? edges

Undirected Graphs

Undirected Graph
- 5 vertices
- 7 edges



Directed Graphs



Directed Graph

- 5 vertices
- 7 edges

Directed vs. Undirected

Directed

- Resource allocation (e.g., Operating Systems; resource to/from requesting process)
- Page rank
- Linguistics (e.g., programming languages)
- Finance

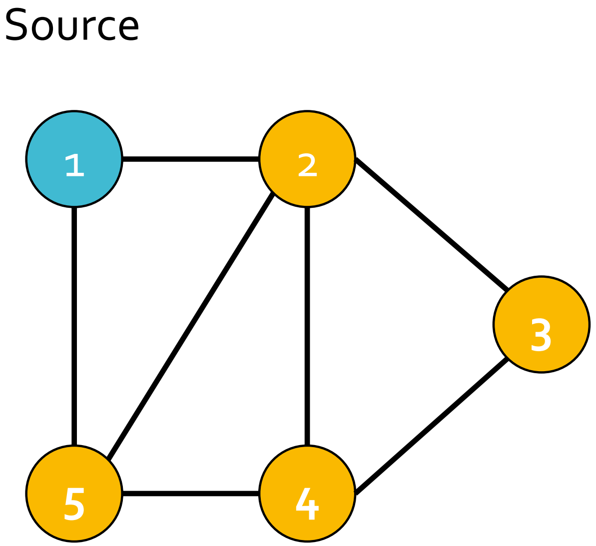
Undirected

- Social network (e.g., Facebook)
- Transportation networks

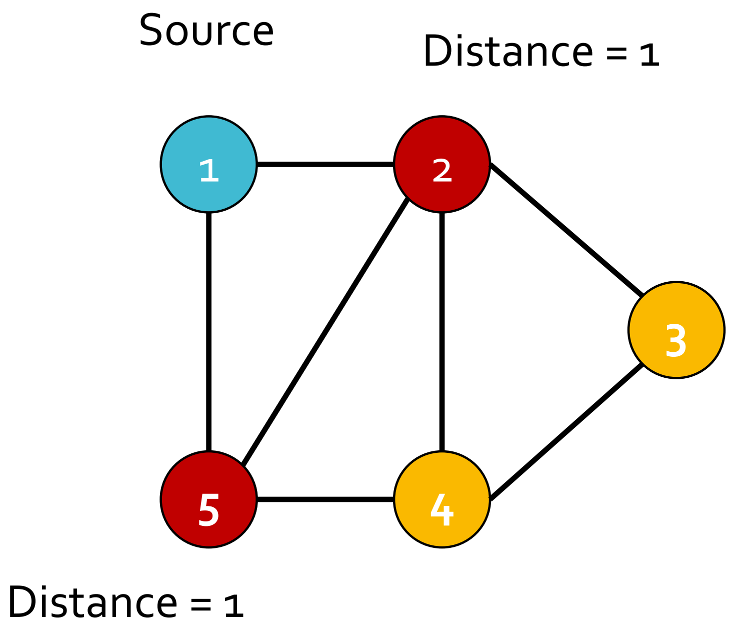
Breadth First Search (BFS)

- Way to search/traverse a graph
- Given a source vertex *S*, find
  - every vertex that is reachable from S
  - the distance (i.e., number of "hops") from S to each vertex
- Breadth-first – search the graph by looking at the vertices that are at the same level (or distance) from the source before looking at vertices that are further away

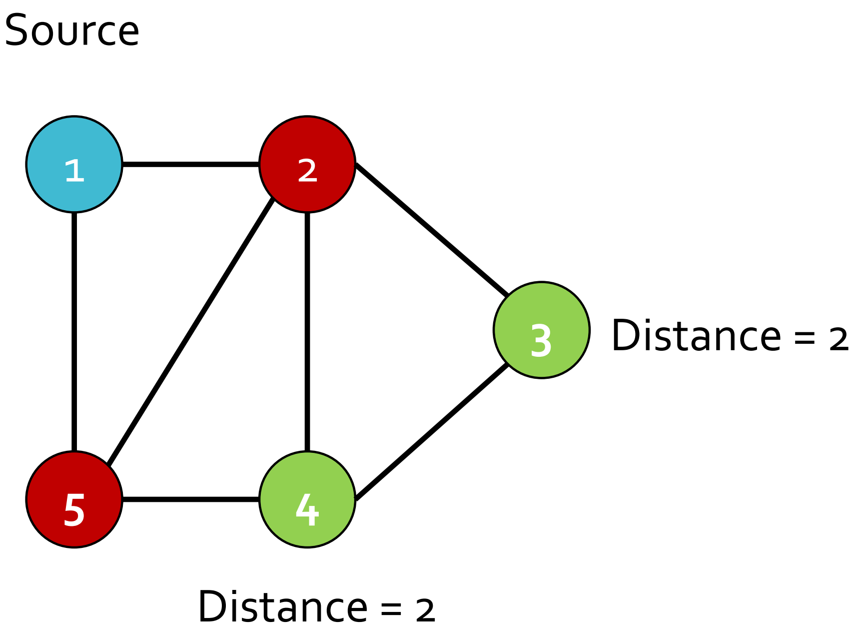
BFS



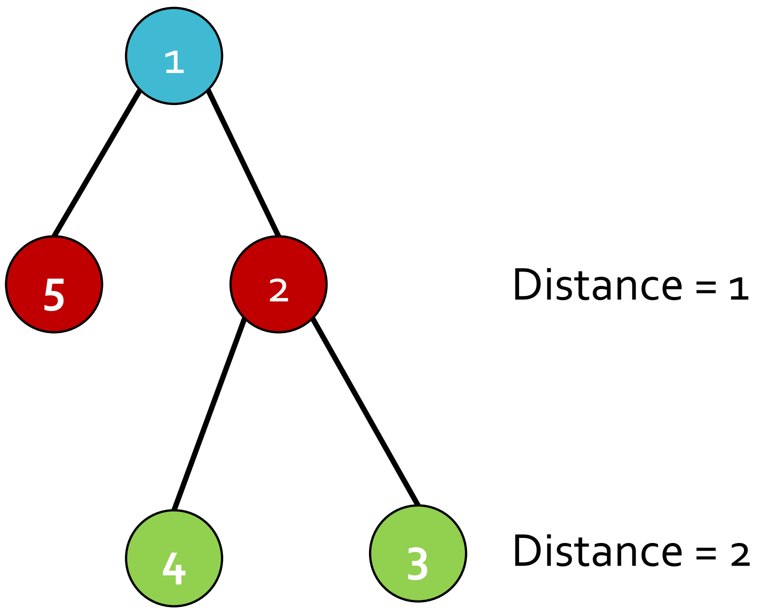
BFS – 1st iteration



BFS – 2nd iteration



BFS Tree

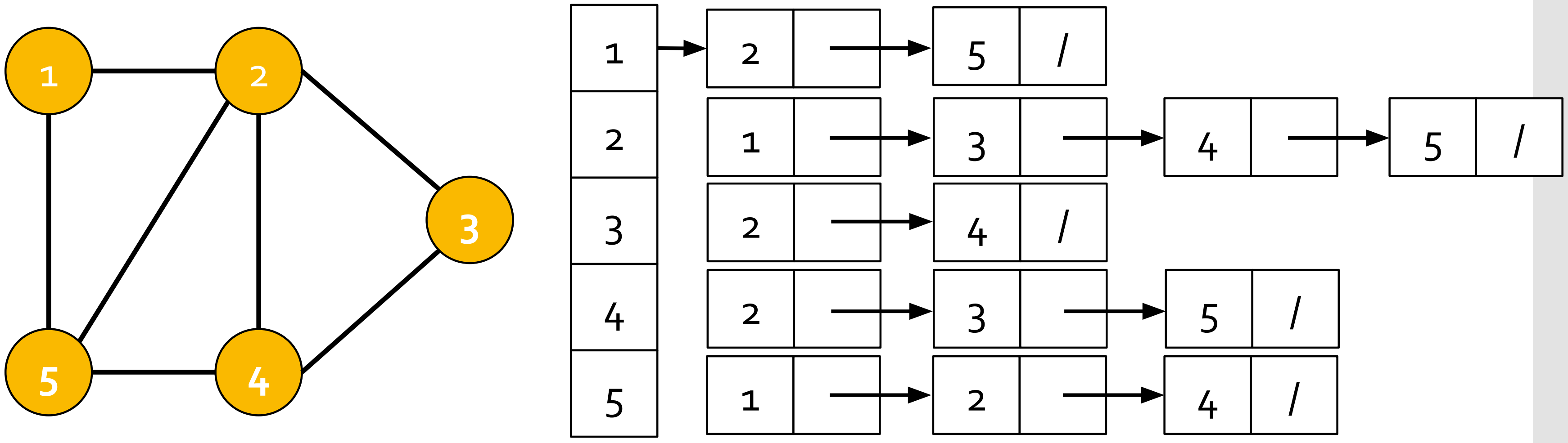


2 is **parent** (or ancestor) to 4 and 3

==

4 and 3 are **children** (or descendant) of 2

Adjacency List



Using Linked List to Represent an Adjacency List

```
typedef struct adj_node_struct_t {
    // vertex ID
    int vid;
   // pointer to next adj node
    struct adj_node_struct_t *next;
} adj_node_t;
```

Using Linked List to Represent an Adjacency List

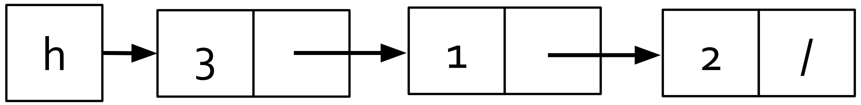
1. adj_node_t *head = NULL;
2. adj_node_t *node1 = (adj_node_t*) malloc(sizeof(adj_node_t));
3. node1->vid = 1;
4. adj_node_t *node2 = (adj_node_t*) malloc(sizeof(adj_node_t));
5. node2->vid = 2;
6. adj_node_t *node3 = (adj_node_t*) malloc(sizeof(adj_node_t));
7. node3->vid = 3;

Using Linked List to Represent an Adjacency List

1. head = node3;
2. node3->next = node1;
3. node1->next = node2;
4. node2->next = NULL;

Using Linked List to Represent an Adjacency List

1. head = node3;
2. node3->next = node1;
3. node1->next = node2;
4. node2->next = NULL;



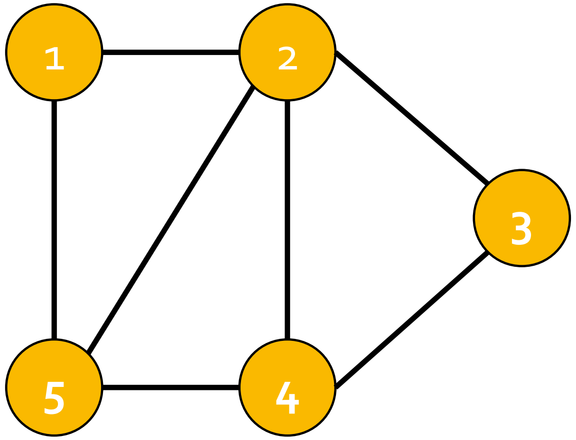
BFS Algorithm

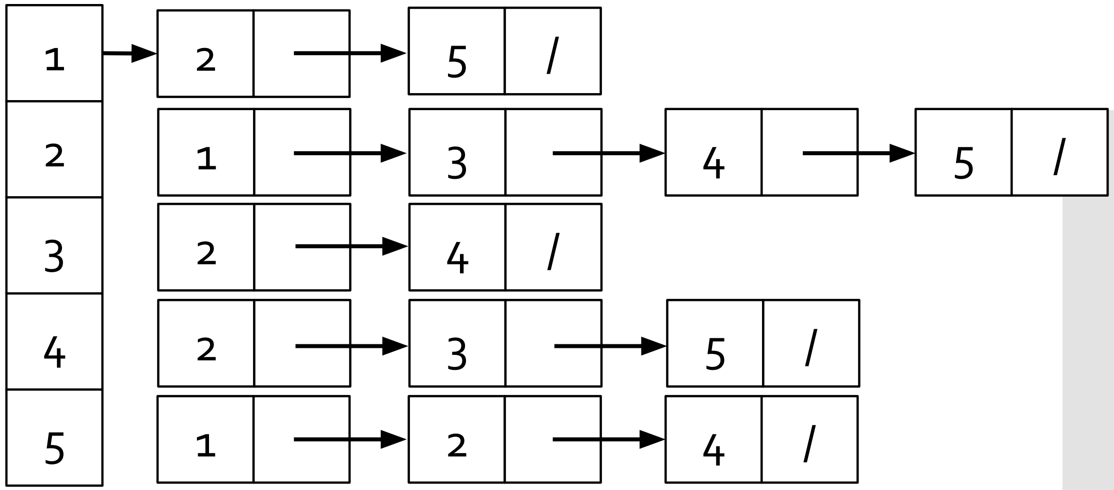
- 3 arrays
  - color – **0** (not visited), **1** (in the queue), **2** (visited)
  - distance – **0** if source, **1** if immediate neighbor to source, **2** if neighbor's neighbor, etc.
  - parent – vertex ID of the vertex's parent

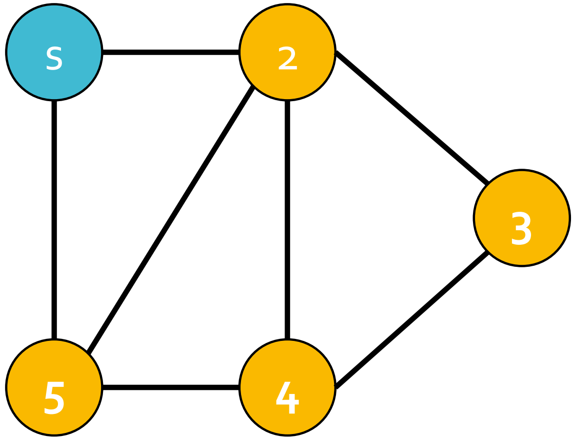
BFS Algorithm

- BFS(G, s) // given a graph G and a source vertex s
  - For each vertex u ∈ V[G] - {s} // initialize the arrays for all but the source
    - color[u] <- 0
    - distance[u] <- ∞
    - Parent[u] <- NIL
  - color[s]= 1 // 1 because it's the first vertex and you will need it in the queue
  - distance[s] = 0 // 0 because it's the source
  - parent[s] = NIL // NIL because it has not parents
  - Q <- { }
  - enqueue(Q, s)
  - while (Q is not empty)
    - u <- dequeue(Q) // first in first out queue
    - for each v ∈ adjacency_list[u]
      - if color[v] == 0 // only add if not already visited or in queue
        - color[v] = 1 // v is now in the queue
        - distance[v] <- distance[u] + 1
        - parent[v] <- u
        - enqueue(Q, v)
    - color[u] = 2 // u has been visited and accounted for

BFS Algorithm

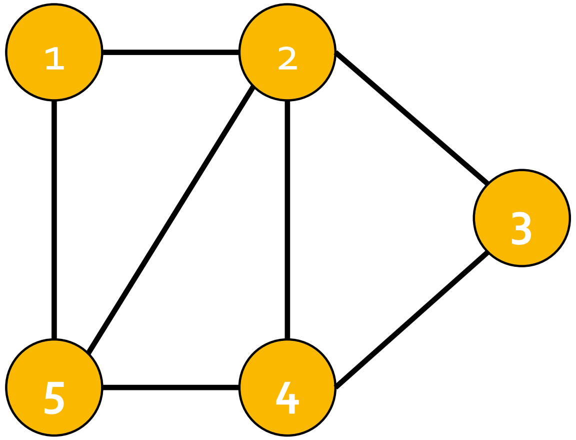


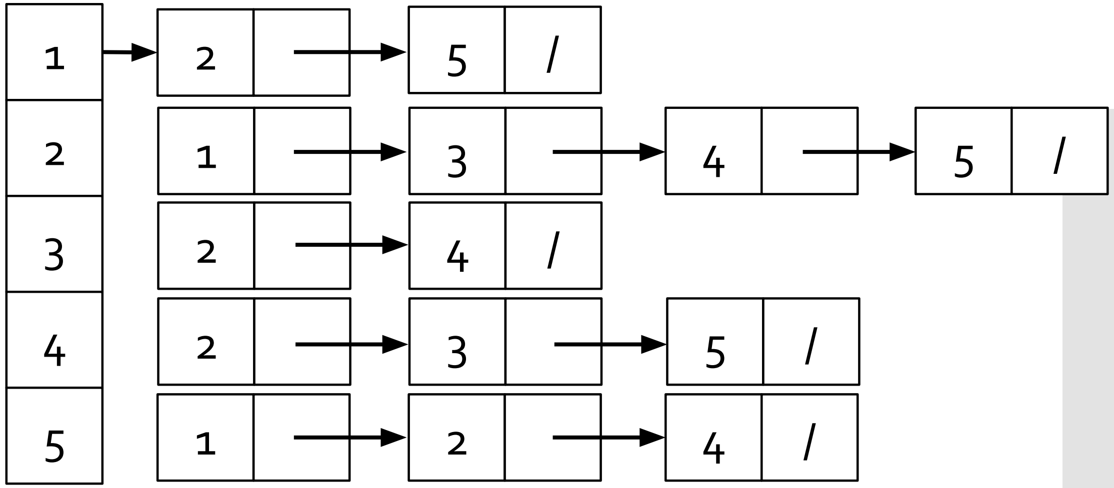


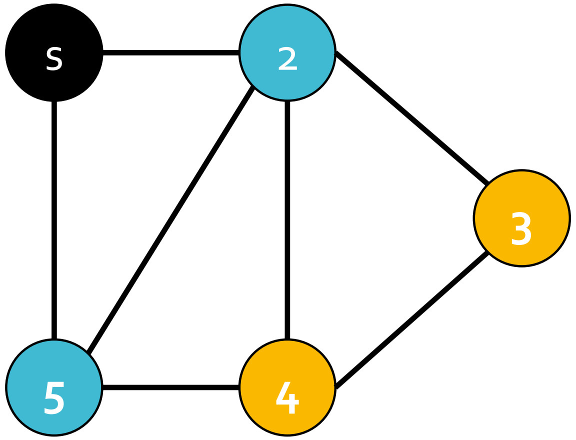


1) Queue: s

BFS Algorithm

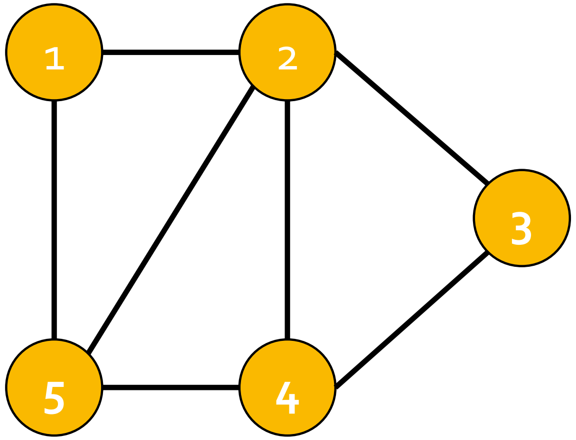


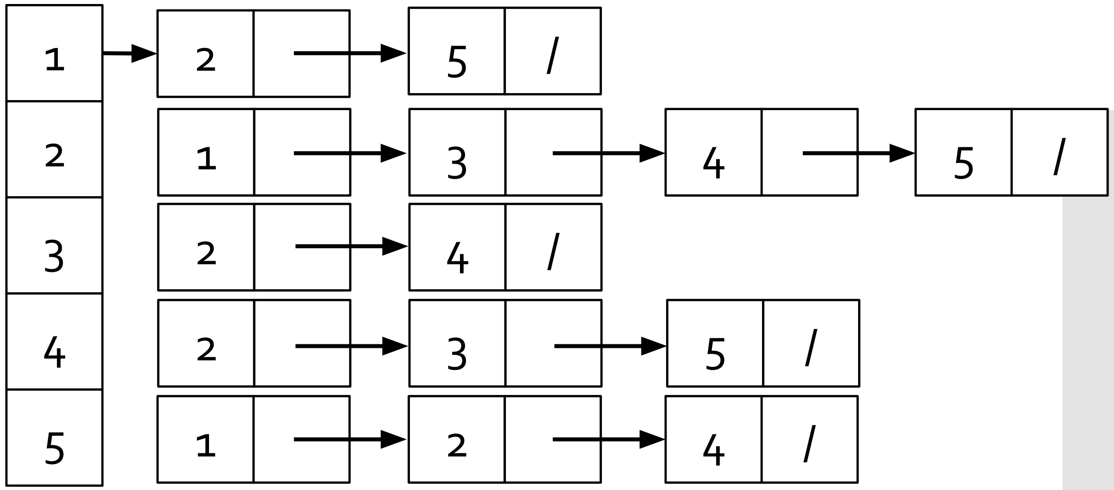


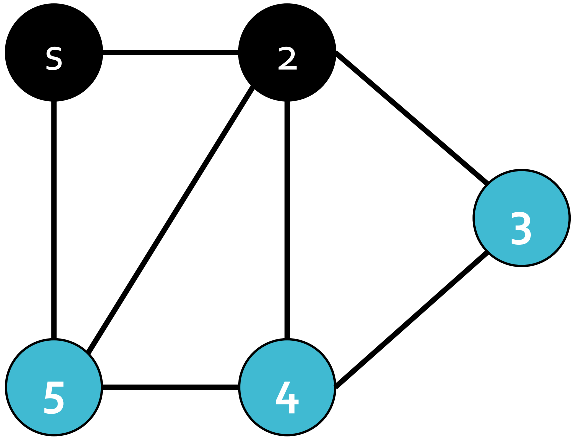


1) Queue: s
   - dequeue s and add 2,5
2) Queue: 2 5

BFS Algorithm

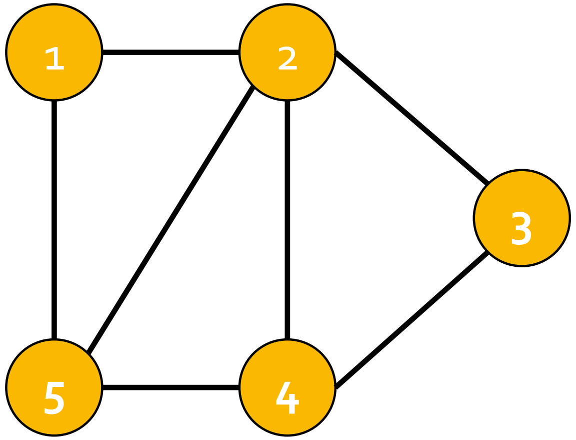


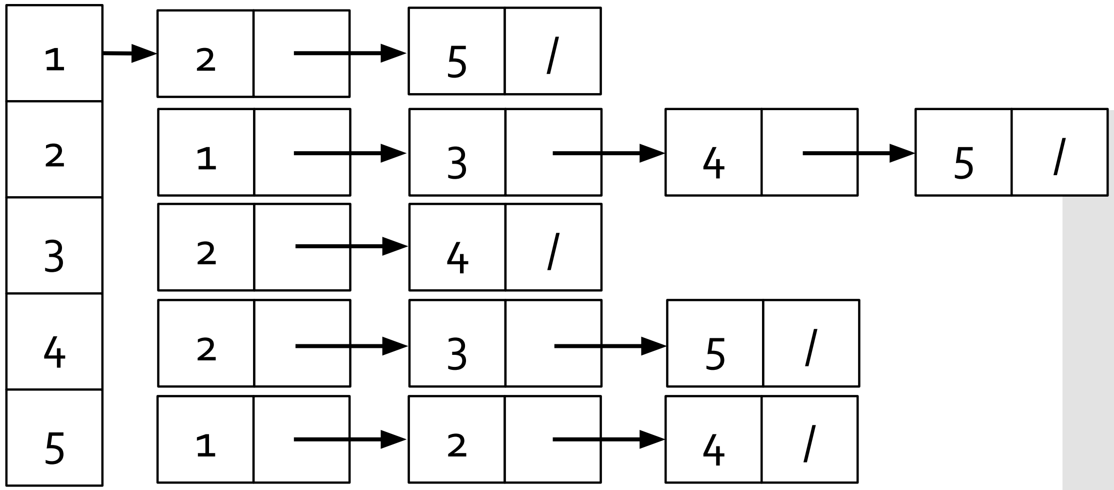


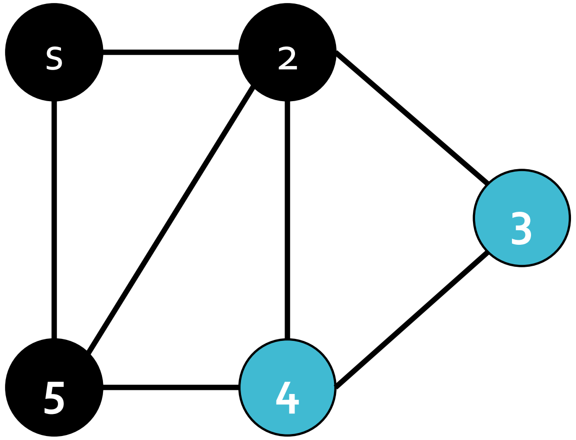


1) Queue: s
   - dequeue s and add 2,5
2) Queue: 2 5
   - dequeue 2 and add 3,4
3) Queue: 5, 3,4

BFS Algorithm

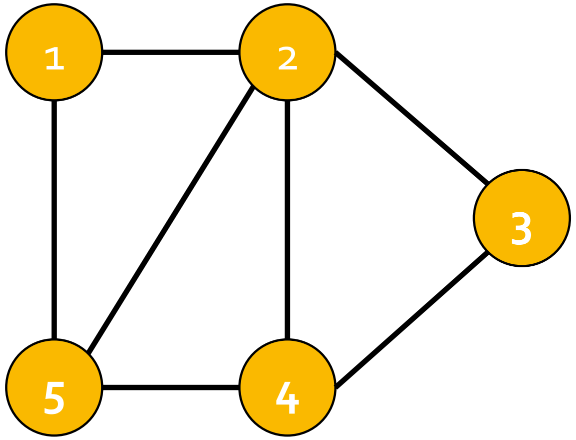


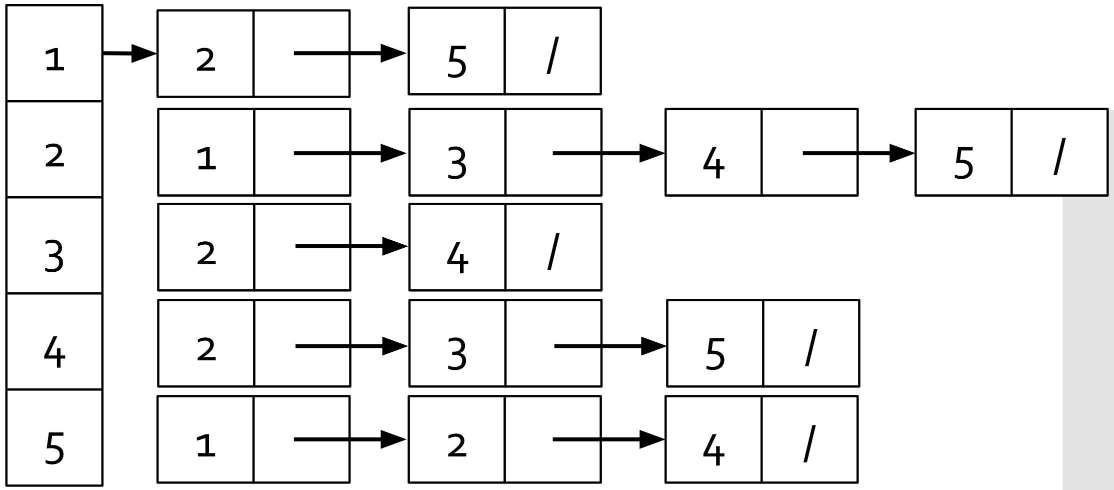


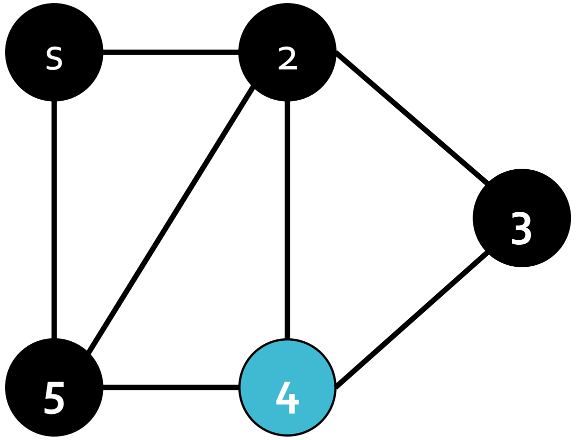


1) Queue: s
   - dequeue s and add 2,5
2) Queue: 2 5
   - dequeue 2 and add 3,4
3) Queue: 5, 3,4
   - dequeue 5
4) Queue: 3,4

BFS Algorithm

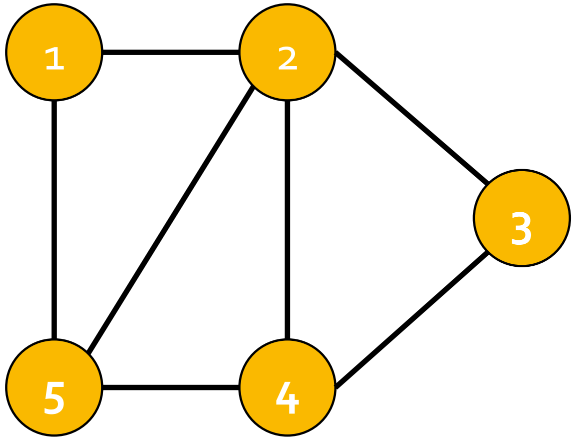


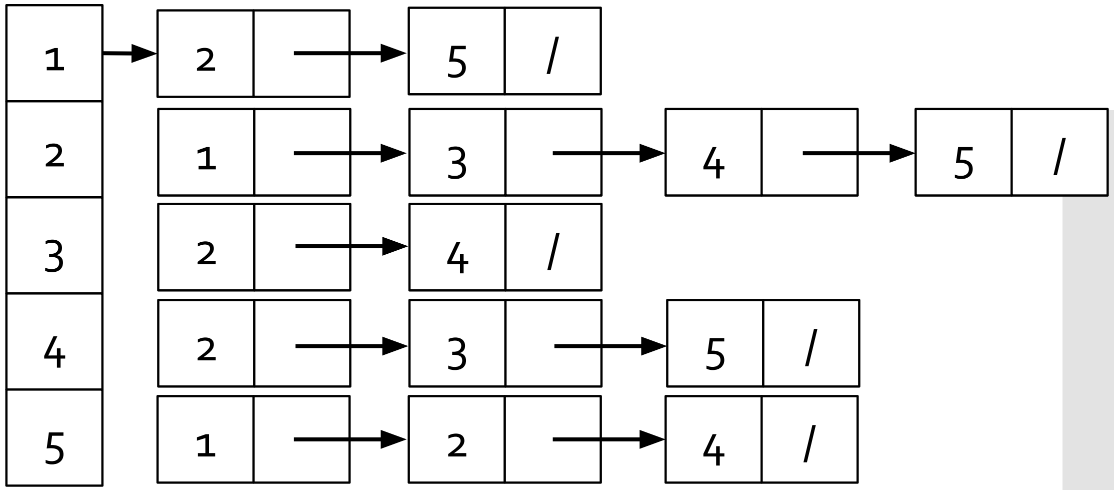


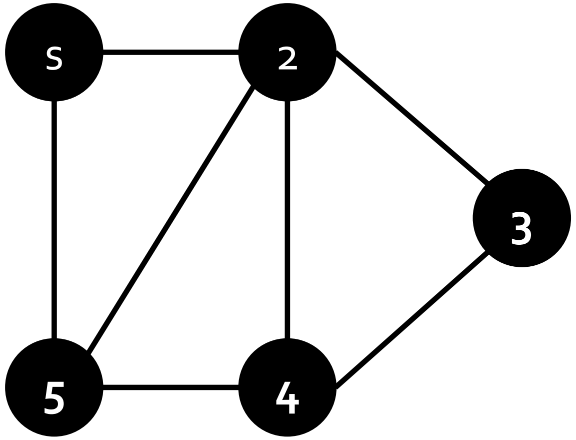


1) Queue: s
   - dequeue s and add 2,5
2) Queue: 2 5
   - dequeue 2 and add 3,4
3) Queue: 5, 3,4
   - dequeue 5
4) Queue: 3,4
   - dequeue 3
5) Queue: 4

BFS Algorithm

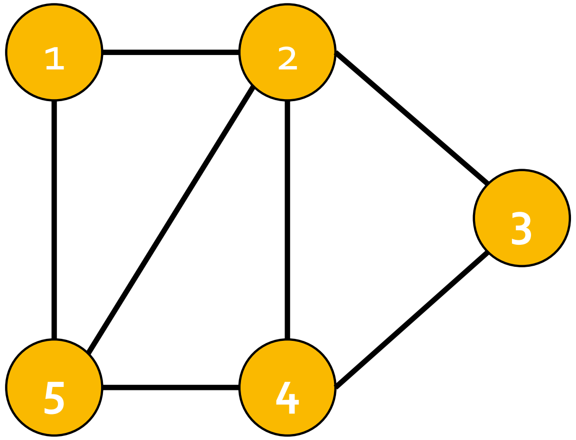


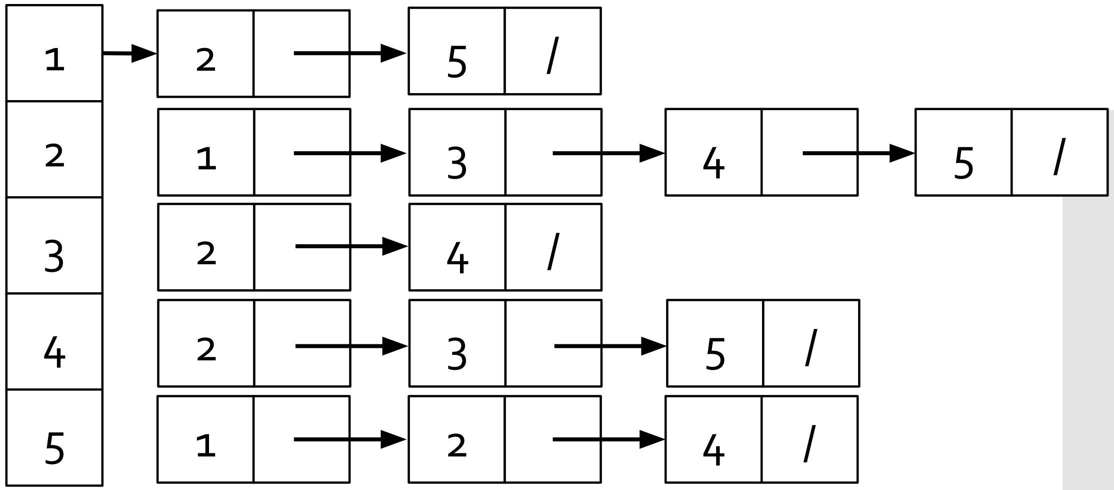


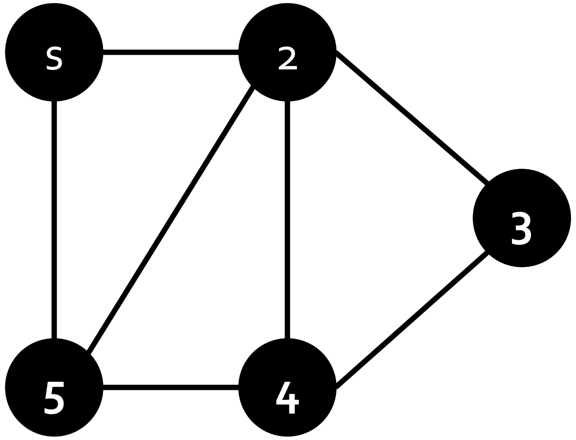


1) Queue: s
   - dequeue s and add 2,5
2) Queue: 2 5
   - dequeue 2 and add 3,4
3) Queue: 5, 3,4
   - dequeue 5
4) Queue: 3,4
   - dequeue 3
5) Queue: 4
   - dequeue 4
6) Queue: (empty, done)

BFS Algorithm







color

2	2	2	2	2

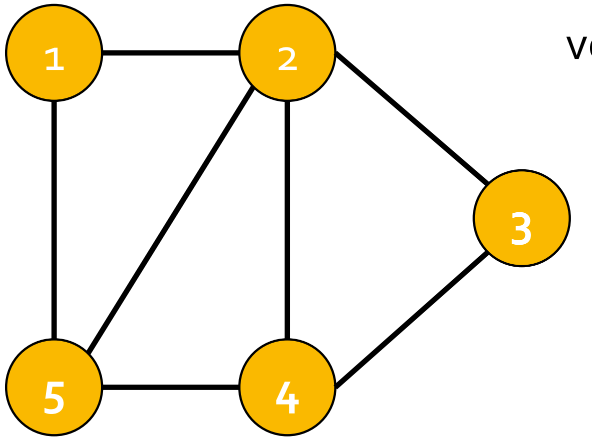
distance

0	1	2	2	1

parent

NIL	1	2	2	1

Adjacency Matrix

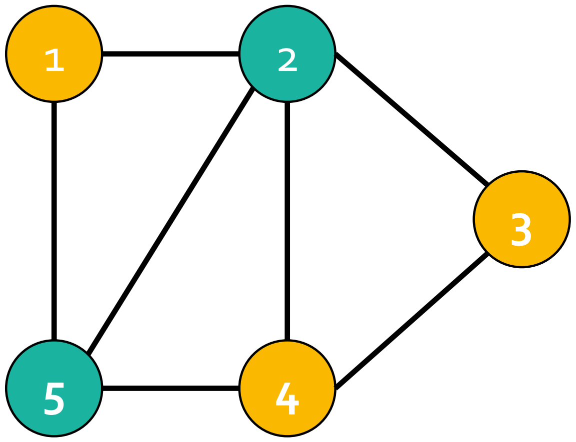


neighbor

vertex

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

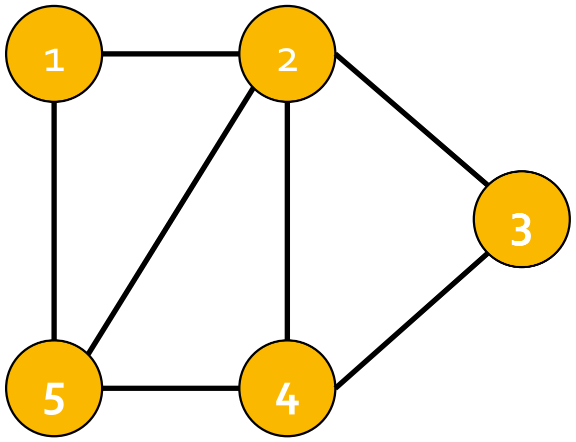
Adjacency Matrix



vertex 1's neighbors are 2 and 5

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjacency Matrix



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

vertex 5's neighbors are 1, 2, and 4

Sparse Matrix Vector Multiply

m x n matrix (A) multiplied by nx1 vector (x)
-> mx1 vector (y)

i^{th} value of y = dot product between i^{th} row of A and x

0	1	2	0	0
3	0	1	0	0
0	0	0	0	0
1	1	0	0	6
0	0	1	9	0

1
3
2
5
1

7
5
0
10
47

Dot Product

p	1	1	2	4	7
q	2	1	2	4	8

dot product = p[1] * q[1] + p[2] * q[2] + … + p[5] * q[5]

Adjacency Matrix

neighbor

vertex	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Transpose

vertex

neighbor	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Transpose:
Swap A[i][j] with A[j][i]

Adjacency Matrix

vertex

neighbor

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

1
0
0
0
0

?
?
?
?
?

Adjacency Matrix

$$
\begin{array}{ccc}
 & & \text{vertex} \\
\text{neighbor} & \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{array}
\end{array}
\qquad
\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
\qquad
\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}
$$

Adjacency Matrix

vertex

neighbor	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

1
0
0
0
0

0
1
0
0
1

equivalent to the
neighbors of vertex 1

BFS Algorithm

- BFS(G, s) // given a graph G and a source vertex s
  - For each vertex u ∈ V[G] - {s} // initialize the arrays for all but the source
    - color[u] <- 0
    - distance[u] <- ∞
    - Parent[u] <- NIL
  - color[s]= 1 // 1 because it'll be the first vertex in the queue
  - distance[s] = 0 // 0 because it's the source
  - parent[s] = NIL // NIL because it has not parents
  - Q <- { }
  - enqueue(Q, s)
  - while (Q is not empty)
    - u <- dequeue(Q) // first in first out queue
    - **for each v ∈ adjacency_list[u]**
      - if color[v] == 0 // only add if not already visited or in queue
        - color[v] = 1 // v is now in the queue
        - distance[v] <- distance[u] + 1
        - parent[v] <- u
        - enqueue(Q, v)
    - color[u] = 2 // u has not been visited and accounted for

Adjacency Matrix

vertex

neighbor	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

1
0
0
0
0

0
1
0
0
1

For corresponding vertices found,
color it appropriately,
fill in the distance
(== iteration)

Adjacency Matrix

vertex

neighbor	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

1
0
0
0
0

0
1
0
0
1

output vector becomes the input for the next iteration.

Make sure you account for already visited vertices

You are done when the result vector is all 0 (no more neighbors left to visit)