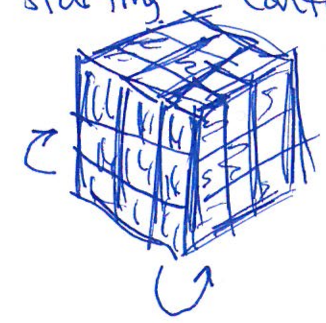
Graph Algorithms

How to solve a Rubik's cube given a starting configuration?

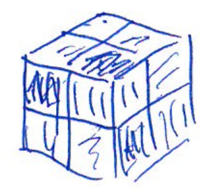


For $3\times3\times3$: 20
$2\times2\times2$: 11

How many steps suffice to solve ~~the~~ Rubik's cube?
(starting from worst position)

Simplest cube: $2\times2\times2$

- 8 cubelets
- Each cubelet has 3 orientations.



$\Rightarrow \frac{8!\cdot 3^8}{24}$ configurations $264,539,320 / 24 = 11,022,480$

($3\times3\times3$ cube has $\approx 4.3\cdot 10^{19}$ configurations)
43,252,003,274,489,856,000

Transitions: 6 faces to twist, 3 ways to twist } 18 transitions

Graph

vertices = configurations.
edges = transitions between configurations.

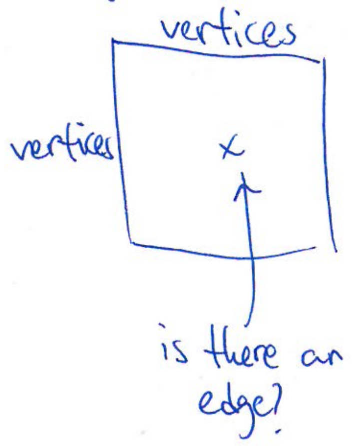
* What's the smallest number of transitions between a configuration and ~~the~~ a ~~the~~ "solved" configuration (different color on each side)?

* What's the furthest configuration from the "solved"?

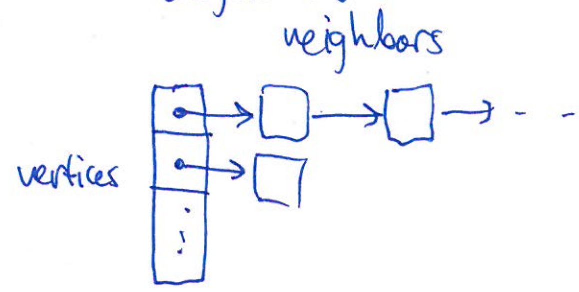
* What are the configurations reachable from "solved"?

Representation of graphs

adjacency matrix



adjacency list



implicit representation

Function: Adj(vertex, neighbor) – returns a vertex.

Search (G, s)

Input $G = (V, E)$ a graph, s a vertex in G.

Output S the set of vertices in G reachable from s.

1. $S \leftarrow \{s\}$
2. Repeat
3. Pick an edge $(x, y) \in E$ where $x \in S$, $y \notin S$ → It's useful to "mark" vertices in S so it's efficient to check whether a vertex is in S.
4. $S \leftarrow S \cup \{y\}$
5. Until no edge (x, y) as above.

Proposition Every vertex added to S is reachable from s.

Pf By induction on the number of iterations i of the loop.

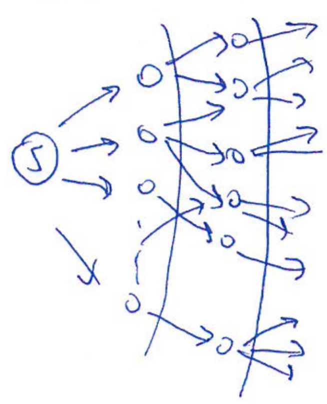
Base $i = 0$ s is reachable from s.

Assume statement is true after i repetitions of loop.

Inductive Step statement is true after $i+1$ repetitions: in the last iteration, x is reachable from s because of inductive hypothesis → y is reachable from s.

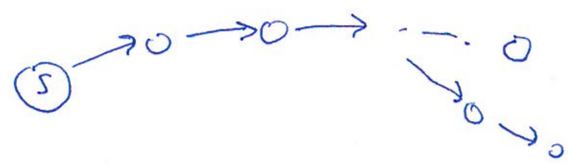
Note One can find the connected components of a graph G by: picking an arbitrary vertex $s \in S$; invoking Search(G,s) to find s's component S; removing the vertices of S from G; and continuing until G has no more vertices.

BFS
Breadth First Search



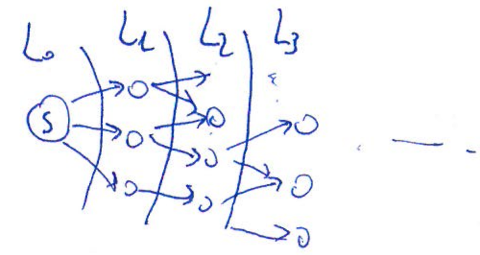
layer by layer

DFS
Depth First Search



go as far as you can; backtrack when stuck.

BFS(G, s) with $G = (V, E)$



<u>Input</u> A graph G and a vertex s in G.

<u>Output</u> $L_0, L_1, L_2 \ldots$ where L_i contains all the vertices at length i from s.

1. $L_0 \leftarrow \{s\}$, mark s.
2. $i \leftarrow 1$
3. Repeat
4. $\quad L_i \leftarrow$ all the neighbors of vertices in L_{i-1} that are unmarked.
   $\quad$ mark the vertices in L_i
5. $\quad i \leftarrow i+1$
6. Until $L_{i-1} = \emptyset$.

<u>Comment</u> At step 4 it's often useful to record for each vertex in L_i what was the vertex in L_{i-1} that neighbored it. ("parent")
This creates a tree structure ("BFS tree").

<u>Proposition</u> $\forall i \geq 0$ Every vertex added to L_i is at length i from s.

<u>Pf</u> By induction on i.

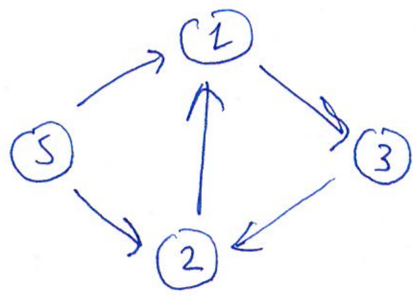
<u>Base</u> $i=0$. $L_0 = \{s\}$ and s is the only vertex at length 0 from s.

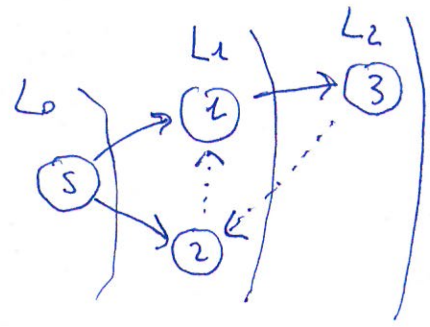
<u>Assume</u> statement is ~~true~~ up to certain i.

<u>Inductive Step</u> we claim that the statement is true for $i+1$: $s \overset{i}{\rightsquigarrow} u \to v$

* every vertex v in L_{i+1} is a neighbor of a vertex u in L_i, hence there exists a length $i+1$ path from s to v.
* if there were a shorter path $s \rightsquigarrow v$ then by hypothesis v should have been in L_j for $j \leq i$, not in L_{i+1}.

Example





BFS run-time

Steps 1-2 O(1) time

Step 5 O(1) time for every iteration $\leq O(|V|)$ time

Step 4 O(1) time per edge $\leq O(|E|)$ time.

also setting up the graph so all vertices are unmarked takes $O(|V|)$

Overall: $O(|V|+|E|)$ time.

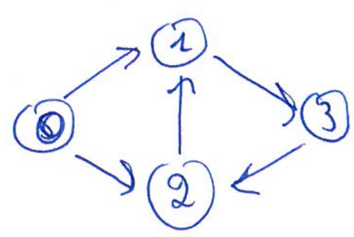
Applications

- shortest path for unweighted graphs
- Web crawling (eg. Google indexing)
- social networking (eg. "people you might know")
- network broadcast
- garbage collection
- model checking ..

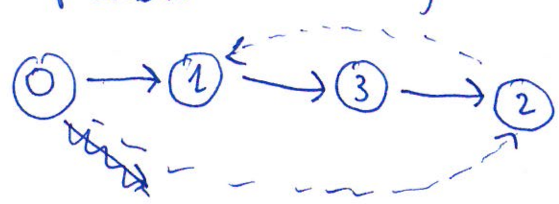
DFS(G, s)

1. Mark s
2. For each neighbor v of s
3. if unmarked then DFS(G, v).

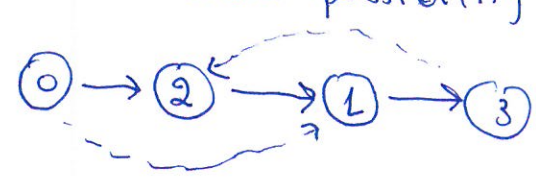
Example



one possible DFS starting 0:



a different possibility starting 0:



DFS run-time

Step 1 O(1) for every vertex = $O(|V|)$.

Step 2 O(1) for every edge = $O(|E|)$

Applications

- cycle detection
- topological sort
- navigating mazes.