Graph Algorithms

L.F.O.A.
Lecture Full Of Acronyms

LFOA Fun Poll:
Which acronym(s) will we not learn about today

- AFS
- BFS
- CFS
- DFS
- MST
- AFSOC
The most basic graph algorithms:

**BFS:** Breadth-first search

**DFS:** Depth-first search

**AFS:** Arbitrary-first search

What problems do these algorithms solve?

---

**Graph Search Algorithms**

Given a graph $G = (V,E)...$

- Check if vertex $s$ can reach vertex $t$.
- Decide if $G$ is connected.
- Identify connected components of $G$.

All reduce to:

“Given $s \in V$, identify all nodes reachable from $s$.”

(We’ll call this set $\text{CONNCOMP}(s)$.)

Algorithm $\text{AFS}(G,s)$ does exactly this.

---

**Bonus of $\text{AFS}(G,s)$:**

Finds a **spanning tree** of $\text{CONNCOMP}(s)$ rooted at $s$.

Given $G = (V,E)$, a **spanning tree** is a tree $T = (V,E')$ such that $E' \subseteq E$.

More informally, a minimal set of edges connecting up all vertices of $G$. 
Bonus of $\text{AFS}(G,s)$:
Finds a **spanning tree** of $\text{CONNCOMP}(s)$ rooted at $s$.

Given $G = (V,E)$, a **spanning tree** is a tree $T = (V,E')$ such that $E' \subseteq E$. 

---

Bonus of $\text{AFS}(G,s)$:
Finds a **spanning tree** of $\text{CONNCOMP}(s)$ rooted at $s$.

Given $G = (V,E)$, a **spanning tree** is a tree $T = (V,E')$ such that $E' \subseteq E$. 

---

Bonus of $\text{AFS}(G,s)$:
Finds a **spanning tree** of $\text{CONNCOMP}(s)$ rooted at $s$.

Given $G = (V,E)$, a **spanning tree** is a tree $T = (V,E')$ such that $E' \subseteq E$. 

---
AFS(G,s): Finding all nodes reachable from s

G

“Duh, it’s these ones.”

But it’s not so obvious when the input looks like...

AFS(G,s): Finding all nodes reachable from s

V = { a,b,c,p,q,r,s,t,u,v,w,x,y,z }

E = { {a,b}, {a,c}, {b,c}, {p,q}, {p,x}, {q,r}, {q,s}, {r,y}, {s,u}, {s,x}, {s,y}, {t,u}, {t,x}, {u,v}, {v,y}, {w,x}, {y,z} }

AFS(G,s):

// Has a “bag” data structure holding
// Each tile has a vertex name written on it
Put into bag
While bag is not empty:
  Pick an Arbitrary tile from bag
  if v is “unmarked”:
    “Mark” v
    For each neighbor w of v:
      Put into bag

Intent:

“Marked” vertices should be those reachable from s.

in bag means we want to keep exploring from w.
\textbf{AFS(G,s):}

- **Put** 5 \text{ into bag}
- While bag is not empty:
  - Pick arbitrary tile $\checkmark$ from bag
  - If $v$ is "unmarked":
    - "Mark" $v$
    - For each neighbor $w$ of $v$:
      - Put $w$ into bag
AFS(G,s):
Put $s$ into bag
While bag is not empty:
Pick arbitrary tile $v$ from bag
If $v$ is "unmarked":
"Mark" $v$
For each neighbor $w$ of $v$:
Put $w$ into bag
AFS(G,s):
Put 5 into bag
While bag is not empty:
Pick arbitrary tile v from bag
If v is "unmarked":
  "Mark" v
  For each neighbor w of v:
  Put w into bag
AFS(G.s):
Put 5 into bag
While bag is not empty:
→ Pick arbitrary tile x from bag
  If v is "unmarked":
    "Mark" v
    For each neighbor w of v:
    Put w into bag

AFS(G.s):
Put 5 into bag
While bag is not empty:
→ Pick arbitrary tile x from bag
  If v is "unmarked":
    "Mark" v
    For each neighbor w of v:
    Put w into bag

AFS(G.s):
Put 5 into bag
While bag is not empty:
→ Pick arbitrary tile x from bag
  If v is "unmarked":
    "Mark" v
    For each neighbor w of v:
    Put w into bag
AFS(G,s):
Put 1 into bag
While bag is not empty:
Pick arbitrary tile from bag
If v is "unmarked":
"Mark" v
For each neighbor w of v:
Put w into bag
AFS(G,s):
Put 5 into bag
While bag is not empty:
Pick arbitrary tile \( v \) from bag
If \( v \) is “unmarked”:
“Mark” \( v \)
For each neighbor \( w \) of \( v \):
Put \( w \) into bag
Analysis of AFS

Want to show: When this algorithm halts,

\[
\{ \text{marked vertices} \} = \{ \text{vertices reachable from } s \}.
\]

\{ \text{marked} \} \subseteq \{ \text{reachable} \}: This is clear.
\{ \text{reachable} \} \subseteq \{ \text{marked} \}:

Wait, why does the algorithm even halt?!

Why does AFS halt?

Every time a bunch of tiles is added to bag,
    it's because some vertex \( v \) just got marked.

\* we add at most \( |V| \) bunches of tiles to the bag (since each vertex is marked \( \leq 1 \) time).
\* at most finitely many tiles ever go into the bag.

Each iteration through loop removes 1 tile.

\* AFS halts after finitely many iterations.

<table>
<thead>
<tr>
<th>AFS((G,s)):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put ( s ) into bag</td>
</tr>
<tr>
<td>While bag is not empty:</td>
</tr>
<tr>
<td>Pick arbitrary tile ( v ) from bag</td>
</tr>
<tr>
<td>If ( v ) is &quot;unmarked&quot;:</td>
</tr>
<tr>
<td>&quot;Mark&quot; ( v )</td>
</tr>
<tr>
<td>For each neighbor ( w ) of ( v ):</td>
</tr>
<tr>
<td>Put ( w ) into bag</td>
</tr>
</tbody>
</table>

A more careful analysis

Every time a bunch of tiles is added to bag,
    it's because some vertex \( v \) just got marked.

In this case, we add \( \deg(v) \) tiles to the bag.

\* total number of tiles that ever enter the bag is

\[ \leq \sum_{v \in V} \deg(v) = 2|E| \]

Each iteration through loop removes 1 tile.

\* AFS halts after finitely many iterations.
A more careful analysis

Every time a bunch of tiles is added to bag, it’s because some vertex \( v \) just got marked.

In this case, we add \( \deg(v) \) tiles to the bag.

\[
\sum_{v \in V} \deg(v) = 2|E|
\]

Each iteration through loop removes 1 tile.

\* AFS halts after \( \leq 2|E| \) many iterations.

**AFS(G,s):**
- Put \( G \) into bag
- While bag is not empty:
  - Pick arbitrary tile \( G \) from bag
  - If \( v \) is “unmarked”:
    - “Mark” \( v \)
    - For each neighbor \( w \) of \( v \):
      - Put \( w \) into bag

A more careful analysis

Every time a bunch of tiles is added to bag, it’s because some vertex \( v \) just got marked.

In this case, we add \( \deg(v) \) tiles to the bag.

\[
\sum_{v \in V} \deg(v) = 2|E|
\]

Each iteration through loop removes 1 tile.

\* AFS halts after \( \leq 2|E|+1 \) many iterations.

**AFS(G,s):**
- Put \( G \) into bag
- While bag is not empty:
  - Pick arbitrary tile \( G \) from bag
  - If \( v \) is “unmarked”:
    - “Mark” \( v \)
    - For each neighbor \( w \) of \( v \):
      - Put \( w \) into bag

When a tile \( w \) is added to the bag, it gets there “because of” a neighbor \( v \) that was just marked.

(Except for the initial \( s \).)

Let’s actually record this info on the tile, writing \( v \rightarrow w \).

Meaning: “We want to keep exploring from \( w \).
By the way, we got to \( w \) from \( v \).”

(And we’ll write \( L \rightarrow S \) initially.)
AFS(G,s):
Put $\square$ into bag
While bag is not empty:
   Pick an Arbitrary tile $\square$ from bag
   If $v$ is "unmarked":
      "Mark" $v$
      For each neighbor $w$ of $v$:
         Put $\square$ into bag

AFS(G,s):
Put $\blacksquare$ into bag
While bag is not empty:
   Pick an Arbitrary tile $\blacksquare$ from bag
   If $v$ is "unmarked":
      "Mark" $v$
      For each neighbor $w$ of $v$:
         Put $\blacksquare$ into bag

AFS(G,s):
Put $\blacklozenge$ into bag
While bag is not empty:
   Pick an Arbitrary tile $\blacklozenge$ from bag
   If $v$ is "unmarked":
      "Mark" $v$ and record parent($v$) := $p$
      For each neighbor $w$ of $v$:
         Put $\blacklozenge$ into bag
AFS(G, s):
Put $1 \rightarrow s$ into bag
While bag is not empty:
    Pick an arbitrary tile $p \rightarrow w$ from bag
    If $v$ is "unmarked":
        "Mark" $v$ and record $\text{parent}(v) := p$
        For each neighbor $w$ of $v$:
            Put $w \rightarrow w$ into bag

Suppose the next few tiles pulled are $6 \rightarrow 2$, $6 \rightarrow 5$, $7 \rightarrow 3$.
Then AFS would reach the following state...

Suppose the next few tiles pulled are $6 \rightarrow 2$, $6 \rightarrow 5$, $7 \rightarrow 3$.
Then AFS would reach the following state...
Then remaining tiles would be pulled & discarded.
AFS(G,s):
Put \( s \) into bag
While bag is not empty:
    Pick an Arbitrary tile \( p \) from bag
    If \( p \) is "unmarked":
        "Mark" \( p \) and record parent(\( p \)) := \( s \)
        For each neighbor \( w \) of \( p \):
            Put \( \langle -w \rangle \) into bag

Theorem: Every vertex in \( \text{CONNComp}(s) \) gets marked.

Theorem: Every vertex in \( \text{CONNComp}(s) \) gets marked.

Equivalently: For all vertices \( y \), if there's a path from \( s \) to \( y \) of length \( k \), then \( y \) gets marked.

Proof: By induction on \( k \).
    Base case \( k = 0 \): Indeed, \( s \) gets marked.
    Induction step: Suppose it's true for some \( k \in \mathbb{N} \).
    Now suppose \( \exists \) a length-(\( k+1 \)) path from \( s \) to some \( y \).
    Write it as \( (s, ..., x, y) \). So \( (s, ..., x) \) is a length-\( k \) path.
    By induction, \( x \) gets marked.
    When \( x \) gets marked by the algorithm, \( \langle x \rightarrow y \rangle \) goes in bag.
    We proved the bag eventually empties.
    Thus \( \langle x \rightarrow y \rangle \) will come out, and the algorithm will mark \( y \).

So we've proved AFS(G,s) indeed marks \( \text{CONNComp}(s) \).

From now on, let's assume \( \text{CONNComp}(s) \) is all of \( G \).

Corollary: The parent() information recorded by AFS encodes a spanning tree of \( G \) rooted at \( s \).

Proof:
It certainly records a bunch of edges.
Each vertex in \( G \), except \( s \), has exactly one parent edge.
Thus there are \( |V| - 1 \) edges.
Further, it's clear that for all vertices \( v \),
parent(parent(\( \ldots \)parent(\( v \)\( \ldots \)))) must reach \( s \).
\( \checkmark \) all vertices are connected to \( s \), hence to each other.
\( \checkmark \) parent edges form a tree (\(|V| - 1 \) edges, connected).
Instantiations of AFS

**DFS: Depth-First Search**

When the bag is a “stack”. LIFO: Last-In First-Out.
(Assume sorted adjacency list representation.)

![Diagram of DFS: Depth-First Search]

(Actually implemented using an array)

---

**DFS: Depth-First Search**

When the bag is a “stack”. LIFO: Last-In First-Out.
(Assume sorted adjacency list representation.)

![Diagram of DFS: Depth-First Search]

(Actually implemented using an array)
DFS: Depth-First Search

When the bag is a "stack".
LIFO: Last-In First-Out.

(Actually implemented using an array.)

DFS: Depth-First Search

When the bag is a "stack".
LIFO: Last-In First-Out.

(Actually implemented using an array.)

DFS: Depth-First Search

When the bag is a "stack".
LIFO: Last-In First-Out.

(Actually implemented using an array.)
RecursiveDFS(v)
if v unmarked
    mark v
    for each w ∈ N(v)
        RecursiveDFS(w)
**BFS: Breadth-First Search**

When the bag is a "queue".
FIFO: First-In First-Out.

(Usually implemented using a linked list)

(Assume sorted adjacency list representation.)
BFS: Breadth-First Search

When the bag is a "queue".
FIFO: First-In First-Out.

(Assume sorted adjacency list representation.)

(usually implemented using a linked list)
**BFS: Breadth-First Search**

When the bag is a "queue".
FIFO: First-in First-Out.

**BFS bonus property:**
Vertices marked in increasing order of distance from \( s \).

**BFS**(G, s)
...
parent(v) := p
\( \text{dist}(v) := \text{dist}(\text{parent}(v)) + 1 \)
...

(usually implemented using a linked list)

---

**BFS: Breadth-First Search**

When the bag is a "queue".
FIFO: First-In First-Out.

**BFS bonus property:**
Vertices marked in increasing order of distance from \( s \).

**Exercise:** Prove this.
So path from \( s \) to any \( v \) in BFS tree is a shortest path.

---

**BFS & DFS: Running time**

Put \( s \) into bag
While bag is not empty:
Pick an Arbitrary tile \( v \) from bag
If \( v \) is "unmarked":
* "Mark" \( v \) and record parent(v) := p
For each neighbor \( w \) of \( v \):
Put \( v \) into bag

Recall: # of tiles put in bag is \( \leq 2|E|+1 \).
Actually, exactly \( 2|E|+1 \), assuming G connected.
Bag operations are \( O(1) \) time for stack/queue.
Each tile engenders \( O(1) \) work.
\( \bullet \) Total run-time: \( O(|E|) \).
BFS & DFS: Running time

AFS(G,s) just finds the connected component of s.

What if we want to find all connected components?

FullAFS(G):
For all vertices v:
If v is unmarked
AFS(G,v)

Overall run-time: \(O(|V|+|E|)\) (Why?)

We have seen AFS, BFS, DFS

Looks like we’re missing something...

CFS! Cheapest-First Search

The goal of CFS is more ambitious than just finding connected components.

Its goal is to find a minimum spanning tree (MST).

Weighted Graphs

Often in life, each edge of a graph \(G = (V,E)\) will have a real number associated to it.

Variously called:

- weight
- length
- distance
- cost.

“Cost function”, \(c : E \rightarrow \mathbb{R}^+\)

Positive values only, unless otherwise specified.
MST
The year: 1926
The place: Brno, Moravia
Our hero: Otakar Borůvka

Borůvka’s had a pal called Jindřich Saxel who worked for Západoravské elektrárny (the West Moravian Power Plant company).

Saxel asked him how to figure out the most efficient way to electrify southwest Moravia.

MST
Edge exists if it’s feasible to connect two towns by power lines.
Edge weights might be distance in km, or cost in 1000’s of koruna to install lines.

MST
Minimum Spanning Tree (MST) problem:
Input: A weighted graph $G = (V,E)$, with cost function $c : E \rightarrow \mathbb{R}^+$.
Output: Subset of edges of minimum total cost such that all vertices connected.

The edges will form a tree:
If you had a cycle, you could delete any edge on it and still be connected, but cheaper.
MST

Example:

In this case, there's a unique solution, of cost $5+2+3+12+16+4=42$.

MST

Convenient assumption: Edges have distinct costs.
In this case, not hard to show the MST is unique.
Thus we can speak of the MST, not just an MST.

A hint for the little trick that shows this is WLOG:

“Whether [the] distance from Brno to Brčlav is 50 km or 50 km and 1 cm is a matter of conjecture.”

MST via Cheapest-First Search

Often known as Prim's Algorithm, due to a 1957 publication by Robert C. Prim.

Actually first discovered by Vojtěch Jarník, who described it in a letter to Borůvka, and published it in 1930.

Borůvka himself had published a different algorithm in 1926.
MST via Cheapest-First Search

Put $\{s\}$ into bag
While bag is not empty:
Pick an Arbitrary edge $[p \rightarrow q]$ from bag
If $v$ is "unmarked":
  "Mark" $v$, record $parent(v) := p$
  For each neighbor $w$ of $v$:
    Put $[w \rightarrow v]$ into bag

JARNÍK-PRIM(G):
Let $s$ be any vertex
Put $\{s\}$ into bag
While bag is not empty:
  Pick the cheapest edge $[p \rightarrow q]$ from bag
  If $v$ is "unmarked":
    "Mark" $v$, record $parent(v) := p$
    For each neighbor $w$ of $v$:
      Put $[w \rightarrow v]$ into bag

Naive implementation: Unsorted list.

$O(|E|)$ time to scan for cheapest edge.
$O(|E|^2)$ total run-time.

JARNÍK-PRIM(G): Sophisticated implementation: "Priority Queue".

$O(\log |E|)$ time for both bag operations.
$O(|E| \log |E|)$ total run-time.
MST via Cheapest-First Search

**Effectively:** CFS grows a tree from \( s \), always adding the cheapest edge next.

**Example:**

```
    8 10 5
   /  /  \
 10 12 3
   \  /  \
    4 14 26
```

MST via Cheapest-First Search

**Theorem:** JARNÍK–PRIM finds the MST.

MST via Cheapest-First Search

**Theorem:** For each \( 0 \leq k \leq n-1 \), the first \( k \) edges added are all in the MST.

**Proof:** By induction on \( k \).

Base case \( k=0 \): Vacuously true.

Induction step: Suppose CFS has added \( k \) edges so far \( (0 \leq k < n-1) \), and all are in MST. We need to show next added edge is also in MST.
MST via Cheapest-First Search
Let $S$ be the set of vertices connected to $s$ so far,

Let $S$ be the set of vertices connected to $s$ so far, and let $e = (v, w)$ be next edge added by CFS.
(By definition of CFS, $e$ is the cheapest edge out of $S$.)
Let $T$ be the MST for $G$.
AFSOC that $e \notin T$.
Since $T$ spans $G$, must exist a path from $v$ to $w$ in $T$.

Let $S$ be the set of vertices connected to $s$ so far, and let $e = (v, w)$ be next edge added by CFS.
(By definition of CFS, $e$ is the cheapest edge out of $S$.)
Let $T$ be the MST for $G$.
AFSOC that $e \notin T$.
Since $T$ spans $G$, must exist a path from $v$ to $w$ in $T$.
Let $e' = (v', w')$ be first edge on that path which exits $S$. 
MST via Cheapest-First Search
Let \( S \) be the set of vertices connected to \( s \) so far, and let \( e = \{v,w\} \) be next edge added by CFS.
(By definition of CFS, \( e \) is the cheapest edge out of \( S \).)
Let \( T \) be the MST for \( G \).
AFSOC that \( e \notin T \).
Since \( T \) spans \( G \), must exist a path from \( v \) to \( w \) in \( T \).
Let \( e' = \{v',w'\} \) be first edge on that path which exits \( S \).

MST via Cheapest-First Search
Claim: \( T' := T - e' \cup \{e\} \) is a spanning tree.
If true, we have a contradiction because \( \text{cost}(e') > \text{cost}(e) \) (why?) and so \( \text{cost}(T') > \text{cost}(T) \).
\( T' \) has \( |V| - 1 \) edges, so we just need to check it's still connected.
Any walk in \( T \) formerly using \( e' = \{v,w\} \) can now take path from \( v' \) to \( v \), then take \( e \), then take path from \( w \) to \( w' \).

Look carefully at our proof that \( e \notin \text{MST} \).
We didn't actually use the fact that the edges inside \( S \) were part of the MST.
All we used: \( e \) was the cheapest edge out of \( S \).
Thus we more generally proved...
**MST Cut Property:**

Let $G=(V,E)$ be a graph with distinct edge costs. Let $S \subseteq V$ (with $S \neq \emptyset$, $S \neq V$). Let $e \in E$ be the cheapest edge with one endpoint in $S$ and the other not in $S$. Then a minimum spanning tree must contain $e$.

---

**MST Cut Property**

Using this, it’s not hard to show that practically any natural “greedy” MST algorithm works.

**Kruskal’s Algorithm:**

Go through edges in order of cheapness. Add edge as long as it doesn’t make a cycle.

**Borůvka’s Algorithm:**

Start with each vertex a connected component. Repeatedly: add the cheapest edge coming out of each connected component.

---

**Run-time Race for MST (an amusing story)**

The “classical” (pre-1960) MST algorithms, Borůvka, Jarník–Prim, Kruskal, all run in time $O(m \log m)$.

That is very good.

In practice, these algorithms are great.

Nevertheless, algorithms & data structures wizards tried to do better.
Run-time Race for MST

1984: Fredman & Tarjan invent the “Fibonacci heap” data structure.
Run-time improved from $O(m \log(m))$ to $O(m \log^*(m))$.

Remember $\log^*(m)$?
It is the number of times you need to take $\log$ to get down to 2.
For all real-world purposes, $\log^*(m) \leq 5$.

Run-time Race for MST

1984: Fredman & Tarjan invent the “Fibonacci heap” data structure.
Run-time improved from $O(m \log(m))$ to $O(m \log^*(m))$.

Also not Fredman

Not Fredman

Tarjan

Run-time Race for MST

1986: Gabow, Galil, T. Spencer, Tarjan improved the algorithm.
Run-time improved from $O(m \log^*(m))$ to...
$O(m \log (\log^*(m)))$. 
Run-time Race for MST

1986: Gabow, Gail, T. Spencer, Tarjan improved the algorithm.
Run-time improved from $O(m \log^*(m))$ to...
$O(m \log \log^*(m))$.

Gabow  Galil  Tarjan & Not-Spencer

Run-time Race for MST

1997: Chazelle invents “soft heap” data structure.
Run-time improved from $O(m \log(\log^*(m)))$ to...
$O(m \alpha(m) \log(\alpha(m)))$.
I will tell you what function $\alpha(m)$ is in a second.
I assure you, it’s comically slow-growing.

Chazelle

Run-time Race for MST

2000: Chazelle improves it down to $O(m \alpha(m))$.

$\alpha(m)$ is called the Inverse-Ackermann function.

$\log^*(m) = \#$ of times you need to do $\log$ to get down to 2
$\log^**(m) = \#$ of times you need to do $\log^*$ to get down to 2
$\log^**(m) = \#$ of times you need to do $\log^**$ to get down to 2
...
$\alpha(m) = \#$ of $*’s$ you need so that $\log^{**...**(m)} \leq 2$
It’s incomprehensibly, preposterously slow-growing!
Run-time Race for MST

1995: Meanwhile, Karger, Klein, and Tarjan give an algorithm with run-time $O(m)$.
It’s a randomized algorithm: $O(m)$ is the expected value of the running time.

Karger  Klein  Tarjan

Run-time Race for MST

2002: Pettie and Ramachandran gave a new deterministic MST algorithm.
They proved its running time is $O(\text{optimal})$.

Pettie  Ramachandran

Run-time Race for MST

2002: Pettie and Ramachandran gave a new deterministic MST algorithm.
They proved its running time is $O(\text{optimal})$.
Would you like to know its running time?
So would we.
Its running time is unknown.
All we know is: whatever it is, it’s optimal.
Study Guide

Definition:
Minimum Spanning Tree

Algorithms and analysis:
AFS
BFS
DFS
CFS (Jarník-Prim algorithm)