CS 350 Algorithms and Complexity

Winter 2019

Lecture 6: Exhaustive Search Algorithms

Andrew P. Black

Department of Computer Science

Portland State University

Brute Force

- A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved
- Examples:

    Computing a^n ($a > 0$, n a nonnegative integer) by repeated multiplication

    Computing $n!$ by repeated multiplication

    Multiplying two matrices following the definition

    Searching for a key in a list sequentially

Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010
- Pattern: happy
  Text: It is never too late to have a happy childhood.
-

Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010

- Pattern: happy
  Text: It is never too late to have a happy childhood.

-

Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010
- Pattern: happy
  Text: It is never too late to have a happy childhood.
-

Portland State UNIVERSITY

Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010

- Pattern: happy
  Text: It is never too late to have a happy childhood.

-

Portland State UNIVERSITY

Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010
- Pattern: happy
  Text: It is never too late to have a happy childhood.
-

Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010

- Pattern: happy
  Text: It is never too late to have a happy childhood.

-

Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010

- Pattern: happy
  Text: It is never too late to have a happy childhood.

-

Portland State University

Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010

- Pattern: happy
  Text: It is never too late to have a happy childhood.

-

Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010

- Pattern: happy
  Text: It is never too late to have a happy childhood.

-

Portland State University

Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010

- Pattern: happy
  Text: It is never too late to have a happy childhood.

-

Portland State UNIVERSITY

Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010

- Pattern: happy
  Text: It is never too late to have a happy childhood.

-

Pseudocode and Efficiency

Pseudocode and Efficiency

```
ALGORITHM  BruteForceStringMatch(T[0..n − 1], P[0..m − 1])
    //Implements brute-force string matching
    //Input: An array T[0..n − 1] of n characters representing a text and
    //       an array P[0..m − 1] of m characters representing a pattern
    //Output: The index of the first character in the text that starts a
    //        matching substring or −1 if the search is unsuccessful
    for i ← 0 to n − m do
        j ← 0
        while j < m and P[j] = T[i + j] do
            j ← j + 1
        if j = m return i
    return −1
```

Pseudocode and Efficiency

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//          an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//            matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Efficiency: A: O(n) B: O(m(n-m)) C: O(m) D: $O(m^2)$

Brute-Force Polynomial Evaluation

Brute-Force Polynomial Evaluation

- Problem: Find the value of polynomial

  $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$ at a point $x = x_0$

Portland State UNIVERSITY

Brute-Force Polynomial Evaluation

- Problem: Find the value of polynomial

  $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$ at a point $x = x_0$

- Brute-force algorithm

```
p ← 0.0
for i ← n downto 0 do
    power ← 1
    for j ← 1 to i do  //compute x^i
        power ← power * x
    p ← p + a[i] * power
return p
```

Brute-Force Polynomial Evaluation

- Problem: Find the value of polynomial

  $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$ at a point $x = x_0$

- Brute-force algorithm

```
p ← 0.0
for i ← n downto 0 do
    power ← 1
    for j ← 1 to i do  //compute x^i
        power ← power * x
    p ← p + a[i] * power
return p
```

- Efficiency:

Brute-Force Polynomial Evaluation

- Problem: Find the value of polynomial

  $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$ at a point $x = x_0$

- Brute-force algorithm

```
p ← 0.0
for i ← n downto 0 do
    power ← 1
    for  j ← 1 to i do  //compute x^i
        power ← power * x
    p ← p + a[i] * power
return p
```

- Efficiency: A: $O(n)$ B: $O(n^2)$ C: $O(\lg n)$ D: $O(n^3)$

Portland State University

Polynomial Evaluation: Improvement

Polynomial Evaluation: Improvement

- We can do better by evaluating from right to left:

Polynomial Evaluation: Improvement

- We can do better by evaluating from right to left:
- Better brute-force algorithm:

Portland State UNIVERSITY

Polynomial Evaluation: Improvement

- We can do better by evaluating from right to left:
- Better brute-force algorithm:

Polynomial Evaluation: Improvement

- We can do better by evaluating from right to left:
- Better brute-force algorithm:

```
p ← a[0]
power ← 1
for i ← 1 to n do
    power ← power ∗ x
    p ← p + a[i] ∗ power
return p
```

Polynomial Evaluation: Improvement

- We can do better by evaluating from right to left:
- Better brute-force algorithm:

```
p ← a[0]
power ← 1
for i ← 1 to n do
    power ← power ∗ x
    p ← p + a[i] ∗ power
return p
```

Polynomial Evaluation: Improvement

- We can do better by evaluating from right to left:
- Better brute-force algorithm:

```
p ← a[0]
power ← 1
for i ← 1 to n do
    power ← power ∗ x
    p ← p + a[i] ∗ power
return p
```

- Efficiency:

Polynomial Evaluation: Improvement

- We can do better by evaluating from right to left:
- Better brute-force algorithm:

$p \leftarrow a[0]$
$\text{power} \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
  $\text{power} \leftarrow \text{power} * x$
  $p \leftarrow p + a[i] * \text{power}$
return p

- Efficiency: A: $O(n)$ B: $O(n^2)$ C: $O(\lg n)$ D: $O(n^3)$

Closest-Pair Problem

- Find the two closest points in a set of n points (in the two-dimensional Cartesian plane).
- Brute-force algorithm:
  - Compute the distance between every pair of distinct points
    - and return the indices of the points for which the distance is the smallest.

Closest-Pair Brute-Force Algorithm (cont.)

ALGORITHM *BruteForceClosestPoints*(P)

//Finds two closest points in the plane by brute force
//Input: A list P of n ($n \geq 2$) points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
//Output: Indices $index1$ and $index2$ of the closest pair of points
$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    for $j \leftarrow i + 1$ **to** n **do**
        $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //$sqrt$ is the square root function
        if $d < dmin$
            $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$
return $index1, index2$

Closest-Pair Brute-Force Algorithm (cont.)

ALGORITHM *BruteForceClosestPoints*(P)

//Finds two closest points in the plane by brute force
//Input: A list P of n ($n \geq 2$) points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
//Output: Indices $index1$ and $index2$ of the closest pair of points
$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //$sqrt$ is the square root function
    if $d < dmin$
      $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$
return $index1$, $index2$

- Efficiency:

Closest-Pair Brute-Force Algorithm (cont.)

ALGORITHM *BruteForceClosestPoints*(P)

//Finds two closest points in the plane by brute force
//Input: A list P of n ($n \geq 2$) points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
//Output: Indices $index1$ and $index2$ of the closest pair of points
$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    for $j \leftarrow i + 1$ **to** n **do**
        $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //$sqrt$ is the square root function
        if $d < dmin$
            $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$
return $index1, index2$

- Efficiency: A: $O(n)$ B: $O(n^2)$ C: $O(\lg n)$ D: $O(n^3)$

Closest-Pair Brute-Force Algorithm (cont.)

ALGORITHM *BruteForceClosestPoints*(P)

//Finds two closest points in the plane by brute force
//Input: A list P of n ($n \geq 2$) points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
//Output: Indices $index1$ and $index2$ of the closest pair of points
$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //$sqrt$ is the square root function
    if $d < dmin$
      $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$
return $index1, index2$

- Efficiency: A: $O(n)$ B: $O(n^2)$ C: $O(\lg n)$ D: $O(n^3)$
- How to make it faster?

Problem:

ALGORITHM *BruteForceClosestPoints*(P)

//Finds two closest points in the plane by brute force
//Input: A list P of n $(n \geq 2)$ points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
//Output: Indices $index1$ and $index2$ of the closest pair of points
$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    for $j \leftarrow i + 1$ **to** n **do**
        $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //$sqrt$ is the square root function
        if $d < dmin$
            $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$
return $index1$, $index2$

If *sqrt* is 10 x slower than × and +, by how much will *BruteForceClosestPoints* speed up when we take out the *sqrt* ?

A. ~ 10 times

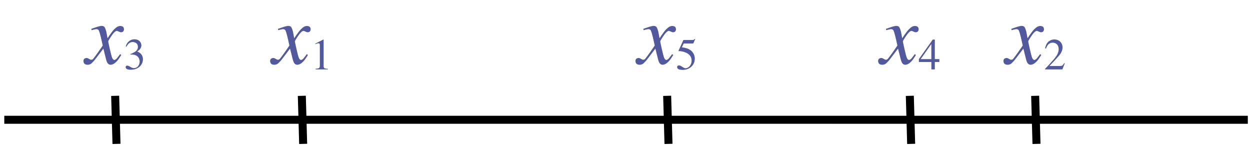
B. ~ 100 times

C. ~ 1000 times

Portland State University

Problem:

Can you design a more efficient algorithm than the one based on the brute-force strategy to solve the closest-pair problem for n points $x_1, \ldots, x_n$ on the real line?



Portland State University

Brute Force Closest Pair

- An Example of a particular kind of Brute Force Algorithm based on:

  Exhaustive search

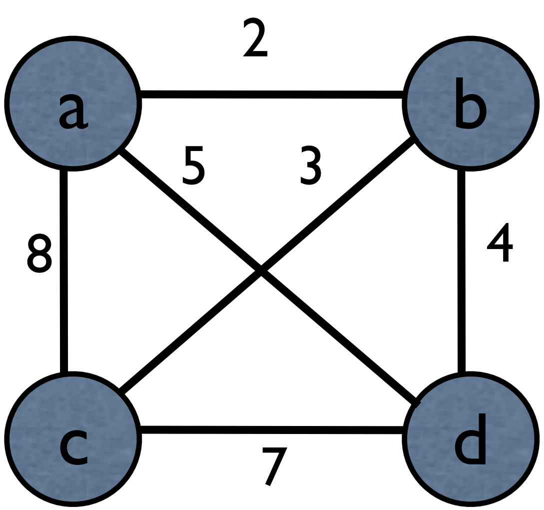
Portland State UNIVERSITY

Exhaustive Search

- A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.
- Method:
  - generate a list of all potential solutions to the problem in a systematic manner (see algorithms in Sec. 4.3)
  - evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
  - when search ends, announce the solution(s) found

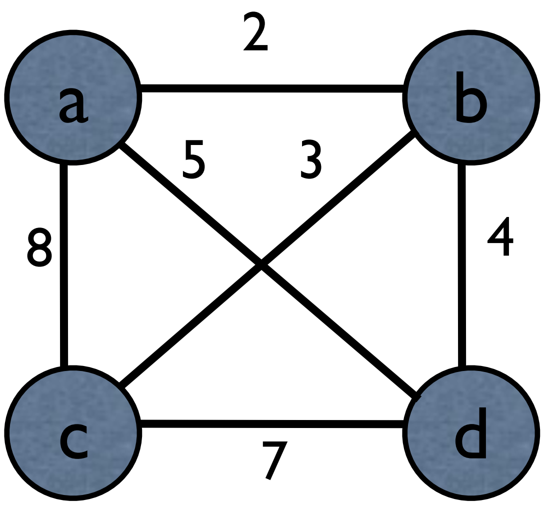
Example 1: Traveling Salesman Problem

- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
- Alternatively: find shortest Hamiltonian circuit in a weighted connected graph
- Example:



TSP by Exhaustive Search

Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17



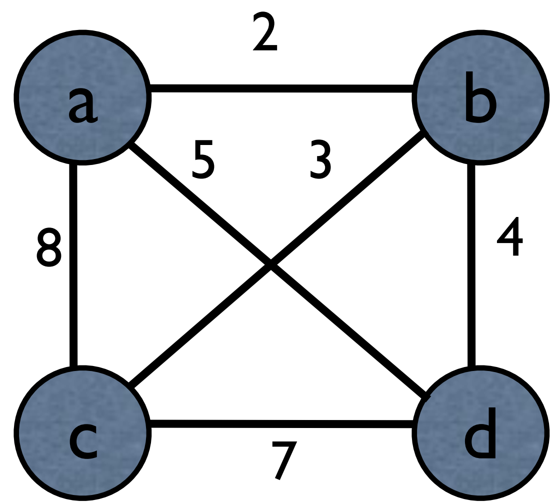
More tours?

Less tours?

Efficiency:

TSP by Exhaustive Search

Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17



More tours?

Less tours?

Efficiency:

A: $O(n)$
B: $O(n^2)$
C: $O(n^3)$
D: $O((n-1)!)$
E: $O(n!)$

Portland State UNIVERSITY

Example 2: Knapsack Problem

- Given *n* items:
  - weights: w_1 w_2 … w_n
  - values: v_1 v_2 … v_n
  - a knapsack of capacity W
- Find most valuable subset of the items that fit into the knapsack
- Example: Knapsack capacity W=16

item	weight	value
1.	2	$20
2.	5	$30
3.	10	$50
4.	5	$10

Knapsack Problem by Exhaustive Search

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	infeasible
{1,2,4}	12	$60
{1,3,4}	17	infeasible
{2,3,4}	20	infeasible
{1,2,3,4}	22	infeasible

item	weight	value
1.	2	$20
2.	5	$30
3.	10	$50
4.	5	$10

Knapsack capacity W=16

Knapsack Problem by Exhaustive Search

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	infeasible
{1,2,4}	12	$60
{1,3,4}	17	infeasible
{2,3,4}	20	infeasible
{1,2,3,4}	22	infeasible

item	weight	value
1.	2	$20
2.	5	$30
3.	10	$50
4.	5	$10

Knapsack capacity W=16

Knapsack Problem by Exhaustive Search

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	infeasible
{1,2,4}	12	$60
{1,3,4}	17	infeasible
{2,3,4}	20	infeasible
{1,2,3,4}	22	infeasible

item	weight	value
1.	2	$20
2.	5	$30
3.	10	$50
4.	5	$10

Knapsack capacity W=16

- Efficiency?
  A: $O(n^2)$
  B: $O(2^n)$
  C: O(n!)
  D: O((n-1)!)

Portland State University

Example 3: The Assignment Problem

- There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *p* to job *j* is C[*i*, *j*]. Find an assignment that minimizes the total cost.

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

- Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.
- How many assignments are there …

Example 3: The Assignment Problem

- There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *p* to job *j* is C[*i*, *j*]. Find an assignment that minimizes the total cost.

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

an *assignment* $\langle a_1, a_2, a_3, a_4 \rangle$ means that person *i* gets job a_i

- Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.

- How many assignments are there …

Example 3: The Assignment Problem

- There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *p* to job *j* is C[*i*, *j*]. Find an assignment that minimizes the total cost.

		Job 1	Job 2	Job 3	Job 4
$\langle a_1,$	Person 1	9	2	7	8
$a_2,$	Person 2	6	4	3	7
$a_3,$	Person 3	5	8	1	8
$a_4\rangle$	Person 4	7	6	9	4

an *assignment* $\langle a_1, a_2, a_3, a_4\rangle$ means that person *i* gets job a_i

- Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.

- How many assignments are there …

Example 3: The Assignment Problem

- There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *p* to job *j* is C[*i*, *j*]. Find an assignment that minimizes the total cost.

		Job 1	Job 2	Job 3	Job 4
⟨2,	Person 1	9	2	7	8
4,	Person 2	6	4	3	7
3,	Person 3	5	8	1	8
1⟩	Person 4	7	6	9	4

an *assignment* $\langle a_1, a_2, a_3, a_4 \rangle$ means that person *i* gets job a_i

- Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.

- How many assignments are there …

Example 3: The Assignment Problem

- There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *p* to job *j* is C[*i*, *j*]. Find an assignment that minimizes the total cost.

		Job 1	Job 2	Job 3	Job 4
⟨2,	Person 1	9	2	7	8
4,	Person 2	6	4	3	7
3,	Person 3	5	8	1	8
1⟩	Person 4	7	6	9	4

an *assignment* $\langle a_1, a_2, a_3, a_4 \rangle$ means that person *i* gets job a_i

- Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.
- How many assignments are there …

Portland State University

Example 3: The Assignment Problem

- There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *p* to job *j* is C[*i*, *j*]. Find an assignment that minimizes the total cost.

		Job 1	Job 2	Job 3	Job 4
⟨2,	Person 1	9	2	7	8
4,	Person 2	6	4	3	7
3,	Person 3	5	8	1	8
1⟩	Person 4	7	6	9	4

an *assignment* $\langle a_1, a_2, a_3, a_4 \rangle$ means that person *i* gets job a_i

- Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.
- How many assignments are there …

Example 3: The Assignment Problem

- There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *p* to job *j* is C[*i*, *j*]. Find an assignment that minimizes the total cost.

		Job 1	Job 2	Job 3	Job 4
⟨2,	Person 1	9	2	7	8
4,	Person 2	6	4	3	7
3,	Person 3	5	8	1	8
1⟩	Person 4	7	6	9	4

an *assignment* $\langle a_1, a_2, a_3, a_4 \rangle$ means that person *i* gets job a_i

- Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.

- How many assignments are there …

Example 3: The Assignment Problem

- There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *p* to job *j* is C[*i*, *j*]. Find an assignment that minimizes the total cost.

		Job 1	Job 2	Job 3	Job 4
⟨2,	Person 1	9	2	7	8
4,	Person 2	6	4	3	7
3,	Person 3	5	8	1	8
1⟩	Person 4	7	6	9	4

an *assignment* $\langle a_1, a_2, a_3, a_4 \rangle$ means that person *i* gets job a_i

- Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.

- How many assignments are there …

Assignment Problem by Exhaustive Search

Assignment Problem by Exhaustive Search

- Consider the problem in terms of the *Cost Matrix C*

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

Assignment Problem by Exhaustive Search

- Consider the problem in terms of the *Cost Matrix C*

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30

Assignment Problem by Exhaustive Search

- Consider the problem in terms of the *Cost Matrix C*

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24

Assignment Problem by Exhaustive Search

- Consider the problem in terms of the *Cost Matrix C*

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26

Portland State University

Assignment Problem by Exhaustive Search

- Consider the problem in terms of the *Cost Matrix C*

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33

Assignment Problem by Exhaustive Search

- Consider the problem in terms of the *Cost Matrix C*

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33
1, 4, 3, 2	9+7+1+6=23

Assignment Problem by Exhaustive Search

- Consider the problem in terms of the *Cost Matrix C*

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33
1, 4, 3, 2	9+7+1+6=23

etc.

Assignment Problem by Exhaustive Search

- Consider the problem in terms of the *Cost Matrix C*

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33
1, 4, 3, 2	9+7+1+6=23

etc.

How many assignments are there?

Assignment Problem by Exhaustive Search

- Consider the problem in terms of the *Cost Matrix C*

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33
1, 4, 3, 2	9+7+1+6=23

etc.

How many assignments are there?

A: $O(n)$ B: $O(n^2)$ C: $O(n^3)$ D: $O(n!)$

Convex Hulls

- What is a Convex Hull?

A. A bad design for a boat
B. A good design for a boat
C. A set of points without any concavities
D. None of the above

Convex Hulls

- What is a Convex Set?

A. A bad design for a boat
B. A good design for a boat
C. A set of points without any concavities
D. None of the above

Convex Hulls

- What is a Convex Set?

Portland State UNIVERSITY

Convex Hulls

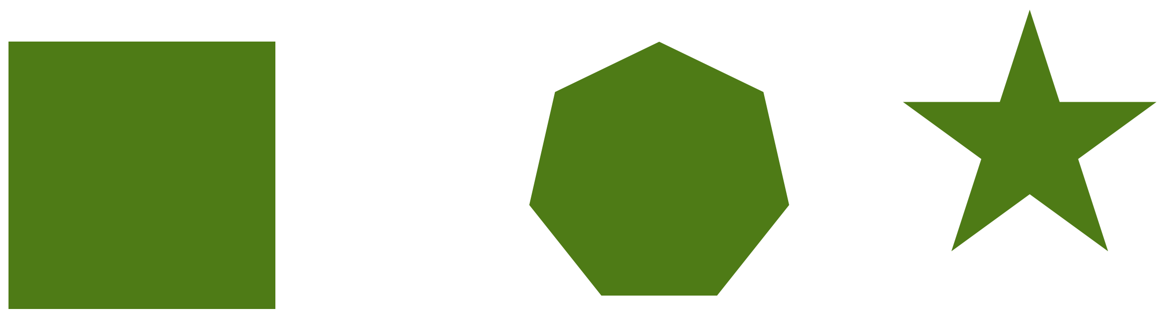
- What is a Convex Set?

Convex Hulls

- What is a Convex Set?

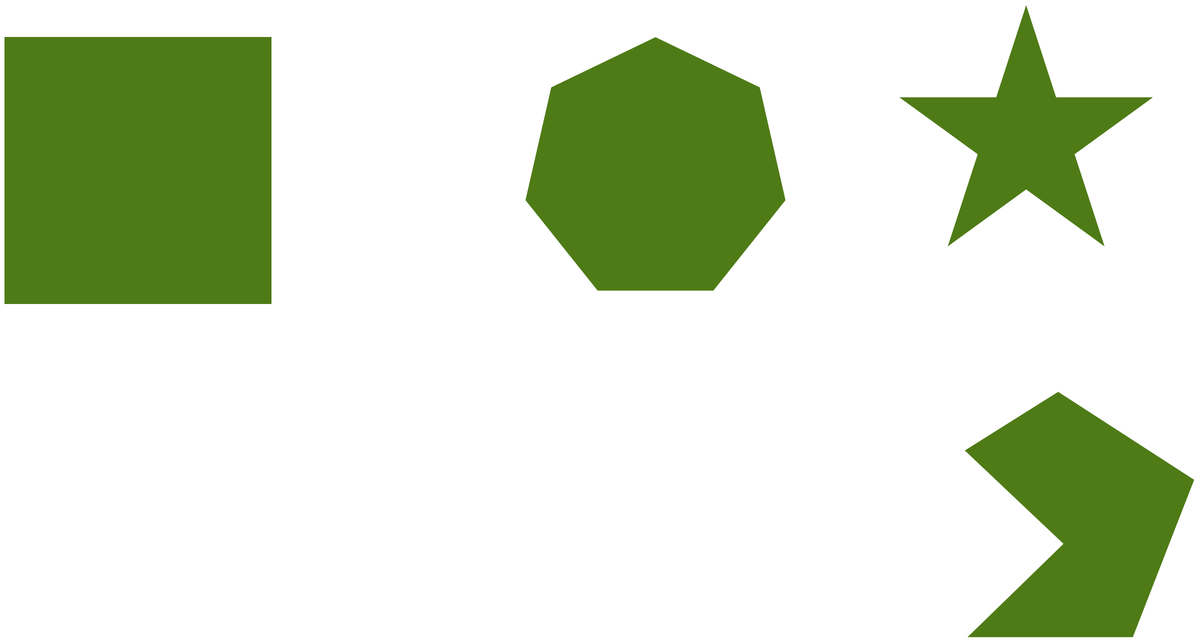
Convex Hulls

- What is a Convex Set?



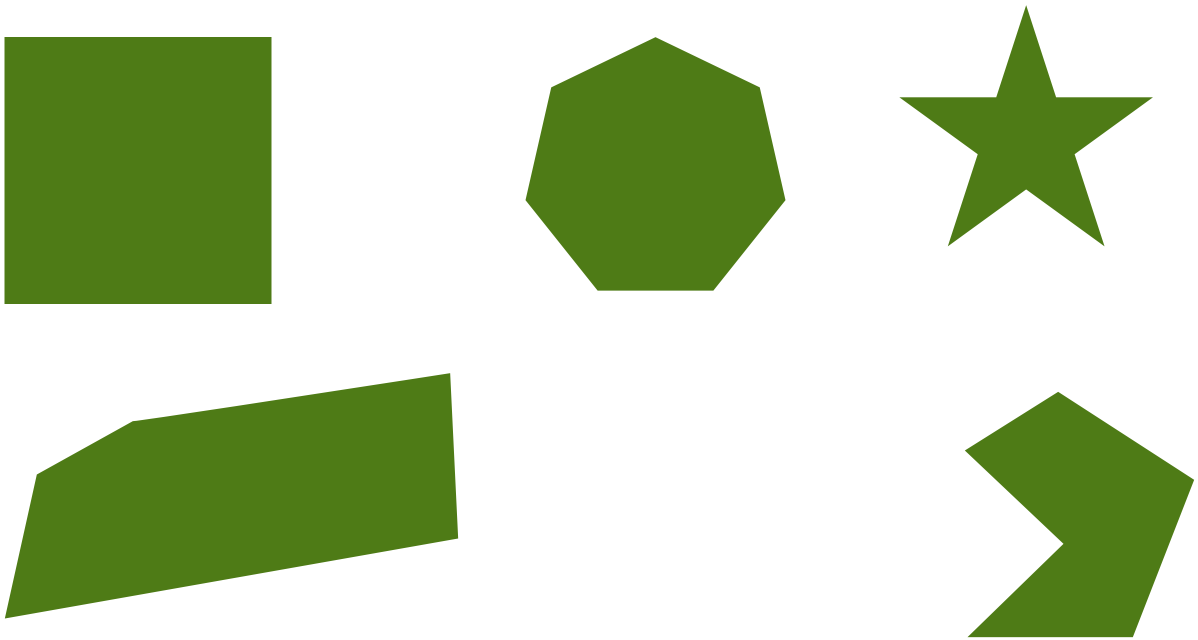
Convex Hulls

- What is a Convex Set?



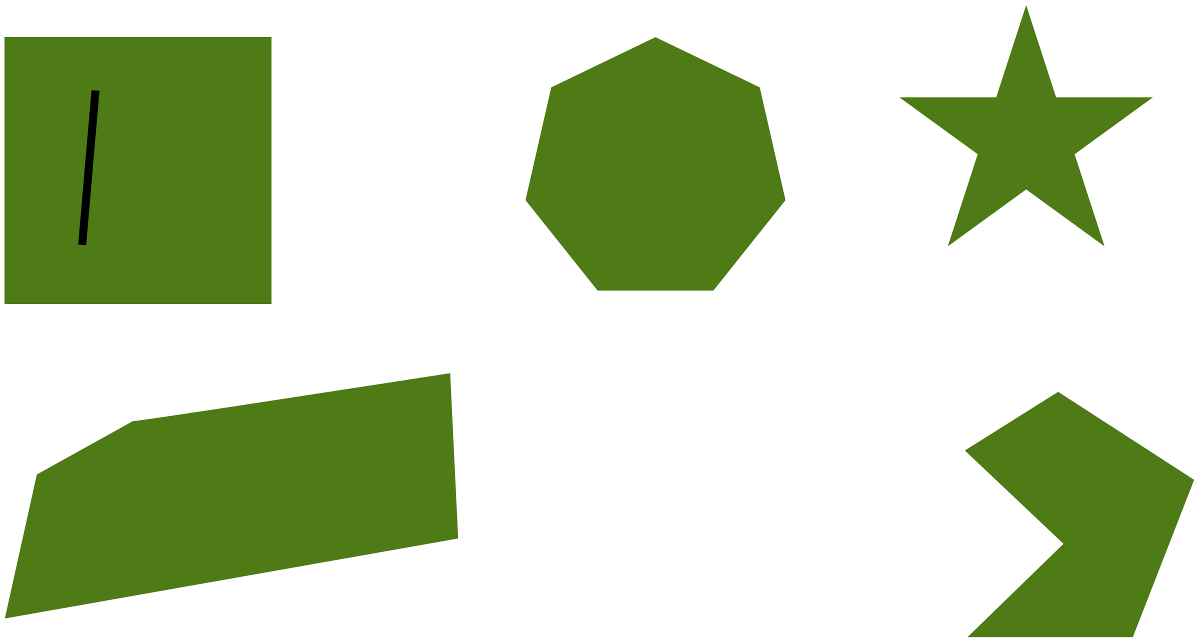
Convex Hulls

- What is a Convex Set?



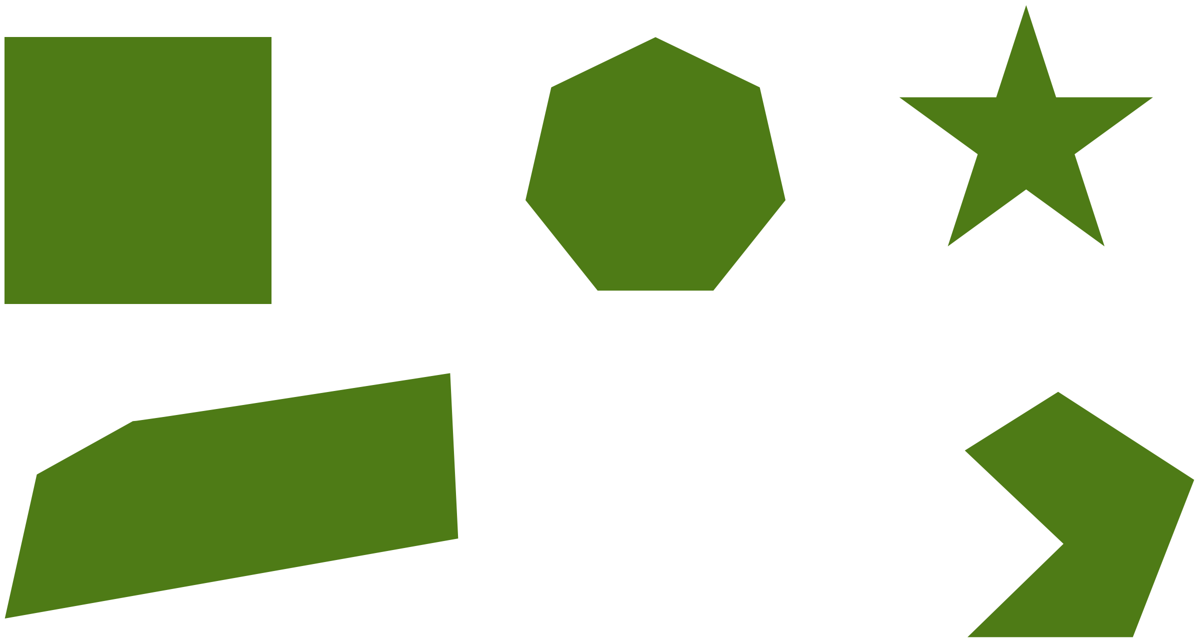
Convex Hulls

- What is a Convex Set?



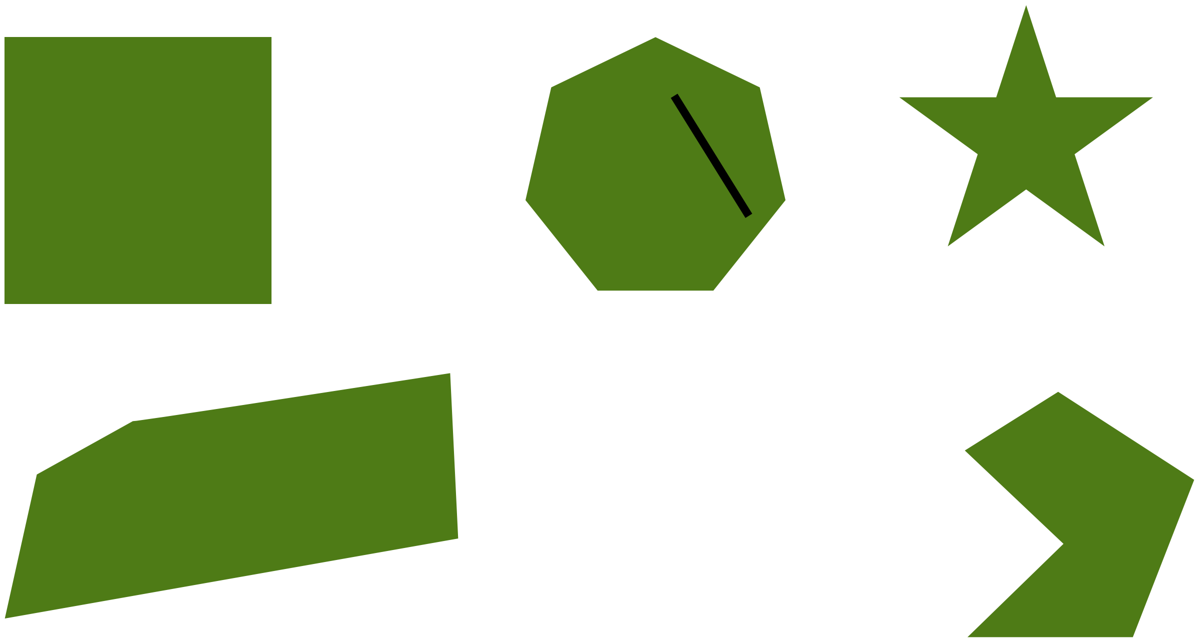
Convex Hulls

- What is a Convex Set?



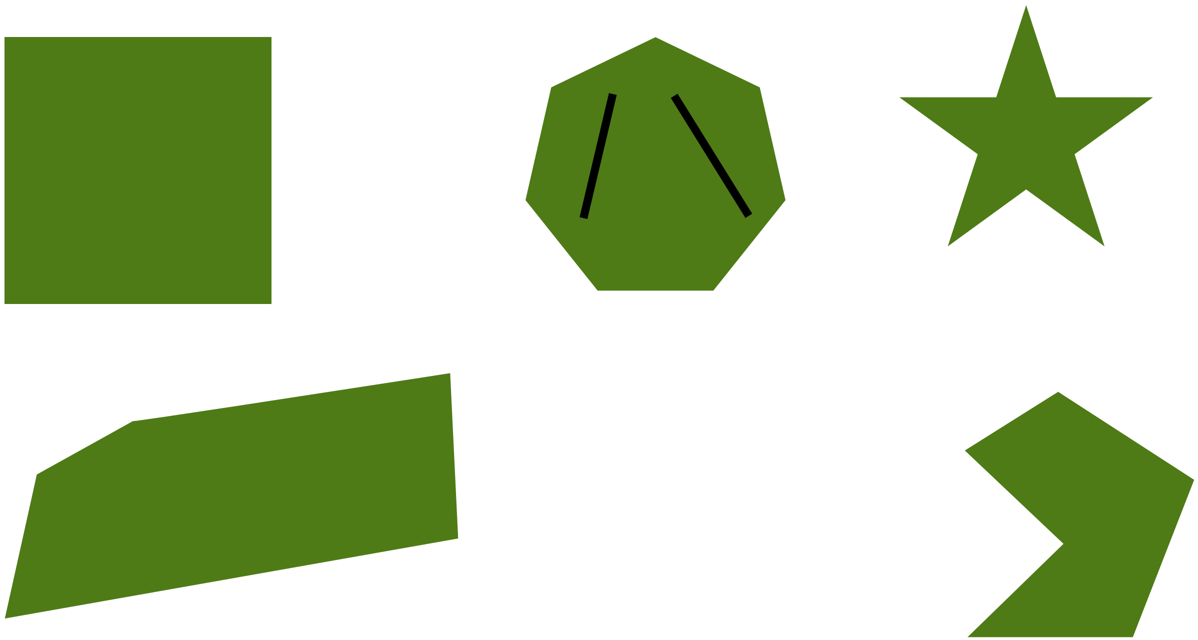
Convex Hulls

- What is a Convex Set?



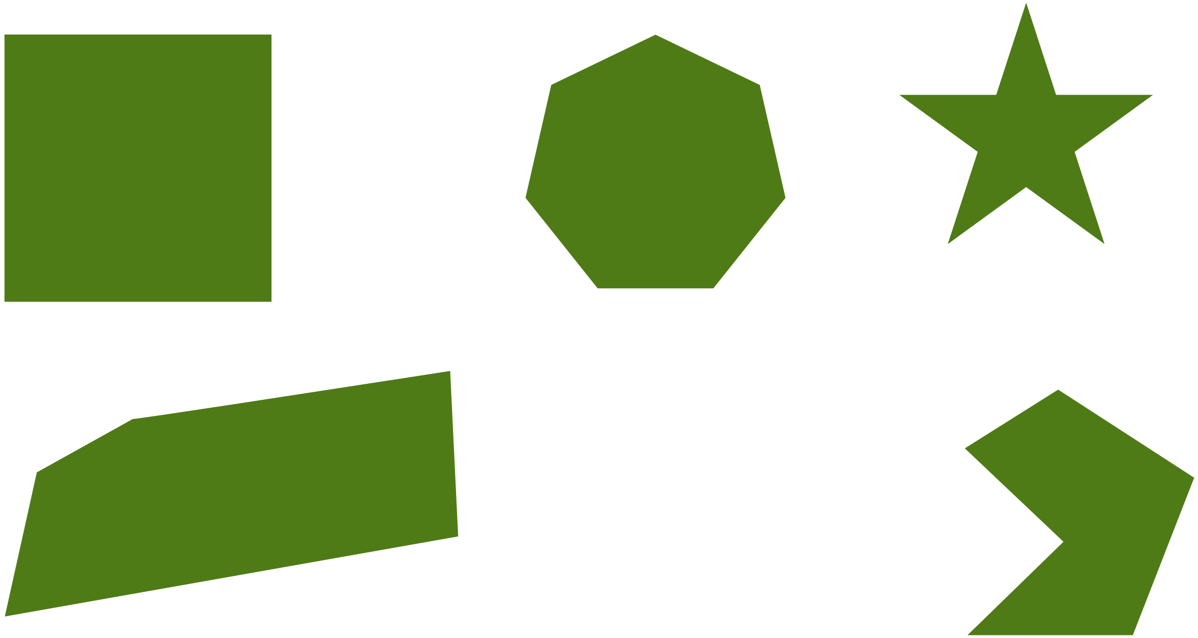
Convex Hulls

- What is a Convex Set?



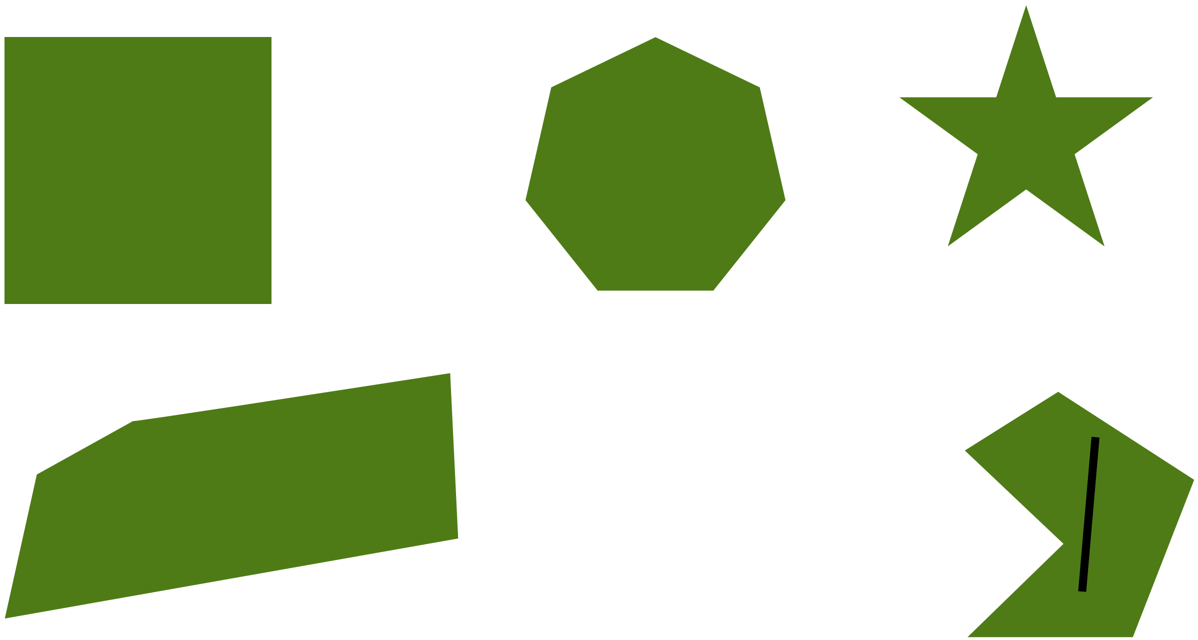
Convex Hulls

- What is a Convex Set?



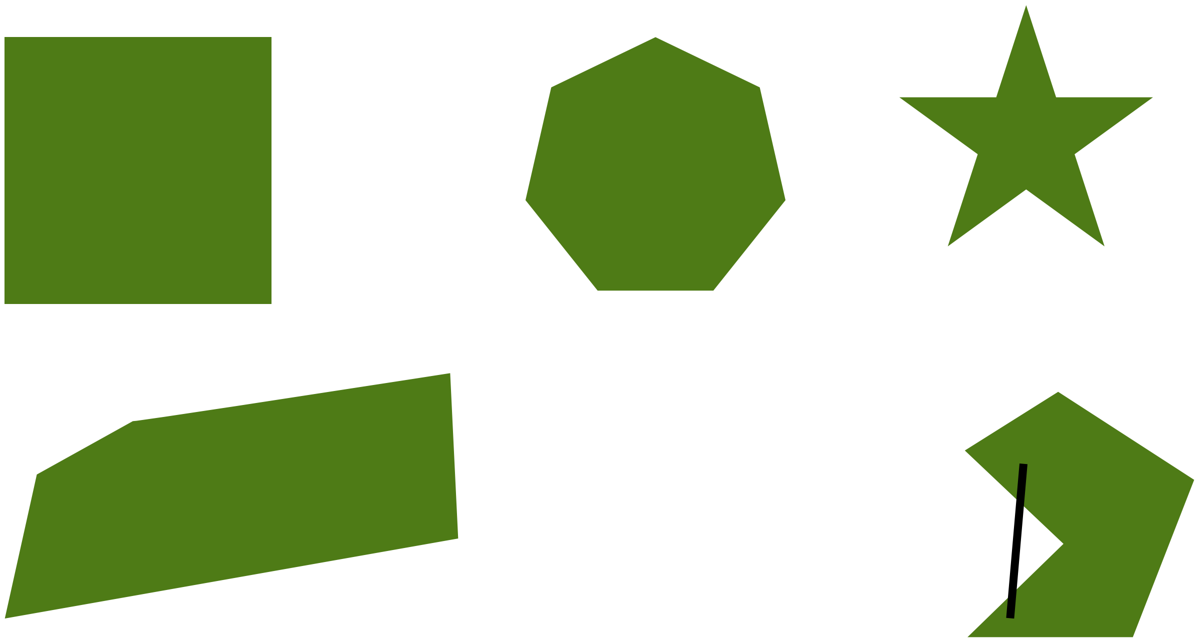
Convex Hulls

- What is a Convex Set?



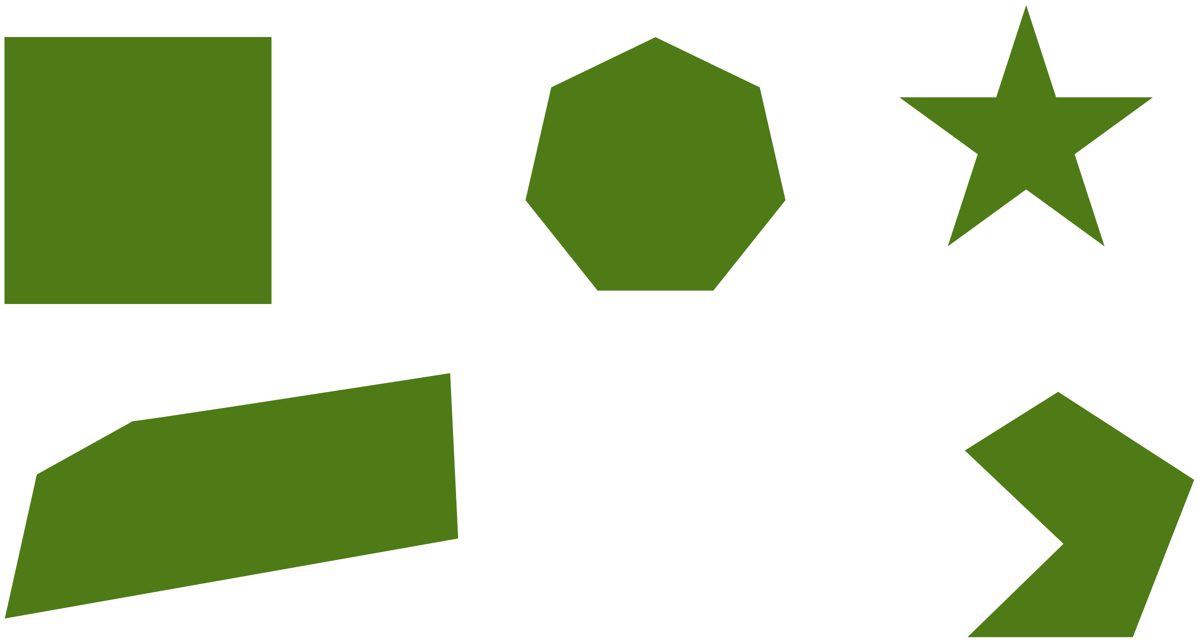
Convex Hulls

- What is a Convex Set?



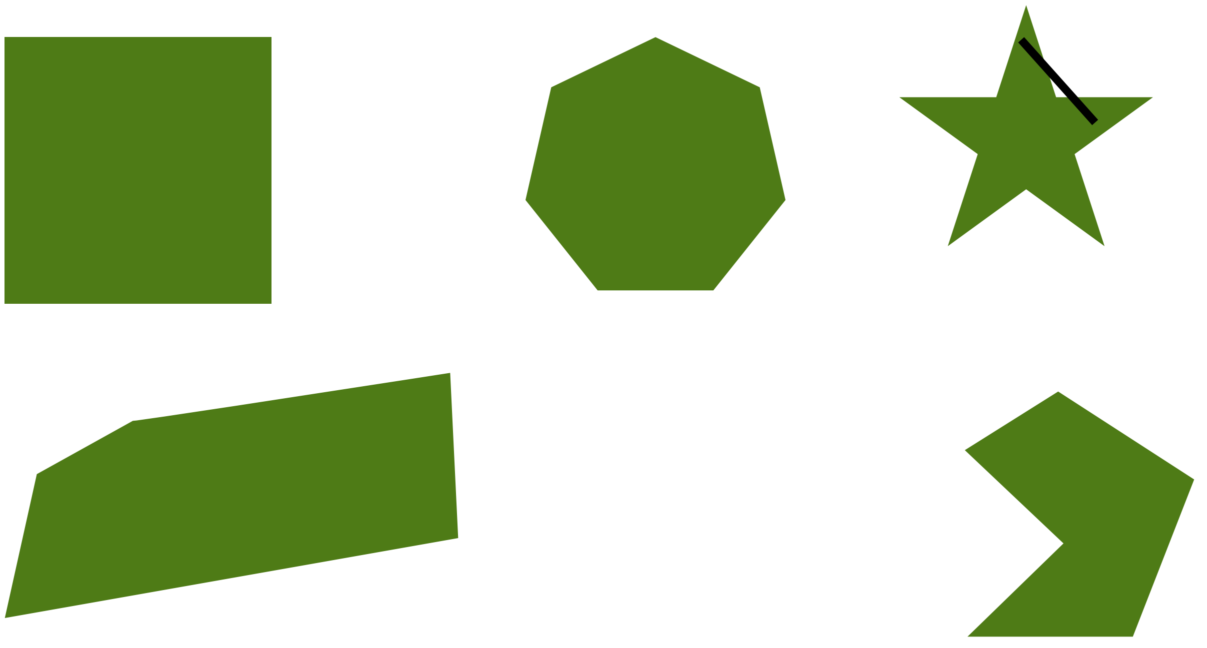
Convex Hulls

- What is a Convex Set?



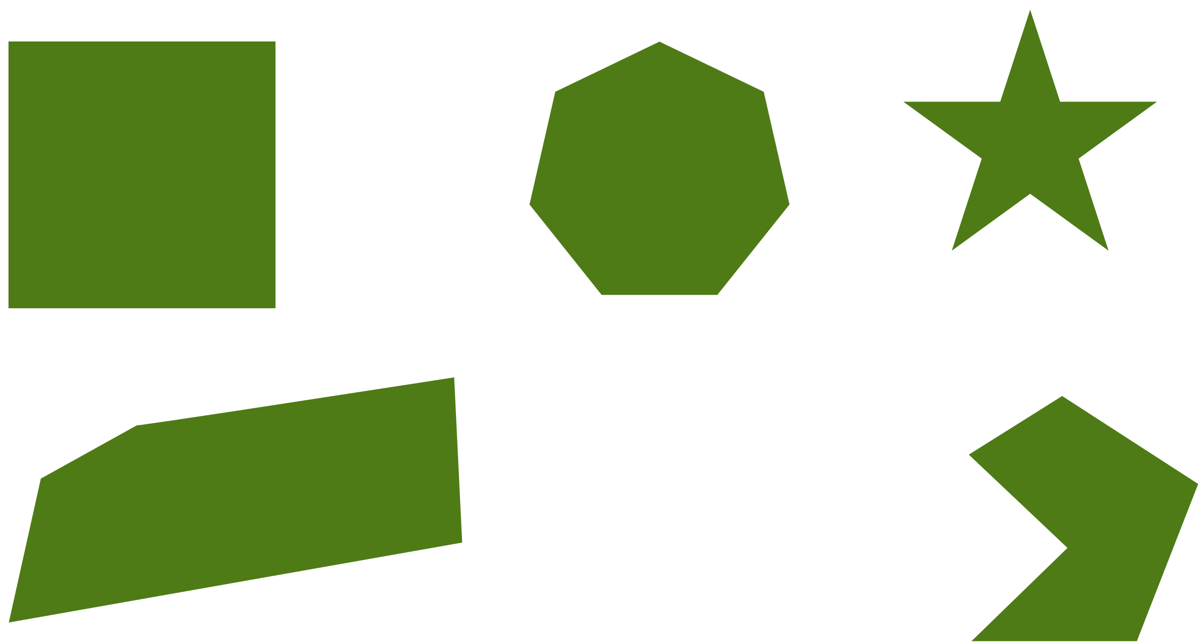
Convex Hulls

- What is a Convex Set?



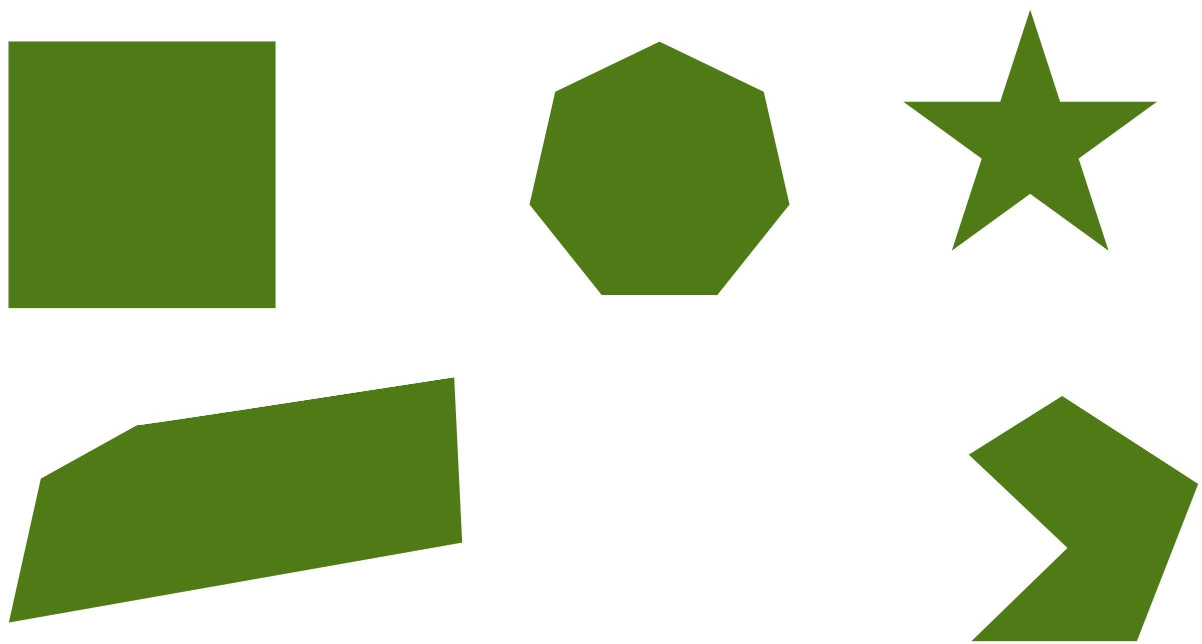
Convex Hulls

- What is a Convex Set?

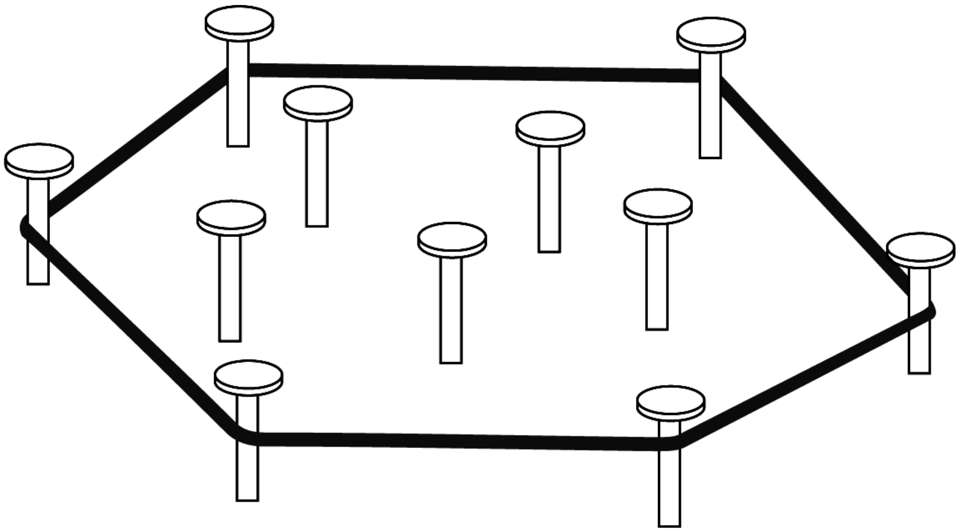


Convex Hulls

A set of points C is *convex* iff ∀ a, b ∈ C, all points on the line segment ab are entirely in C

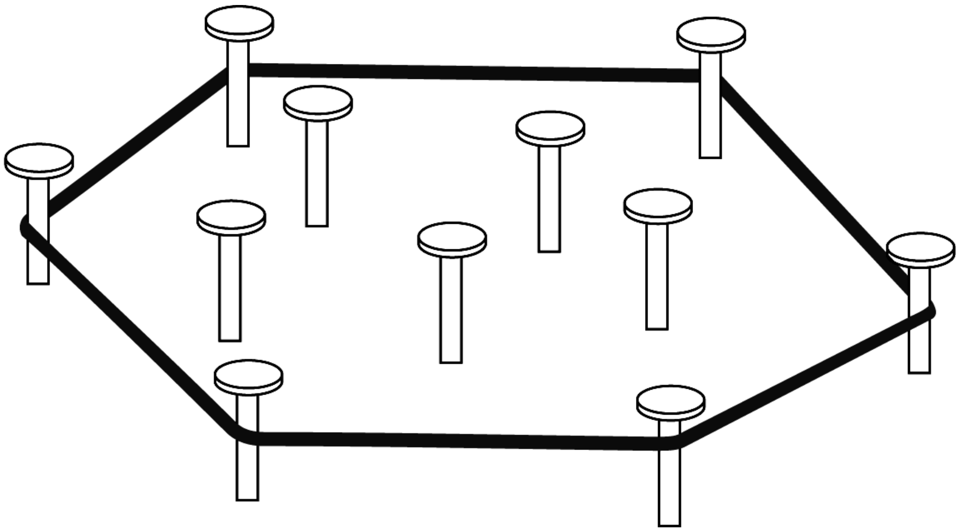


Convex Hulls



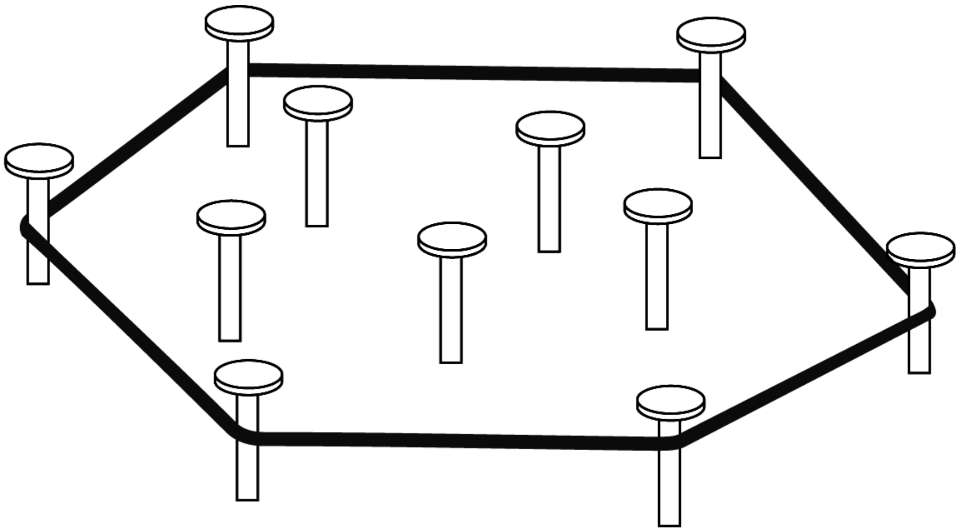
Convex Hulls

- Given an arbitrary set of points S, the convex hull of S is the smallest convex set that contain all the points in S.



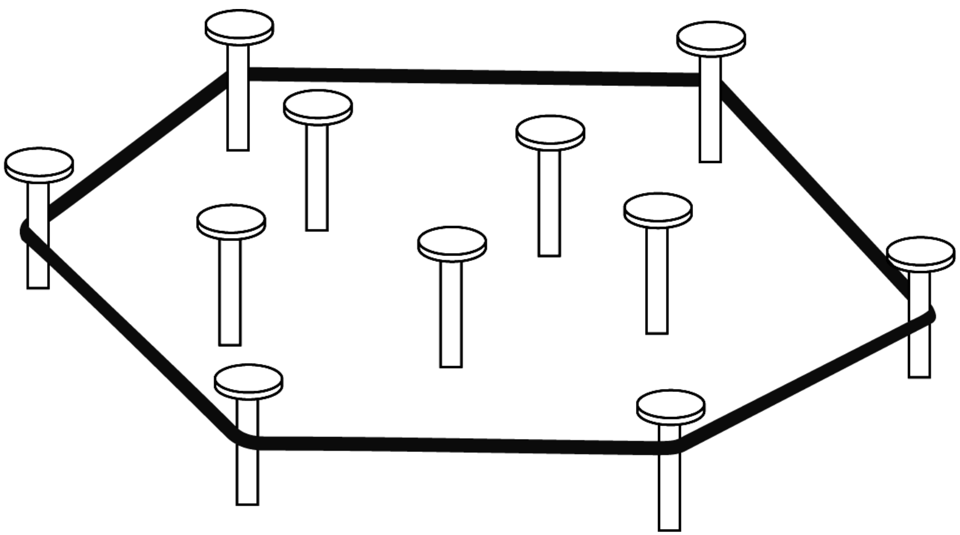
Convex Hulls

- Given an arbitrary set of points S, the convex hull of S is the smallest convex set that contain all the points in S.
  - Barricading sleeping tigers



Convex Hulls

- Given an arbitrary set of points S, the convex hull of S is the smallest convex set that contain all the points in S.
  - Barricading sleeping tigers
  - Rubber-band around nails

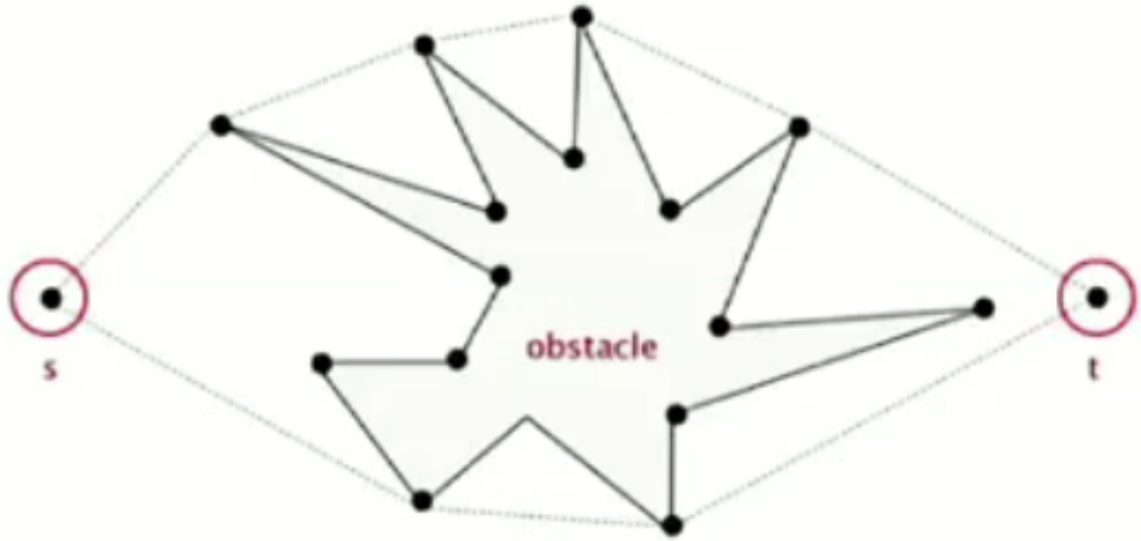


Applications of Convex Hull

- Collision-detection in video games

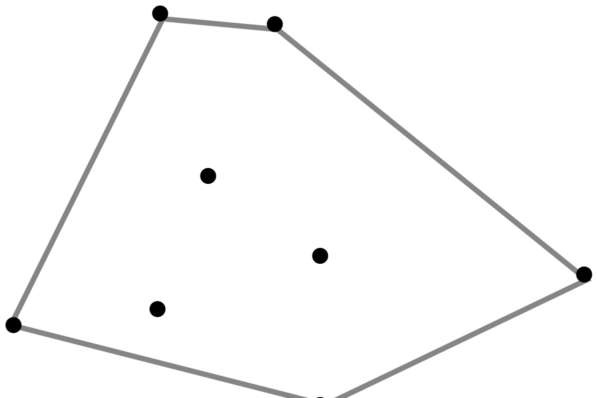
Applications of Convex Hull

- Collision-detection in video games
- Robot motion planning



Theorems about Convex Hulls

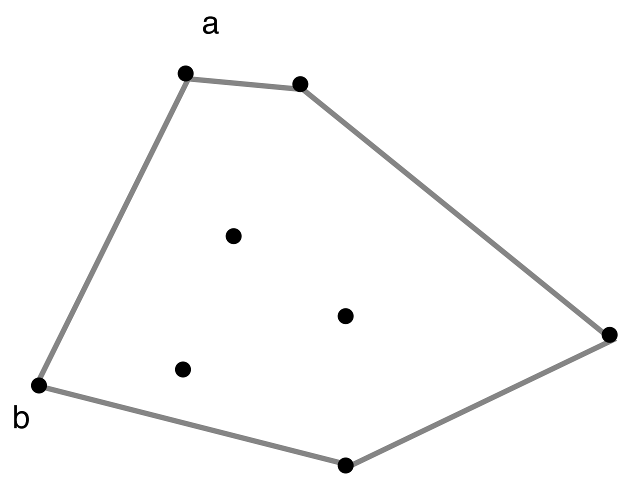
- The convex hull of a set S is a convex polygon all of whose vertices are at some of the points of S.
- A line segment ab is part of the boundary of the convex hull of S iff all the points of S lie *on the same side* of ab (or *on* ab)



Portland State UNIVERSITY

Theorems about Convex Hulls

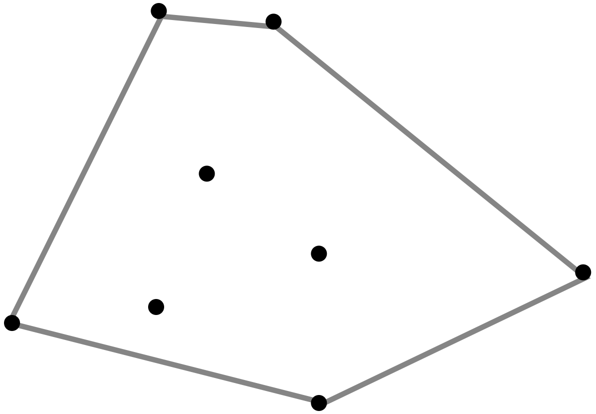
- The convex hull of a set S is a convex polygon all of whose vertices are at some of the points of S.
- A line segment ab is part of the boundary of the convex hull of S iff all the points of S lie *on the same side* of ab (or *on* ab)



Portland State University

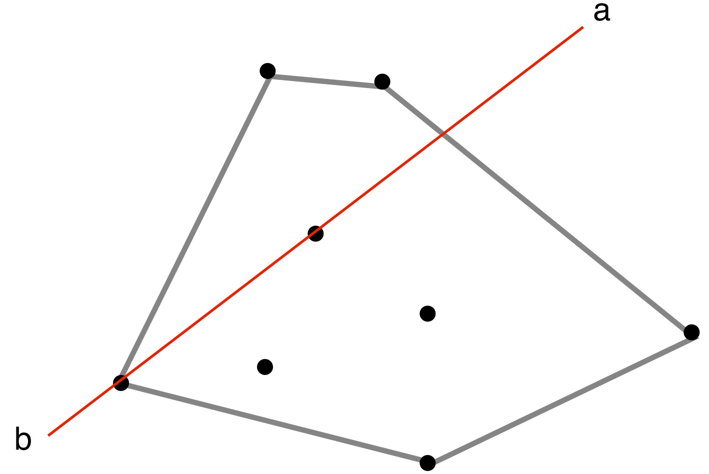
Theorems about Convex Hulls

- The convex hull of a set S is a convex polygon all of whose vertices are at some of the points of S.
- A line segment ab is part of the boundary of the convex hull of S iff all the points of S lie *on the same side* of ab (or *on* ab)



Theorems about Convex Hulls

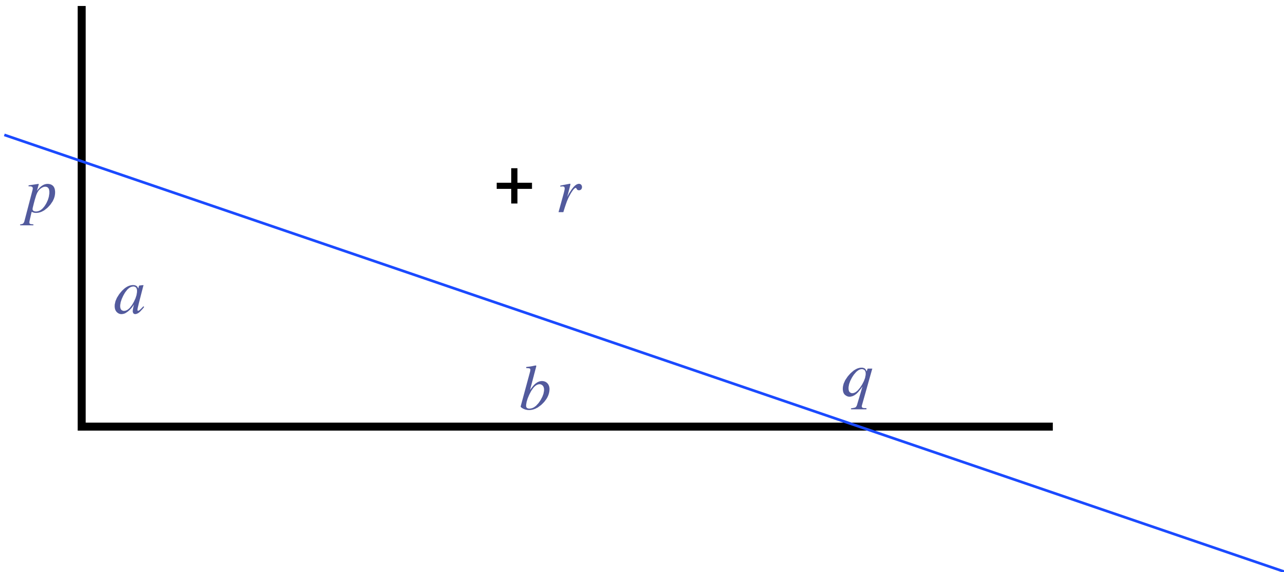
- The convex hull of a set S is a convex polygon all of whose vertices are at some of the points of S.
- A line segment ab is part of the boundary of the convex hull of S iff all the points of S lie *on the same side* of ab (or *on* ab)



Portland State University

Brute-Force Algorithm for Convex Hull

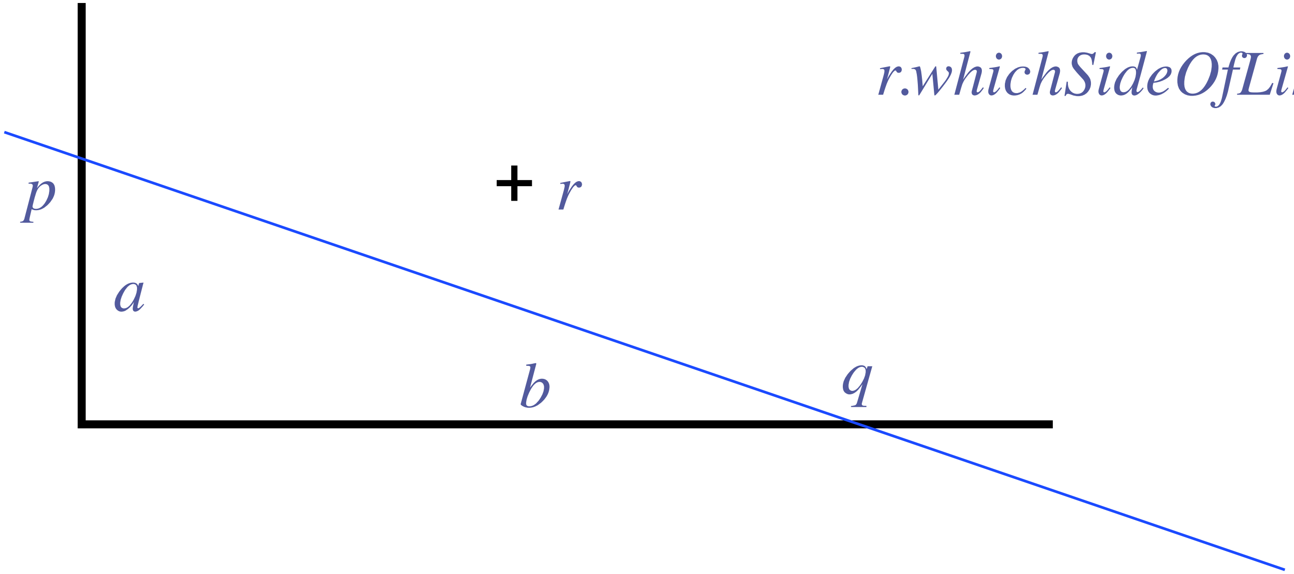
- write it down!
  - Assume that you have a method for ascertaining if a point r is on a line pq, on the –ve side of line pq, or on the +ve side of pq



Brute-Force Algorithm for Convex Hull

- write it down!
  - Assume that you have a method for ascertaining if a point r is on a line pq, on the –ve side of line pq, or on the +ve side of pq

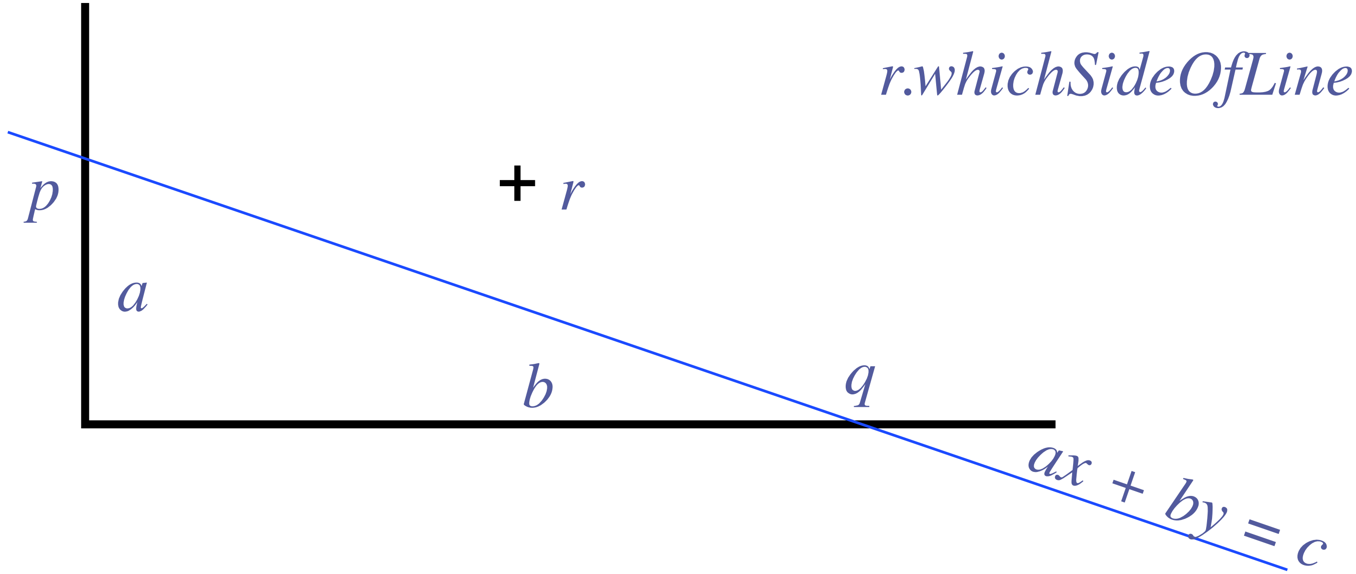
$$r.whichSideOfLine(pq)$$



Brute-Force Algorithm for Convex Hull

- write it down!
  - Assume that you have a method for ascertaining if a point r is on a line pq, on the –ve side of line pq, or on the +ve side of pq

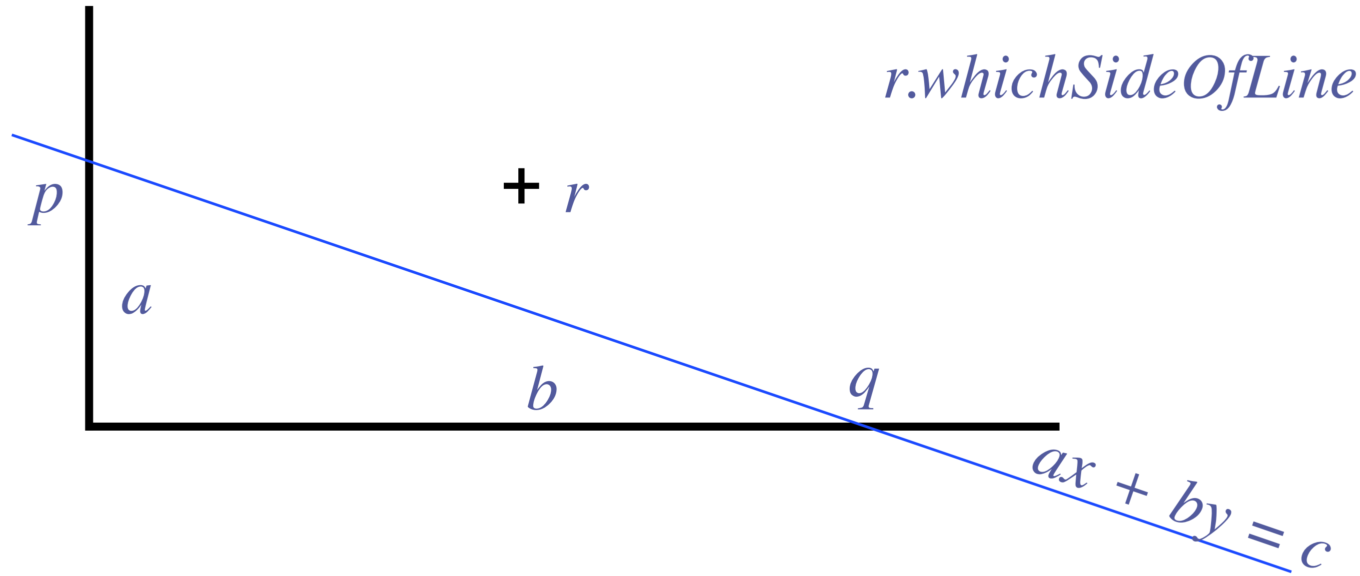
$$r.whichSideOfLine(pq)$$



Brute-Force Algorithm for Convex Hull

- write it down!
  - Assume that you have a method for ascertaining if a point r is on a line pq, on the –ve side of line pq, or on the +ve side of pq

$r.whichSideOfLine(pq)$



$c = p_x q_y - q_x p_y$

Brute-Force Algorithm for Convex Hull

```
edgeSet ← {}
P: for p in S do:
  Q: for q in S, q ≠ p do:
      goodSide ← 0
      R: for r in S, r≠p ∧ r≠q do:
          side ← r.whichSideOfLine(pq)
          if  side ≠ 0 then
              if  goodSide = 0 then goodSide ← side
              if  goodSide ≠ side then exit Q.
      edgeSet ← edgeSet ∪ {pq}
```

Final Comments on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time *only* on *very small* instances
- In some cases, there are *much* better alternatives!
  - Euler circuits
  - shortest paths
  - minimum spanning tree
  - assignment problem
- However, in many cases, exhaustive search (or a variation) is the only known way to find an exact solution

Searching in Graphs

Exhaustively search a graph, by traversing the edges, visiting every node once

Two approaches:

- Depth-first search and
- Breadth-first search

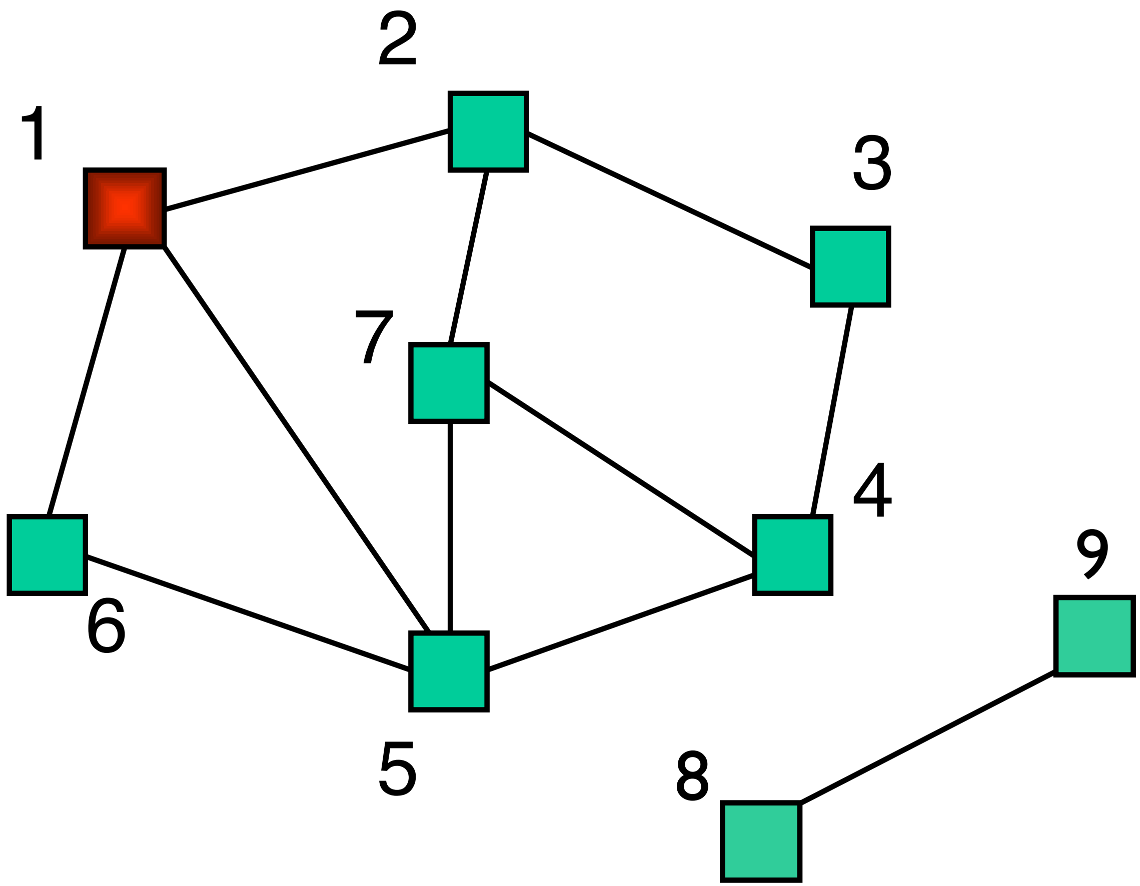
ALGORITHM *DFS*(G)

//Implements a depth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
// in the order they are first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
$count \leftarrow 0$
for each vertex v in V **do**
    if v is marked with 0
        $dfs(v)$

$dfs(v)$
//visits recursively all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are encountered
//via global variable *count*
$count \leftarrow count + 1$; mark v with *count*
for each vertex w in V adjacent to v **do**
    if w is marked with 0
        $dfs(w)$

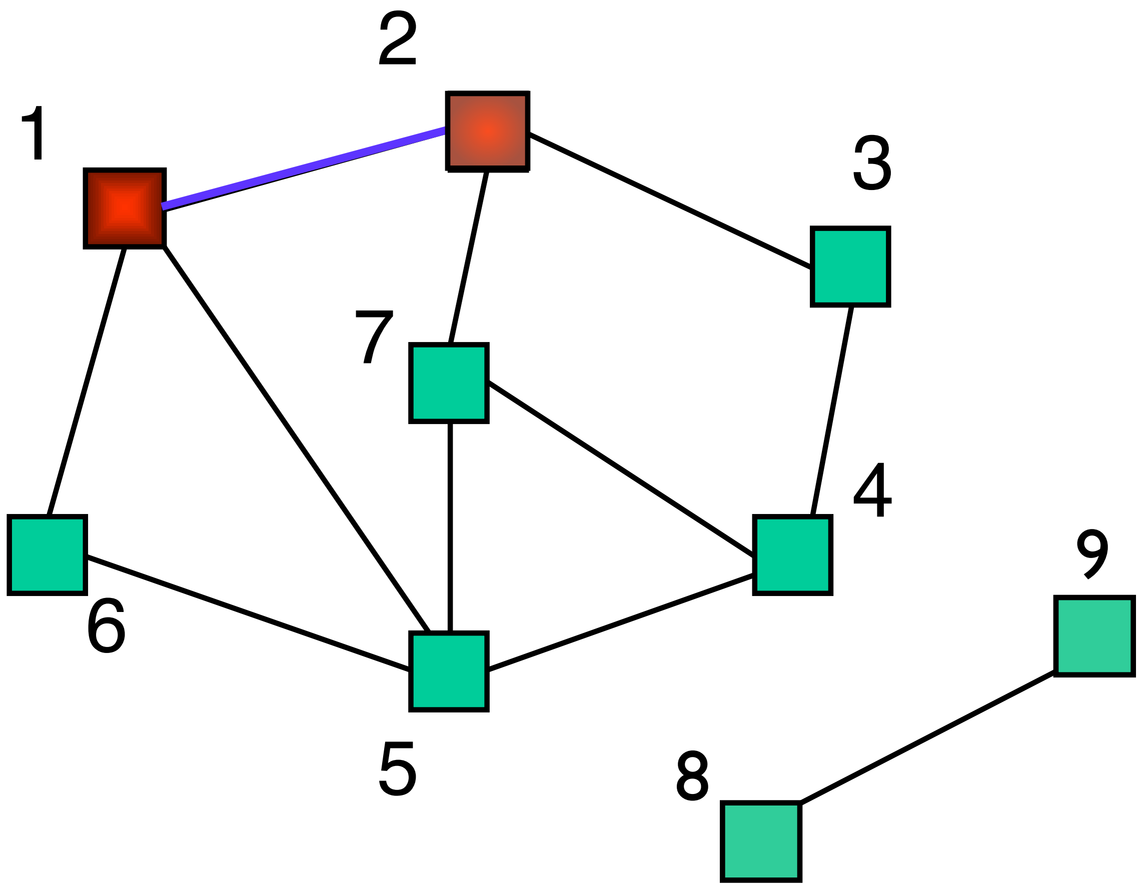
Example

dfs(1)



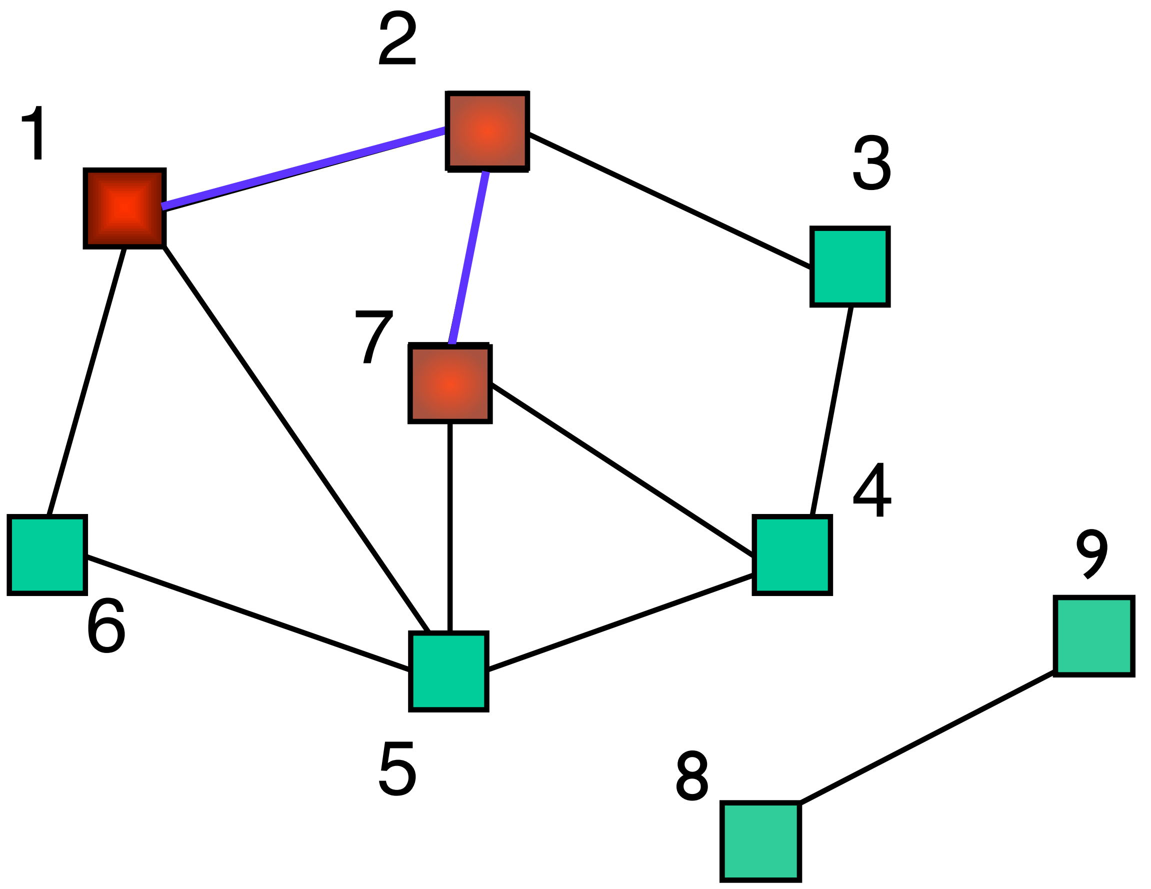
Example

dfs(1)
dfs(2)



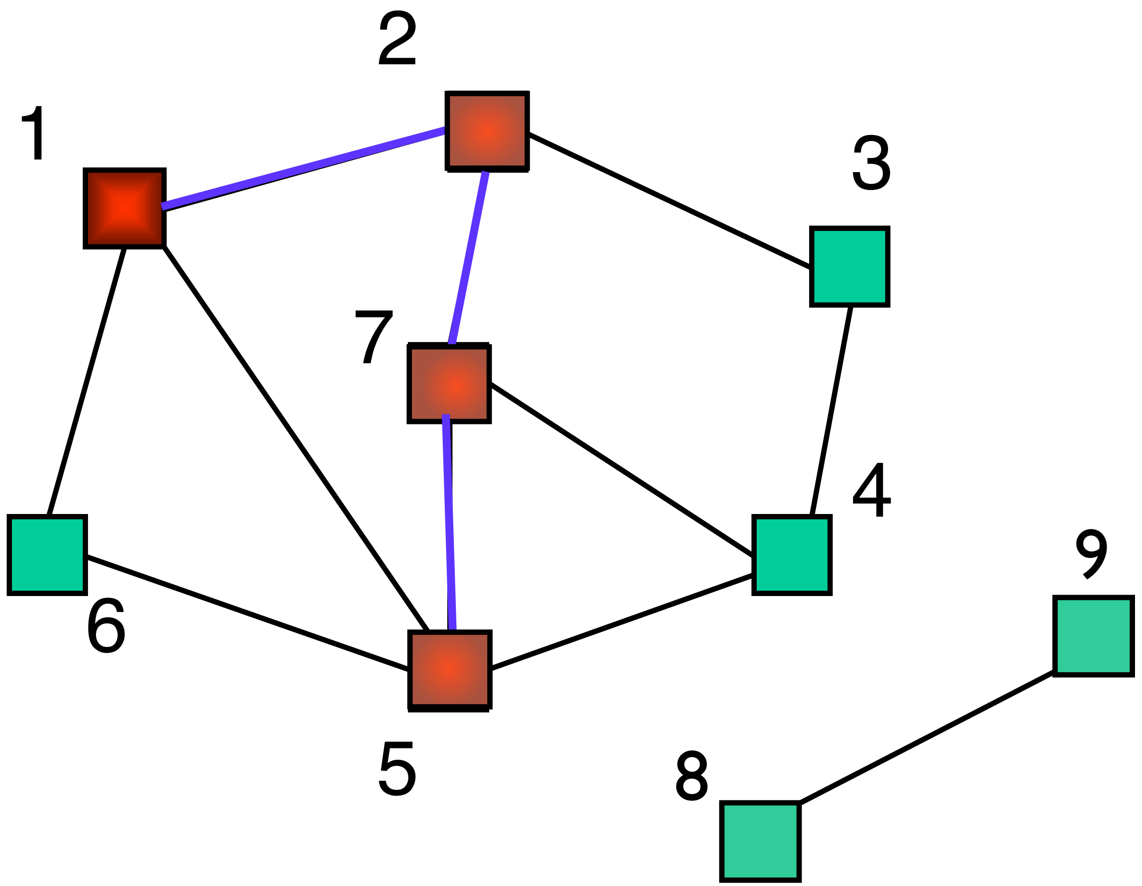
Example

dfs(1)
dfs(2)
dfs(7)



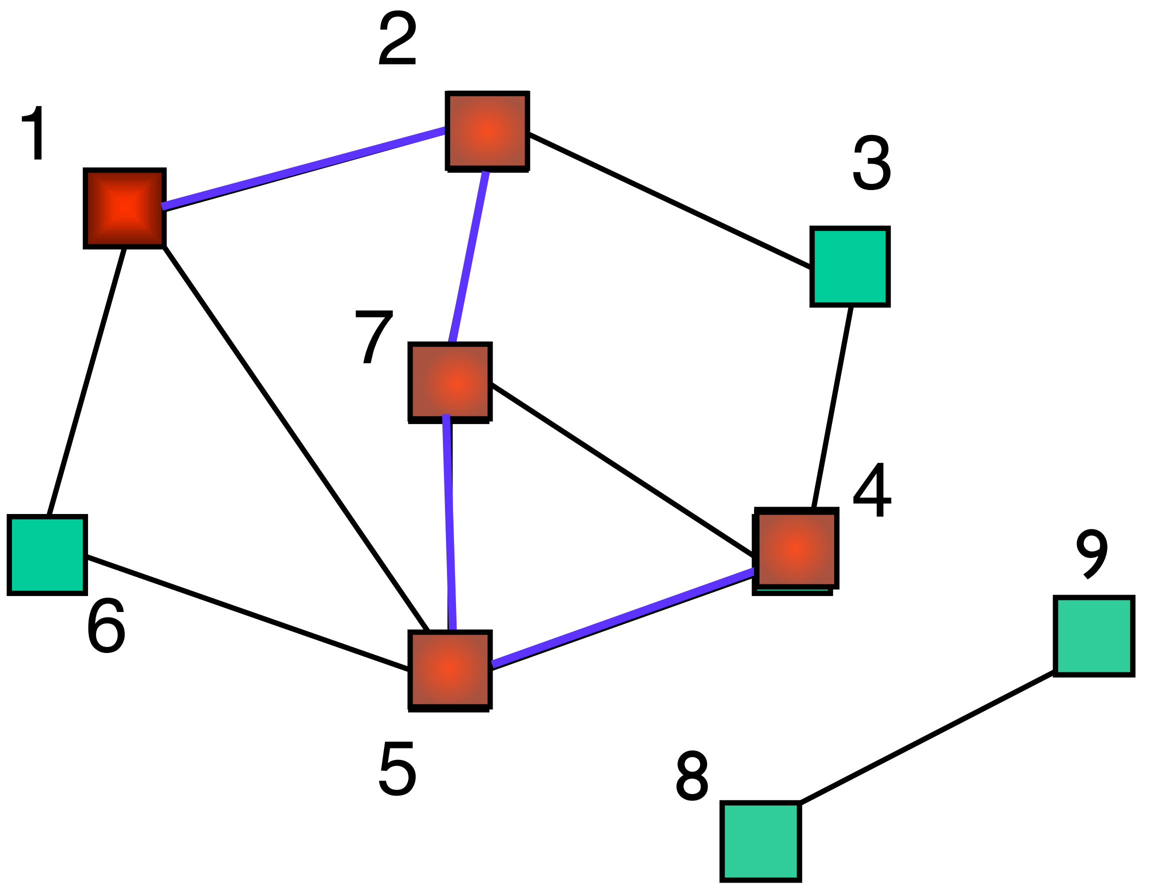
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)



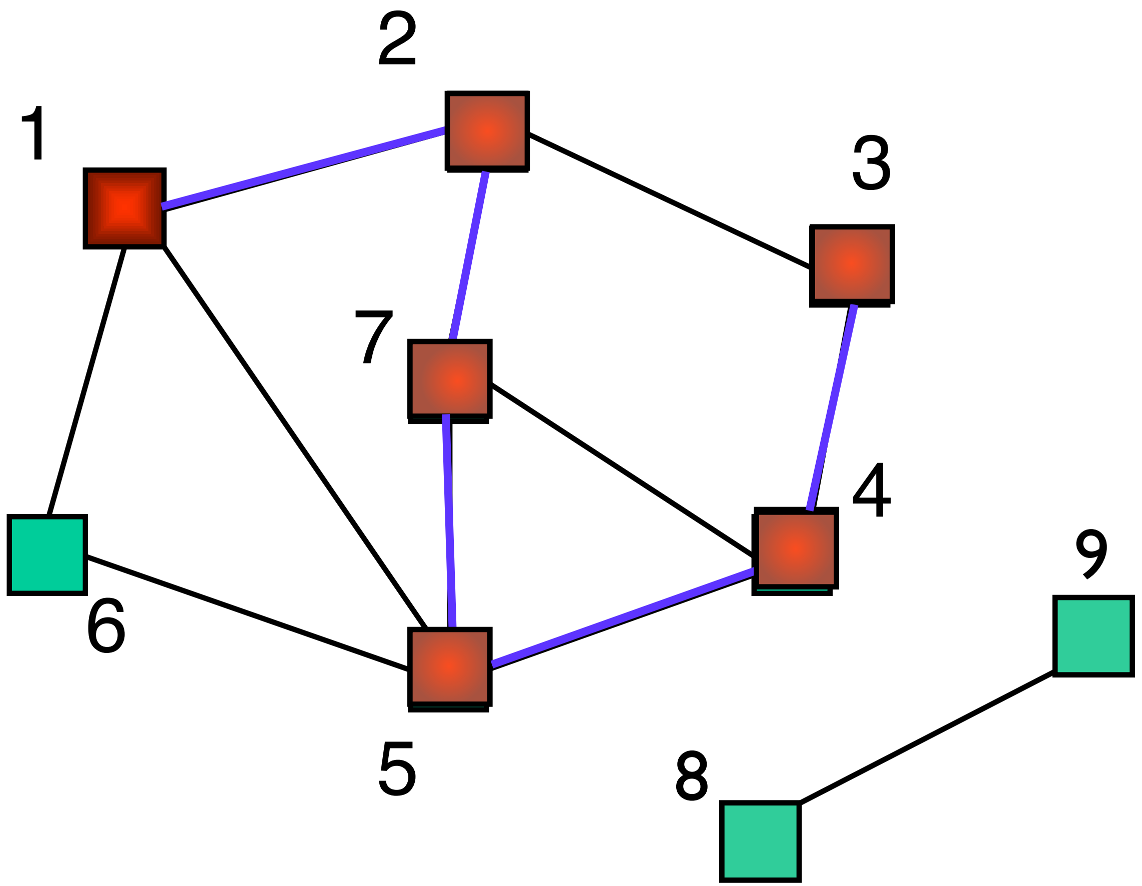
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
dfs(4)



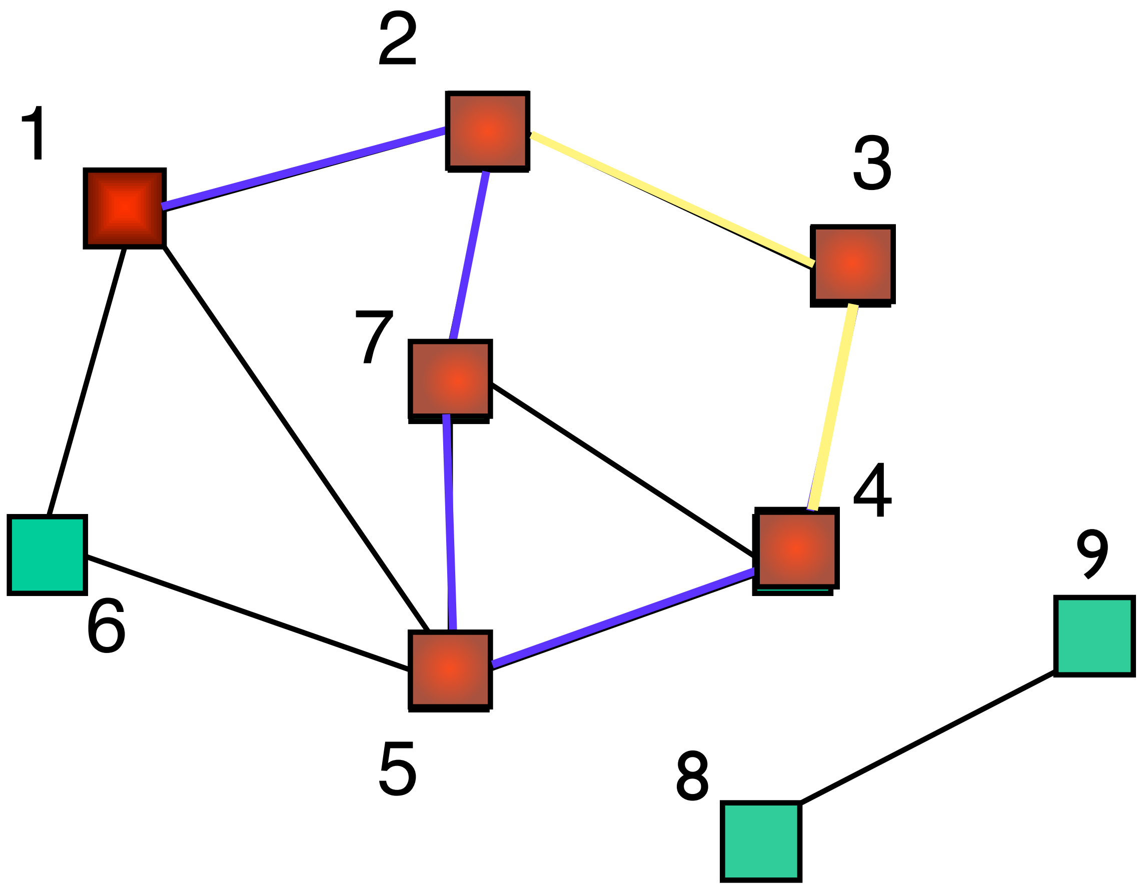
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
dfs(4)
dfs(3)



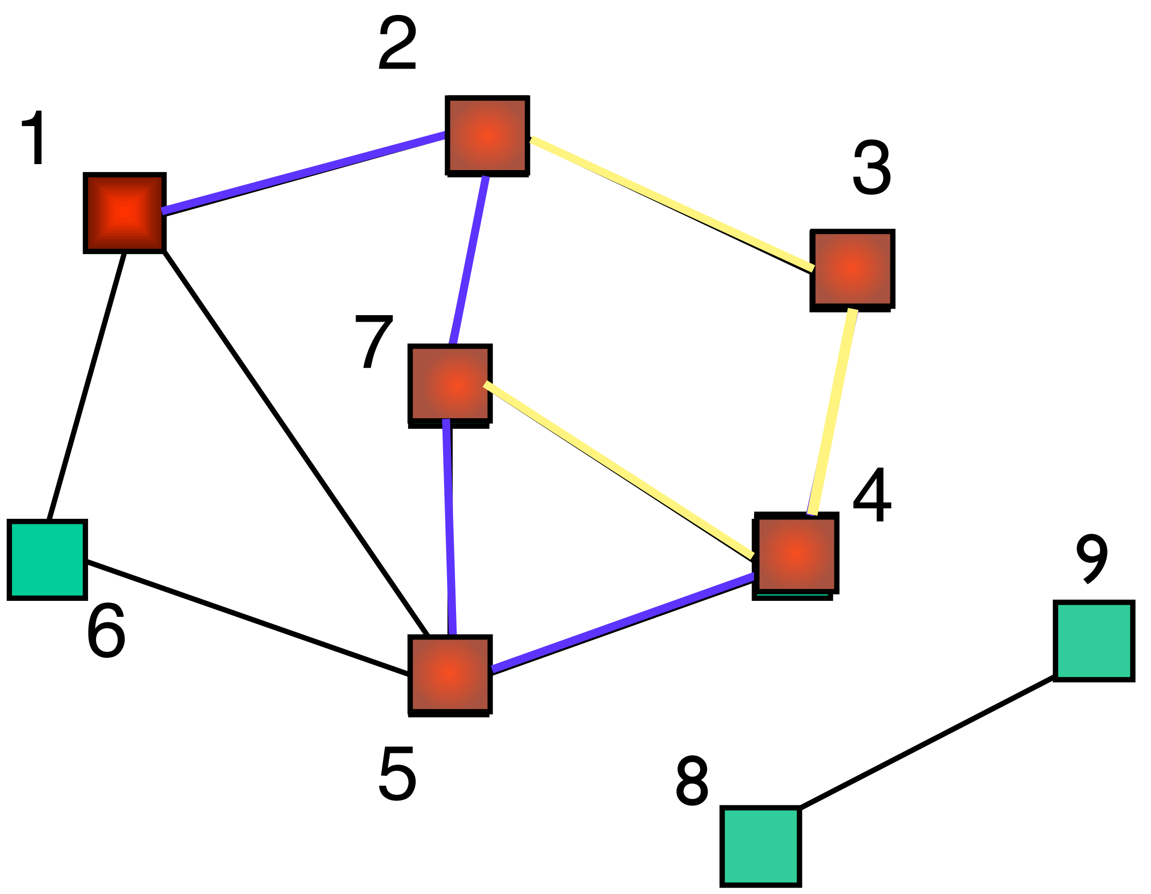
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
dfs(4)



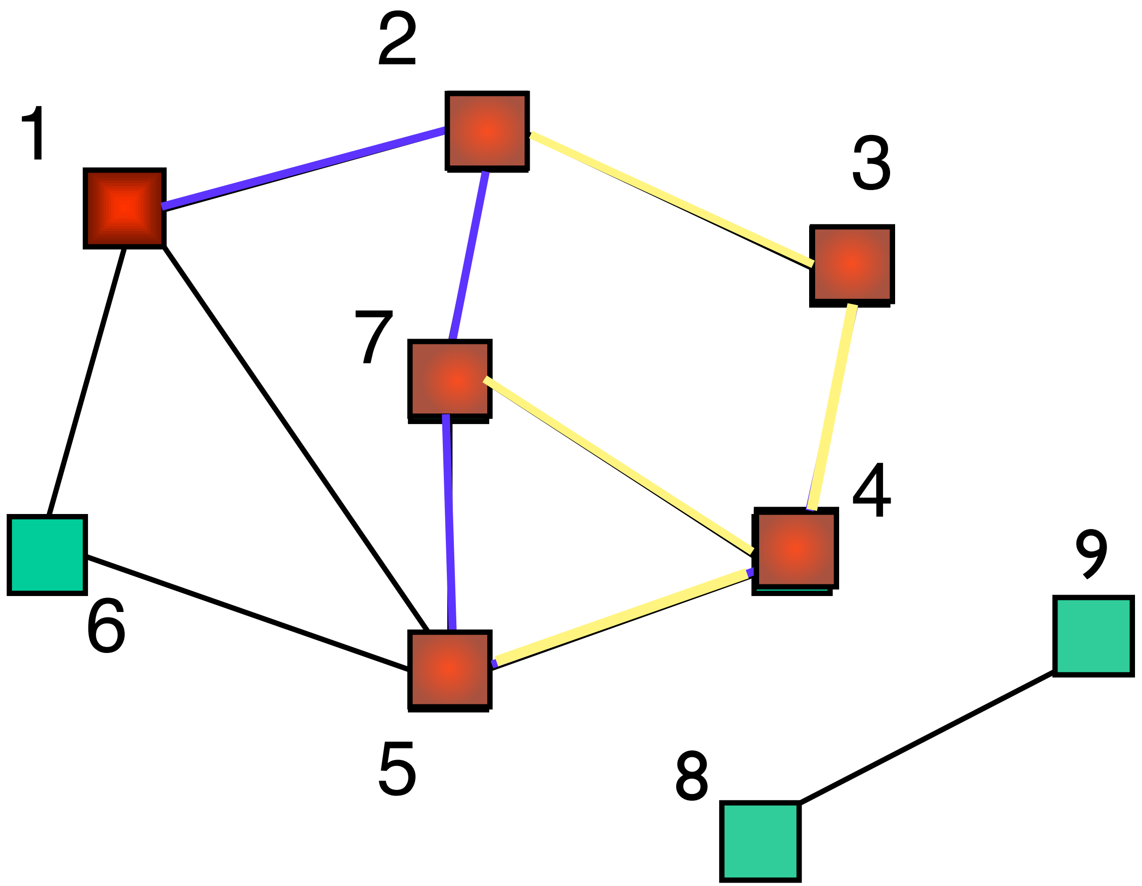
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
dfs(4)



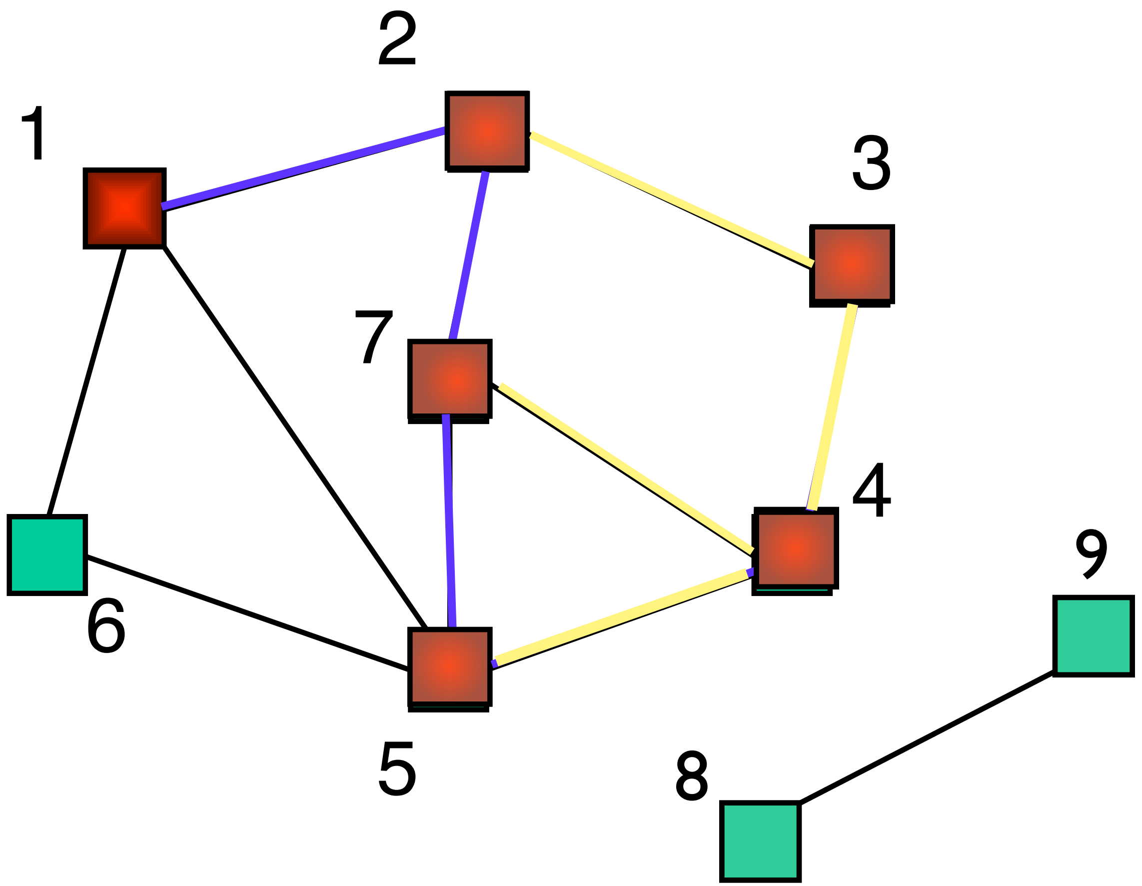
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
dfs(4)



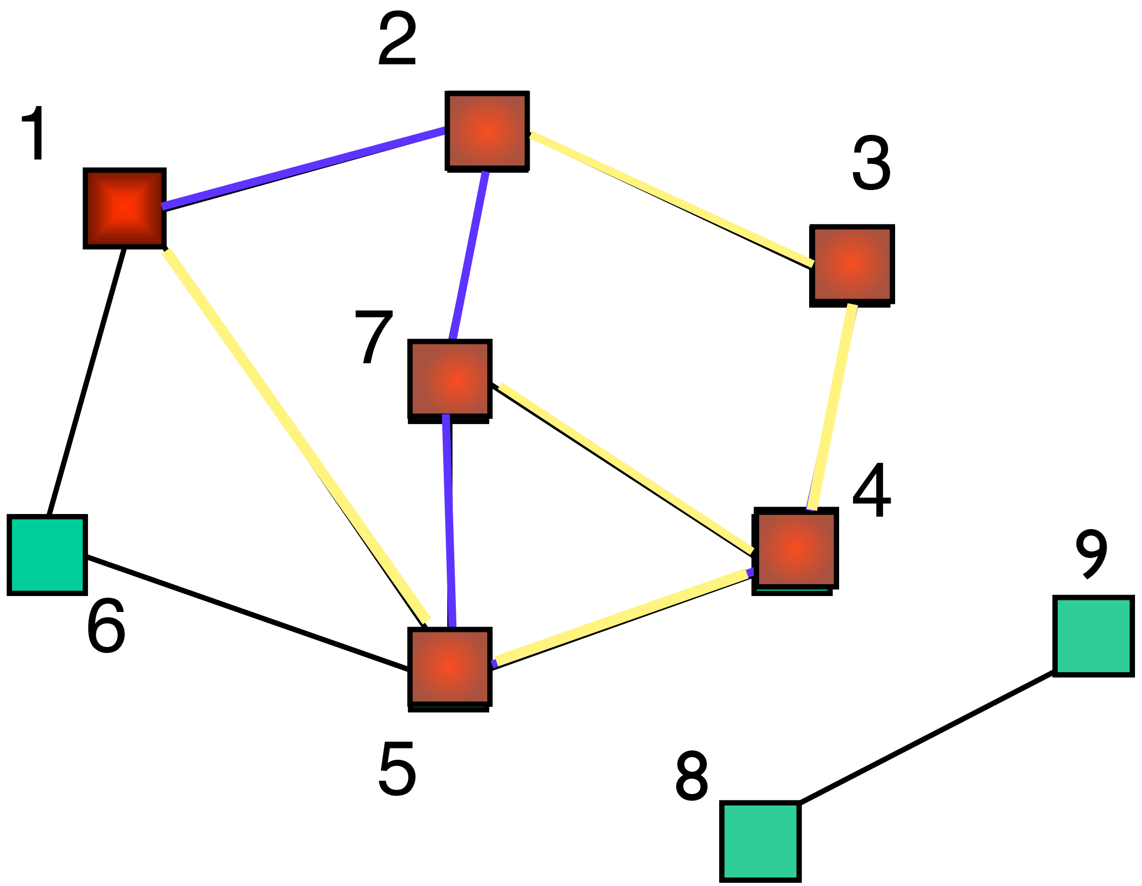
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)



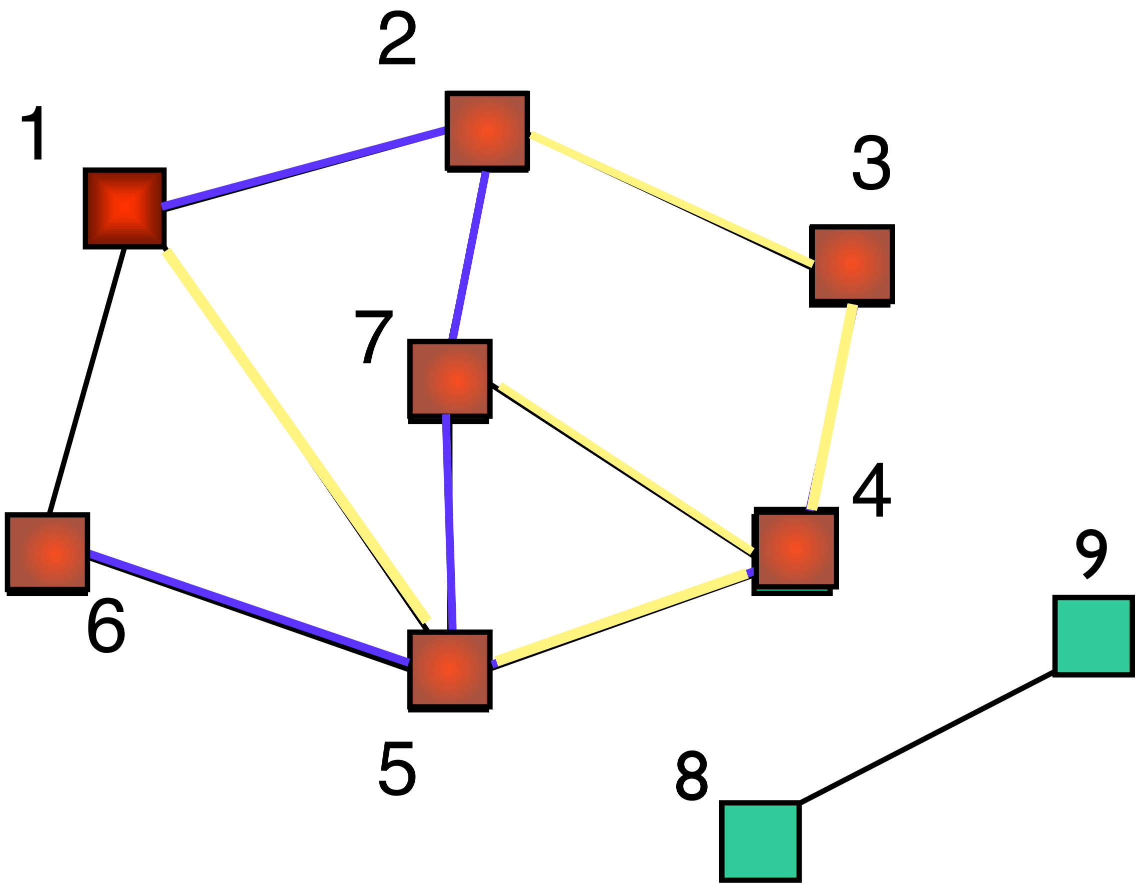
Example

$dfs(1)$
$dfs(2)$
$dfs(7)$
$dfs(5)$



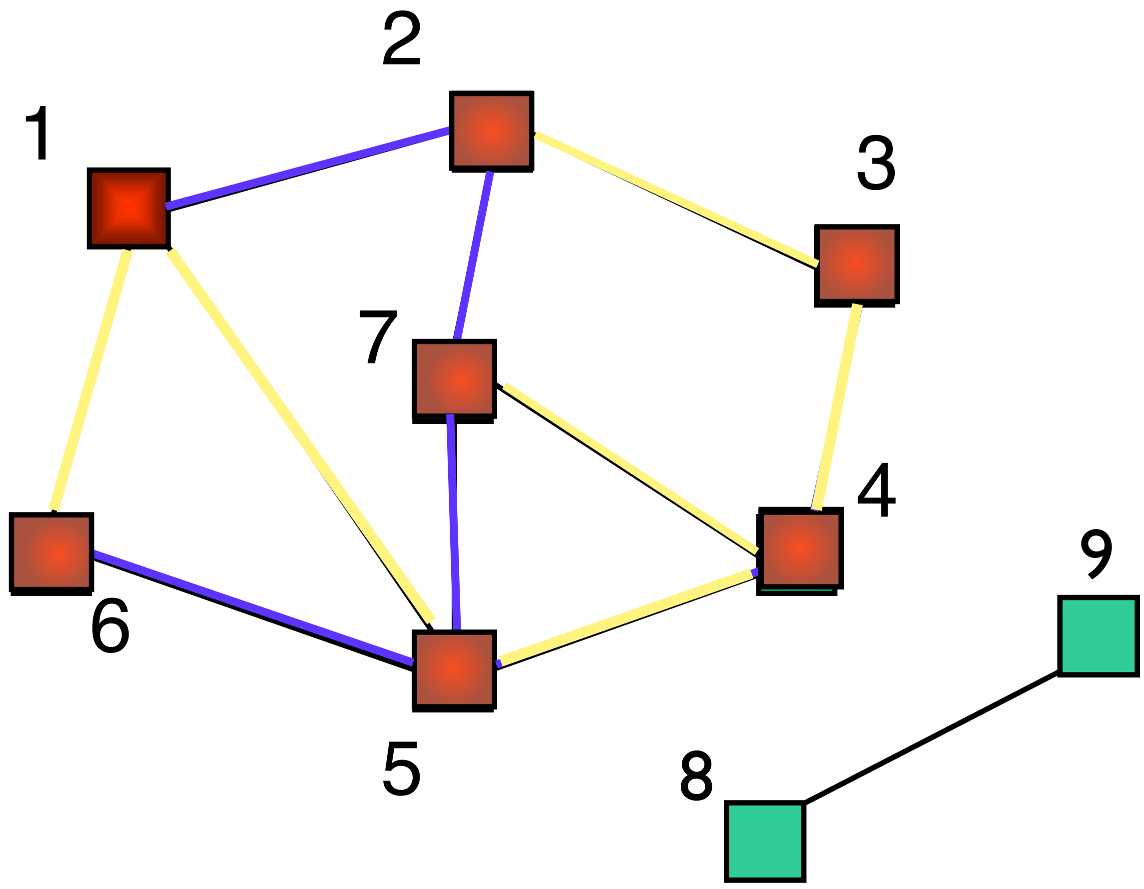
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
dfs(6)



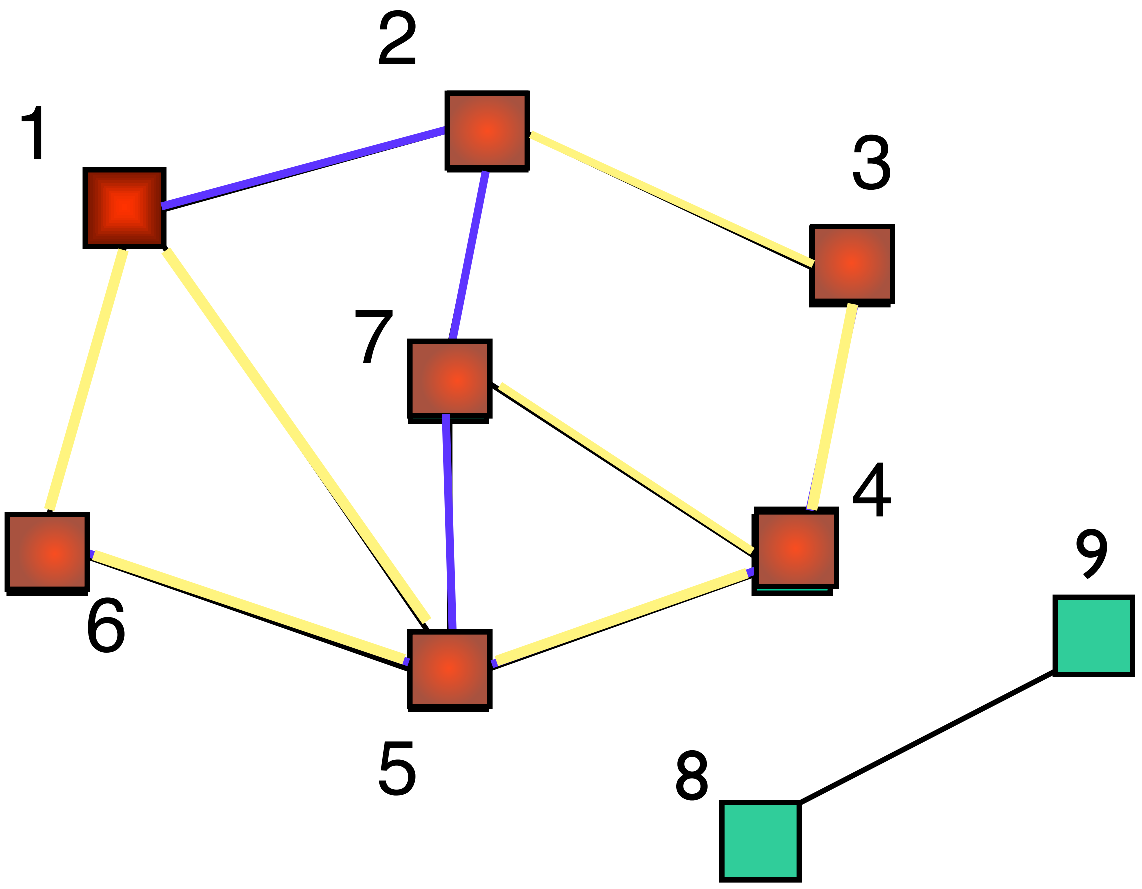
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
dfs(6)



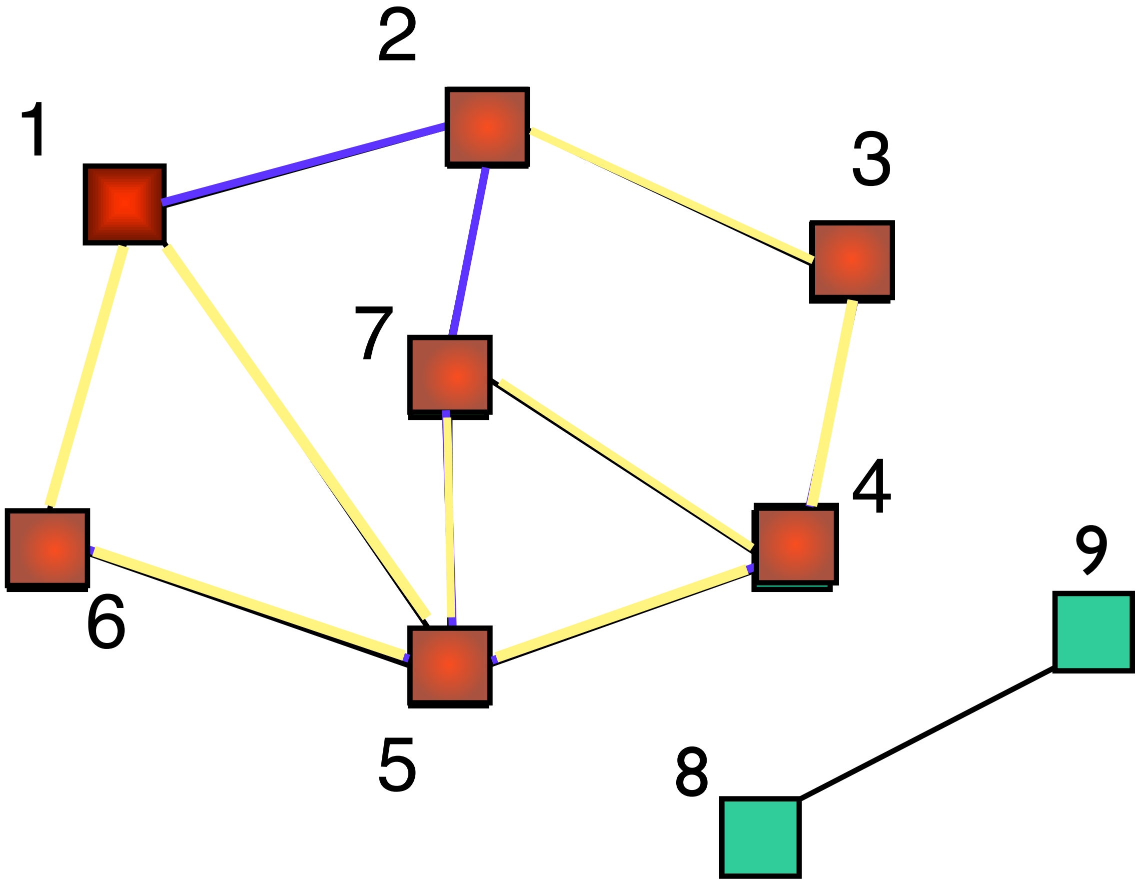
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)



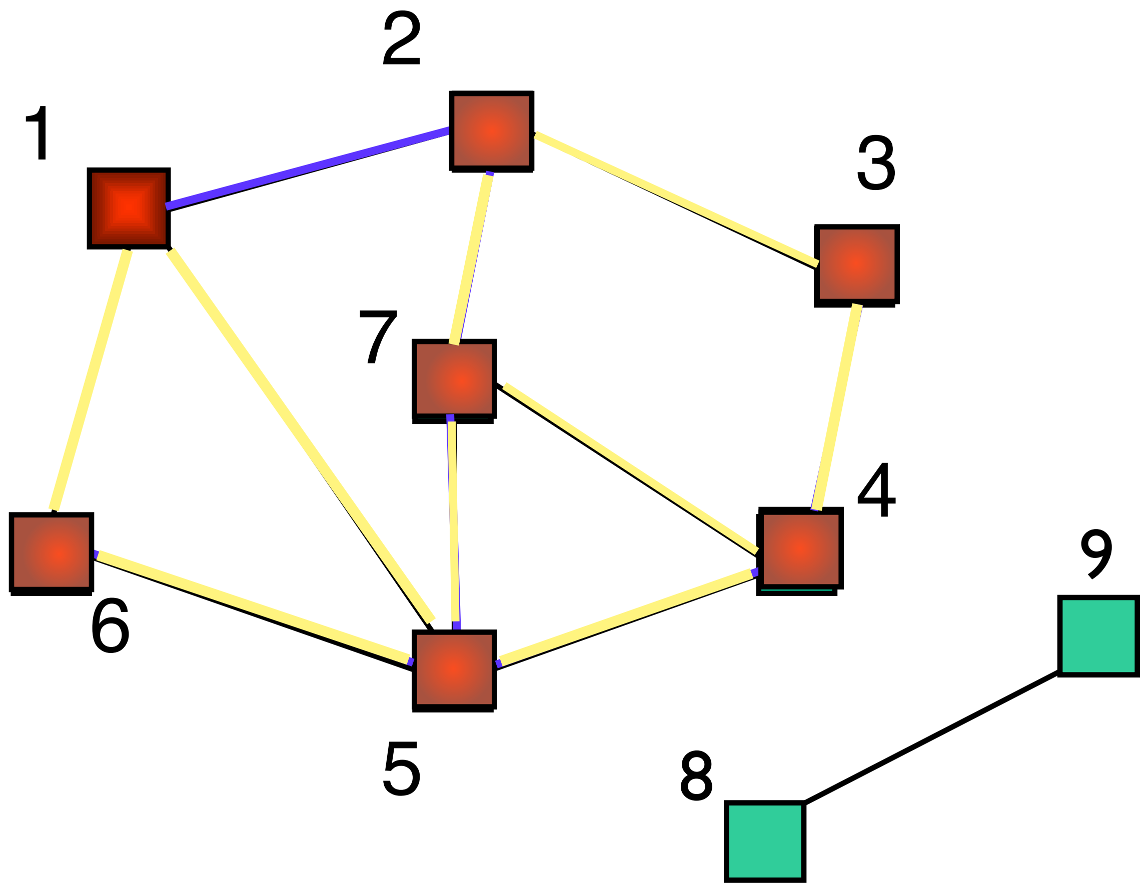
Example

dfs(1)
dfs(2)
dfs(7)



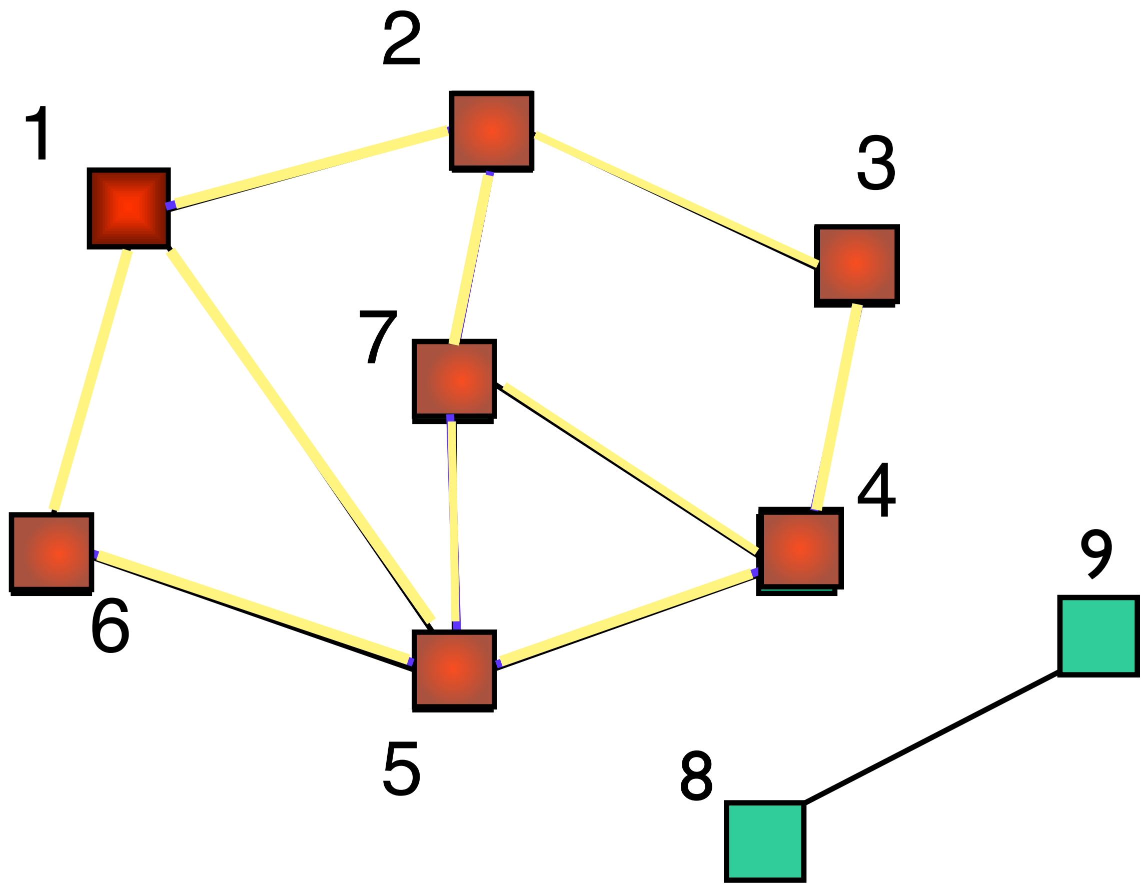
Example

dfs(1)
dfs(2)



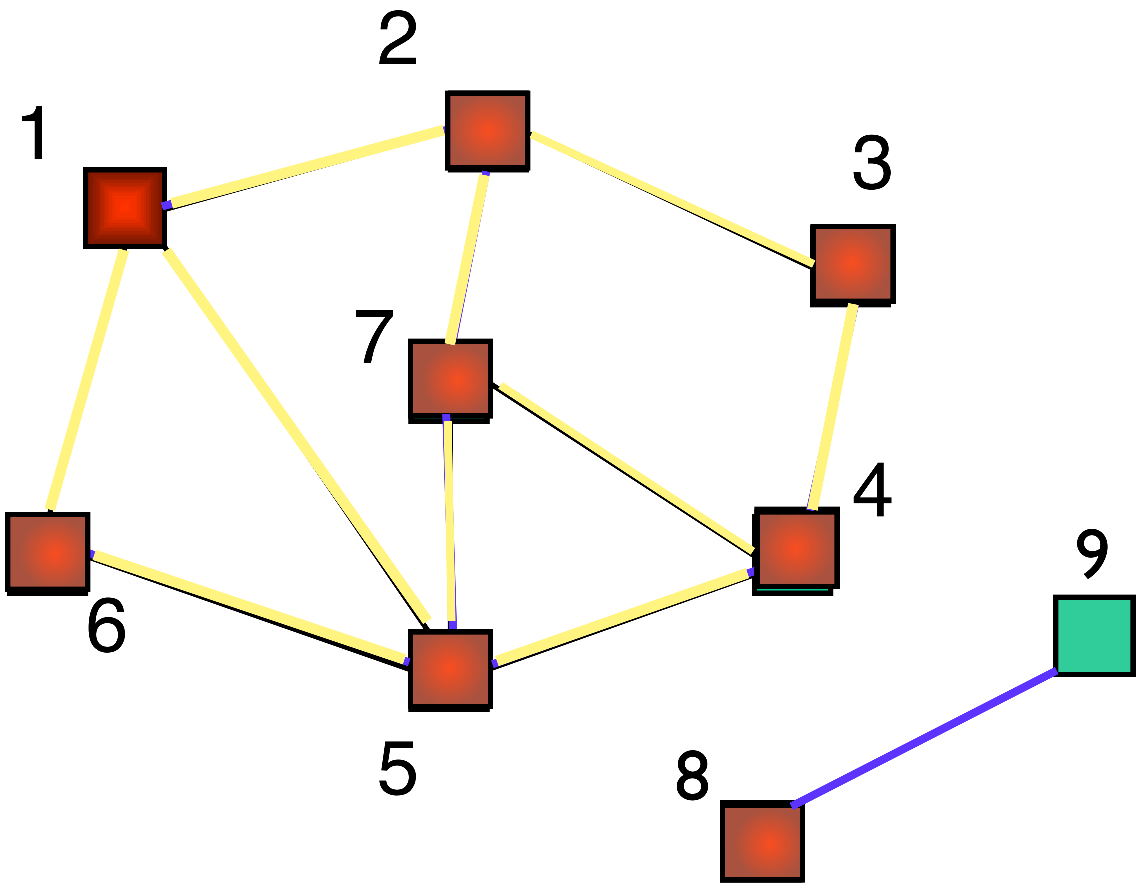
Example

$dfs(1)$



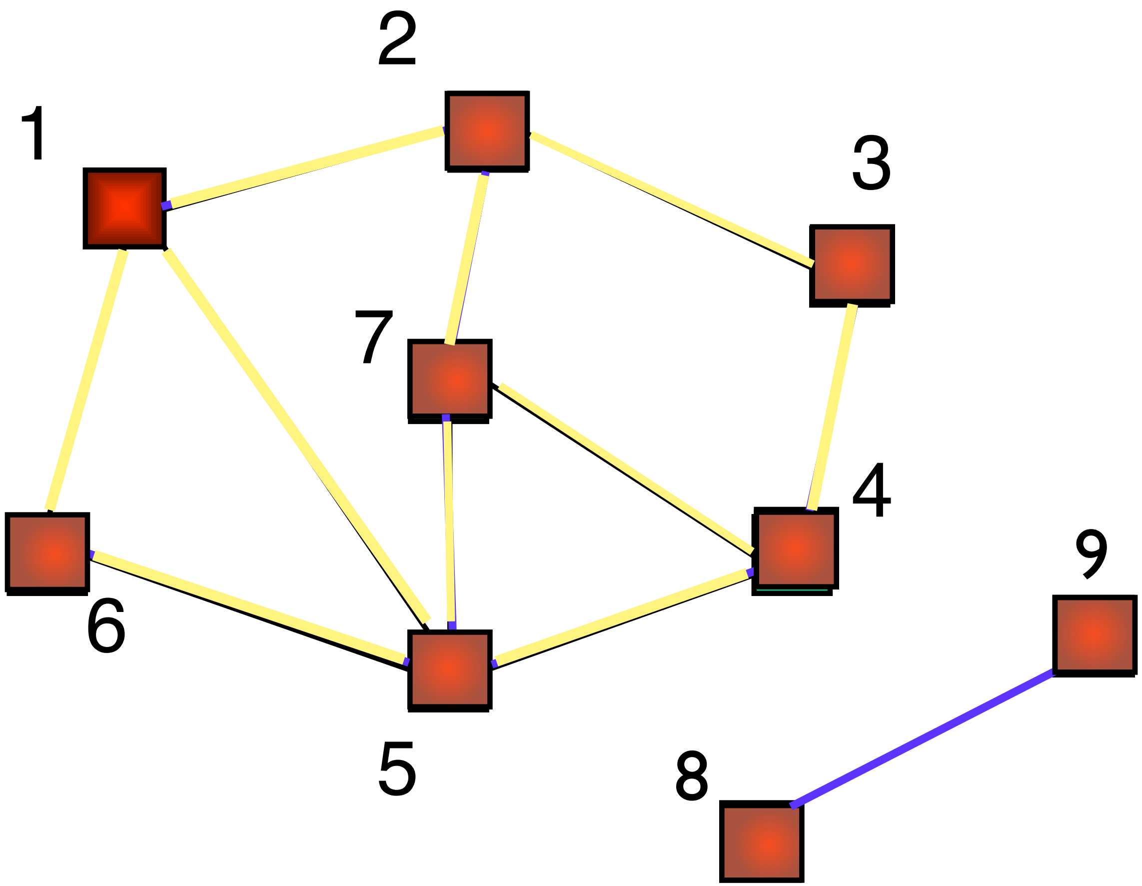
Example

dfs(8)



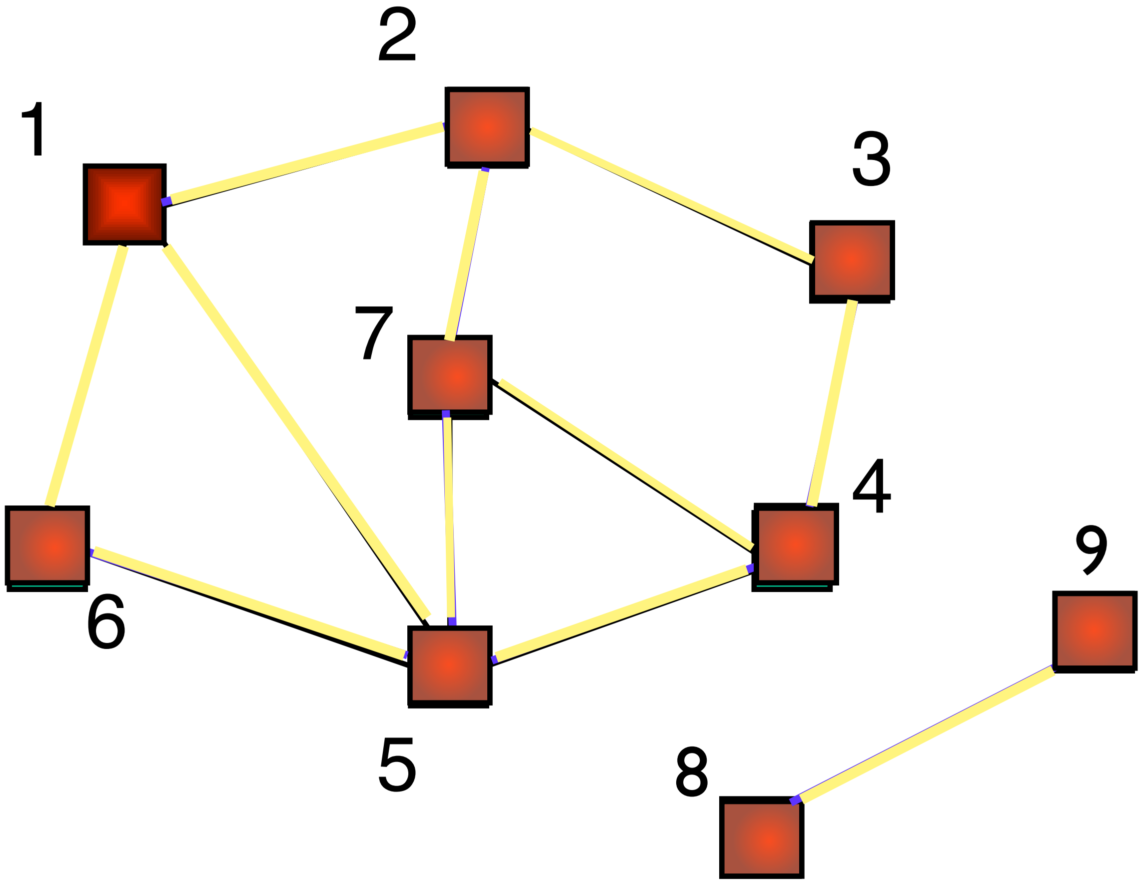
Example

dfs(8)
dfs(9)

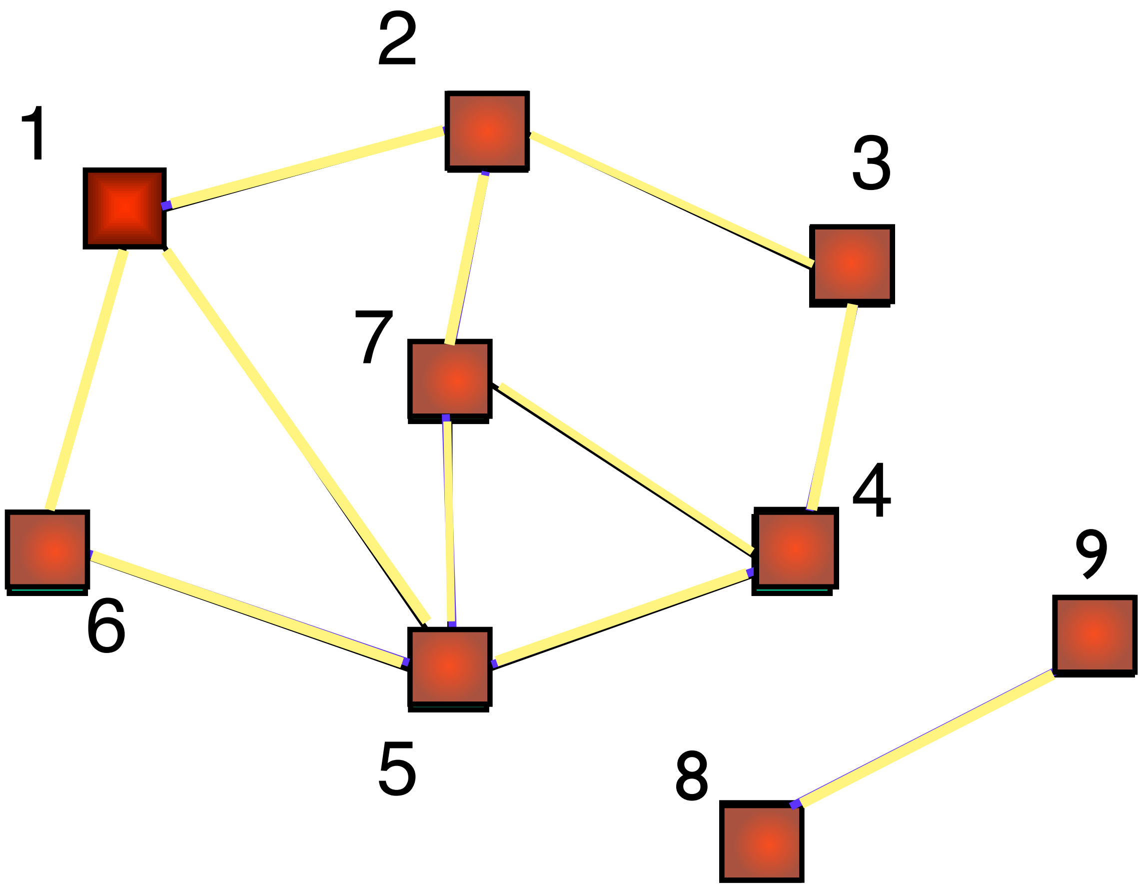


Example

dfs(8)



Example



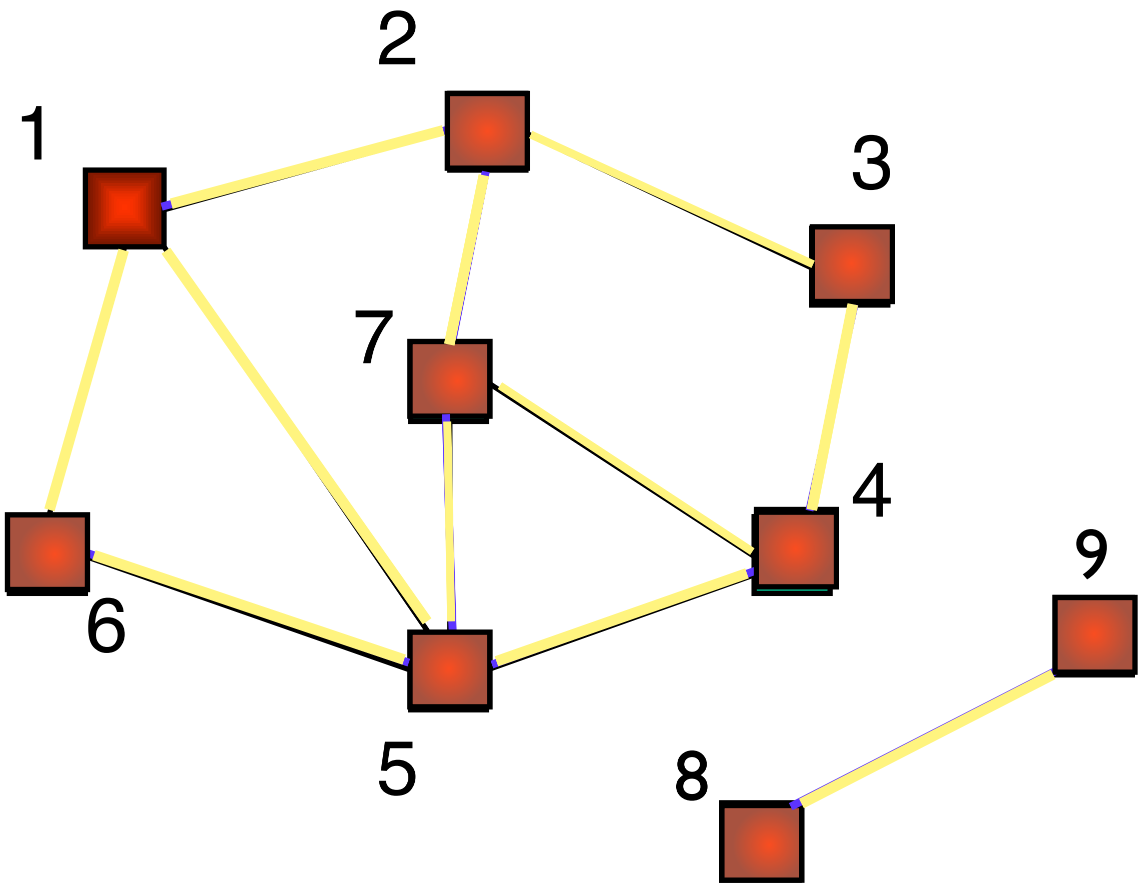
Complexity?

- What's the basic operation?
  - finding all the Vertices in the graph?
  - making a mark?
  - checking a mark?
  - finding all the neighbors of a node?
- Cost depends on the data structure used to represent the graph

Two choices of data structure:

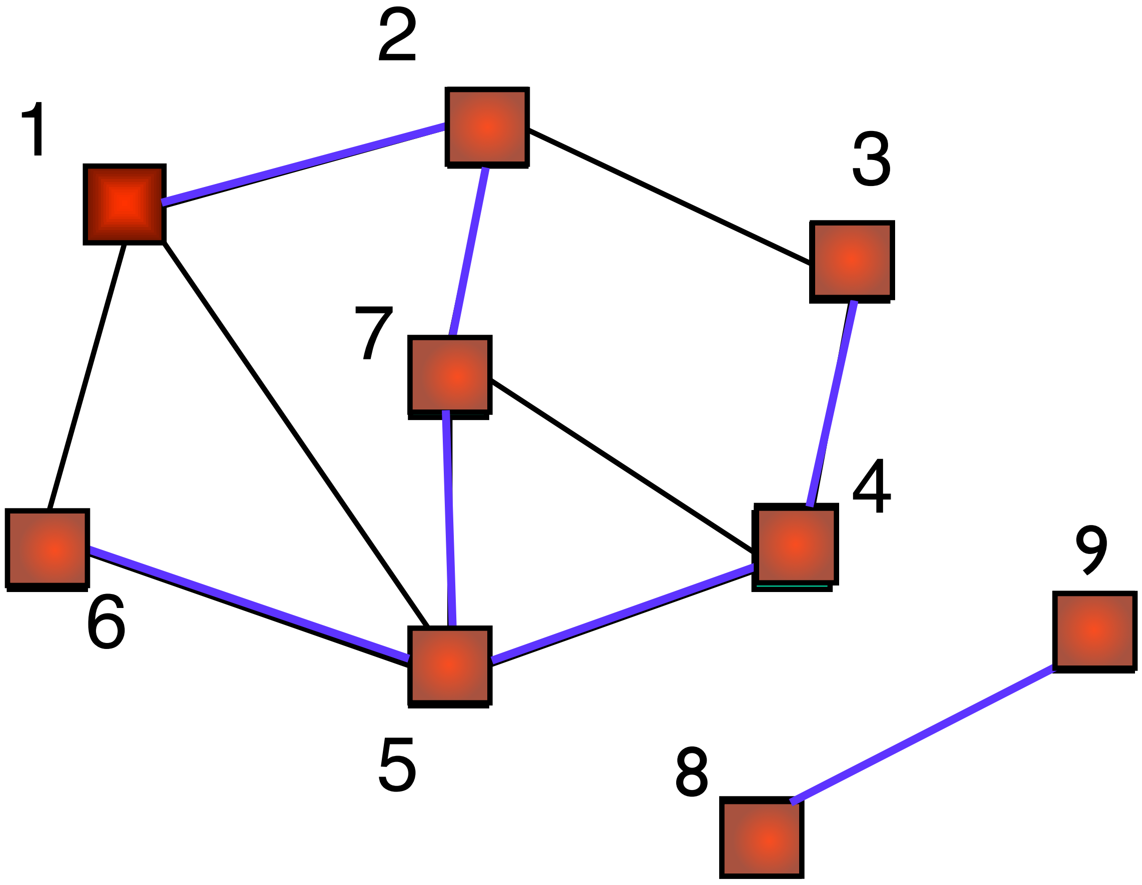
- Adjacency Matrix: $\Theta(|V|^2)$
- Adjacency List: $\Theta(|V| + |E|)$

One Last look at the Example



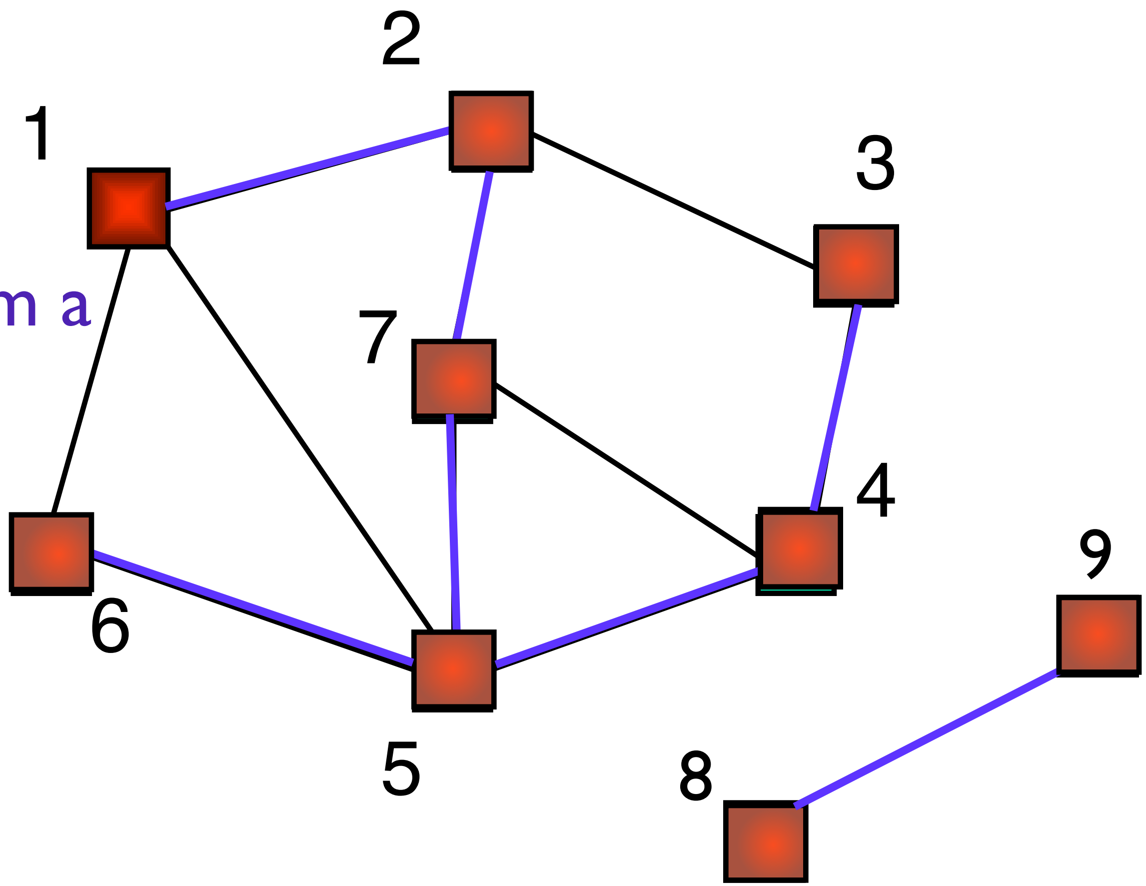
Portland State UNIVERSITY

One Last look at the Example

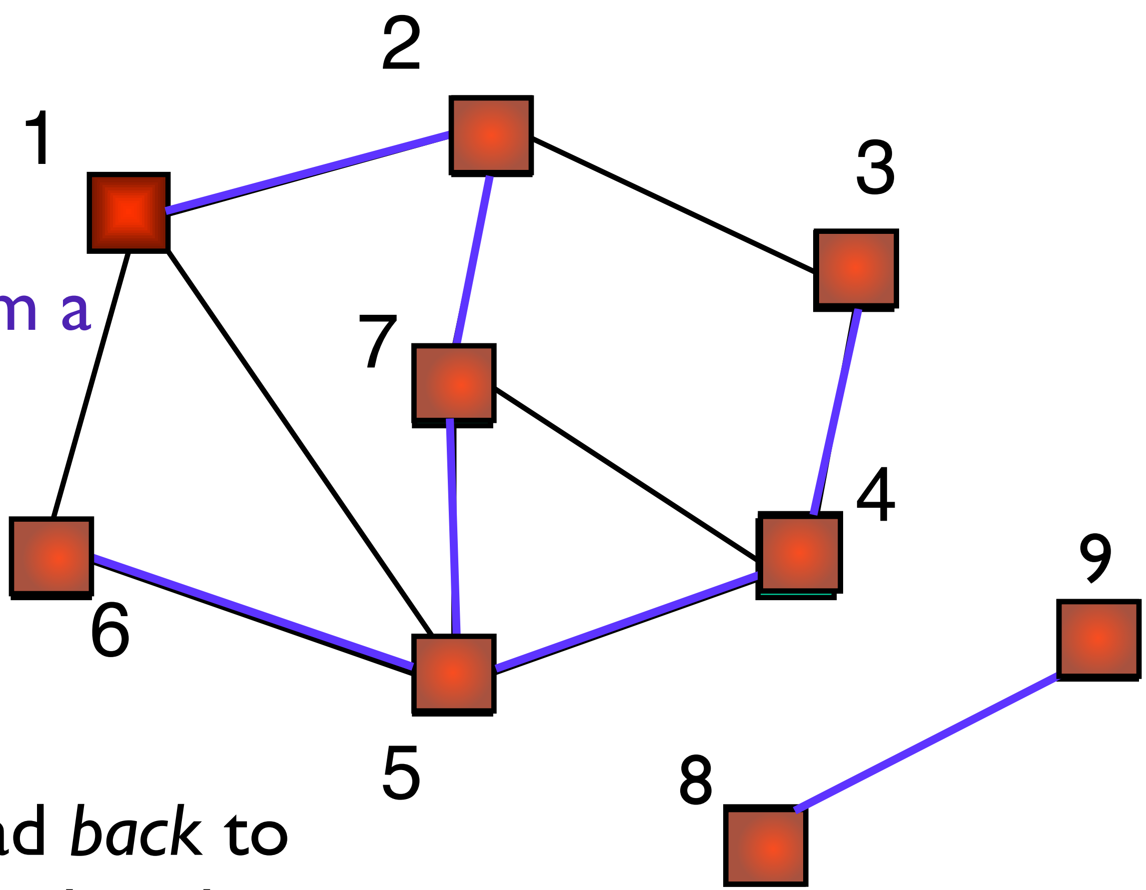


One Last look at the Example

Blue edges form a spanning three



One Last look at the Example



Blue edges form a spanning three

Black edges lead *back* to an already-visited node

Applications

- Checking for connectivity
  - How?
- Checking for Cycles
  - How?