1. **Brute-force search (n.d.). In *Wikipedia*.**

Retrieved from <https://en.wikipedia.org/wiki/Brute-force_search>

Drawing on this and the following Wikipedia reading, we discuss in class these two types of algorithms: brute-force and greedy. Students should understand the difference between a **brute-force algorithm and a greedy one**. So, for this reading and the one below, students may need to find additional resources if the above are insufficient, but any mathematical discussion of running times (big-O notation, etc.) is optional reading.

1. **Greedy algorithm (n.d.). In *Wikipedia.***

Retrieved from <https://en.wikipedia.org/wiki/Greedy_algorithm>

Read the introduction which defines a **greedy algorithm**, as well as the section on Specifics. The rest of the article is optional.

1. **Kun, J. (2013). Depth- and Breadth-First Search.**

Retrieved from [http://web.archive.org/web/20150316153709/http://jeremykun.com/2013/01/22/depth-and-breadth-first-search/](http://web.archive.org/web/20150316153709/http:/jeremykun.com/2013/01/22/depth-and-breadth-first-search/)

This reading explains **depth- and breadth-first search in detail along with Python implementation**. We will be directly discussing this code in an activity in class, so try to learn exactly how it functions. You can also use this code for the optional question of the pre-class work. If you are struggling with this reading, you should read the Bryukhanov reading first.

1. **(Optional). Patel, A. (n.d.). Introduction to A\*.**

Retrieved from <http://www.redblobgames.com/pathfinding/a-star/introduction.html>

A\* ("A star") is a heuristic path-finding algorithm often used in games and other commercial applications, including navigation tools and maps. We will not be covering A\* in class but you may be interested to learn about this widely used algorithm.

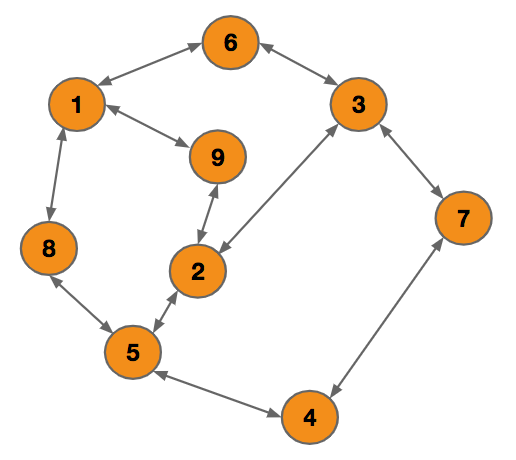
1. **Bryukhanov, V. (N.D). Labyrinth Algorithms.**

Retrieved from <http://bryukh.com/labyrinth-algorithms/>

This reading **visually illustrates depth-first and breadth-first strategies and shows simple Python code implementation**. Pay attention to the strategies and main steps of each algorithm. The section on A\* is optional.

**Study Guide**

1. Define brute-force and greedy algorithms and explain the differences between them, particularly with respect to searching.
   1. **Brute force search** – find all possible outcomes for the solution and check if it satisfies the problem statement
   2. **Greedy algorithm** – make locally optimal choice at each stage to approximate globally optimal solution in a reasonable amount of time
      1. **Check the immediate next option and pick the best immediate option**
      2. **Eg** – find min number of 0.5 and 0.2 coins that can make up 17.8
      3. Drawback: if the immediate choice is not the right route, it will still take it as the algo will not consider further choices down each route
2. What is the difference between a local (relative) optimum and global (absolute) optimum?
   1. **Local optimum** – the most optimal immediate choice in each node connection
   2. **Global optimum** – the most optimal choice in the whole network
3. Explain the mechanics of breadth-first and depth-first searches. There are many easily-searchable video tutorials available to supplement your reading if you need additional material.
   1. **Breadth-first search** **(use queue to store node, first in first out)**
      1. start at vertex, pick any unvisited vertex and check if is goal
      2. Visit neighbour of earlier visited node before neighbour of recently visited ones
      3. Look around vertex before going into depths of graph then recursively apply search
      4. **Only for short distance node**
   2. **Depth-first search (use stack instead of queue to store nodes, first in last out)**
      1. start at vertex, pick any unvisited vertex and check if is goal
      2. Recursively apply depth first search to vertex, ignore those that already visited
      3. Repeat until all adjacent vertices are visited
4. Explain intuitively the difference between a stack and a queue.
   1. **Stack** – first in last out
   2. **Queue** – first in first out
5. Be able to determine whether a path exists between two nodes of a graph using both depth-first and breadth-first search. Explain how to do so.
   1. Directed graph: triple G= (V, E, phi) vertices, edges and phi is adjacency function, which edge connect to which vertices
   2. Undirected graph: edges are undirected
   3. Depth first search:
      1. Start somewhere, check if node value is the same as the sought value.
      2. If no, then go to the adjacent value
6. *Practice*! Assume that the starting node is node 3 and the target goal is node 5. Which brute-force technique (depth-first search or breadth-first search) could have explored the paths given below before stopping--if even possible? For each path, your answer may be either one of them, both of them, or neither.



1. 3, 2, 5: DFS, BFS
2. 3, 7 ,4, 5: DFS BFS
3. 3, 9, 5: DFS
4. 3, 6, 1, 8, 5: DFS
5. 3, 6, 1, 5: DFS
6. 3, 6, 1, 9, 2, 5: DFS

**Pre-class Work**

Come to class with your answers to questions 1-3 ready to paste into a pre-class work poll.

1. Observe the breadth-first and depth-search strategies at <https://course-resources.minerva.kgi.edu/2015-Fall/CS50/1.09.2/draw-streets.html>.
2. Under what circumstances does a breadth-first search find the path faster than depth-first? Under what circumstances does a depth-first search find the path faster than breadth-first? How does this compare to the graphs in the Bryukhanov Labyrinth Algorithms reading?
   1. breadth first search is faster: when the nearest intersection is in another section of the map
   2. depth first search faster: when there are no obstructions at all between nodes and need to find it in an empty route
   3. **Dijkstra’s algorithm** : greedy search, least cost path given by initial node to goal node, follow path of lowest expected total cost/ distance, sorted priority queue (usually faster than both BFS and DFS)
3. Write a clear step-by-step natural language algorithm for finding the connections between nodes in the San Francisco graph using breadth-first and depth-first strategies.
   1. **Breadth-first search** **(use queue to store node, first in first out)**
      1. start at vertex, pick any unvisited vertex and check if is goal
      2. Visit neighbour of earlier visited node before neighbour of recently visited ones
      3. Look around vertex of immediate node before going into depths of street map then recursively apply search
   2. **Depth-first search (use stack instead of queue to store nodes, first in last out)**
      1. start at vertex, pick any unvisited vertex and check if is goal
      2. Recursively apply depth first search to vertex, ignore those that already visited
      3. Repeat until all adjacent vertices are visited
4. ***Optional***: Download the following Python code that constructs a network (graph) of street intersections in San Francisco: <https://drive.google.com/a/minerva.kgi.edu/file/d/0BzwpC_ph6YT0RjN0c1ZwX0RraEk/view?usp=sharing>. Inside this folder you will find sanfrancisco.py, which contains a graph of all the intersections, and tests.py, a file that provides extra code to help you run depth- and breadth-first search functions. Using the code from the Depth- and Breadth-First Search reading, determine whether there is a path in the San Francisco graph from node (1501, 4118) to node (6173, 7065).
5. ***Optional***: modify your depth- and breadth-first search functions to print the actual path found by the algorithm; view those paths at <https://course-resources.minerva.kgi.edu/2015-Fall/CS50/1.09.2/draw-streets.html>.