
Search Strategies

Our standing at graph algorithms:

1. Search Strategies (Current)
2. Combinatorial Search(Chapter)
3. Advanced Graph Algorithm(Current)
4. Graph Problem Patterns(Future Chapter)

Searching ¹ is one of the most effective tools in algorithms. We have seen them being widely applied in the field of artificial intelligence to offer either exact or approximate solutions for complex problems such as puzzles, games, routing, scheduling, motion planning, navigation, and so on. On the spectrum of discrete problems, nearly every single one can be modeled as a searching problem together with enumerative combinatorics and **optimizations**. The searching solutions serve as either naive baselines or even as the only existing solutions for some problems. Understanding common searching strategies as the main goal of this chapter along with the search space of the problem lays the foundation of problem analysis and solving, it is just indescribably **powerful** and **important**!

0.1 Introduction

Linear, tree-like data structures, they are all subsets of graphs, making graph searching universal to all searching algorithms. There are many searching

¹https://en.wikipedia.org/wiki/Category:Search_algorithms

strategies, and we only focus on a few decided upon the completeness of an algorithm—being absolutely sure to find an answer if there is one.

Searching algorithms can be categorized into the following two types depending on if the domain knowledge is used to guide selection of the best path while searching:

1. Uninformed Search: This set of searching strategies normally are handled with basic and obvious problem definition and are not guided by estimation of how optimistic a certain node is. The basic algorithms include: Depth-first-Search(DFS), Breadth-first Search(BFS), Bidirectional Search, Uniform-cost Search, Iterative deepening search, and so on. We choose to cover the first four.
2. Informed(Heuristic) Search: This set of searching strategies on the other hand, use additional domain-specific information to find a *heuristic function* which estimates the cost of a solution from a node. Heuristics means “serving to aid discovery”. Common algorithms seen here include: Best-first Search, Greedy Best-first Search, A^* Search. And we only introduce Best-first Search.

Following this introductory chapter, in Chapter Combinatorial Search, we introduce combinatorial problems and its search space, and how to prune the search space to search more efficiently.

Because the search space of a problem can either be of linear or tree structure—an implicit free tree, which makes the graph search a “big deal” in practice of problem solving. Compared with reduce and conquer, searching algorithms treat states and actions atomically: they do not consider any internal/optimal structure they might possess. We recap the **linear search** given its easiness and that we have already learned how to search in multiple linear data structures.

Linear Search As the naive and baseline approach compared with other searching algorithms, linear search, a.k.a sequential search, simply traverse the linear data structures sequentially and checking items until a target is found. It consists of a **for/while** loop, which gives as $O(n)$ as time complexity, and no extra space needed. For example, we search on list A to find a target t :

```

1 def linearSearch(A, t): #A is the array, and t is the target
2     for i,v in enumerate(A):
3         if A[i] == t:
4             return i
5     return -1

```

Linear Search is rarely used practically due to its lack of efficiency compared with other searching methods such as hashmap and binary search that we will learn soon.

Searching in Un-linear Space For the un-linear data structure, or search space comes from combinatorics, they are generally be a graph and sometimes be a rooted tree. Because mostly the search space forms a search tree, we introduce searching strategies on a search tree first, and then we specifically explore searching in a tree, recursive tree traversal, and search in a graph.

Generatics of Search Strategies

Assume we know our state space, searching or state-space search is the process of searching through a state space for a solution by making explicit a sufficient portion of an implicit state-space graph, in the form of a search tree, to include a goal node.

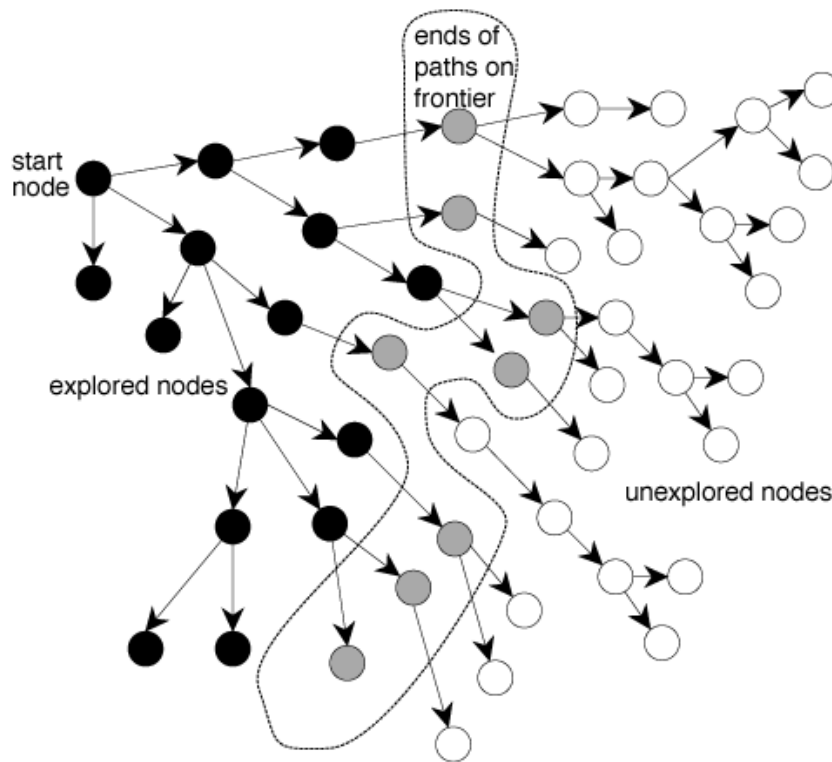


Figure 1: Graph Searching

Nodes in Searching Process In the searching process, nodes in the targeting data structure can be categorized into three sets as shown in Fig.1 and we distinguish the state of a node—which set they are at with a color each.

- Unexplored set–WHITE: initially all nodes in the graph are in the unexplored set, and we assign WHITE color. Nodes in this set have not yet being visited yet.
- Frontier set–GRAY: nodes which themselves have been just discovered/visited and they are put into the *frontier* set, waiting to be expanded; that is to say their children or adjacent nodes (through outgoing edges) are about to be discovered and have not all been visited—not all being found in the frontier set yet. This is an intermediate state between WHITE and BLACK, which is ongoing, visiting but not yet completed. Gray vertex might have adjacent vertices of all three possible states.
- Explored set–BLACK: nodes have been fully explored after being in the frontier set; that is to say none of their children is not explored and being in the unexplored set. For black vertex, all vertices adjacent to them are nonwhite.

All searching strategies follow the general tree search algorithm:

1. At first, put the state node in the frontier set.

```
1 frontier = {S}
```

2. Loop through the frontier set, if it is empty then searching terminates. Otherwise, pick a node n from frontier set:
 - (a) If n is a goal node, then return solution
 - (b) Otherwise, generate all of n 's successor nodes and add them all to frontier set.
 - (c) Remove n from frontier set.

Search process constructs a *search tree* where the root is the start state. Loops in graph may cause the search tree to be infinite even if the state space is small. In this section, we only use either acyclic graph or tree for demonstrating the general search methods. In acyclic graph, there might exist multiple paths from source to a target. For example, the example shown in Fig. ?? has multiple paths from to. Further in graph search section, we discuss how to handle cycles and explain single-path graph search. Changing the ordering in the frontier set leads to different search strategies.

0.2 Uninformed Search Strategies

Through this section, we use Fig. 2 as our exemplary graph to search on. The data structure to represent the graph is as:

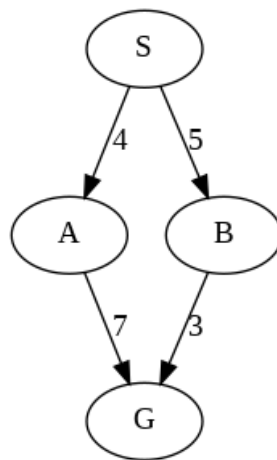


Figure 2: Exemplary Acyclic Graph.

```

1 from collections import defaultdict
2 al = defaultdict(list)
3 al['S'] = [( 'A', 4), ( 'B', 5)]
4 al['A'] = [( 'G', 7)]
5 al['B'] = [( 'G', 3)]

```

With uninformed search, we only know the goal test and the adjacent nodes, but without knowing which non-goal states are better. Assuming and limiting the state space to be a tree for now so that we won't worry about repeated states.

There are generally two ways to order nodes in the frontier without domain-specific information:

- Queue that nodes are first in and first out (FIFO) from the frontier set. This is called breath-first search.
- Stack that nodes are last in but first out (LIFO) from the frontier set. This is called depth-first search.
- Priority queue that nodes are sorted increasingly in the path cost from source to each node from the frontier set. This is called Uniform-Cost Search.

0.2.1 Breath-first Search

Breath-first search always expand the shallowest node in the frontier first, visiting nodes in the tree level by level as illustrated in Fig. 3. Using Q to denote the frontier set, the search process is explained:

```

Q=[A]
Expand A, add B and C into Q

```

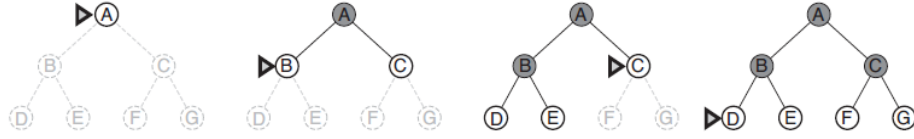


Figure 3: Breadth-first search on a simple search tree. At each stage, the node to be expanded next is indicated by a marker.

```

Q=[B, C]
Expand B, add D and E into Q
Q=[C, D, E]
Expand C, add F and G into Q
Q=[D, E, F, G]
Finish expanding D
Q=[E, F, G]
Finish expanding E
Q=[F, G]
Finish expanding F
Q=[G]
Finish expanding G
Q=[]

```

The implementation can be done with a FIFO queue iteratively as:

```

1 def bfs(g, s):
2     q = [s]
3     while q:
4         n = q.pop(0)
5         print(n, end = ' ')
6         for v, _ in g[n]:
7             q.append(v)

```

Call the function with parameters as `bfs(a1, 'S')`, the output is as:

```
S A B G G
```

Properties Breadth-first search is **complete** because it can always find the goal node if it exists in the graph. It is also **optimal** given that all actions(arcs) have the same constant cost, or costs are positive and non-decreasing with depth.

Time Complexity We can clearly see that BFS scans each node in the tree exactly once. If our tree has n nodes, it makes the time complexity $O(n)$. However, the search process can be terminated once the goal is found, which can be less than n . Thus we measure the time complexity by counting the number of nodes expanded while searching is running. Assume the tree has a branching factor b at each non-leaf node and the goal node locates at depth d , we sum up the number of nodes from depth 0 to depth d , the total

number of nodes expanded are:

$$n = \sum_{i=0}^d b^i \quad (1)$$

$$= \frac{b^{d+1} - 1}{b - 1} \quad (2)$$

Therefore, we have a time complexity of $O(b^d)$. It is usually very slow to find solutions with a large number of steps because it must look at all shorter length possibilities first.

Space Complexity The space is measured in terms of the maximum size of frontier set during the search. In BFS, the maximum size is the number of nodes at depth d , resulting the total space cost to $O(b^d)$.

0.2.2 Depth-first Search

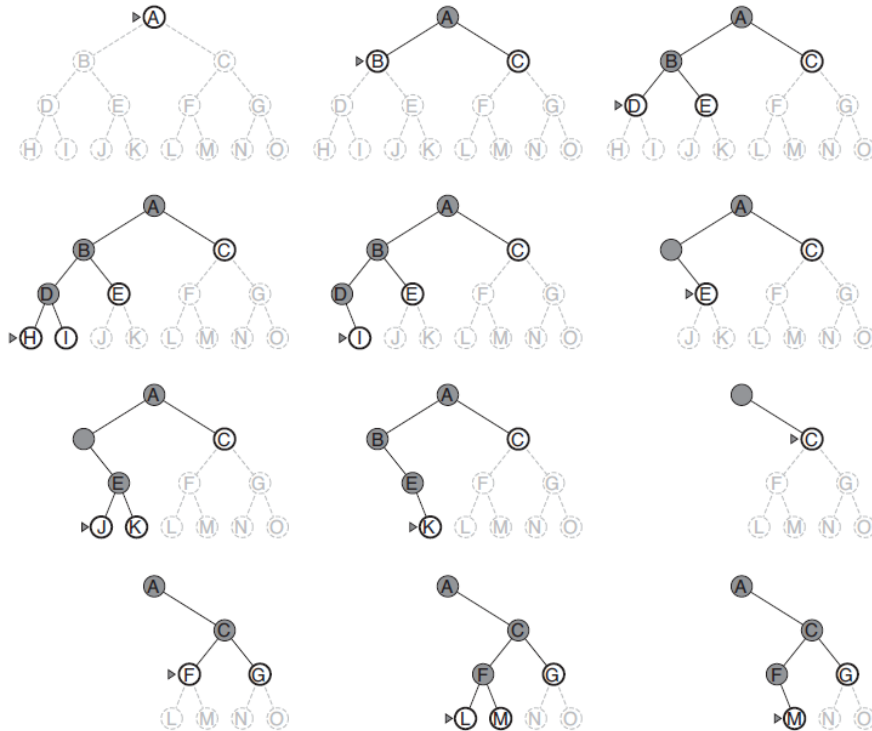


Figure 4: Depth-first search on a simple search tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory as node L disappears. Dark gray marks nodes that is being explored but not finished.

Depth-first search on the other hand always expand the deepest node from the frontier first. As shown in Fig. 4, Depth-first search starts at the root node and continues branching down a particular path. Using S to denote the frontier set which is indeed a stack, the search process is explained:

```
S=[A]
Expand A, add C and B into S
S=[C, B]
Expand B, add E and D into S
S=[C, E, D]
Expand D
S=[C, E]
Expand E
S=[C]
Expand C, add G and F into S
S=[C, G, F]
Expand F
S=[C, G]
Expand G
S=[C]
Expand C
S=[]
```

Depth-first can be implemented either recursively or iteratively.

Recursive Implementation In the recursive version, the recursive function keeps calling the recursive function itself to expand its adjacent nodes. Starting from a source node, it always deepen down the path until a leaf node is met and then it backtrack to expand its other siblings (or say other adjacent nodes). The code is as:

```
1 def dfs(g, vi):
2     print(vi, end=' ')
3     for v, _ in g[vi]:
4         dfs(g, v)
```

Call the function with parameters as `dfs(a1, 'S')`, the output is as:

```
S A G B G
```

Iterative Implementation According to the definition, we can implement DFS with LIFO **stack** data structure. The code is similar to that of BFS other than using different data structure from the frontier set.

```
1 def dfs_iter(g, s):
2     stack = [s]
3     while stack:
4         n = stack.pop()
5         print(n, end=' ')
6         for v, _ in g[n]:
7             stack.append(v)
```


Call the function with parameters as `dfs_iter(al, 'S')`, the output is as:

```
S B G A G
```

We observe that the ordering is not exactly the same as of the recursive counterpart. To keep the ordering consistent, we simply need to add the adjacent nodes in reversed order. In practice, we replace $g[n]$ with $g[n][::-1]$.

Properties DFS may not terminate without a fixed depth bound to limit the amount of nodes that it expand. DFS is **not complete** because it always deepens the search and in some cases the supply of nodes even within the cutting off fixed depth bound can be infinitely. DFS is **not optimal**, in our example, of our goal node is C, it goes through nodes A, B, D, E before it finds node C. While, in the BFS, it only goes through nodes A and C. However, when we are lucky, DFS can find long solutions quickly.

Time Complexity For DFS, it might need to explore all nodes within graph to find the target, thus its worst-case time and space complexity is not decided upon by the depth of the goal, but the total depth of the graph, d instead. DFS has the same time complexity as BFS, which is $O(b^d)$.

Space Complexity The stack will at most stores a single path from the root to a leaf node (goal node) along with the remaining unexpanded siblings so that when it has visited all children, it can backward to a parent node, and know which sibling to explore next. Therefore, the space that needed for DFS is $O(bd)$. In most cases, the branching factor is a constant, which makes the space complexity be mainly influenced by the depth of the search tree. Obviously, DFS has great efficiency in space, which is why it is adopted as the basic technique in many areas of computer science, such as solving constraint satisfaction problems(CSPs). The backtracking technique we are about to introduce even further optimizes the space complexity on the basis of DFS.

0.2.3 Uniform-Cost Search(UCS)

When a priority queue is used to order nodes measured by the path cost of each node to the root in the frontier, this is called uniform-cost search, aka Cheapest First Search. In UCS, frontier set is expanded only in the direction which requires the minimum cost to travel to from root node. UCS only terminates when a path has explored the goal node, and this path is the cheapest path among all paths that can reach to the goal node from the initial point. When UCS is applied to find shortest path in a graph, it is called Dijkstra's Algorithm.

We demonstrate the process of UCS with the example shown in Fig. 2.

Here, our source is ‘S’, and the goal is ‘G’. We are set to find a path from source to goal with minimum cost. The process is shown as:

```
Q = [(0, S)]
Expand S, add A and B
Q = [(4, A), (5, B)]
Expand A, add G
Q = [(5, B), (11, G)]
Expand B, add G
Q = [(8, G), (11, G)]
Expand G, goal found, terminate.
```

And the Python source code is:

```
1 import heapq
2 def ucs(graph, s, t):
3     q = [(0, s)] # initial path with cost 0
4     while q:
5         cost, n = heapq.heappop(q)
6         # Test goal
7         if n == t:
8             return cost
9         else:
10            for v, c in graph[n]:
11                heapq.heappush(q, (c + cost, v))
12    return None
```

Properties Uniformed-Cost Search is **complete** as a similar search strategy compared with breath-first search(using queue). It is optimal even if there exist negative edges.

Time and Space Complexity Similar to BFS, both the worst case time and space complexity is $O(b^d)$. When all edge costs are c , and C^* is the best goal path cost, the time and space complexity can be more precisely represented as $O(b^{C^*/c})$.

0.2.4 Iterative-Deepening Search

Iterative-Deepening Search(IDS) is a modification on top of DFS, more specifically depth limited DFS(DLS); as the name suggests, IDS sets a maximum depth as a “depth bound”, and it calls DLS as a subroutine looping from depth zero to maximum depth to expand nodes just as DFS will do and it only does goal test for nodes at the testing depth.

Using the graph in Fig. 2 as an example. The process is shown as:

```
maxDepth = 3

depth = 0: S = [S]
Test S, goal not found

depth = 1: S = [S]
```

```

Expand S, S = [B, A]
Test A, goal not found
Test B, goal not found

depth = 2: S=[S]
Expand S, S=[B, A]
Expand A, S=[B, G]
Test G, goal found, STOP

```

The implementation of the DLS goes easier with recursive DFS, we use a count down to variable `maxDepth` in the function, and will only do goal testing until this variable reaches to zero. The code is as:

```

1 def dls(graph, cur, t, maxDepth):
2     # End Condition
3     if maxDepth == 0:
4         if cur == t:
5             return True
6     if maxDepth < 0:
7         return False
8
9     # Recur for adjacent vertices
10    for n, _ in graph[cur]:
11        if dls(graph, n, t, maxDepth - 1):
12            return True
13    return False

```

With the help of function `dls`, the implementation of DLS is just an iterative call to the subroutine:

```

1 def ids(graph, s, t, maxDepth):
2     for i in range(maxDepth):
3         if dls(graph, s, t, i):
4             return True
5     return False

```

Analysis It appears to us that we are undermining the efficiency of the original DFS since the algorithm ends up visiting top level nodes of the goal multiple times. However, it is not as expensive as it seems to be, since in a tree most of the nodes are in the bottom levels. If the goal node locates at the bottom level, DLS will not have an obvious efficiency decline. But if the goal locates on topper levels on the right side of the tree, it avoids to visit all nodes across all depths on the left half first and then be able to find this goal node.

Properties Through the depth limited DFS, IDS has advantages of DFS:

- Limited space linear to the depth and branching factor, giving $O(bd)$ as space complexity.
- In practice, even with redundant effort, it still finds longer path more quickly than BFS does.

By iterating through from lower to higher depth, IDS has advantages of BFS, which comes with **completeness** and **optimality** stated the same as of BFS.

Time and Space Complexity The space complexity is the same as of BFS, $O(bd)$. The time complexity is slightly worse than BFS or DFS due to the repetitive visiting nodes on top of the search tree but it still has the same worst case exponential time complexity, $O(b^d)$.

0.2.5 Bidirectional Search**

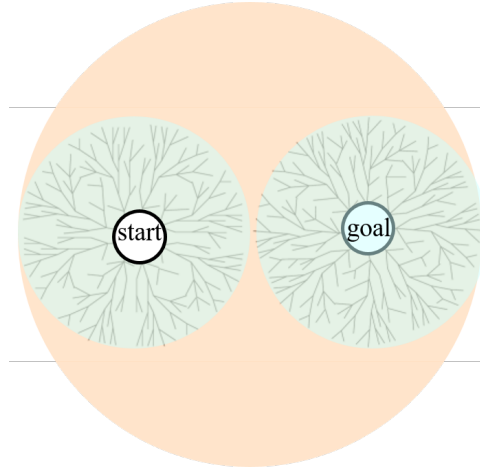


Figure 5: Bidirectional search.

Bidirectional search applies breadth-first search from both the start and the goal node, with one BFS from start moving forward and one BFS from the goal moving backward until their frontiers meet. This process is shown in Fig. 5. As we see, each BFS process only visit $O(b^{d/2})$ nodes comparing with one single BFS that visits $O(b^d)$ nodes. This will improve both the time and space efficiency by $b^{d/2}$ times compared with vanilla BFS.

Implementation Because the BFS that starts from the goal needs to move backwards, the easy way to do this is to create another copy of the graph wherein each edge has opposite direction compared with the original. By creating a reversed graph, we can use a forward BFS from the goal.

We apply level by level BFS instead of updating the queue one node by one node. For better efficiency of the intersection of the frontier set from both BFS, we use **set** data structure instead of simply a **list** or a **FIFO** queue.

Use Fig. 2 as an example, if our source and goal is ‘S’ and ‘G’ respectively, if we proceed both BFS simultaneously, the process looks like this:

```

qs = ['S']
qt = ['G']
Check intersection, and proceed
qs = ['A', 'B']
qt = ['A', 'B']
Check intersection, frontier meet, STOP

```

No process in this case, however, the above process will end up missing the goal node if we change our goal to be 'A'. This process looks like:

```

qs = ['S']
qt = ['A']
Check intersection, and proceed
qs = ['A', 'B']
qt = ['S']
Check intersection, and proceed
qs = ['G']
qt = []
STOP

```

This because for source and goal nodes that has a shortest path with even length, if we proceed the search process simultaneously, we will always end up missing the intersection. Therefore, we process each BFS iteratively—one at a time to avoid such troubles.

The code for one level at a time BFS with `set` and for the intersection check is as:

```

1 def bfs_level(graph, q, bStep):
2     if not bStep:
3         return q
4     nq = set()
5     for n in q:
6         for v, c in graph[n]:
7             nq.add(v)
8     return nq
9
10 def intersect(qs, qt):
11     if qs & qt: # intersection
12         return True
13     return False

```

The main code for bidirectional search is as:

```

1 def bis(graph, s, t):
2     # First build a graph with opposite edges
3     bgraph = defaultdict(list)
4     for key, value in graph.items():
5         for n, c in value:
6             bgraph[n].append((key, c))
7     # Start bidirectional search
8     qs = {s}
9     qt = {t}
10    step = 0
11    while qs and qt:
12        if intersect(qs, qt):

```

```

13     return True
14     qs = bfs_level(graph, qs, step%2 == 0)
15     qt = bfs_level(bgraph, qt, step%2 == 1)
16     step = 1 - step
17     return False

```

0.2.6 Summary

Table 1: Performance of Search Algorithms on Trees or Acyclic Graph

| Method | Complete | Optimal | Time | Space |
|----------------------|----------|---------|--------------|--------------|
| BFS | Y | Y, if | $O(b^d)$ | $O(b^d)$ |
| UCS | Y | Y | $O(C^*/c)$ | $O(C^*/c)$ |
| DFS | N | N | $O(b^m)$ | $O(bm)$ |
| IDS | Y | Y, if | $O(b^d)$ | $O(bd)$ |
| Bidirectional Search | Y | Y, if | $O(b^{d/2})$ | $O(b^{d/2})$ |

Using b as branching factor, d as the depth of the goal node, and m is the maximum graph depth. The properties and complexity for the five uninformed search strategies are summarized in Table. 1.

0.3 Graph Search

Cycles This section is devoted to discuss more details about two search strategies—BFS and DFS in more general graph setting. In the last section, we just assumed our graph is either a tree or acyclic directional graph. In more general real-world setting, there can be cycles within a graph which will lead to infinite loops of our program.

Print Paths Second, we talked about the paths, but we never discuss how to track all the paths. In this section, we would like to see how we can track paths first, and then with the tracked paths, we detect cycles to avoid getting into infinite loops.

More Efficient Graph Search Third, the last section is all about tree search, however, in a large graph, this is not efficient by visiting some nodes multiple times if they happen to be on the multiple paths between the source and any other node in the graph. Usually, depends on the application scenarios, graph search which remembers already-expanded nodes/states in the graph and avoids expanding again by checking any about to be expanded node to see if it exists in frontier set or the explored set. This section, we introduce graph search that suits for general purposed graph problems.

Visiting States We have already explained that we can use three colors: WHITE, GREY, and BLACK to denote nodes within the unexpanded, frontier, and explored set, respectively. We are doing so to avoid the hassles of tracking three different sets, with visiting state, it is all simplified to a color check. We define a `STATE` class for convenience.

```
class STATE:
    white = 0
    gray = 1
    black = 2
```

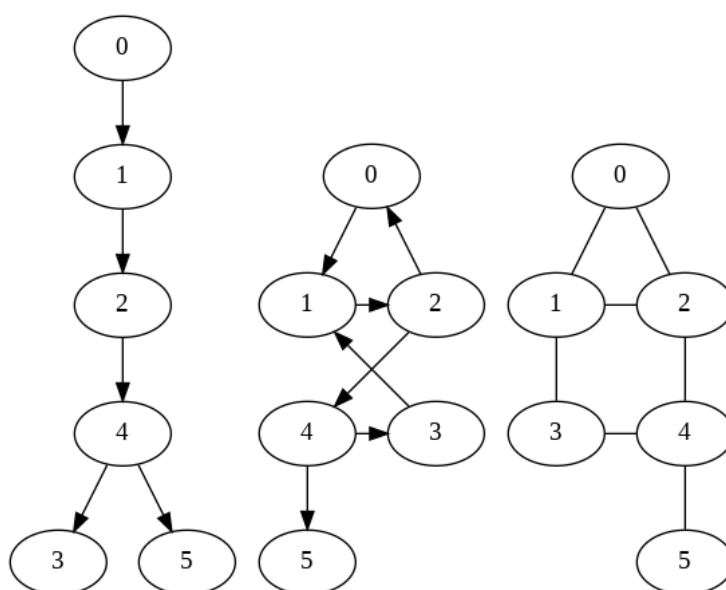


Figure 6: Exemplary Graph: Free Tree, Directed Cyclic Graph, and Undirected Cyclic Graph.

In this section, we use Fig. 6 as our exemplary graphs. Each's data structure is defined as:

- Free Tree:

```
1 ft = [[1], [2], [4], [], [3, 5], []]
```

- Directed Cyclic Graph:

```
1 deg = [[1], [2], [0, 4], [1], [3, 5], []]
```

- Undirected Cyclic Graph

```
1 ucg = [[1, 2], [0, 2, 3], [0, 1, 4], [1, 4], [2, 3, 5],
         [4]]
```

Search Tree It is important to realize the Searching ordering is always forming a tree, this is terminologized as **Search Tree**. In a tree structure, the search tree is itself. In a graph, we need to figure out the search tree and it decides our time and space complexity.

0.3.1 Depth-first Search in Graph

In this section we will further the depth-first tree search and explore depth-first graph search to compare their properties and complexity.

Depth-first Tree Search

Vanilla Depth-first Tree Search Our previous code slightly modified to suit for the new graph data structure works fine with the free tree in Fig. 6. The code is as:

```
1 def dfs(g, vi):
2     print(vi, end=' ')
3     for nv in g[vi]:
4         dfs(g, nv)
```

However, if we call it on the cyclic graph, `dfs(dcg, 0)`, it runs into stack overflow.

Cycle Avoiding Depth-first Tree Search So, how to avoid cycles? We know the definition of a cycle is a closed path that has at least one node that repeats itself; in our failed run, we were stuck with cycle `[0, 1, 2, 0]`. Therefore, let us add a **path** in the recursive function, and whenever we want to expand a node, we check if it forms a cycle or not by checking the membership of a candidate to nodes comprising the path. We save all paths and the visiting ordering of nodes in two lists: **paths** and **orders**. The recursive version of code is:

```
1 def dfs(g, vi, path):
2     paths.append(path)
3     orders.append(vi)
4     for nv in g[vi]:
5         if nv not in path:
6             dfs(g, nv, path+[nv])
7     return
```

Now we call function `dfs` for `ft`, `dcg`, and `ucg`, the **paths** and **orders** for each example is listed:

- For the free tree and the directed cyclic graph, they have the same output. The **orders** are:

```
[0, 1, 2, 4, 3, 5]
```

And the **paths** are:


```
[[0], [0, 1], [0, 1, 2], [0, 1, 2, 4], [0, 1, 2, 4, 3], [0, 1, 2, 4, 5]]
```

- For the undirected cyclic graph, **orders** are:

```
[0, 1, 2, 4, 3, 5, 3, 4, 2, 5, 2, 1, 3, 4, 5, 4, 3, 1, 5]
```

And the **paths** are:

```
[[0],
[0, 1],
[0, 1, 2],
[0, 1, 2, 4],
[0, 1, 2, 4, 3],
[0, 1, 2, 4, 5],
[0, 1, 3],
[0, 1, 3, 4],
[0, 1, 3, 4, 2],
[0, 1, 3, 4, 5],
[0, 2],
[0, 2, 1],
[0, 2, 1, 3],
[0, 2, 1, 3, 4],
[0, 2, 1, 3, 4, 5],
[0, 2, 4],
[0, 2, 4, 3],
[0, 2, 4, 3, 1],
[0, 2, 4, 5]]
```

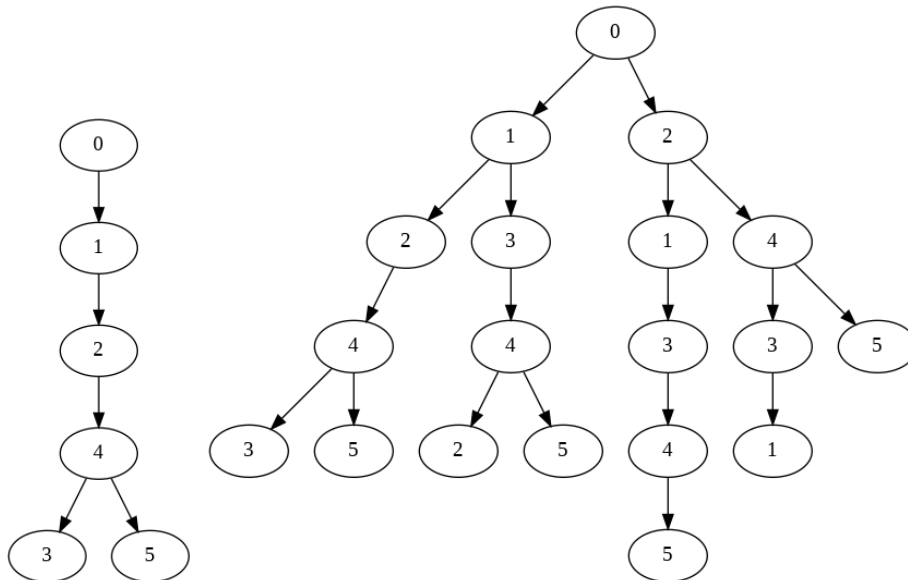


Figure 7: Search Tree for Exemplary Graph: Free Tree and Directed Cyclic Graph, and Undirected Cyclic Graph.

These paths mark the search tree, we visualize the search tree for each exemplary graph in Fig. 7.

Depth-first Graph Search

We see that from the above implementation, for a graph with only 6 nodes, we have been visiting nodes for a total of 19 times. A lot of nodes have been repeating. 1 appears 3 times, 3 appears 4 times, and so on. As we see the visiting order being represented with a **search tree** in Fig. 7, our complexity is getting close to $O(b^h)$, where b is the branching factor and h is the total vertices of the graph, marking the upper bound of the maximum depth that the search can traverse. If we simply want to search if a value or a state exists in the graph, this approach insanely complicates the situation. What we do next is to avoid revisiting the same vertex again and again by tracking the visiting state of a node.

In the implementation, we only track the longest path—from source vertex to vertex that has no more unvisited adjacent vertices.

```

1 def dfgs(g, vi, visited, path):
2     visited.add(vi)
3     orders.append(vi)
4     bEnd = True # node without unvisited adjacent nodes
5     for nv in g[vi]:
6         if nv not in visited:
7             if bEnd:
8                 bEnd = False
9                 dfgs(g, nv, visited, path + [nv])
10    if bEnd:
11        paths.append(path)

```

Now, we call this function with **ucg** as:

```

1 paths, orders = [], []
2 dfgs(ucg, 0, set(), [0])

```

The output for **paths** and **orders** are:

```

([ [0, 1, 2, 4, 3], [0, 1, 2, 4, 5], [0, 1, 2, 4, 3, 5] ])

```

Did you notice that the depth-first graph search on the undirected cyclic graph shown in Fig. 6 has the same visiting order of nodes and same search tree as the free tree and directed cyclic graph in Fig. 6?

Efficient Path Backtrace In graph search, each node is added into the frontier and expanded only once, and the search tree of a $|V|$ graph will only have $|V| - 1$ edges. Tracing paths by saving each path as a list in the frontier set is costly; for a partial path in the search tree, it is repeating itself multiple times if it happens to be part of multiple paths, such as partial path $0 \rightarrow 1 \rightarrow 2 \rightarrow 4$. We can bring down the memory cost to $O(|v|)$ if we only save edges by using a **parent dict** with key and value referring as the node and

its parent node in the path, respectively. For example, edge 0->1 is saved as `parent[1] = 0`. Once we find out goal state, we can backtrace from this goal state to get the path. The backtrace code is:

```

1 def backtrace(s, t, parent):
2     p = t
3     path = []
4     while p != s:
5         path.append(p)
6         p = parent[p]
7     path.append(s)
8     return path[::-1]

```

Now, we modify the dfs code as follows to find a given state (vertex) and obtaining the path from source to target:

```

1 def dfgs(g, vi, s, t, visited, parent):
2     visited.add(vi)
3     if vi == t:
4         return backtrace(parent, s, t)
5
6     for nv in g[vi]:
7         if nv not in visited:
8             parent[nv] = vi
9             fpath = dfgs(g, nv, s, t, visited, parent)
10            if fpath:
11                return fpath
12
13    return None

```

The whole Depth-first graph search tree constructed from the `parent` dict is delineated in Fig. 8 on the given example.

Properties The completeness of DFS depends on the search space. If your search space is finite, then Depth-First Search is complete. However, if there are infinitely many alternatives, it might not find a solution. For example, suppose you were coding a path-search problem on city streets, and every time your partial path came to an intersection, you always searched the left-most street first. Then you might just keep going around the same block indefinitely.

The depth-first graph search is **nonoptimal** just as Depth-first tree search. For example, if the task is to find the shortest path from source 0 to target 2. The shortest path should be 0->2, however depth-first graph search will return 0->1->2. For the search tree using depth-first tree search, it can find the shortest path from source 0 to 2. However, it will explore the whole left branch starts from 1 before it finds its goal node on the right side.

Time and Space Complexity For the depth-first graph search, we use aggregate analysis. The search process covers all edges, $|E|$ and vertices,

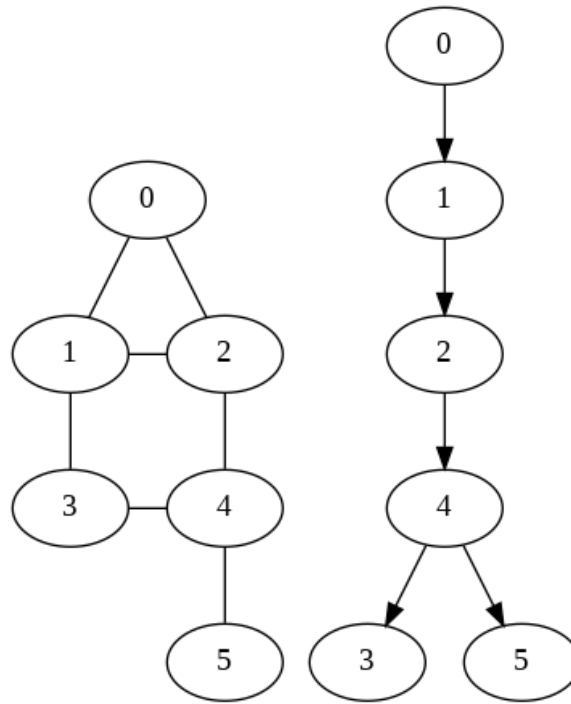


Figure 8: Depth-first Graph Search Tree.

$|V|$, which makes the time complexity as $O(|V| + |E|)$. For the space, it uses space $O(|V|)$ in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices.

Applications

Depth-first tree search is adopted as the basic workhorse of many areas of AI, such as solving CSP, as it is a brute-force solution. In Chapter Combinatorial Search, we will learn how “backtracking” technique along with others can be applied to speed things up. Depth-first graph search is widely used to solve graph related tasks in non-exponential time, such as Cycle Check(linear time) and shortest path.



Questions to ponder:

- Only track the longest paths.
- How to trace the edges of the search tree?
- Implement the iterative version of the recursive code.

0.3.2 Breath-first Search in Graph

We further breath-first tree search and explore breath-first graph search in this section to grasp better understanding of one of the most general search strategies. Because that BFS is implemented iteratively, the implementation in this section of sheds light to the iterative counterparts of DFS's recursive implementations from last section.

Breath-first Tree Search

Similarly, out vanilla breath-first tree search shown in Section. ?? will get stuck with the cyclic graph in Fig. 6.

Cycle Avoiding Breath-first Tree Search We avoid cycles with similar strategy to DFS tree search that traces paths and checks membership of node. In BFS, we track paths by explicitly adding paths to the `queue`. Each time we expand from the frontier (queue), the node we need is the last item in the path from the queue. In the implementation, we only track the longest paths from the search tree and the visiting orders of nodes. The Python code is:

```

1 def bfs(g, s):
2     q = [[s]]
3     paths, orders = [], []
4     while q:
5         path = q.pop(0)
6         n = path[-1]
7         orders.append(n)
8         bEnd = True
9         for v in g[n]:
10             if v not in path:
11                 if bEnd:
12                     bEnd = False
13                 q.append(path + [v])
14         if bEnd:
15             paths.append(path)
16     return paths, orders

```

Now we call function `bfs` for `ft`, `dgcg`, and `ucg`, the `paths` and `orders` for each example is listed:

- For the free tree and the directed cyclic graph, they have the same output. The `orders` are:

```
[0, 1, 2, 4, 3, 5]
```

And the `paths` are:

```
[[0, 1, 2, 4, 3], [0, 1, 2, 4, 5]]
```

- For the undirected cyclic graph, `orders` are:

```
[0, 1, 2, 2, 3, 1, 4, 4, 4, 3, 3, 5, 3, 5, 2, 5, 4, 1, 5]
```

And the **paths** are:

```
[[0, 2, 4, 5], [0, 1, 2, 4, 3], [0, 1, 2, 4, 5], [0, 1, 3,
4, 2], [0, 1, 3, 4, 5], [0, 2, 4, 3, 1], [0, 2, 1, 3, 4,
5]]
```

Properties We can see the visiting orders of nodes are different from Depth-first tree search counterparts. However, the corresponding search tree for each graph in Fig. 6 is the same as its counterpart–Depth-first Tree Search illustrated in Fig. 7. This highlights how different searching strategies differ by visiting ordering of nodes but not differ at the search-tree which depicts the search space—all possible paths.

Applications However, the Breath-first Tree Search and path tracing is extremely more costly compared with DFS counterpart. When our goal is to enumerate paths, go for the DFS. When we are trying to find shortest-paths, mostly use BFS.

Breath-first Graph Search

Similar to Depth-first Graph Search, we use a **visited** set to make sure each node is only added to the frontier(queue) once and thus expanded only once.

BFGS Implementation The implementation of Breath-first Graph Search with goal test is:

```
1 def bfgs(g, s, t):
2     q = [s]
3     parent = {}
4     visited = {s}
5     while q:
6         n = q.pop(0)
7         if n == t:
8             return backtrace(s, t, parent)
9         for v in g[n]:
10             if v not in visited:
11                 q.append(v)
12                 visited.add(v)
13                 parent[v] = n
14     return parent
```

Now, use the undirected cyclic graph as example to find the path from source 0 to target 5:

```
1 bfgs(ucg, 0, 5)
```

With the found path as:

[0, 2, 4, 5]

While this found path is the shortest path between the two vertices measured by the length. The whole Breath-first graph search tree constructed from the `parent` dict is delineated in Fig. 9 on the given example.

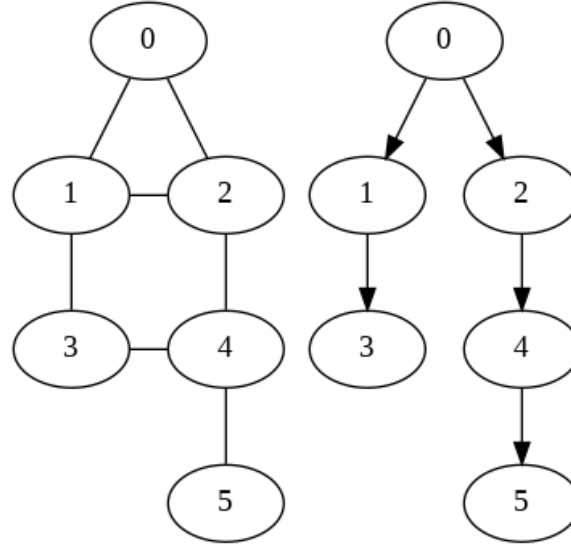


Figure 9: Breath-first Graph Search Tree.

Time and Space Complexity Same to DFBS, the time complexity as $O(|V| + |E|)$. For the space, it uses space $O(|V|)$ in the worst case to store vertices on the current search path, the set of already-visited vertices, as well as the dictionary used to store edge relations. The shortage that comes with costly memory usage of Breath-first Graph Search to Depth-first Graph Search is less obvious compared to Breath-first Tree Search to Depth-first Graph Search.

Tree Search VS Graph Search

There are two important characteristics about tree search and graph search:

- Within a graph $G = (V, E)$, either it is undirected or directed, acyclic or cyclic, both the breath-first and depth-first tree search results the same search tree: They both enumerate all possible states (paths) of the search space.
- The conclusion is different for breath-first and depth-first graph search. For acyclic and directed graph (tree), both search strategies result the

same search tree. However, whenever there exists cycles, the depth-first graph search tree might differ from the breath-first graph search tree.

0.3.3 Depth-first Graph Search

Within this section and the next, we focus on explaining more characteristics of the graph search that avoids repeatedly visiting a vertex. Seemingly these features and details are not that useful judging from current context, but we will see how it can be applied to solve problems more efficiently in Chapter Advanced Graph Algorithms, such as detecting cycles, topological sort, and so on.

As shown in Fig. 10 (a directed graph), we start from 0, mark it gray, and visit its first unvisited neighbor 1, mark 1 as gray, and visit 1's first unvisited neighbor 2, then 2's unvisited neighbor 4, 4's unvisited neighbor 3. For node 3, it doesn't have white neighbors, we mark it to be complete with black. Now, here, we "backtrack" to its predecessor, which is 4. And then we keep the process till 5 become gray. Because 5 has no edge out any more, it becomes black. Then the search backtracks to 4, to 2, to 1, and eventually back to 0. We should notice the ordering of vertices become gray or black is different. From the figure, the gray ordering is [0, 1, 2, 4, 3, 5], and for the black is [3, 5, 4, 2, 1, 0]. Therefore, it is necessary to distinguish the three states in the depth-first graph search at least.

Three States Recursive Implementation We add additional `colors` list to track the color of each vertices, `orders` to track the ordering of the gray, and `completed_orders` for ordering vertices by their ordering of turning into black—when all of a node's neighbors become black which is after the recursive call in the code.

```

1 def dfs(g, s, colors, orders, complete_orders):
2     colors[s] = STATE.gray
3     orders.append(s)
4     for v in g[s]:
5         if colors[v] == STATE.white:
6             dfs(g, v, colors, orders, complete_orders)
7     colors[s] = STATE.black
8     complete_orders.append(s)
9     return

```

Now, we try to call the function with the undirected cyclic graph in Fig. 6.

```

1 v = len(ucg)
2 orders, complete_orders = [], []
3 colors = [STATE.white] * v
4 dfs(ucg, 0, colors, orders, complete_orders)

```

Now, the `orders` and `complete_orders` will end up differently:

```
[0, 1, 2, 4, 3, 5] [3, 5, 4, 2, 1, 0]
```

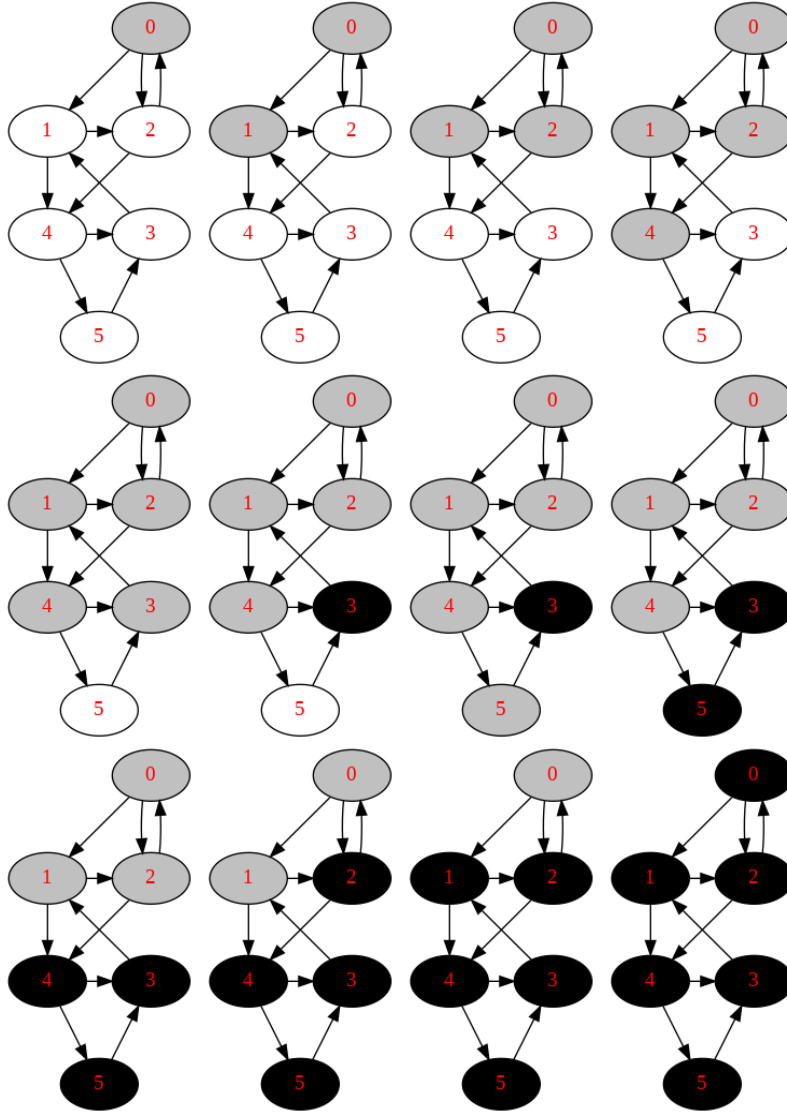



Figure 10: The process of Depth-first Graph Search in Directed Graph. The black arrows denotes the relation of u and its not visited neighbors v . And the red arrow marks the backtrack edge.

Three States and Edges Depth-first Graph Search on graph $G = (V, E)$ connects all reachable vertices from a given source in the graph in the form of a depth-first forest G_π . Edges within G_π are called **tree edges**. Tree edges are edges marked with black arrows in Fig. 11. Other edges in G can be classified into three categories based on Depth-first forest G_π , they are:

1. Back edges which connect a node back to one of its ancestors in the depth-first forest G_π . Marked as red edges in Fig. 11.

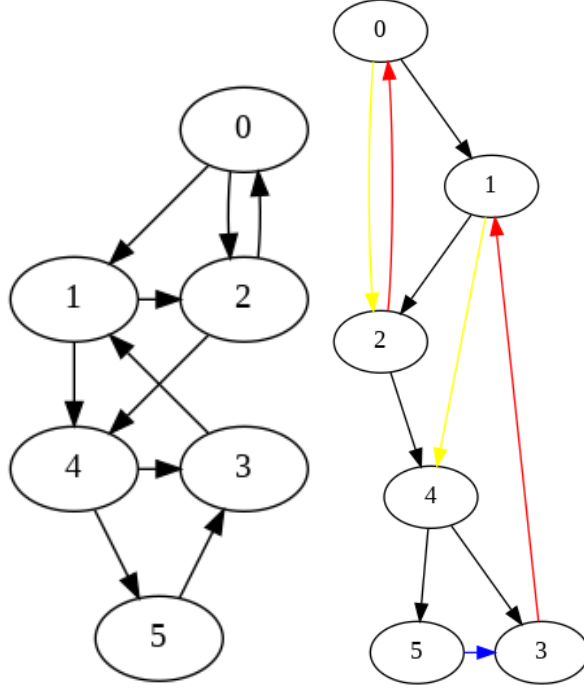


Figure 11: Classification of Edges: black marks tree edge, red marks back edge, yellow marks forward edge, and blue marks cross edge.

2. Forward edges point from a node to one of its descendants in the depth-first forest G_π . Marked as yellow edges in Fig. 11.
3. Cross edges point from a node to a previously visited node that is neither an ancestor nor a descendant in the depth-first forest G_π . Marked as blue edges in Fig. 11.

We can decide the type of tree edge using the DFS execution with the states: for an edge (u, v) , depends on whether we have visited v before in the DFS and if so, the relationship between u and v .

1. If v is WHITE, then the edge is a tree edge.
2. If v is GRAY—both u and v are both being visited—then the edge is a back edge. In directed graph, this indicates that we meet a cycle.
3. If v is BLACK, that v is finished, and that the $start_time[u] < start_time[v]$, then the edge is a forward edge.
4. If v is BLACK, but the $start_time[u] > start_time[v]$, then the edge is a cross edge.

In undirected graph, there is no forward edge or cross edge. Therefore, it does not really need three colors. Usually, we can simply mark it as visited or not visited.

Classification of edges provide important information about the graph, e.g. to if we detect a back edge in directed graph, we find a cycle.

Parenthesis Structure In either undirected or directed graph, the discovered time when state goes from WHITE to GRAY and the finish time when state turns to BLACK from GRAY has the parenthesis structure. We modify `dfs` to track the time: a static variable `t` is used to track the time, `discover` and `finish` is used to record the first discovered and finished time. The implementation is shown:

```

1 def dfs(g, s, colors):
2     dfs.t += 1 # static variable
3     colors[s] = STATE.gray
4     dfs.discover[s] = dfs.t
5     for v in g[s]:
6         if colors[v] == STATE.white:
7             dfs(g, v, colors)
8     # complete
9     dfs.t += 1
10    dfs.finish[s] = dfs.t
11    return

```

Now, we call the above function with directed graph in Fig. 11.

```

1 v = len(dcg)
2 colors = [STATE.white] * v
3 dfs.t = -1
4 dfs.discover, dfs.finish = [-1] * v, [-1] * v
5 dfs(dcg, 0, colors)

```

The output for `dfs.discover` and `dfs.finish` are:

```

([0, 1, 2, 4, 3, 6], [11, 10, 9, 5, 8, 7])

```

From `dfs.discover` and `dfs.finish` list, we can generate a new list of merged order, `merge_orders` that arranges nodes in order of their discovered and finish time. The code is as:

```

1 def parenthesis(dt, ft, n):
2     merge_orders = [-1] * 2 * n
3     for v, t in enumerate(dt):
4         merge_orders[t] = v
5     for v, t in enumerate(ft):
6         merge_orders[t] = v
7
8     print(merge_orders)
9     nodes = set()
10    for i in merge_orders:
11        if i not in nodes:
12            print('(' , i, end = ', ')

```

```

13     nodes.add(i)
14     else:
15         print(i, ' '), end = ' '

```

The output is:

```

1 [0, 1, 2, 4, 3, 3, 5, 6, 6, 5, 4, 2, 1, 0]
2 ( 0, ( 1, ( 2, ( 4, ( 3, 3 ), ( 5, ( 6, 6 ), 5 ), 4 ), 2 ), 1 ),
   0 ),

```

We would easily find out that the ordering of nodes according to the discovery and finishing time makes a well-defined expression in the sense that the parentheses are properly nested.



Questions to ponder:

- Implement the iterative version of the recursive code.

0.3.4 Breadth-first Graph Search

We have already known how to implement BFS of both the tree and graph search versions. In this section, we want to first exemplify the state change process of BFGS with example shown in Fig. 10. Second, we focus on proving that within the breath-first graph search tree, a path between root and any other node is the shortest path.

Three States Iterative Implementation When a node is first put into the frontier set, it is marked with gray color. A node is complete only if all its adjacent nodes turn into gray or black. With the visiting ordering of the breath-first graph search, the state change of nodes in the search process is shown in Fig. 12. The Python code is:

```

1 def bfgs_state(g, s):
2     v = len(g)
3     colors = [STATE.white] * v
4
5     q, orders = [s], [s]
6     complete_orders = []
7     colors[s] = STATE.gray # make the state of the visiting node
8     while q:
9         u = q.pop(0)
10        for v in g[u]:
11            if colors[v] == STATE.white:
12                colors[v] = STATE.gray
13                q.append(v)
14                orders.append(v)
15
16        # complete
17        colors[u] = STATE.black
18        complete_orders.append(u)
19    return orders, complete_orders

```

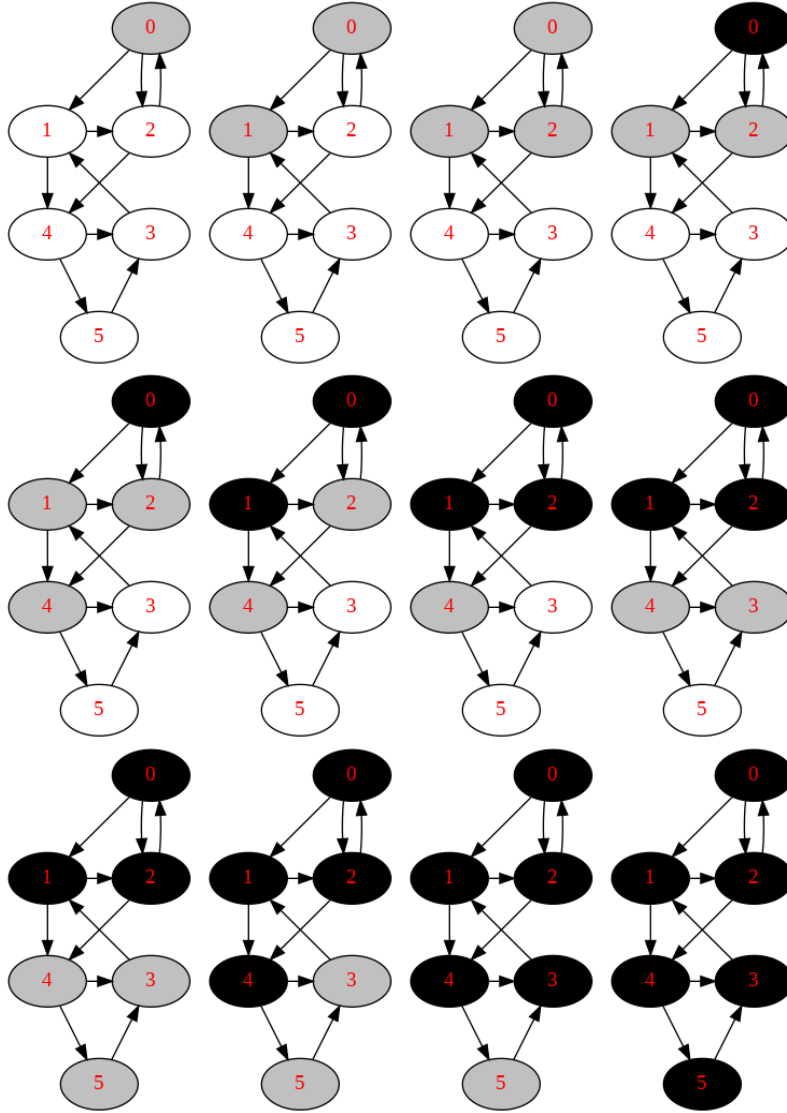


Figure 12: The process of Breath-first Graph Search. The black arrows denotes the the relation of u and its not visited neighbors v . And the red arrow marks the backtrack edge.

The printout of `orders` and `complete_orders` are:

```
([0, 1, 2, 4, 3, 5], [0, 1, 2, 4, 3, 5])
```

Properties In breath-first graph search, the first discovery and finishing time are different for each node, but the discovery ordering and the finishing ordering of nodes are the same ordering.

Shortest Path

Applications

The common problems that can be solved by BFS are those only need one solution: the best one such like getting the shortest path. As we will learn later that breath-first-search is commonly used as archetype to solve graph optimization problems, such as Prim’s minimum-spanning-tree algorithm and Dijkstra’s single-source-paths algorithm.

0.4 Tree Traversal

0.4.1 Depth-First Tree Traversal

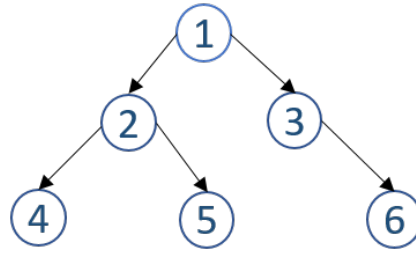


Figure 13: Exemplary Binary Tree

Introduction

Depth-first search starts at the root node and continues branching down a particular path; it selects a child node that is at the deepest level of the tree from the frontier to expand next and defers the expansion of this node’s siblings. Only when the search hits a dead end (a node that has no child) does the search “backtrack” to its parent node, and continue to branch down to other siblings that were deferred. A recursive tree can be traversed recursively. We print out the value of current node, then apply recursive call on the left and right node; by treating each node as a subtree, naturally a recursive call to a node can be thought of handling the traversal of that subtree. The code is quite straightforward:

```
1 def recursive(node):  
2     if not node:  
3         return  
4     print(node.val, end=' ')  
5     recursive(node.left)  
6     recursive(node.right)
```

Now, we call this function with a tree as shown in Fig. 13, the output that indicates the traversal order is:

```
1 1 2 4 5 3 6
```

Three Types of Depth-first Tree Traversal

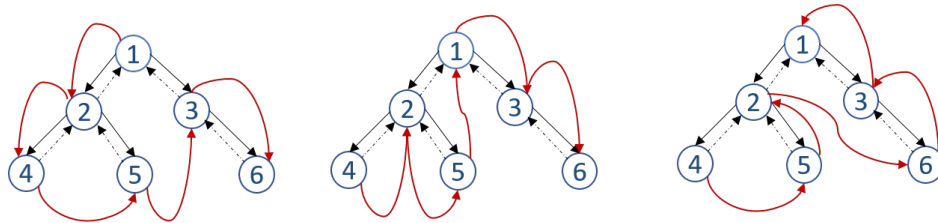



Figure 14: Left: PreOrder, Middle: InOrder, Right: PostOrder. The red arrows marks the traversal ordering of nodes.

The visiting ordering between the current node, its left child, and its right child decides the following different types of recursive tree traversals:

- (a) Preorder Traversal with ordering of [**current node**, **left child**, **right child**]: it visits the nodes in the tree with ordering [1, 2, 4, 5, 3, 6]. In our example, the recursive function first prints the root node 1, then goes to its left child, which prints out 2. Then it goes to node 4. From node 4, it next moves to its left child which is empty and leads to the termination of the recursive call and then the recursion backward to node 4. Since node 4 has no right child, it further backwards to node 2, and then it check 2's right child 5. The same process of node 4 happens on node 5. It backwards to node 2, backwards to node 1, and keep visiting its right child 3, and the process goes on. We draw out this process in Fig. 14.
- (b) Inorder Traversal with ordering of [**left child**, **current node**, **right child**]: it traverses the nodes in ordering of [4, 2, 5, 1, 3, 6]. Three segments will appear with the inorder traversal for a root node: nodes in left subtree, root, and nodes in the right subtree.
- (c) Postorder Traversal with ordering of [**left child**, **right child**, **current node**]: it traverses the nodes in ordering of [4, 5, 2, 6, 3, 1].

We offer the code of Inorder Traversal:

```
1 def inorder_traversal(node):
2     if not node:
3         return
4     inorder_traversal(node.left)
5     print(node.val, end=' ')
6     inorder_traversal(node.right)
```

 Try to check the other two orderings: [left child, current node, right child] and [left child, right child, current node] by hand first and then write the code to see if you get it right?

Return Values

Here, we want to do the task in a different way: We do not want to just print out the visiting orders, but instead write the ordering in a list and return this list. How would we do it? The process is the same, other than we need to return something (not `None` which is default in Python). If we only have empty node, it shall return us an empty list `[]`, if there is only one node, returns `[1]` instead.

Let us use PreOrder traversal as an example. To make it easier to understand, the same queen this time wants to do the same job in a different way, that she wants to gather all the data from these different states to her own hand. This time, she assumes the two generals A and B will return a **list** of the subtree, safely and sound. Her job is going to combine the list returned from the left subtree, her data, and the list returned from the right subtree. Therefore, the left general brings back $A = [2, 4, 5]$, and the right general brings back $B = [3, 6]$. Then the final result will be $queue + A + B = [1, 2, 4, 5, 3, 6]$. The Python code is given:

```

1 def PreOrder(root):
2     if root is None:
3         return []
4     ans = []
5     left = PreOrder(root.left)
6     right = PreOrder(root.right)
7     ans = [root.val] + left + right
8     return ans

```

An Example of Divide and Conquer Be able to understand the returned value and combine them is exactly the method of **divide and conquer**, one of the fundamental algorithm design principles. This is a seemingly trivial change, but it approaches the problem solving from a totally different angle: atomic searching to divide and conquer that highlights the structure of the problem. The printing traversal and returning traversal represents two types of problem solving: the first is through searching—searching and treating each node more separately and the second is through reduce and conquer—reducing the problem to a series of smaller subproblems (subtrees where the smallest are empty subtrees) and construct the result by using the information of current problem and the solutions of the subproblems.

Complexity Analysis

It is straightforward to see that it only visit all nodes twice, one in the forward pass and the other in the backward pass of the recursive call, making the time complexity linear to total number of nodes, $O(n)$. The other way is through the recurrence relation, we would write $T(n) = 2 \times T(n/2) + O(1)$, which gives out $O(n)$ too.

0.4.2 Iterative Tree Traversal

In Chapter Iteration and Recursion, we would know that the recursive function might suffer from the stack overflow, and in Python the recursion depth is 1000. This section, we explore iterative tree traversals corresponding to PreOrder, InOrder, and PostOrder tree traversal. We know that the recursion is implemented implicitly with call stack, therefore in our iterative counterparts, they all use an explicit stack data structure to mimic the recursive behavior.

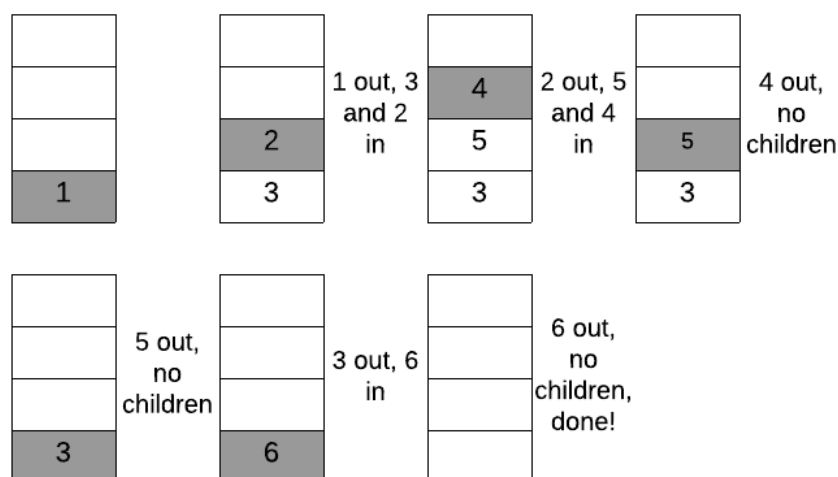


Figure 15: The process of iterative preorder tree traversal.

Simple Iterative Preorder Traversal If we know how to implement a DFS iteratively with stack in a graph, we know our iterative preorder traversal. In this version, the stack saves all our frontier nodes.

- At first, we start from the root, and put it into the stack, which is 1 in our example.
- Our frontier set has only one node, thus we have to pop out node 1 and expand the frontiner set. When we are expanding node 1, we add its children into the frontier set by pushing them into the stack. In

the preorder traversal, the left child should be first expanded from the frontier stack, indicating we should push the left child into the stack afterward the right child is pushed into. Therefore, we add node 3 and 2 into the stack.

- We continue step 2. Each time, we expand the frontier stack by pushing the toppest node's children into the stack and after popping out this node. This way, we use the first come last ordering of the stack data structure to replace the recursion.

We illustrate this process in Fig. 15. The code is shown as:

```

1 def PreOrderIterative(root):
2     if root is None:
3         return []
4     res = []
5     stack = [root]
6     while stack:
7         tmp = stack.pop()
8         res.append(tmp.val)
9         if tmp.right:
10            stack.append(tmp.right)
11        if tmp.left:
12            stack.append(tmp.left)
13    return res

```

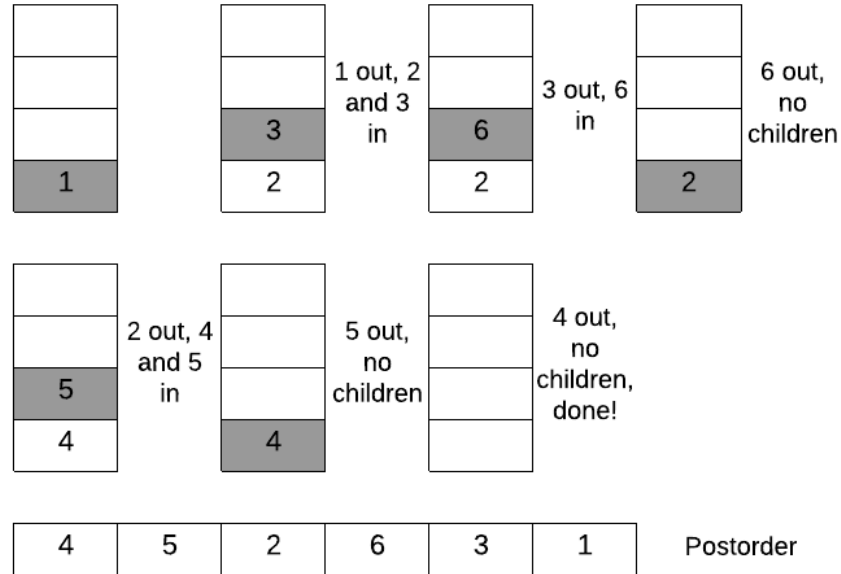


Figure 16: The process of iterative postorder tree traversal.

Simple Iterative Postorder Traversal Similar to the above preorder traversal, the postordering is the ordering of nodes finishing the expanding of both its left and right subtree, thus with the ordering of **left subtree**, **right subtree**, and **root**. In preorder traversal, we obtained the ordering of **root**, **left subtree**, and **right subtree**. We try to reverse the ordering, it becomes **right subtree**, **left subtree**, and **root**. This ordering only differs with postorder by a single a swap between the left and right subtree. So, we can use the same process as in the preorder traversal but expanding a node's children in the order of left and right child instead of right and left. And then the reversed ordering of items being popped out is the postorder traversal ordering. The process is shown in Fig. 16. The Python implementation is shown as:

```

1 def PostOrderIterative(root):
2     if root is None:
3         return []
4     res = []
5     stack = [root]
6     while stack:
7         tmp = stack.pop()
8         res.append(tmp.val)
9         if tmp.left:
10            stack.append(tmp.left)
11        if tmp.right:
12            stack.append(tmp.right)
13    return res[::-1]

```

General Iterative Preorder and Inorder Traversal In the depth-first-traversal, we always branch down via the left child of the node at the deepest level in the frontier. The branching only stops when it can no longer find a left child for the deepest node in the frontier. Only till then, it will look around at expanding the right child of this deepest node, and if no such right child exists, it backtracks to its parents node and continues to check its right child to continue the branching down process.

Inspired by this process, we use a pointer, say **cur** to point to the root node of the tree, and we prepare an empty **stack**. The iterative process is:

- The branching down process can be implemented with visiting **cur** node, and pushing it into the **stack**. And then we set **cur=cur.left**, so that it keeps deepening down.
- When one branch down process terminates, we pop out a node from **stack**, and we set **cur=node.right**, so that we expand the branching process to its right sibling.

We illustrate this process in Fig. 17. The ordering of items pushed into the stack is the preorder traversal ordering, which is [1, 2, 4, 5, 3, 6]. And

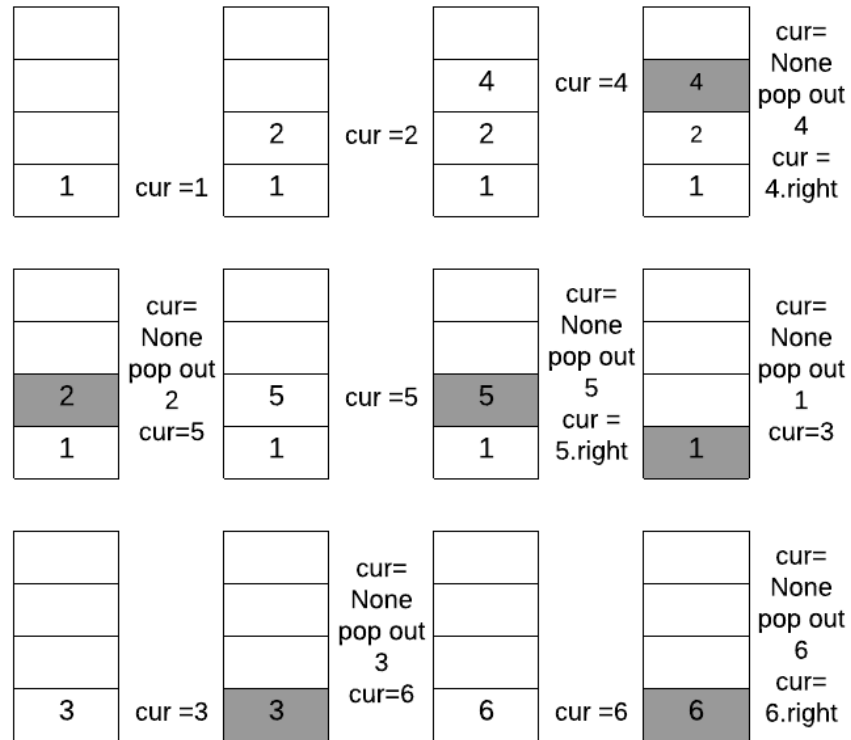


Figure 17: The process of iterative tree traversal.

the ordering of items being popped out of the stack is the inorder traversal ordering, which is [4, 2, 5, 1, 3, 6].

Implementation We use two lists—`preorders` and `inorders`—to save the traversal orders. The Python code is:

```

1 def iterative_traversal(root):
2     stack = []
3     cur = root
4     preorders = []
5     inorders = []
6     while stack or cur:
7         while cur:
8             preorders.append(cur.val)
9             stack.append(cur)
10            cur = cur.left
11        node = stack.pop()
12        inorders.append(node.val)
13        cur = node.right
14    return preorders, inorders

```

0.4.3 Breath-first Tree Traversal

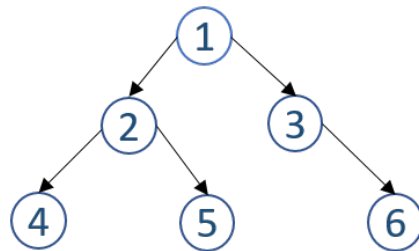


Figure 18: Draw the breath-first traversal order

Instead of traversing the tree recursively deepening down each time, the alternative is to visit nodes level by level, as illustrated in Fig. ?? for our exemplary binary tree. We first visit the root node 1, and then its children 2 and 3. Next, we visit 2 and 3's children in order, we go to node 4, 5, and 6. This type of Level Order Tree Traversal uses the **breath-first search strategy** which differs from our covered depth-first search strategy. As we see in the example, the root node is expanded first, then all successors of the root node are expanded next, and so on, following a level by level ordering. We can also find the rule, the nodes first come and get first expanded. For example 2 is first visited and then 3, thus we expand 2's children first. Then we have 4 and 5. Next, we expand 3's children. This First come first expanded tells us we can rely on a queue to implement BFS.

Simple Implementation We start from the root, say it is our first level, put it in a list named `nodes_same_level`. Then we use a `while` loop, and each loop we visit all children nodes of `nodes_same_level` from the last level. We put all these children in a temporary list `temp`, before the loop ends, we assign `temp` to `nodes_same_level`, until the deepest level where no more children nodes will be found and leave our `temp` list to be empty and our `while` loop terminates.

```

1 def LevelOrder(root):
2     if not root:
3         return
4     nodes_same_level = [root]
5     while nodes_same_level:
6         temp = []
7         for n in nodes_same_level:
8             print(n.val, end=' ')
9             if n.left:
10                temp.append(n.left)
11            if n.right:
12                temp.append(n.right)
13        nodes_same_level = temp
  
```

The above will output follows with our exemplary binary tree:

```
1 1 2 3 4 5 6
```

Implementation with Queue As we discussed, we can use a FIFO queue to save the nodes waiting for expanding. In this case, at each **while** we only handle one node that are at the front of the queue.

```
1 def bfs(root):
2     if not root:
3         return
4     q = [root]
5     while q:
6         node = q.pop(0) # get node at the front of the queue
7         print(node.val, end=' ')
8         if node.left:
9             q.append(node.left)
10        if node.right:
11            q.append(node.right)
```

0.5 Informed Search Strategies**

0.5.1 Best-first Search

Best-first search is a search algorithm which explores a graph by expanding the most promising node chosen according to a specified rule. The degree of promising of a node is described by a **heuristic evaluation function** $f(n)$ which, in general, may depend on the description of the node n , the description of the goal, and the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain.

Breath-first search fits as a special case in Best-first search if the objective of the problem is to find the shortest path from source to other nodes in the graph; it uses the estimated distance to source as a heuristic function. At the start, the only node in the frontier set is the source node, expand this node and add all of its unexplored neighboring nodes in the frontier set and each comes with distance 1. Now, among all nodes in the frontier set, choose the node that is the most promising to expand. In this case, since they all have the same distance, expand any of them is good. Next, we would add nodes that have $f(n) = 2$ in the frontier set, choose any one that has smaller distance.

A Generic best-first search will need a priority queue to implement instead of a FIFO queue used in the breath-first search.

0.5.2 Hands-on Examples

Get a more straightforward example

Add an example

Triangle (L120)

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

Example:

Given the following triangle:

```
[
  [2],
  [3,4],
  [6,5,7],
  [4,1,8,3]
]
```

The minimum path sum from top to bottom is 11 (i.e., $2 + 3 + 5 + 1 = 11$).

Analysis Solution: first we can use dfs traverse as required in the problem, and use a global variable to save the minimum value. The time complexity for this is $O(2^n)$. When we try to submit this code, we get LTE error. The code is as follows:

```
1 import sys
2 def min_path_sum(t):
3     '''
4     Purely Complete Search
5     '''
6     min_sum = sys.maxsize
7     def dfs(i, j, cur_sum):
8         nonlocal min_sum
9         # edge case
10        if i == len(t) or j == len(t[i]):
11            # gather the sum
12            min_sum = min(min_sum, cur_sum)
13            return
14        # only two edges/ choices at this step
15        dfs(i+1, j, cur_sum + t[i][j])
16        dfs(i+1, j+1, cur_sum + t[i][j])
17    dfs(0, 0, 0)
18    return min_sum
```

0.6 Exercises

0.6.1 Coding Practice

Property of Graph

1. 785. Is Graph Bipartite? (medium)
2. 261. Graph Valid Tree (medium)
3. 797. All Paths From Source to Target(medium)