Non-linear Recursive Backtracking

Li Yin[1]

April 5, 2019

[1]www.liyinscience.com

Non-linear Recursive Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, that *incrementally* builds candidates to the solutions. As soon as it determines that a candidate cannot possibly lead to a valid *complete solution*, it abandons this *partial candidate* and "backtracks" (return to the upper level) and reset to the upper level's state so that the search process can continue to explore the next branch. Backtracking is all about choices and consequences, this is why backtracking is the most common algorithm for solving *constraint satisfaction problem (CSP)*[1], such as Eight Queens puzzle, Map Coloring problem, Sudoku, Crosswords, and many other logic puzzles.

Properties and Applications To generalize the characters of backtracking:

1. **No Repetition and Completion:** It is a systematic generating method that avoids repetitions and missing any possible right solution. This property makes it ideal for solving combinatorial problems such as combination and permutation which requires us to enumerate all possible solutions.

2. **Search Pruning:** Because the final solution is built incrementally, in the process of working with partial solutions, we can evaluate the partial solution and prune branches that would never lead to the acceptable complete solution: either it is invalid configuration, or it is worse than known possible complete solution.

In this chapter, the organization is as follows:

1. Show Property 1: We will first show how backtrack construct the complete solution incrementally and how it backtrack to its previous state.

[1] CSPs are mathematical questions defined as a set of objects whose state must satisfy a number of constraints or limitations, visit `https://en.wikipedia.org/wiki/Constraint_satisfaction_problem` for more information

(a) **On Implicit Graph:** start with the combination and permutation problem to show us how the backtracking works in Section **??**.

(b) **On Explicit Graph:** Enumerating all pahts between the source and target vertex in a graph drawing in Section **??**.

2. Show Property 2: we demonstrate the application of search pruning in backtracking through CSP problems in Section **??**.

0.1 Enumeration

0.1.1 Permutation

Before I throw you more theoretical talking, let us look at an example: Given a set of integers 1, 2, 3, enumerate all possible permutations using all items from the set without repetition. A permutation describes an arrangement or ordering of items. It is trivial to figure out that we can have the following six permutations: $[1, 2, 3]$, $[1, 3, 2]$, $[2, 1, 3]$, $[2, 3, 1]$, $[3, 1, 2]$, and $[3, 2, 1]$.

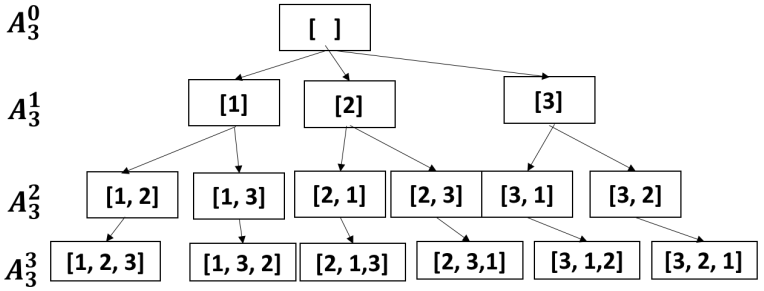

Figure 1: The state transfer graph of Permutation

This is a typical combinatorial problem, the process of generating all valid permutations is visualized in Fig. 1. To construct the final solution, we can start from an empty ordering shown at the first level, []. Then we try to add one item where we have three choices :1, 2, and 3. We get three partial solutions $[1], [2], [3]$ at the second level. Next, for each of these partial solutions, we have two choices, for [1], we can either put 2 or 3 first. Similarly, for [2], we can do either 1 and 3, and so on. Given n distinct items, the number of possible permutations are $n * (n - 1) * ... * 1 = n!$.

Implicit Graph In the graph, each node is either a partial or final solution. If we look it as a tree, the internal node is a partial solution and all leaves are final solutions. One edge represents generating the next solution based on the current solution. The vertices and edges are not given by an

explicitly defined graph or trees, the vertices are generated on the fly and the edges are implicit relation between these nodes.

Backtracking and DFS The implementation of the state transfer we can use either BFS or DFS on the implicit vertices. DFS is preferred because theoretically it took $O(\log n!)$ space used by stack, while if use BFS, the number of vertices saved in the queue can be close to $n!$. With recursive DFS, we can start from node [], and traverse to [1,2], then [1,2,3]. Then we backtrack to [1,2], backtrack to [1], and go to [1, 3], to [1, 3, 2]. To clear the relation between backtracking and DFS, we can say backtracking is a complete search technique and DFS is an ideal way to implement it.

We can generalize Permutation, Permutations refer to the permutation of n things taken k at a time without repetition, the math formula is $A_n^k = n*(n-1)*(n-2)*...*k$. In Fig. 1, we can see from each level k shows all the solution of A_n^k. The generation of A_n^k is shown in the following Python Code.

```
def A_n_k(a, n, k, depth, used, curr, ans):
  '''
  Implement permutation of k items out  of n items
  depth: start from 0, and represent the depth of the search
  used: track what items are  in the partial solution from the
    set of n
  curr: the current partial solution
  ans: collect all the valide solutions
  '''
  if depth == k: #end condition
    ans.append(curr[::])
    return

  for i in range(n):
    if not used[i]:
      # generate the next solution from curr
      curr.append(a[i])
      used[i] = True
      # move to the next solution
      A_n_k(a, n, k, depth+1, used, curr, ans)

      #backtrack to previous partial state
      curr.pop()
      used[i] = False
  return
```

Give the input of `a=[1, 2, 3]`, we call the above function with the following code:

```
a = [1, 2, 3]
n = len(a)
ans = [[None]]
used = [False] * len(a)
ans = []
```

```
6 A_n_k(a, n, n, 0, used, [], ans)
7 print(ans)
```

The output is:

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2,
    1]]
```

```
1  [1]
2  [1, 2]
3  [1, 2, 3]
4  backtrack:  [1, 2]
5  backtrack:  [1]
6  [1, 3]
7  [1, 3, 2]
8  backtrack:  [1, 3]
9  backtrack:  [1]
10 backtrack:  []
11 [2]
12 [2, 1]
13 [2, 1, 3]
14 backtrack:  [2, 1]
15 backtrack:  [2]
16 [2, 3]
17 [2, 3, 1]
18 backtrack:  [2, 3]
19 backtrack:  [2]
20 backtrack:  []
21 [3]
22 [3, 1]
23 [3, 1, 2]
24 backtrack:  [3, 1]
25 backtrack:  [3]
26 [3, 2]
27 [3, 2, 1]
28 backtrack:  [3, 2]
29 backtrack:  [3]
30 backtrack:  []
```

To notice, in line `10` of `A_n_k` we used deep copy `curr[::]`. Because if not, we need up getting all empty list due to the fact that `curr` is used to track all vertices, when it backtrack to the root node, it eventually become [].

Two Passes Therefore, we can say backtrack visit these implicit vertices in two passes: First forward pass to build the solution incrementally, second backward pass to backtrack to previous state. We can see within these two passes, the curr list is used as all vertices, and it start with [] and end with []. This is the character of backtracking.

Time Complexity of Permutation In the example of permutation, we can see that backtracking only visit each state once. The complexity of this is similar to the graph traveral of $O(|V|+|E|)$, where $|V| = \sum_{i=0}^{n} A_n^k$, because

it is a tree structure, $|E| = |v| - 1$. This actually makes the permutation problem NP-hard.

0.1.2 Combination

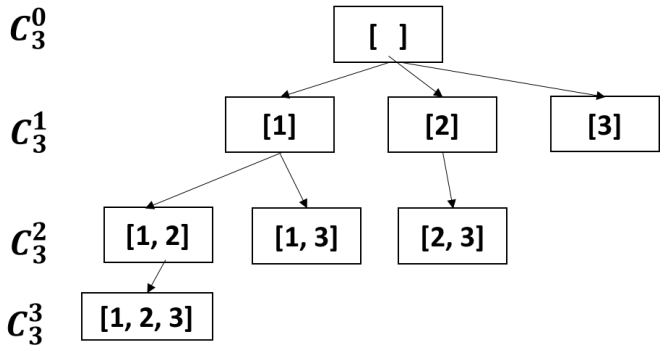

Figure 2: The state transfer graph of Combination

Continue from the permuation, combination refers to the combination of n things taken k at a time without repetition, the math formula C_n^k. Combination does not care about the ordering between chosen elements, such that [a, b] and [b, a] are considered as the same solution, thus only one of them can exist. Some useful formulation with combinations include $C_n^k = C_n^{n-k}$.

Here, we will demonstrate how we can use backtracking to implement an algorithm that just can traverse all the states and generate the power set (all possible subsets). We can see at each level, for each parent node, we would go through all of the elements in the array that has not be used before. For example, for [], we get [1], [2], [3]; for [1], we need [2], [3] to get [1, 2], [1, 3]. Thus we use an index in the designed function to denote the start position for the elements to be combined. Also, we use DFS, which means the path is []->[1]->[1, 2]->[1, 2, 3], we would use recursive function because it is easier to implement and also we can spare us from using a stack to save these nodes, which can be long and it would not really bring the benefit of using iterative implementation. The key point here is after the recursive function returns to the last level, say after [1, 2, 3] is generated, we would return to the previous state (this is why it is called backtrack!! Incremental and Backtrack).

Implementation The process is the similar to the implementation of permutation, excepth that we have one different variable, `start`. `start` is used to track the start index of the next candidate instead of use the `used` array to track the state of each item in the `curr` solution. `curr.pop()` is the soul for showing it is a backtracking algorithm!

```
def C_n_k(a, n, k, start, depth, curr, ans):
  '''
  Implement combination of k items out  of n items
  start: the start of candinate
  depth: start from 0, and represent the depth of the search
  curr: the current partial solution
  ans: collect all the valide solutions
  '''
  if depth == k: #end condition
    ans.append(curr[::])
    return

  for i in range(start, n):
    # generate the next solution from curr
    curr.append(a[i])
    # move to the next solution
    C_n_k(a, n, k, i+1, depth+1, curr, ans)

    #backtrack to previous partial state
    curr.pop()
  return
```

Similarly, we call the following code:

```
a = [1, 2, 3]
n = len(a)
ans = [[None]]
ans = []
C_n_k(a, n, 2, 0, 0, [], ans)
print(ans)
```

The output is:

```
[[1, 2], [1, 3], [2, 3]]
```

Try to modify the above code so that you can collect the power set.

Note: To generate the power set, backtracking is NOT the only solution, if you are interested right now, check out Section **??**.

Time Complexity of Combination Because backtracking ensures efficiency by visiting each state no more than once. For the combination(subset) problem, the total nodes of the implicit search graph/tree is $\sum_{k=0}^{n} C_n^k = 2^n$. We can look it as another way, there are in total n objects, and each object we can make two decisions: inside of the subset or not, therefore, this makes 2^n.

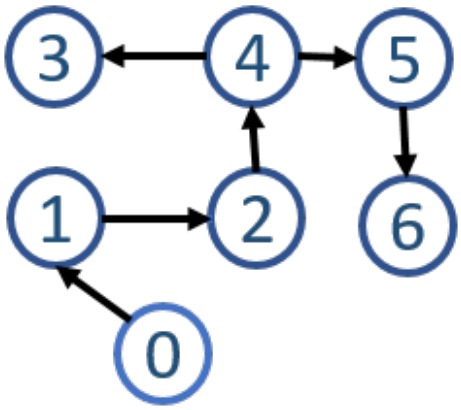

Figure 3: All paths from 0, include 0->1, 0->1->2,0->1->2->4, 0->1->2->4->3, 0->1->2->4->5, 0->1->2->4->5->6

0.1.3 All Paths

Backtracking technique can be naturally used in graph path traversal. One example is to find all possible paths from a source to the target. One simpler occasion is when the graph has no cycles. Backtrack technique can enumerate all paths in the graph exactly once for each.

The implementation is as follow: we still use dfs, because there has no cycles, we have no need to track the visiting state of each node. We generate the possible answer with backtracking technique through the `path` variable to track each state.

```
def all_paths(g, s, path, ans):
  '''generate all pahts with backtrack'''
  ans.append(path[::])
  for v in g[s]:
    path.append(v)
    print(path)
    all_paths(g, v, path, ans)
    path.pop()
    print(path)
```

Feed in the above network and run the following code:

```
al = [[1], [2], [4], [], [3, 5], [6], []]
ans = []
path = [0]
all_paths(al, 0, path, ans)
```

With the printing, we can see the whole process, `path` changes as the description of backtrack.

```
[0, 1]
[0, 1, 2]
```

```
[0, 1, 2, 4]
[0, 1, 2, 4, 3]
[0, 1, 2, 4] backtrack
[0, 1, 2, 4, 5]
[0, 1, 2, 4, 5, 6]
[0, 1, 2, 4, 5] backtrack
[0, 1, 2, 4] backtrack
[0, 1, 2] backtrack
[0, 1] backtrack
[0] backtrack
```

We can see each state, we can always have a matching backtrack state.

0.1.4 Analysis

Backtracking VS DFS The implementation of Backtracking is equivalent to a DFS on the implicit or explicit search space and visiting each vertex no more than once. Backtracking is a technique to build up solution spaces incrementally and each exactly only once. And once one path reaches to the end, it backtrack to its previous state and try another candidate. DFS is a natural way to implement backtarck technique.

Backtracking VS Exhaustive Search Backtracking helps in solving an overall problem by incrementally builds candidates, which is equivalent to finding a solution to the first sub-problem and then recursively attempting to resolve other sub-problems bases on the solution of the first sub-problem. Therefore, **Backtracking can be considered as a Divide-and-conquer method for exhaustive search.** Problems which are typically solved using backtracking technique have such property in common: they can only be solved by trying every possible configuration and each configuration is tried only once(every node one time). A Naive exhaustive search solution is to generate all configurations and "pick" a configuration that follows given problem constraints. Backtracking however works in incremental way and **prunes** branches that can not give a result. It is an optimization over the exhaustive search where all possible(possible still with constraints) configurations are generated and tried. This comparison is called named as **Generating VS Filtering**.

Visualize Backtracking A backtracking algorithm can be thought of as a *tree of possibilities*. In this tree, the root node is our starting point, we traverse the possible choices for the next step (all the children nodes). If we reach to end condition (leaf candidates) we succeed and the DFS search stop. If the partial candidate can not satisfy the constraint, we return to the root node, and reset the state ('backtrack'). The process is shown in Fig. 4.

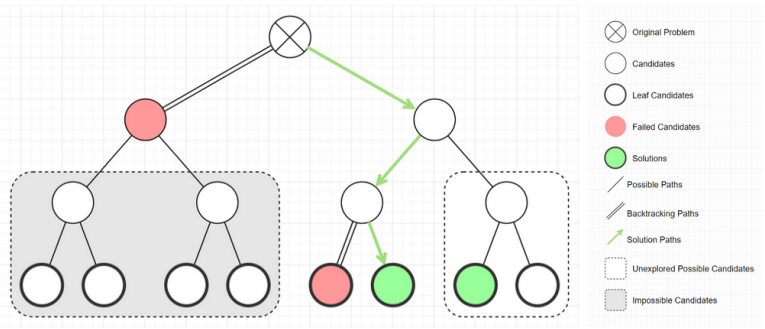

Figure 4: Tree of possibilities for a typical backtracking algorithm

Applications Backtracking can be applied where we can incrementally build up our final soultion from partial solution. It is a searching technique that applied on implicit graph which is built on-the-fly. It guarentees that it only visit each search vertex no more than once. The problems that backtracking can be used are these three types: (1) Combinations (Section 0.1.2), (2) Permutations (Section **??**), (3) enumerate all paths from a to b in graph. and (3) Optimization problems with constraints such as classical travels salesman, puzzles, and sudoku (Section 0.2). In the first three cases, backtracking visit each implicit or explicit vertex exactly once in the searching space. And for problems with restraints, we can do search pruning and we end up amortizely visiting each vertex less than once which is more efficient compared with an exhaustive graph search such as DFS and BFS.

0.2 Constraint Satisfaction Problems with Search Pruning

In previous sections, we have learned how backtracking can be applied to enumerating based combinatorial tasks such combination, permutation, and all paths in graph. In this section, we state how backtracking can be optimized with search prunning in CSP.

0.2.1 Sudoku

We will start with a sudoku problem due to its popularity in magazines, and we will discuss classical eight queen problem, traveling salesman probelm (TSP), and we leave exercises to such problems such as puzzles, sudoku and so on.

Problem Definition Given a, possibly, partially filled grid of size $n \times n$, completely fill the grid with number between 1 and n. The constraint is defined as:

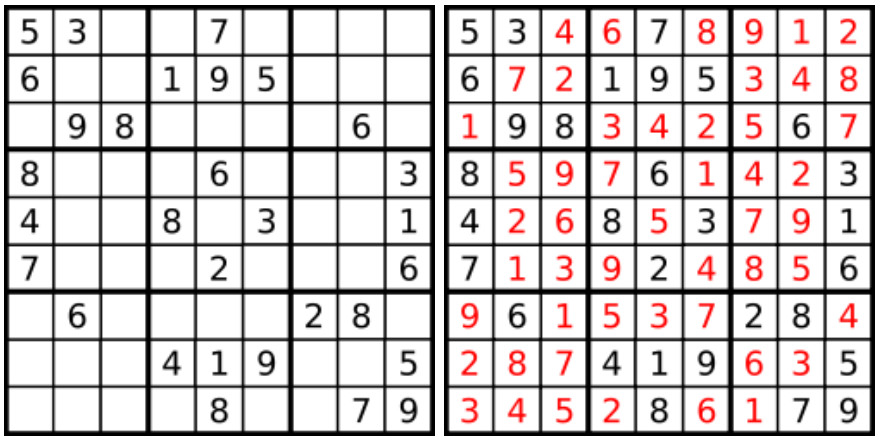

Figure 5: Example sudoko puzzle and its solution

1. Each row has all numbers form 1 to 'n'.
2. Each column has all numbers form 1 to 'n'.
3. Each sub-grid ($\sqrt{n} \times \sqrt{n}$) has all numbers form 1 to n.

Only all constraint are satisfied can we have a valid candidate. How many possible candidates here? Suppose we have an empty table, the brute force is to try 1 to n at each grid, we have possible solution space of n^{n^2}. How many of them are valid solutions? We can get closer by permutating numbers from 1 to 9 at each row, with $9!^9$ possible search space. This is already a lot better than the first. How to know the exact possible solutions? This site `http://pi.math.cornell.edu/~mec/Summer2009/Mahmood/Count.html` demonstrates that the actual $N = 6670903752021072936960$ which is approximately 6.671×10^{21} possible solutions. This shows that sudo problem is actually NP-hard problem.

Solving Sudoku with Backtracking Now, let us see how we can use backtrack and search prunning to implement a sudoku solver.

Step 1: Get Empty Spots and Record Initial States Assuming we scan the whole grid shown in Fig. 5 and find all empty spots that waiting for filling in. From the constraints, we use the following data structures to record the state of each row, each column, and each grid.

```
row_state = [0]*9
col_state = [0]*9
grid_state = [0]*9
```

We use (i,j) to denote the position of a grid. It correspond position i in `row_state`, and j in `col_state`, and `(i//3)*3 + (j//3)` for corresponding sub-grid. We see 5 we need to set `row_state[0]`, `col_state[0]` and

`grid_state[0]`. We treat each state as a serial of bits of maximum length of 9. For 5, we set the 5th bit into 1 for each state, which is through XOR with mask that left shift 1 by 4 digits. The details can be found in Chapter**??**. To check if 5 is in the row, or column or current grid we just need to get corresponding bit value, if it is one or not. Through the bit manipulation, we can use 27 extra spaces and able to check if a number if valid here in $O(1)$ time. Therefore, for each empty spot, we can find all possible values. The functions used to set/reset/check state is implemented as follows:

```
def setState(i, j, v, row_state, col_state, grid_state):
  row_state[i] |= 1 << v
  col_state[j] |= 1 << v
  grid_index = (i//3)*3 + (j//3)
  grid_state[grid_index] |= 1 << v

def resetState(i, j, v, row_state, col_state, grid_state):
  row_state[i] &= ~(1 << v)
  col_state[j] &= ~(1 << v)
  grid_state[grid_index] &= ~(1 << v)

def checkState(i, j, v, row_state, col_state, grid_state):
  row_bit = (1 << v) & row_state[i] != 0
  col_bit = (1 << v) & col_state[j]  != 0
  grid_index = (i//3)*3 + (j//3)
  grid_bit = (1 << v) & grid_state[grid_index]  != 0
  return not row_bit and not col_bit and not grid_bit
```

The following function is implement to find empty spots and state record.

```
def getEmptySpots(board, rows, cols, row_state, col_state,
    grid_state):
    ''' get empty spots and find its corresponding values in O(n
    *n)'''
    empty_spots = {}
    # initialize the state, and get empty spots
    for i in range(rows):
      for j in range(cols):
        if board[i][j]:
            # set that bit to 1
            setState(i, j, board[i][j]-1, row_state, col_state,
    grid_state)
        else:
            empty_spots[(i,j)] = []

    # get possible values for each spot
    for i, j in empty_spots.keys():
      for v in range(9):
        if checkState(i, j, v, row_state, col_state, grid_state)
    :
          empty_spots[(i, j)].append(v+1)

    return empty_spots
```

Now, with our existing grid, we get the empty spots and sort them by the length of possible values.

Step 2: Fill In with Backtracking and Search Pruning Now, we iterate the empty spots with the sorted ordering. This is a human-inituitive way and also a more efficient way than not sorting it and keep them in the original order. The implicit graph is composed of empty spots as vertices. Now, we implement the backtracking technique in DFS.

```
def dfs_backtrack(empty_spots, index):
  if index == len(empty_spots):
    return True
  (i, j), vl = empty_spots[index]

  for v in vl: #try each value
    # check the state
    if checkState(i, j, v-1, row_state, col_state, grid_state):
      # set the state
      setState(i, j, v-1, row_state, col_state, grid_state)
      # mark the board
      board[i][j] = v
      if dfs_backtrack(empty_spots, index+1):
        return True
      else:
        #backtack to previouse state
        resetState(i, j, v-1, row_state, col_state, grid_state)
        #unmark the board
        board[i][j] = None
  return False
```

For the above example, we run the following code:

```
ans = dfs_backtrack(sorted_empty_spots, 0)
print(ans)
print(board)
```

And the board is:

```
[[5, 3, 4, 6, 7, 8, 9, 1, 2], [6, 7, 2, 1, 9, 5, 3, 4, 8], [1,
    9, 8, 3, 4, 2, 5, 6, 7], [8, 5, 9, 7, 6, 1, 4, 2, 3], [4, 2,
    6, 8, 5, 3, 7, 9, 1], [7, 1, 3, 9, 2, 4, 8, 5, 6], [9, 6, 1,
    5, 3, 7, 2, 8, 4], [2, 8, 7, 4, 1, 9, 6, 3, 5], [3, 4, 5, 2,
    8, 6, 1, 7, 9]]
```

The prunning appears when the `checkState` returns `False`. We prun braches that appears invalid already and only handle brachs that up till now are valid.

Experiment Let us do an experiment, with the same input of board, we track the time that we use the sorted or unsorted empty spots and see what is the time difference. The code is provided in colab. The time is 0.025 seconds for unsorted and 0.0005 seconds for sorted.

0.3 Summary

BFS and DFS are two of the most universal algorithms for solving practical problems. Each suits better than the other to specific type of problems.

- For BFS, it suits problems that ask the shortest paths(unweighted) from a certain source node to a certain destination, whether it is single sourced or all-pairs. Or in some cases, the questions requires us only traverse the graph for a certain steps (levels). Because BFS is iterative and can traverse the nodes level by level.
- For DFS, it is better for the weighted optimization problem, that we are required to count all possible paths or to get the best out of all possible paths. Because DFS has the advantage of saving the result of overlapping subproblem to avoid extra computation.
- Use either BFS or DFS when we just need to check correctness (whether we can reach a given state to another.

0.4 Bidirectional Search: Two-end BFS

Definition In normal graph search using BFS/DFS we begin our search in one direction usually from source vertex s toward the goal vertex t, but what if we start search form both direction simultaneously. Bidirectional search is a graph search algorithm which find *smallest path* from source to goal vertex. We just learned that Breadth-first-search suits well for shortest path problem. Because in Level-by-level BFS, it controls the visiting order of nodes by its order to the starting vertex. Therefore, Bidirectional search runs *two simultaneous level-by-level BFS searches* which eventually "meet in the middle" (when two searches intersect) and terminates.

1. Forward search starts form source/initial vertex s toward goal vertex t.
2. Backward search form goal/target vertex t toward source vertex s

Time and Space Complexity Suppose if branching factor of tree is b and distance of goal vertex from source is h, then the normal BFS/DFS searching complexity would be $O(b^h)$. On the other hand, if we execute two search operation then the complexity would be $O(b^{h/2})$ for each search and total complexity would be $O(b^{h/2} + b^{h/2})$ which is far less than $O(b^h)$. Therefore, in many cases bidirectional search is way faster and dramatically reduce the amount of required exploration. Because we need to save nodes at each level in the queue, the maximum nodes we get at the middle $h/2$ will be b^h, this makes the space complexity to $O(b^h)$.

When and How **When** Bidirectional search can find the shortest path successfully if all the paths are assigned uniform costs.

How When the graph from each side is not balanced: each node has various branch factors. We take two queues: sourceQueue for BFS in forward direction from source to target and targetQueue which is used to do the BFS from the target towards the source in backward direction. We try to alternate between the two queues: sourceQueue and targetQueue; basically in every iteration we choose the smaller queue for the next iteration for the processing which effectively helps in alternating between the two queues only when the swapping between the two queues is profitable.

This helps in the following way: As we go deeper into a graph the number of edges can grow exponentially. Since we are approaching in a balanced way, selecting the queue which has smaller number of nodes for the next iteration, we are avoiding processing a large number of edges and nodes by trying to having the intersection point somewhere in the middle.

Since we are processing both the target and source queue we are not going to much depth from any direction, either in forward direction (i.e, while processing source queue) or in backward direction (i.e, target queue which searches from target to source in backward manner).

Implementation

0.5 Bonus

Generating combinations, permutation uniformly at random.

0.6 Exercises

0.6.1 Knowledge Check

0.6.2 Coding Practice

Cycle Detection

1. 207. Course Schedule (medium)

Topological Sort

Connected Component

1. 323. Number of Connected Components in an Undirected Graph (medium).
2. 130. Surrounded Regions(medium)
3.

Mix

1. 210. Course Schedule II (medium, cycle detection and topological sort).

Backtracking

1. 77. Combinations

```
Given two integers n and k, return all possible
    combinations of k numbers out of 1 ... n.

Example:

Input: n = 4, k = 2
Output:
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

2. 17. Letter Combinations of a Phone Number

```
Given a digit string, return all possible letter
    combinations that the number could represent.

A mapping of digit to letters (just like on the telephone
    buttons) is given below.

Input:Digit string "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "
    cf"].

Note:
Although the above answer is in lexicographical order, your
     answer could be in any order you want.
```

3. 797. All Paths From Source to Target (medium).

4. **37. Sudoku Solver (hard).** Write a program to solve a Sudoku puzzle by filling the empty cells.

   A sudoku solution must satisfy all of the following rules:

   (a) Each of the digits 1-9 must occur exactly once in each row.

   (b) Each of the digits 1-9 must occur exactly once in each column.

   (c) Each of the the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

Empty cells are indicated by the character '.'.