Combinatorial Search

Li Yin[1]

March 29, 2020

[1]www.liyinscience.com

Combinatorial Search

So far, we have learned the most fundamental search strategies on general data structures such as array, linked list, graph, and tree. In this chapter, instead of searching on explicit and well defined data structures, we extend and discuss more *exhaustive search* algorithms that can solve rather obscure and challenging *combinatorial problems*, such as sudoku and the famous Travels Salesman Problem. For combinatorial problems, we have to figure out the potential search space, and rummage a solution.

0.1 Introduction

Combinatorial search problems consists of n items and a requirement to find a solution, i.e., a set of $L < N$ items that satisfy specified conditions or constraints. For example, a sudoku problem where a 9×9 grid is partially filled with number between 1 and 9, fill the empty spots with numbers that satisfy the following conditions:

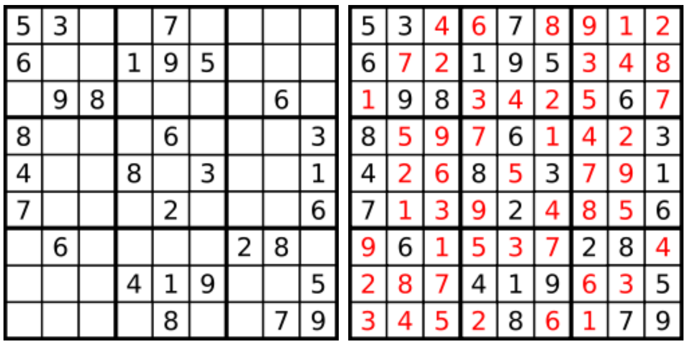


Figure 1: A Sudoku puzzle and its solution

1. Each row has all numbers form 1 to 9.

2. Each column has all numbers form 1 to 9.

3. Each sub-grid (3×3) has all numbers form 1 to 9.

This sudoku together with one possible solution is shown in Fig. 1. In this case, we have 81 items, and we are required to fill 51 empty items with the above three constraints.

Model Combinatorial Search Problems We can model the combinatorial search solution as a vector $s = (s_0, s_1, ..., s_{L-1})$, where each variable s_i is selected from a finite set A, which is called the *domain* for each variable. Such a vector might represent an arrangement where s_i contains the i-th item of a permutation, in the combination problem, a boolean denotes if the i-th item is selected already, or it can represent a path in a graph or a sequence of moves in a game. In the sudoku problem, each s_i can choose from a number in range $[1, 9]$.

Problem Categories Combinatorial search problems arise in many areas of computer science such as artificial intelligence, operations search, bioinformatics, and electronic commerce. These problems typically involve finding a *grouping*, *ordering*, or *assignment* of a discrete, finite set of objects that satisfy given conditions or constraints. We introduce two well-studied types of problems that are more likely to be NP-hard and of at least exponential complexity:

1. Constraint Satisfaction Problems (CSP) are mathematical questions defined as a set of variables whose state must satisfy a number of constraints or limitations(mathematical equations or inequations), such as Sudoku, N-queen, map coloring, Crosswords, and so on. The size of the search space of CSPs can be roughly given as:

$$O(cd^L) \tag{1}$$

   Where there are L variables, each with domain size d, and there are c constraints to check out.

2. Combinatorial optimization problems consist of searching for maxima or minima of an objective function F whose domain is a discrete but large configuration space. Some classic examples are:

   - Travelling Salesman Problems (TSP): given position (x, y) of n different cities, find the shortest possible path that visits each city exactly once.
   - Integer Linear Programming: maximize a specified linear combination of a set of integers $X_1, .., X_n$ subject to a set of linear constraints each of the form:

$$a_1X_1 + ... + a_nX_n \leq c \tag{2}$$

- Knapsack Problems: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Search Strategies From Chapter Discreet Programming, we have learned the basic enumerative combinatorics, including counting principles and knowledge on permutations, combinations, partitions, subsets, and subsequences. Combinatorial Search builds atop this subject, and together through different search strategies such as depth-first search and best-first search, it is able to enumerate the search space and find the solution(s) with necessary speedup methods. In this chapter, we only discuss about complete search and only acknowledge the existence of approximate search techniques.

Backtracking is a process of depth-first based search where it "builds" the search tree on the fly incrementally instead of having a tree/graph structure beforehand to search through. Backtracking fits to solve combinatorial search problems because:

1. It is space efficient for the usage of a DFS and the candidates are built incrementally and their validity to fit a solution is checked right away.

2. It is time efficient for that some partial candidates can be pruned if the algorithm believes that it will not lead to our final complete solution.

Because the ordering of variables $s_0, ..., s_{L-1}$ can potentially affect the size of the search space sometimes. Thus, backtracking search relies on one or more heuristics to select which variable to consider next. *Look-ahead* is one such heuristic that is preferably applied to check the effects of choosing a given variable to evaluate or to decide the order of values to give to it.

There are other Breath-first Search based strategies that might work better than backtracking, such as for combinatorial optimization problems, best-first branch and bound search might be more efficient than its depth-first counterpart.

Speedups The speedup methods are well studied in computer science, and we list two general ways to prune unqualified or unnecessary branches during the search of backtracking:

1. Branch and Prune: This method prunes the unqualified branches with constraints of the problems. This is usually applied to solve constraint restricted problems (CSPs).

2. Branch and Bound: This method prunes unnecessary branches via comparing an estimation of a partial candidate with a found global best solution. If the estimation states that the partial candidate will

never lead us to a better solution, we cut off this branch. This technique can be applied to solve a general optimization problems, such as Travel Salesman Problems (TSP), knapsack problems, and so.

0.2 Backtracking

In this section, we first introduce the technique of backtracking, and then demonstrate it by implementing common enumerative combinatorics seen in Chapter Discreet Programming.

0.2.1 Introduction

Backtracking search is an exhaustive search algorithm(depth-first search) that systematically assigns all possible combinations of values to the variables and checks if these assignments constitute a solution. Backtracking is all about choices and consequences and it shows the following two properties:

1. **No Repetition and Completion:** It is a systematic generating method that enumerates all possible states exactly at most once: it will not miss any valid solution but avoids repetitions. If there exists "correct" solution(s), it is guaranteed to be found. This property makes it ideal for solving combinatorial problems where the search space has to be constructed and enumerated. Therefore, the worst-case running time of backtracking search is exponential with the length of the state (b^L, b is the average choice for each variable in the state).

2. **Search Pruning:** Along the way of working with partial solutions, in some cases, it is possible for us to decide if they will lead to a valid *complete solution.* As soon as the algorithm is confident to say the partial configuration is either invalid or nonoptimal, it abandons this *partial candidate*, an then "backtracks" (return to the upper level), and resets to the upper level's state so that the search process can continue to explore the next branch for the sake of efficiency. This is called *search pruning* with which the algorithm ends up amortizely visiting each vertex less than once. This property makes backtracking the most promising way to solve CSPs and combinatorial optimization problems.

Solving sudoku problem with backtracking algorithm, each time at a level in the DFS, it tries to extend the last partial solution $s = (s_0, s_1, ..., s_k)$ by trying out all 9 numbers at s_{k+1}, say we choose 1 at this step. It testifies the partial solution with the desired solution:

1. If the partial solution $s = (s_0, s_1, ..., s_k, 1)$ is still valid, move on to the next level and work on trying out s_{k+2}.

2. If the partial solution is invalid and is impossible to lead to a complete solution, it "backtracks" to the last level and resets the state as $s = (s_0, s_1, ..., s_k)$ so that it can try our other choices if there are some left(which in our example, we will try $s_{k+1} = 2$) or keep "backtracking" to even upper level.

The process should be way clearer once we have learned the examples in the following subsections.

0.2.2 Permutations

Given a list of items, generate all possible permutations of these items. If the set has duplicated items, only enumerate all unique permutations.

No Duplicates(L46. Permutations)

When there are no duplicates, from Chapter Discreet Programming, we know the number of all permutations are:

$$p(n, m) = \frac{n!}{(n-m)!} \tag{3}$$

where m is the number of items we choose from the total n items to make the permutations.

```
For example:
a = [1, 2, 3]
There are 6 total permutations:
[1, 2, 3], [1, 3, 2],
[2, 1, 3], [2, 3, 1],
[3, 1, 2], [3, 2, 1]
```

Analysis Let us apply the philosophy of backtracking technique. We have to build a state with length 3, and each variable in the state has three choices: 1, 2, and 3. The constraint here comes from permutation which requires that no two variables in the state will be having the same value. To build this incrementally with backtracking, we state with an empty state `[]`. At first, we have three options, we get three partial results [1], [2], and [3]. Next, we handle the second variable in the state: for [1], we can choose either 2 or 3,getting $[1, 2]$ and $[1, 3]$; same for [2], where we end up with $[2, 1]$ and $[2, 3]$; for [3], we have $[3, 1]$ and $[3, 2]$. At last, each partial result has only one option, we get all permutations as shown in the example. We visualize this incrementally building candidates in Fig. 2.

However, we only managed to enumerate the search space, but not systematically or recursively with the Depth-first search process. With DFS, we depict the traverse order of the vertexes in the virtual search space with red arrows in Fig. 2. The backward arrows mark the "backtracking" process, where we have to reset the state to the upper level.

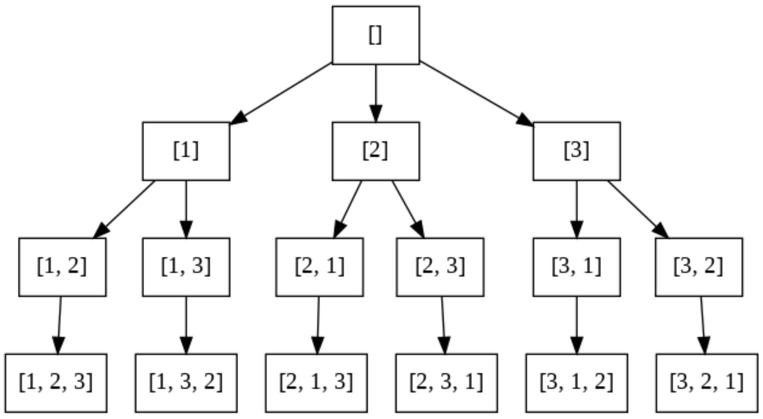


Figure 2: The search tree of permutation

Implementation We use a list of boolean `bUsed` to track which item is used in the search process. `n` is the total number of items, `d` is the depth of the depth-first search process, `curr` is the current state, and `ans` is to save all permutations. The following code, we generate $p(n, m)$

```
def p_n_m(a, n, m, d, used, curr, ans):
  if d == m: #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
      print(curr)
      # move to the next solution
      p_n_m(a, n, m, d + 1, used, curr, ans)
      #backtrack to previous partial state
      curr.pop()
      used[i] = False
  return
```

Check out the running process in the source code.

Alternative: Swapping Method We first start with a complete state, such that $s = [1, 2, 3]$ in our case. By swapping 1 and 2, we get $[2, 1, 3]$ and $[2, 3, 1]$ can be obtained by swapping 1 and 3 on top of $[2, 1, 3]$. With all permutations as leaves in the search space, the generating process is similar to Fig. 2. We show this alternative process in Fig. 3. At first, we swap index 0 with all other indexes, including 0, 1, and 2. At the second layer, we move on to swap index 1 with all other successive indexes, and so on for all other layers. The Python code is as:

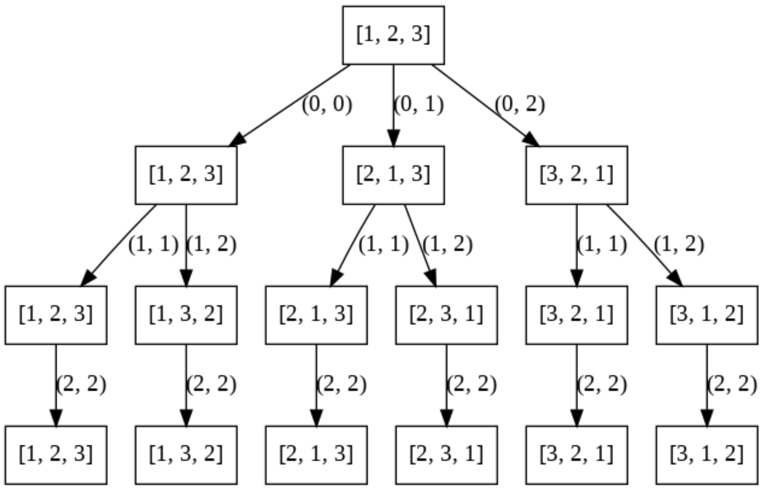


Figure 3: The search tree of permutation by swapping. The indexes of items to be swapped are represented as a two element tuple.

```python
ans = []
def permutate(a, d):
  global ans
  if d == len(a):
    ans.append(a[::])
  for i in range(d, len(a)):
    a[i], a[d] = a[d], a[i]
    permutate(a, d+1)
    a[i], a[d] = a[d], a[i]
  return
```

There is Johnson-Trotter algorithm that utilizes such swapping method, which avoids recursion, and instead computes the permutations by an iterative method.

With Duplicates(47. Permutations II)

We have already know that $p(n, n)$ is further decided by the duplicates within the n items. Assume we have in total of d items are repeated, and each item is repeated x_i times, then the number of all arrangements $pd(n, n)$ are:

$$pd(n,n) = \frac{p(n,n)}{x_0!x_1!...x_{d-1}}, \tag{4}$$

$$\texttt{w.r.t} \sum_{i=0}^{d-1} x_i \leq n \tag{5}$$

For example, when $a = [1, 2, 2, 3]$, there are $\frac{4!}{2!}$ unique permutations, which is 12 in total, and are listed as bellow:

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

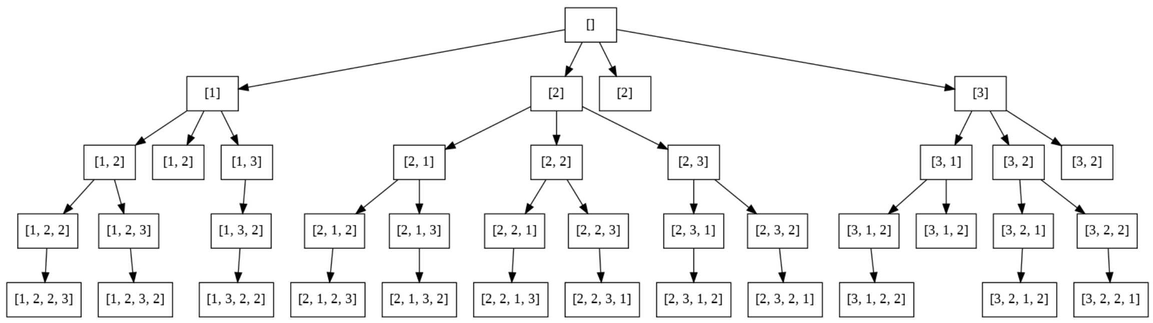


Figure 4: The search tree of permutation with repetition

Analysis The enumeration of these all possible permutations can be obtained with backtracking exactly the same as if there are no duplicates. However, this is not efficient since it has doubled the search space with repeated permutations. Here comes to our first time applying the Branch and Prune method: we avoid repetition by pruning off redundant branches.

One main advantage of backtracking is not to save all intermediate states, thus we should find a mechanism that avoids generating these intermediate states at the first place. One solution is that we sort all n items, making all repeat items adjacent to each other. We know if the current intermediate state is redundant by simply comparing this item with its predecessor: if it equals, we move on from building state with this item to the next item in line. The search tree of our example is shown in Fig. 4.

Implementation The implementation is highly similar to previous standard permutation code other than three different points:

1. Before the items are called by `permutate`, they are sorted first.
2. A simple condition check to avoid generating repeat states.
3. We used a dictionary data structure `tracker` which has all unique items as keys and each item's corresponding occurrence as values to replace the boolean vector `used` for slightly better space efficiency.

The Python code is as:

```python
from collections import Counter
def permuteDup(nums, k):
    ans = []
    def permutate(d, n, k, curr, tracker):
      nonlocal ans
      if d == k:
          ans.append(curr)
          return
      for i in range(n):
          if tracker[nums[i]] == 0:
            continue
          if i - 1 >= 0 and nums[i] == nums[i-1]:
              continue
          tracker[nums[i]] -= 1
          curr.append(nums[i])

          permutate(d+1, n, k, curr[:], tracker)
          curr.pop()
          tracker[nums[i]] += 1
      return

    nums.sort()
    permutate(0, len(nums), k, [], Counter(nums))
    return ans
```

Can you extend the swap method based permutation to handle duplicates?

Discussion

From the example of permutation, we have demonstrated how backtracking works to construct candidates with an implicit search tree structure: the root node is the initial state, any internal node represents intermediate states, and all leaves are our candidates which in this case there are $n!$ for $p(n, n)$ permutation. In this subsection, we want to point out the unique properties and its computational and space complexities.

Two Passes Backtracking builds an implicit search tree on the fly, and it does not memorize any intermediate state. It visits the vertices in the search tree in two passes:

1. Forward pass: it builds the solution incrementally and reaches to the leaf nodes in a DFS fashion. One example of forward pass is $[]->[1]->[1,2]->[1,2,3]$.

2. Backward pass: as the returning process from recursion of DFS, it also backtracks to previous state. One example of backward pass is $[1,2,3]->[1,2],->[1]$.

First, the forward pass to build the solution **incrementally**. The change of `curr` in the source code indicates all vertices and the process of backtracking, it starts with [] and end with []. This is the core character of backtracking. We print out the process for the example as:

```
[]->[1]->[1, 2]->[1, 2, 3]->backtrack: [1, 2]
backtrack: [1]
[1, 3]->[1, 3, 2]->backtrack: [1, 3]
backtrack: [1]
backtrack: []
[2]->[2, 1]->[2, 1, 3]->backtrack: [2, 1]
backtrack: [2]
[2, 3]->[2, 3, 1]->backtrack: [2, 3]
backtrack: [2]
backtrack: []
[3]->[3, 1]->[3, 1, 2]->backtrack: [3, 1]
backtrack: [3]
[3, 2]->[3, 2, 1]->backtrack: [3, 2]
backtrack: [3]
backtrack: []
```

Time Complexity of Permutation In the search tree of permutation in Fig. 2, there are in total V nodes, which equals to $\sum_{i=0}^{n} p_n^k$. Because in a tree the number of edges $|E|$ is $|v|-1$, making the time complexity $O(|V|+|E|)$ the same as of $O(|V|)$. Since $p(n,n)$ itself alone takes $n!$ time, making the permutation an NP-hard problem.

Space Complexity A standard depth-first search consumes $O(bd)$ space in worst-case to execute, where b is branching factor and d is the depth of the search tree. In the combinatorial search problems, usually depth and branching is decided by the total number of variables in the state, making $b \sim d \sim n$. In backtracking, we have space complexity $O(n^2)$. However, in normal standard DFS, the input–tree or graph data structure–is given and not attributed to space complexity. For a NP-hard combinatorial search problem, this input is often exponential. Backtracking search outcompetes the standard DFS by avoiding such space consumption; it only keeps a dynamic data structure(`curr`) to construct node on the fly.

0.2.3 Combinations

Given a list of n items, generate all possible combinations of these items. If the input has duplicated items, only enumerate unique combinations.

No Duplicates (L78. Subsets)

From Chapter Discrete Programming, we list the powerset–all m-subset, $m \in [0, n]$ as:

$$C(n,m) = \frac{P(n,m)}{P(m,m)} = \frac{n!}{(n-m)!m!} \tag{6}$$

For example, when $a = [1, 2, 3]$, there are in total 7 m-subsets, they are:

```
C(3, 0): []
C(3, 1): [1], [2], [3]
C(3, 2): [1, 2], [1, 3], [2, 3]
C(3, 3): [1, 2, 3]
```

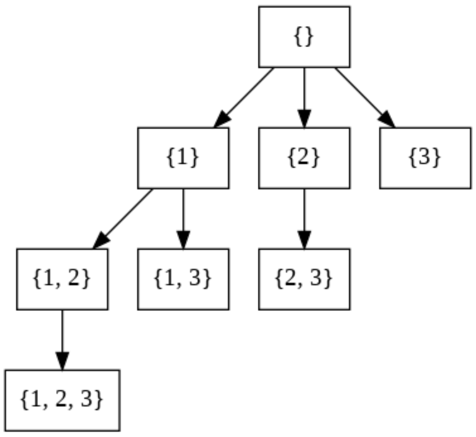


Figure 5: The Search Tree of Combination.

Analysis We can simply reuse the method of permutation, but with a problem that it generates lots of duplicates. For example, $P(3, 2)$ includes $[1, 2]$ and $[2, 1]$ which are indeed the same subset. Of course, we can check redundancy with saved m-subsets, but its not ideal. A systematical solution that avoids duplicates all along is preferred. If we limit the items we put into the m-subsets to be only increasing(of indexes of items or of values of items), in which case $[2, 1]$, $[3, 1]$, and $[3, 2]$ will never be generated. The enumeration of combination through backtracking search is shown in Fig. 5.

Implementation Two modifications based on permutation code:

1. `for` loop: in the loop to iterate all possible candidates, we limit the candidates to be having larger indexes only.

2. We do not have to use a data structure to track the state of each candidate because any candidate that has larger index is a valid candidate.

We use `start` to track the starting position of valid candidates. The code of combination is:

```
def C_n_k(a, n, k, start, d, curr, ans):
  if d == k: #end condition
    ans.append(curr[::])
    return

  for i in range(start, n):
    curr.append(a[i])
    C_n_k(a, n, k, i+1, d+1, curr, ans)
    curr.pop()
  return
```

Alternative: 0 and 1 Selection We have discussed that a powerset written as $P(S)$. With each item either being appear or not appear in the resulting set makes the value set $\{0, 1\}$, resulting $|P(S)| = 2^n$. Follow this pattern, with our given example, we can alternatively generate a powerset like this:

```
s   sets
1   {1}, {}
2   {1,2}, {1}, {2}, {}
3   {1,2,3}, {1,2}, {1, 3}, {3}, {2, 3}, {2}, {3}, {}
```

This process can be better visualized in a tree as in Fig. **??**. We can see this process results 2^n leaves compared with our previous implementation which has a total of 2^n nodes is slightly less efficient. The code is as:

```
def powerset(a, n, d, curr, ans):
  if d == n:
    ans.append(curr[::])
    return

  # Case 1: select item
  curr.append(a[d])
  powerset(a, n, d + 1, curr, ans)
  # Case 2: not select item
  curr.pop()
  powerset(a, n, d + 1, curr, ans)
  return
```

Time Complexity The total nodes within the implicit search space of combination shown in Fig. 5 is $\sum_{k=0}^{n} C_n^k = 2^n$, which was explained in Chapter Discreet Programming. Thus, the time complexity of enumerating the powset is $O(2^n)$ and is less compared with $O(n!)$ that comes with the permutation.

Space Complexity Similarly, combination with backtracking search uses slightly less space. But, we can still acclaim the upper bound to be $O(n^2)$.

With Duplicates(L90. Subsets II)

Assume we have m unqiue items, and the frequency of each is marked as x_i, with $\sum_{i=0}^{m-1} x_i = n$.

$$\sum_{k=0}^{n} c(n,k) = \prod_{i=0}^{m-1}(x_i + 1) \tag{7}$$

For example, when $a = [1, 2, 2, 3]$, there are $2 \times 3 \times 2 = 12$ combinations in the powerset, they are listed as bellow:

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

However, counting $c(n, k)$ with duplicates in the input replies on the specific input with specific distribution of these items. We are still able to count by enumerating with backtracking search.

Analysis and Implementation The enumeration of the powerset with backtracking search is the same as handling the iterations of choice in the enumeration of permutation with duplicates. We first sort our items in increasing order of the values. Then we replace the `for` loop from the above code with the following code snippet to handle the repetition of items from the input:

```
for i in range(start, n):
  if i - 1 >= start and a[i] == a[i-1]:
          continue
  ...
```

0.2.4 More Combinatorics

In this section, we supplement more use cases of backtracking search in the matter of other types of combinatorics.

All Paths in Graph

For a given acyclic graph, enumerate all paths from a starting vertex s. For example, for the graph shown in Fig. 6, and a starting vertex 0, print out the following paths:

```
0, 0->1, 0->1->2, 0->1->2->5, 0->1->3, 0->1->4, 0->2, 0->2->5
```

Analysis The backtracking search here is the same as how to apply a DFS on an explicit graph, with rather one extra point: a state *path* which might have up to n items (the total vertices of a graph). In the implementation, the `path` vector will dynamically be modified to track all paths constructed as the go of the DFS. The code is offered as:

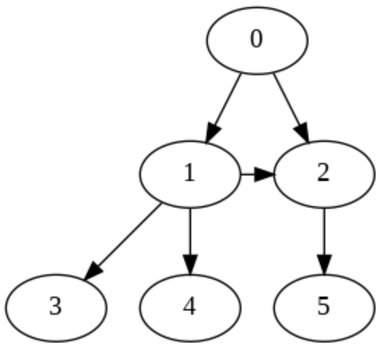


Figure 6: Acyclic graph

```python
def all_paths(g, s, path, ans):
  ans.append(path[::])
  for v in g[s]:
    path.append(v)
    all_paths(g, v, path, ans)
    path.pop()
```

You can run the above code in the Goolge Colab to see how it works on our given example.

Subsequences(940. Distinct Subsequences II)

Given a string, list all unique subsequences. There may or may not exist duplicated characters in the string. For example, when $s ='123'$, there are in total 7 subsequences, which are:

```
'', '1', '2', '3', '12', '13', '23', '123'
```

When $s ='1223'$ which comes with duplicates, there are 12 subsequences:

```
'', '1', '2', '3', '12', '13', '22', '23',
'122', '123', '223',
'1223'
```

Analysis From Chapter Discrete Programming, we have explained that we can count the number of unique subsequences through recurrence relation and pointed out the relation of subsquences with subsets(combinations). Let the number of unique subsequences of a sequence as $seq(n)$ and the number of unique subsets of a set as $set(n)$ with n items in the input. All subsequences are within subsets, and the subsequence set has larger cardinality than subsets, $|seq(n)| \geq |set(n)|$. From the above example, we can also see that when there are only unique items in the sequence or when there are duplicates but all duplicates of an item are adjacent to each other:

- The cardinality of subsequences and subsets equals, $|seq(n) = set(n)|$.

- The subsequences and subsets share the same items when the ordering of the subsequences are ignored.

This indicates that the process of enumerating subsequences is almost the same as of enumerating a powerset. This should give us a good start.

Implementation However, if we change the ordering of the duplicated characters in the above string as $s =' 1232'$, there are in total 14 subsequences instead:

```
'', '1', '2', '3', '12', '13', '23', '22', '32',
'123', '122', '132', '232',
'1232'
```

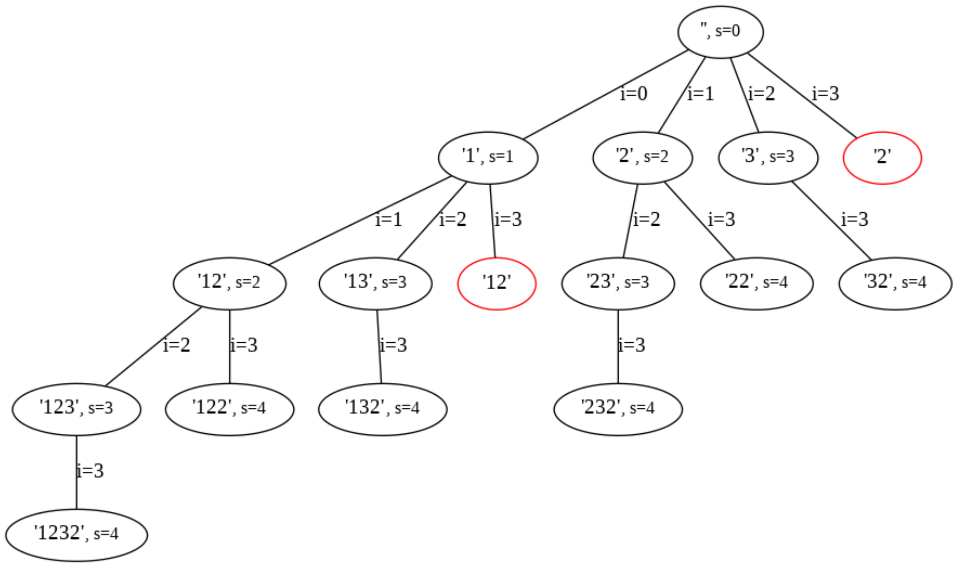


Figure 7: The Search Tree of subsequences.The red circled nodes are redundant nodes. Each node has a variable s to indicate the starting index of candidates to add to current subsequence. i indicate the candidate to add to the current node.

Therefore, our code to handle duplicates should differ from that of a powerset. In the case of powerset, the algorithm first sorts items so that all duplicates are adjacent to each other, making the checking of repetition as simple as checking the equality of item with its predecessor. However, in a given sequence, the duplicated items are not adjacent most of the time, we have to do things differently. We draw the search tree of enumerating all subsequences of string "1232" in Fig. 7. From the figure, we can observe that to avoid redundant branches, we simply check if a current new item in the subsequence is repeating by comparing it with all of its predecessors in range $[s, i]$. The code for checking repetition is as:

```python
def check_repetition(start, i, a):
  for j in range(start, i):
    if a[i] == a[j]:
      return True
  return False
```

And the code to enumerate subsequences is:

```python
def subseqs(a, n, start, d, curr, ans):
  ans.append(''.join(curr[::]))
  if d == n:
    return

  for i in range(start, n):
    if check_repetition(start, i, a):
      continue
    curr.append(a[i])
    subseqs(a, n, i+1, d+1, curr, ans)
    curr.pop()
  return
```

0.2.5 Backtracking in Action

So far, we have applied backtracking search to enumerate combinatorics. In this section, we shall see how backtracking search along with search pruning speedup methods solve two types of challenging NP-hard problems: Constraint Satisfication Problems (CSPs) and Combinatorial Optimization Problems.

As we have briefly introduced the speedup methods needed to solve larger scale of CSPs and COPs. For example, assume within the virtual search tree, the algorithm is currently at level 2 with state $s = [s_0, s_1]$. If there are c choices for state s_1, and if one choice is testified to be invalid, this will prune off $\frac{1}{c}$ of the whole search space. In this section, we demonstrate backtracking search armored with Branch and Prune method solving CSPs and Branch and Bound solving COPs.

0.3 Solving CSPs

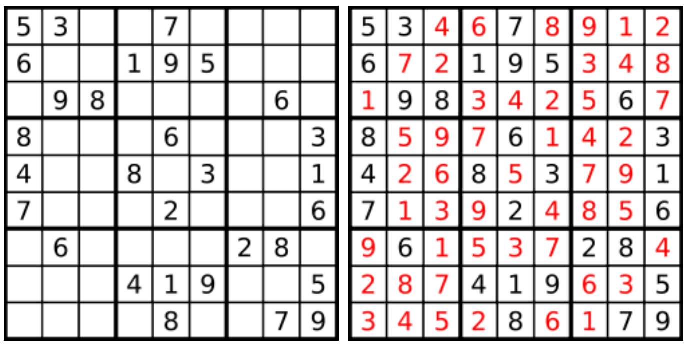
Officially, a constraint satisfaction problem(CSP) consists of a set of n variables, each denoted as s_i, $i \in [0, n-1]$; their respective value domains, each denoted as d_i; and a set of m constraints, each denoted as c_j, $j \in [0, m-1]$. A *solution* to a CSP is an assignment of values to all the variables such that no constraint is violated. A *binary* CSP is one in which each of the constraints involves at most two variables. A CSP can be represented by a *constraint graph* which has a node for each variable and each constraint, and an arc connecting variable nodes contained in a constraint to the corresponding constraint node.

We explain a few strategies from the CSP-solver's arsenal that can potentially speedup the process:

1. Forward Checking: The essential idea is that when a variable X from s_i is instantiated with a value x from its domain d_i, the domain of each future uninstantiated variable Y is examined. If a value y is found such that $X = x$ conflicts with $Y = y$, then y is temporarily removed from the domain of Y.

2. Variable Ordering: The order in which variables are considered while solving a CSP method can have a substantial impact on the search space. One effective ordering is always select the next variable with the smallest remaining domain. In a dynamic variable ordering, the order of variables is determined as the search progresses, and often goes with forward checking which keeps updating the uninstantiated variables' domains. Selecting variable with the minimal domain first can pinpoint the solution quickly given the fact that the branch is still early on, and branch pruning at this stage is more rewarding. Another reasoning is that each step, when we are multiplying d_i to the cost, we are adding the least expensive one, making this a greedy approach.

Sudoku (L37)



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

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

Figure 8: A Sudoku puzzle and its solution

A Sudoku grid shown in Fig. 8 is a $n^2 \times n^2$ grid, arranged into n $n \times n$ mini-grids each containing the values $1, \ldots, n$ such that no value is repeated in any row, column (or mini-grid).

Search Space First, we analyze the number of distinct states in the search space which relies on how we construct the intermediate states and our knowledge in Enumerative combinatorics. We discuss two different formulations on 9×9 grid:

1. For each empty cell in the puzzle, we create a set by taking values $1, ..., 9$ and removing from it those values that appear as a given in the same row, column, or mini-grid as that cell. Assume we have m spots and the corresponding candidate set of each spot is c_i, and initial cost estimation can be obtained which is:

$$T(n) = \prod_{i=0}^{m-1} c_i \tag{8}$$

2. Each row can be presented by a 9-tuples, there will be 9 rows in total, resulting 9 9-tuples to represent the search state. With c_i as the number of non-given values in the i-th 9-tuples, there are $c_i!$ ways of ordering these values by permuting.The number of different states in the search space is thus:

$$T(n) = \prod_{i=0}^{8} c_i! \tag{9}$$

The two different ways each takes a different approach to formulate the state space, making its corresponding backtracking search differs too. We mainly focus on the first formulation with backtracking search.

Speedups Assume we have known all empty spots(variables) to fill in and we construct the search tree using backtracking. In our source code, we did an experiment comparing the effect of ordering variables with minimal domain first rule with arbitrary ordering. The experiment shows that the first method is more than 100 times faster than the second solving the our exemplary Sudoku puzzle. Therefore, we decide to always select the variable that has the least domain set to proceed next in the backtracking.

Further, we apply forward checking, for the current variable and a value we are able to assign, we recompute all the remaining empty spots' domain sets, and use the updated domain sets to decide:

- If this assigment will lead to empty domain for any of other remaining spots, and if so, we terminate the search and backtrack.
- The spot to select next time with the ordering rule we choose.

Implementation We set aside three vectors of length 9, `row_state`, `col_state`, and `block_state` to track the state of all 9 rows, columns, and grids. The list has `set()` data structures as items, saving the numbers filled already in that row, col, and grid respectively. Two stages in the implementation:

1. Initialization: We scan the whole each spot in the 9×9 grid to record the states of the filled spots and to find all empty spots that waiting to

be filled in. With (i, j) to denote the position of a spot, it corresponds to `row_state[i]`, `col_state[j]`, and `block_state[i//3][j//3]`. We also write two functions to set and reset state with one assignment in the backtracking. The Python code is as follows:

```python
from copy import deepcopy
class Sudoku():
  def __init__(self, board):
    self.org_board = deepcopy(board)
    self.board = deepcopy(board)

  def init(self):
    self.A = set([i for i in range(1,10)])
    self.row_state = [set() for i in range(9)]
    self.col_state = [set() for i in range(9)]
    self.block_state = [[set() for i in range(3)] for i in
   range(3)]
    self.unfilled = []

    for i in range(9):
      for j in range(9):
          c = self.org_board[i][j]
          if c == 0:
              self.unfilled.append((i, j))
          else:
              self.row_state[i].add(c)
              self.col_state[j].add(c)
              self.block_state[i//3][j//3].add(c)

  def set_state(self, i, j, c):
    self.board[i][j] = c
    self.row_state[i].add(c)
    self.col_state[j].add(c)
    self.block_state[i//3][j//3].add(c)

  def reset_state(self, i, j, c):
    self.board[i][j] = 0
    self.row_state[i].remove(c)
    self.col_state[j].remove(c)
    self.block_state[i//3][j//3].remove(c)
```

2. Backtracking search with speedups: In the initialization, we have another variable A used as the domain set of the current processing spot. To get the domain set according to the constraints, a simple set operation is executed as: $A-(row\_state[i]|col\_state[j]|block\_state[i//3][j//3])$. In the solver, each time, to pick a spot, we first update all remaining spots in the `unfilled` and then choose the one with minimal domain. This process takes $O(n)$ which is trivial compared with the cost of the searching, with 9 for computing domain set of a single spot, $9n$ for n spots, and adding another n to $9n$ to choose the one with the smallest size. The solver is implemented as:

```python
def _ret_len(self, args):
  i, j = args
  option = self.A - (self.row_state[i] | self.col_state[j
  ] | self.block_state[i//3 ][j//3])
  return len(option)

def solve(self):
  if len(self.unfilled) == 0:
    return True
  # Dynamic variables ordering
  i, j = min(self.unfilled, key = self._ret_len)
  # Forward looking
  option = self.A - (self.row_state[i] | self.col_state[j
  ] | self.block_state[i//3 ][j//3])
  if len(option) == 0:
    return False
  self.unfilled.remove((i, j))
  for c in option:
    self.set_state(i, j, c)
    if self.solve():
      return True
    # Backtracking
    else:
      self.reset_state(i, j, c)
  # Backtracking
  self.unfilled.append((i, j))
  return False
```

0.4 Solving Combinatorial Optimization Problems

Combinatorial optimization is an emerging field at the forefront of combinatorics and theoretical computer science that aims to use combinatorial techniques to solve discrete optimization problems. From a combinatorics perspective, it interprets complicated questions in terms of a fixed set of objects about which much is already known: sets, graphs, polytopes, and matroids. From the perspective of computer science, combinatorial optimization seeks to improve algorithms by using mathematical methods either to reduce the size of the set of possible solutions or to make the search itself faster.

Genuinely, the inner complexity of a COP is at least of exponential, and its solutions fall into two classes: exact methods and heuristic methods. In some cases, we may be able to find efficient exact algorithms with either greedy algorithms or dynamic programming technique such as finding the shortest paths on a graph can be solved by the Dijkstra (greedy) or Bellman-Ford algorithms(dynamic programming) to provide exact optimal solutions in polynomial running time. For more complex problems, COP can be mathematically formulated as a Mixed Linear Programming(MILP) model and which is generally solved using a linear-programming based branch-and-bound algorithm. But, in other cases no exact algorithms are feasible, and the following randomized heuristic search algorithms though we do not cover in this section should be applied:

1. Random-restart hill-climbing.
2. Simulated annealing.
3. Genetic Algorithms.
4. Tabu search.

Model Combinatorial Optimization Problems It is a good practice to formulate COPs with mathematical equations/inequations, which includes three steps:

1. Choose the decision variables that typically encode the result we are interested in, such that in a superset problem, each item is a variable, and each variable includes two decisions: take or not take, making its value set as $0, 1$.
2. Express the problem constraints in terms of these decision variables to specify what the feasible solutions of the problem are.
3. Express the objective function to specify the quality of each solution.

There are generally many ways to model a COP.

Branch and Bound Branch and bound (BB, B&B, or BnB) is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as mathematical optimization. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm. “Branching” is to split problem into a number of subproblems, and “bounding” is to find an optimistic estimation of the best solution to the the subproblems to either maximize the upper bound or minimize the lower bound. To get the optimistic estimation, we have to *relax constraints*. In this section, we will exemplify both the minimization(TSP) and maximization problem(Knapsack).

Search Strategies In practice, we can apply different search strategies to enumerate the search space of the problem, such as depth-first, best-first, and least-discrepancy search. The way of how each listed strategy is applied in the combinatorial optimization problems is:

- Depth-First: it prunes when a node estimation is worse than the best found solution.
- Best-First: it selects the node with the best estimation among the frontier set to expand each time. Worst scenario, the whole search tree have to be saved as long the best estimation is extremely optimistic and not a single branch is pruned in the process.
- Least-Discrepancy: it trusts a greedy heuristic, and then move away from the heuristic in a very systematic fashion.

In this section, we discuss exact algorithms using Branch and Bound with a variation of search strategies. During the interviews, questions that have polynomial exact solutions are more likely to appear, proving your mastery of dynamic programming or greedy algorithms design methodologies. However, it is still good to discuss this option.

0.4.1 Knapsack Problem

Given n items with weight and value indicated by two vectors W and V respectively. Now, given a knapsack with capacity c, maximize the value of items selected into the knapsack with the total weight being bounded by c. Each item can be only used at most once. For example, given the following data, the optimal solution is to choose item 1 and 3, with total weight of 8, and optimal value of 80.

```
c = 10
W = [5, 8, 3]
V = [45, 48, 35]
```

Search Space In this problem, x_i denotes each item, and w_i, v_i for its corresponding weight and value, with $i \in [0, n-1]$. Each item can either be selected or left behind, indicating $x_i \in 0, 1$. The selected items can not exceed the capacity, making $\sum_{i=0}^{n-1} w_i x_i \leq c$. And we capture the total value of the selected items as $\sum_{i=0}^{n-1} v_i x_i$. Putting it all together:

$$\max_{v,x} \quad \sum_{i=0}^{n-1} v_i x_i \tag{10}$$

$$\text{s.t.} \quad \sum_{i=0}^{n-1} w_i x_i \leq c \tag{11}$$

$$x_i \in 0, 1 \tag{12}$$

With each variable having two choices, our search space is as large as 2^n.

Branch and Bound To bound the search, we have to develop a heuristic function to estimate an optimistic–maximum–total value a branch can lead to.

In the case of knapsack problem, the simplest estimation is summing up the total values of selected items so far, and estimate the maximum value by adding the accumulated values of all remaining unselected items along the search.

A tighter heuristic function can be obtained with **constraint relaxation**. By relaxing the condition of simply choose $\{0, 1\}$ to $[0, 1]$, that a fraction of an item can be chosen at any time. By sorting the items by the value per unit $\frac{v_i}{w_i}$, then a better estimate can be obtained by filling the remaining capacity of knapsack with unselected items, with larger unit value first be considered. A branch is checked on the optimal solution so far against the lower estimated bound in our case, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm. Both heuristic functions are more optimistic compared with the true value, but the later is a tighter bound, being able to prune more branches along the search and making it more time efficient. We demonstrate branch and bound with two different search strategies: DFS(backtracking) and Best-First search.

Depth-First Branch and Bound

i	v	w	v/w
0	35	3	11.66
1	45	5	9
2	48	8	6

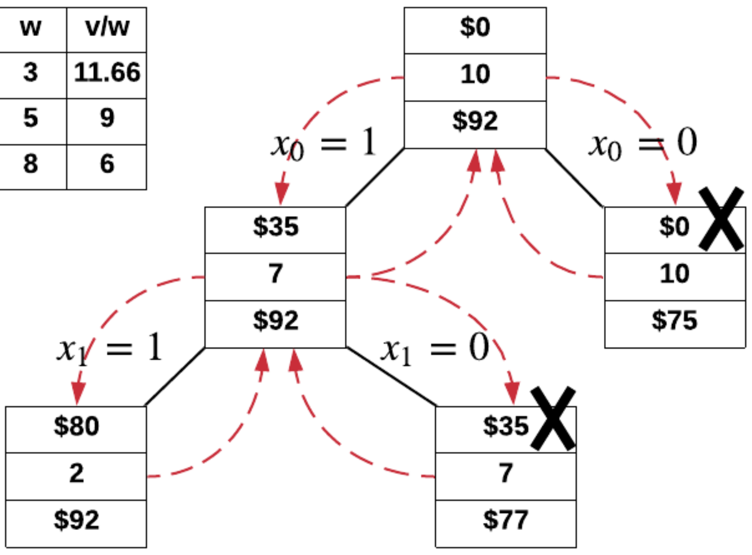


Figure 9: Depth-First Branch and bound

We set up a class `BranchandBound` to implement this algorithm. First, in the initiation, we add additional $\frac{v_i}{w_i}$ to mark each item's value per unit, and sort these items by this value in decreasing order. Second, we have a function `estimate` which takes three parameters: `idx` as start index of the remaining items, `curval` is the total value based on all previous decision, and `left_cap` as the left capacity of the knapsack. The code snippet is:

```python
import heapq

class BranchandBound:
  def __init__(self, c, v, w):
    self.best = 0
    self.c = c
    self.n = len(v)
    self.items = [(vi/wi, wi, vi) for _, (vi, wi) in enumerate(
    zip(v, w))]
    self.items.sort(key=lambda x: x[0], reverse=True)

  def estimate(self, idx, curval, left_cap):
    est = curval
    # use the v/w to estimate
    for i in range(idx, self.n):
      ratio, wi, _ = self.items[i]
      if left_cap - wi >= 0: # use all
        est += ratio * wi
        left_cap -= wi
      else: # use part
        est += ratio * (left_cap)
        left_cap = 0
    return est
```

In the Depth-first search process, it consists of two main branches: one considering to choose the current item, and the other to handle the case while the item is not selected. For the first branch, it has to be bounded by the capacity and the comparison of the best found solution against to the estimation. Additional `status` is to assist to visualize the process of the search, which tracks the combination of items. The process is shown in Fig. 9. And the code is as:

```python
  def dfs(self, idx, est, val, left_cap, status):
      if idx == self.n:
        self.best = max(self.best, val)
        return
      print(status, val, left_cap, est )

      _, wi, vi = self.items[idx]
      # Case 1: choose the item
      if left_cap - wi >= 0: # prune by constraint
        # Bound by estimate, increase value and volume
        if est > self.best:
          status.append(True)
          nest = self.estimate(idx+1, val+vi, left_cap - wi)
          self.dfs(idx+1, nest, val+vi, left_cap - wi, status)
          status.pop()

      # Case 2: not choose the item
      if est > self.best:
        status.append(False)
        nest =  self.estimate(idx+1, val, left_cap)
        self.dfs(idx+1, nest, val, left_cap, status)
        status.pop()
      return
```

Best-First Branch and Bound

Within Best-First search, we use priority queue with the estimated value, and each time the one with the largest estimated value within the frontier set is expanded first. Similarly, with branch and bound, we prune branch that has estimated value that would never surpass the best solution up till then. The search space is the same as in Fig. 9 except that the search process is different from depth-first. In the implementation, the priority queue is implemented with a min-heap where the minimum value is firstly popped out, thus we use the negative estimated value to make it always pop out the largest value conveniently instead of write code to implement a max-heap.

```
  def bfs(self):
      # track val, cap, and idx is which item to add next
      q = [(-self.estimate(0, 0, self.c), 0, self.c, 0)] # estimate, val, left_cap, idx
      self.best = 0
      while q:
        est, val, left_cap, idx = heapq.heappop(q)
        est = -est
        _, wi, vi = self.items[idx]
        if idx == self.n - 1:
          self.best = max(self.best, val)
          continue

        # Case 1: choose the item
        nest = self.estimate(idx + 1, val + vi, left_cap - wi)
        if nest > self.best:
          heapq.heappush(q, (-nest, val + vi, left_cap - wi, idx + 1))

        # Case 2: not choose the item
        nest = self.estimate(idx + 1, val, left_cap)
        if nest > self.best:
          heapq.heappush(q, (-nest, val, left_cap, idx + 1))
      return
```

0.4.2 Travelling Salesman Problem

Given a set of cities and the distances between every pair, find the shortest possible path that visits every city exactly once and returns to the origin

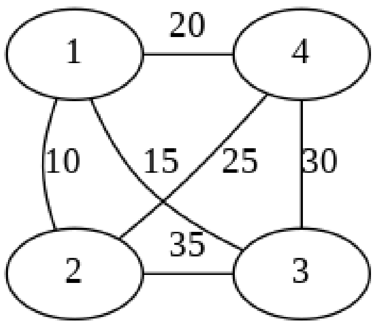


Figure 10: A complete undirected weighted graph.

city. For example, with the graph shown in Fig. 10, such shortest path is $[0, 1, 3, 2, 0]$ with a path weight 80.

Search Space In TSP, a possible complete solution is a *Hamiltonian cycle*, a graph cycle that visits each vertex exactly once. Since it is a cycle, it does not matter where it starts. For convenience, we choose vertex 0 as the origin city. Therefore, in our example, our path starts and ends at 0, and the remaining $n-1$ vertices between will be a permutation of these vertices, making the complexity as $(n-1)!$.

Because this is a complete graph, it might be tempting to apply backtracking on the graph to enumerate all possible paths and find and check possible solutions. However, this path searching will build a $n-1$-ary search tree with height equals to $n-1$, making the complexity as $\frac{(n-1)^n-1}{n-2}$, which is larger than the space of permutation among $n-1$ items. Therefore, in our implementation, we apply backtracking to enumerate all permutations of $n-1$ vertices and check its corresponding cost.

Speedups Since we only care about the minimum cost, then any partial result that has cost larger than the minimum cost of all known complete solutions can be prunned. This is the *Branch and bound* method that we have introduced that is often used in the combinatorial optimization.

Implementation We built the graph as a list of dictionaries, each dictionary stores the indexed vertex's other cites and its corresponding distance as key and value respectively. Compared with standard permutation with backtracking, we add four additional variables: `start` to track the starting vertex, `g` to pass the graph to refer the distance information, `mincost` to save the minimum complete solution so far found, and `cost` to track the current partial path's cost. The code is shown as:

```
def tsp(a, d, used, curr, ans, start, g, mincost, cost):
  if d == len(a):
    # Add the cost from last vertex to the start
    c = g[curr[-1]][start]
    cost += c
    if cost < mincost[0]:
      mincost[0] = cost
      ans[0] = curr[::] + [start]
    return

  for i in a:
    if not used[i] and cost + g[curr[-1]][i] < mincost[0] :
      cost += g[curr[-1]][i]
      curr.append(i)
      used[i] = True
      tsp(a, d + 1, used, curr, ans, start, g, mincost, cost)
      curr.pop()
      cost -= g[curr[-1]][i]
      used[i] = False
  return
```

TSP is a NP-hard problem, and there is no known polynomial time solution so far.

Other Solutions

Whenever we are faced with optimization, we are able to consider the other two algorithm design paradigm–Dynamic Programming and Greedy Algorithms. In fact, the above two problems both have its corresponding dynamic programming solutions: for knapsack problem, polynomial solution is possible; for TSP, though it is still of exponential time complexity, it is much better than $O(n!)$. We will further discuss these two problems in Chapter Dynamic Programming.

0.5 Exercises

1. 77. Combinations

2. 17. Letter Combinations of a Phone Number

3. 797. All Paths From Source to Target

4. N-bit String: enumerate all n-bit strings with backtracking algorithm, for example:

   ```
   n = 3, all 3-bit strings are:
   000, 001, 010, 011, 100, 101, 110, 111
   ```

5. 940. Distinct Subsequences II

6. N-queen

7. Map-coloring

8. 943. Find the Shortest Superstring (hard). Can be moduled as traveling salesman problem and dynamic programming