The Knapsack problem

<u>Problem Definition:</u> Given n items of known *weight*, *value* pairs $\{(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)\}$ and a knapsack of capacity W, find the most valuable subset of items which fit into the knapsack.

- How many copies x_i of each kind of item should we add to the knapsack? Answer leads to variations of the knapsack problem

More formally

$$\text{Maximize } \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} w_i\,, x_i \quad \leq W$$

Variants of the Knapsack problem

- 0-1 knapsack problem: restricts the number of copies of each item to either 0 or 1. i. e, $x_i \in \{0, 1\}$
- Bounded knapsack problem: restricts the number of copies of each item to a maximum non-negative integer p. i.e, $0 \leq x_i \leq p$
- Unbounded knapsack problem: only restriction is that number of copies of each item is non-negative. i.e, $x_i \geq 0$
- Fractional knapsack problem: items can be broken into fractions

Brute-Force algorithm for solve the 0-1 KNP

```
optimalValue ← 0
optimalSubset ← none
for each subset s of the n items do
    compute the total weight w of the items in s
    if w is less than the knapsack capacity W then
        // s is called a feasible subset
        compute the total value v of the items in s
        if v is greater than optimalValue then
            optimalValue ← v
            optimalSubset ← s
return optimalSubset, optimalValue
```

Complexity of the brute-force algorithm of the Knapsack problem

- How many subsets are in a set of size n?

- If we assume that we can generate any subset, compute its weight and value in constant time c, total runtime will be

$$T(n) = \sum_{i=1}^{2^n} c = c * 2^n = O(2^n)$$

- Runtime is exponential making the knapsack problem intractable for all but small values of n
- In practice, not all the 2^n subsets of the n items are generated

Depth-first search

<u>Rough Sketch:</u>

1. Choose an arbitrary vertex as the starting point. Mark that vertex as visited
2. Proceed to an unvisited vertex adjacent to the current vertex and mark it as visited. If there're several unvisited vertices, employ a tie resolution strategy
3. Repeat Step 2 until there's no unvisited vertex adjacent to the current vertex. We call that situation a dead end
4. Return (back up) to the vertex from which the current vertex was reached.
5. Repeatedly perform Steps 2, 3 and 4 until the starting vertex becomes a dead-end.
6. Choose a vertex which has not yet been visited. Repeat the algorithm.

Depth-first search

- Yields two orderings of the graph's vertices
  - The order in which they are first encountered by DFS
  - The order in which they become dead ends

  Both orderings are super useful and often employed for different purposes

- Produces the DSF traversal tree composed of tree edges and may also contain back edges
- Often implemented recursively (using system stack). The iterative implementation (using explicit stack) is also common

```
procedure DFS(G)
    //Input: Graph G = ⟨V, E⟩
    //Output: Graph G with its vertices marked consecutive integers
    // marking the order in which they are first encountered by the DFS
    mark each vertex in V with 0 as a mark of being unvisited
    count ← 0
    for each vertex v in V do
        if v is marked with 0 then
            dfs(v)


procedure dfs(v)
    // traverses the neighbours of a vertex v recursively
    //global variable count. Used to number the vertices
    //count ← count + 1
    mark v with count
    for each unvisited neighbour of v do
        dsf(v)
```

```
procedure DFS_EXPLICITSTACK(G)
    //Input: Graph G = ⟨V, E⟩
    //Output: Graph G with its vertices marked consecutive integers
    // marking the order in which they are first encountered by the DFS
    mark each vertex in V with 0 as a mark of being unvisited
    count ← 0
    for each vertex v in V do
        if  v is marked with 0 then
            dfs(v)

procedure dfs(v)
    count ← count + 1; mark v with count
    create a stack S and push v into it.
    while S is not empty do
        for each vertex w adjacent to the top vertex u in S do
            if w is marked with 0 then
                count ← count + 1; ; mark w with count
                push w into S
        pop u off S
```

Complexity of Depth-first search algorithm

- We visit each vertex exactly once. Thus, vertex visit costs $\Theta(|V|)$

- For each vertex v, we scan all its adjacent vertices. This results in

$$\sum_{v \in V} adj[v] = 2|E| = \Theta(|E|)$$

- Total runtime is $\Theta(|V| + |E|)$ for adjacency list implementation.
- The adjacency matrix implementation results in $\Theta(|V|^2)$

Breadth-first search

- Traverses a graph by visiting all vertices adjacent to a starting vertex
- It then visit all unvisited vertices two edges away from the starting vertex, and then 3 edges away, and so on until all vertices are traversed
- Breadth-first is implemented using a queue

- Produces the breadth-first traversal tree composed of tree edge and may also contain cross edges

```
procedure BFS(G)
    //Input: Graph G = ⟨V, E⟩
    //Output: Graph G with its vertices marked consecutive integers
    // marking the order in which they are first encountered by the DFS
    mark each vertex in V with 0 as a mark of being unvisited
    count ← 0
    for each vertex v in V do
        if  v is marked with 0 then
            bfs(v)

procedure bfs(v)
    count ← count + 1; mark v with count
    create a queue Q and add v to it.
    while Q is not empty do
        for each vertex w adjacent to the front vertex u in Q do
            if w is marked with 0 then
                count ← count + 1; ; mark w with count
                add w to Q
        remove u from Q
```

Complexity of BFS

- The breadth-first search algorithm has the same complexity as the depth-first search algorithm.

- Thus, BFS has $\Theta(|V| + |E|)$ for adjacency list representation and $\Theta(|V|^2)$ for adjacency matrix representation