CSCI 2270 – Data Structures

Recitation 6, Summer 2020

Dijkstra's Algorithm and Hash Tables

Objectives

1. BFS for Shortest Path
2. Weighted graphs and Dijkstra's algorithm
3. Hashing
4. Why not Arrays?
5. Components of Hashing

The shortest path between two vertices in a graph is the one where the sum of weights of edges in the path is minimal.

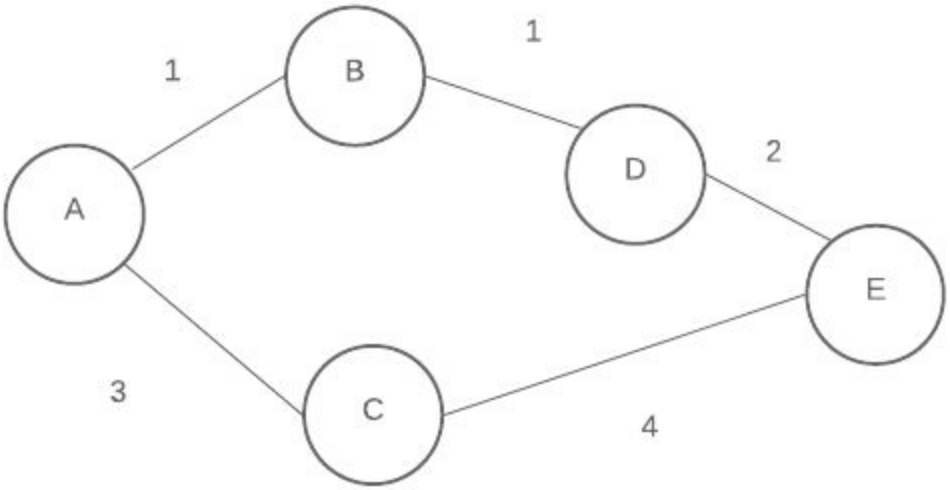
BFS for Shortest Path:

BFS(Breadth-First Search) is used to compute the shortest path between any two vertices in an **unweighted** graph. In an unweighted graph, the **path length** is proportional to the number of vertices in the path.

BFS exhausts all possible vertices at a level (say n) before moving on to vertices at a level - n+1. The claim for BFS is that the first time a node is discovered during the traversal, that distance from the source would give us the shortest path for that node.

The same claim cannot be applied for a weighted graph.
The following example will make it more clear.

Dijkstra's Algorithm and Hash Tables



For the above graph, if we apply BFS, it would give the shortest path as:

A ->C->E (**length** : 3+4 = 7)

But, the path **A->B->D->E** is shorter (**length**: 1+1+2 = 4)

Thus, in a weighted graph, a smaller number of vertices in the path need not imply a shorter path. (Since each edge could have any weight).
To solve this, we use **Dijkstra's** algorithm.

How does **Dijkstra's** solve it?

Dijkstra's algorithm works by marking one vertex at a time as it discovers the shortest path to that vertex. In this process, it helps to get the shortest distance from the source vertex to every other vertex in the graph. It keeps updating these distances for the vertices in increasing order of path length and re-uses them to compute the shortest distance of the destination. Let's look at the algorithm.

CSCI 2270 – Data Structures

Recitation 6, Summer 2020

Dijkstra's Algorithm and Hash Tables

Dijkstra's algorithm:

Each vertex has a **distance** measure and a **solved** boolean flag.

- **distance -** stores the shortest path length from the source vertex.
- **solved -** tells if the shortest path for that vertex from the source has been found.
- **parent -** stores the parent of a vertex found during the traversal.

Initialization:

The **solved** fields for each vertex are initialized using below:

	solved
Source (A)	TRUE
Every other vertex	FALSE

Once, the shortest path for a vertex is found, we update the **solved** flag (to TRUE) and set the **distance** field for that vertex**.** We also have a **solvedList** which at any point in time stores the list of vertices for which the shortest distance has been computed. **Initially,** it has just the **Source(A)** in it.

The **algorithm** can be described as below:

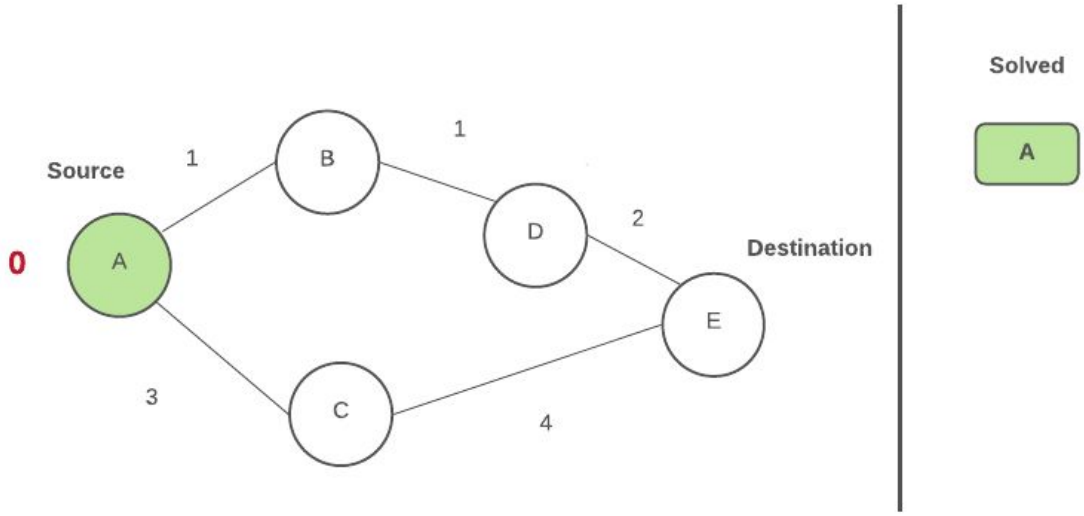
Till **destination** is not solved, do:

- For each element **s** in the **solvedList,** we look at the adjacent vertices that aren't solved yet.
- Among all these vertices, get the vertex whose distance (this would be parent's distance + weight of the edge) from the source is the shortest. Then mark it as **solved, set** its **distance field,** and add it to the **solvedList.**

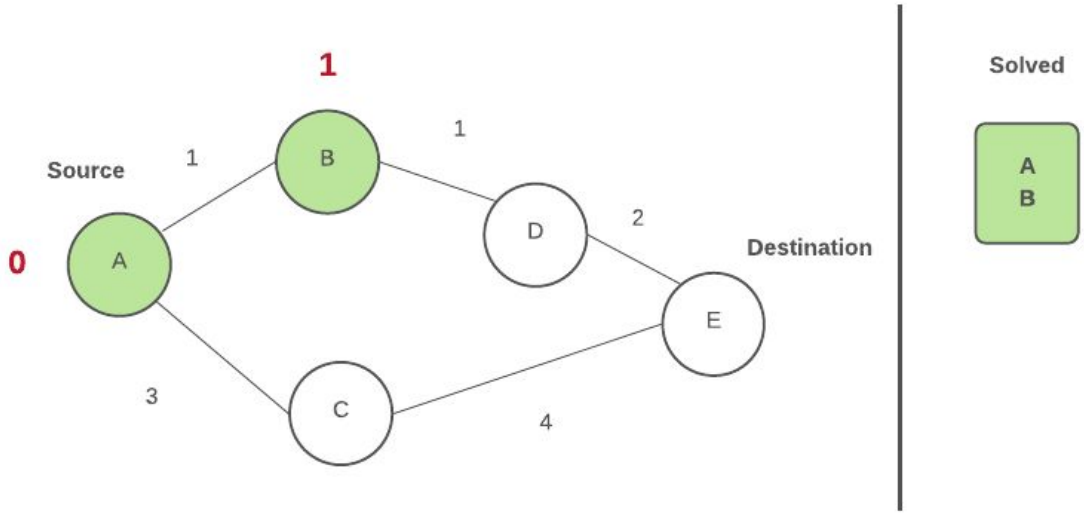
Look at the illustration below to understand it better!

Dijkstra's Algorithm and Hash Tables

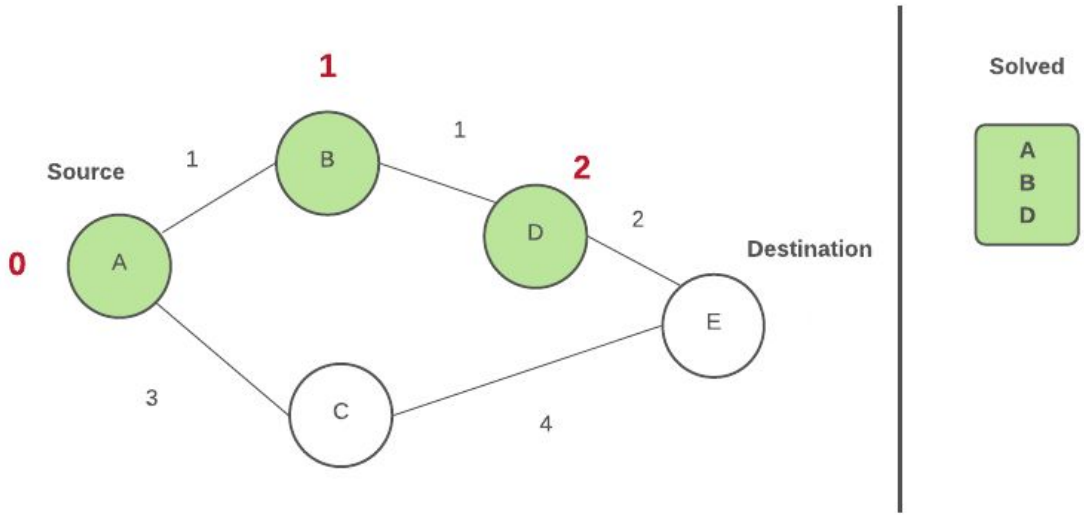
1- Initial State: (**Green** - **Solved** vertex)



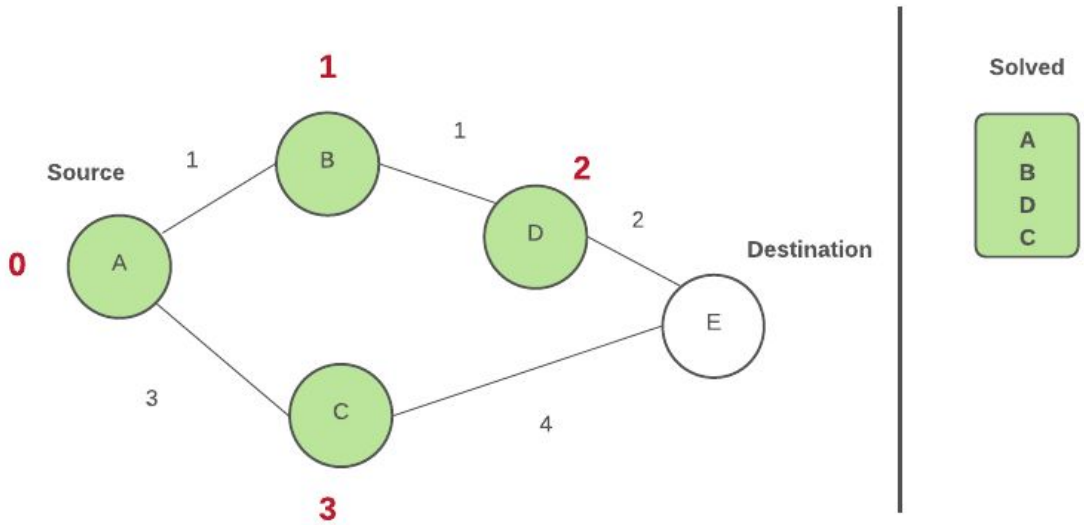
2 - Iterate over all adjacent vertices of A, set B's distance from Source(A):



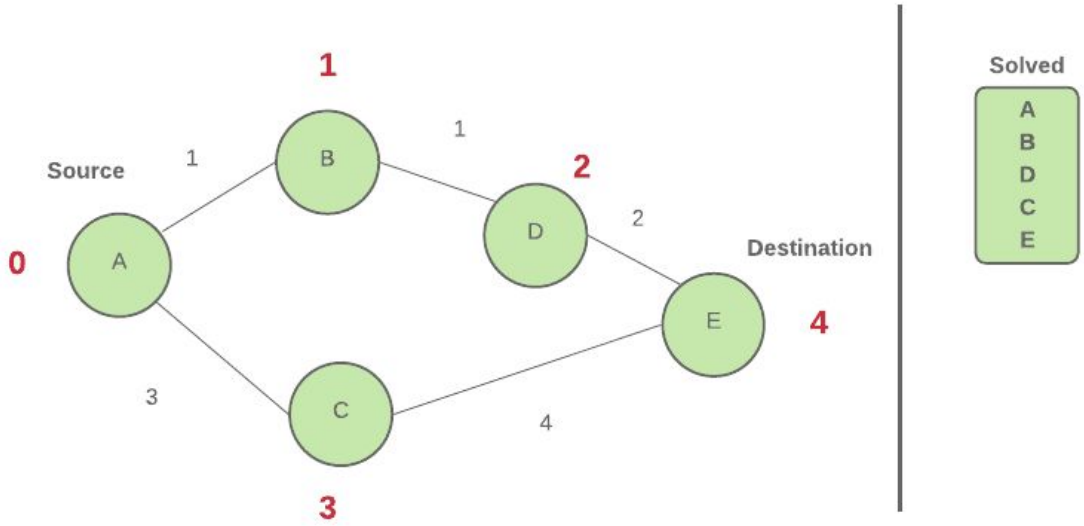
3 - Iterate over all adjacent vertices of A and B, set D's distance from Source(A):



4 - Iterate over all adjacent vertices of A, B, and D and set C's distance from Source(A):



5 - Iterate over all adjacent vertices of A, B, D, and C and set E's distance from Source(A):



In doing the above steps, we get the shortest path length from source A to all the vertices in the graph. Hence, Dijkstra is one of the ways to compute single-source shortest paths to every vertex. Note that, we have **solved** the vertices in increasing order of shortest path length from the source.

3. What is Hashing

Hashing is a technique used for storing and retrieving information as fast as possible. It is used to perform an optimal search. Hashing can be used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples, the students and books are hashed to a unique number.

Why Hashing

We have seen that Balanced Binary Search Trees supports operations such as insert, delete, and search in O(logn) time. In application, if we need these operations in *O(1)*, then hashing is one of the possible methods to do that. The worst-case complexity of hashing is still *O(n)*, but it gives *O(1)* on the average.

Understanding Hashing

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array, i.e in simple terms, we can treat the array as a hash table. To understand the use of hash tables and hashing in general, let us consider the following example:

Let us consider a string S. You are required to give an algorithm for printing the first repeated character in S. The simplest way of solving is for each character in the string, check if the character is repeated or not. The time complexity of this approach is O(N^2) where N is the size of the string.

Now, let us apply hashing to this problem. Take an array of size 256, since we know that the total number of possible ASCII characters is 256. Then, iterate over the string, for each of the input characters go to the corresponding location in the array and increase the count. While scanning the input, if we come across a character whose counter is already 1, we have our answer. The complexity of this approach is O(N).

4. Why not Arrays?

Suppose we have an array of numbers instead of the string above and we want to solve the same problem with it. In this case, the set of possible values is infinite, or at least very big. Creating this huge array and storing the counters is not possible. If we want to solve this problem we need to somehow map all these possible keys to the possible memory locations.
As a result, using simple arrays is not the correct choice when possible keys are very big.

The process of mapping the keys to locations is called Hashing. In hashing, large keys are converted into small keys by using hash functions, and the values are then stored in a data structure called a hash table. We will be talking about these components of Hashing now.

Before we jump to the components of Hashing, enumerating here some common operations performed on Hash Tables:

1) Search - Searches the key in Hash table
2) Insert - Inserts a new key into Hash table
3) Delete - Deletes a key from Hash table

A very simple example of Hashing using a mapping to obtain indices in an array:
Let us consider string S. You are required to count the frequency of all the characters in this string.

```
string s = "ababcd"
```

Take an array frequency of size 26 (assuming we are taking only 26 characters: 'a'-'z') and hash these 26 characters with indices of the array by using the following function. This function maps every character to an index in the array.

```
int hashFunc(char c)
{
     return (c - 'a');
}
```

Then, iterate over the string and increase the value in the frequency array at the corresponding index for each character.

5.Components of Hashing

Hashing has four key components:

1. Hash Table
2. Hash Functions
3. Collisions
4. Collision Resolution Techniques

Hash Table

Hash table is a generalization of an array. With an array, we store the element whose key is k at a position k of the array. That means, given a key k, we find the element whose key is k by just looking in the kth position of the array. This is called direct addressing.

Direct addressing is applicable when we can afford to allocate an array with one position for every possible key. Suppose we do not have enough space to allocate a location for every possible key then we need a mechanism to handle this case. Another way of defining the scenario is if we have fewer locations and more possible keys then simple array implementation is enough.

In these cases, one option is to use hash tables. Hash table or hash map is a data structure that stores the keys and their associated values. Hash table uses a hash function to map keys to their associated values. The general convention is that we use a hash table when the number of keys actually stored is small relative to the number of possible keys.

Dijkstra's Algorithm and Hash Tables

Hash Function

The hash function is used to transform the key into the index. Ideally, the hash function should map each possible key to a unique slot index, but it is difficult to achieve in practice.

How to choose hash functions?
The basic problems associated with the creation of hash tables are:

- An efficient hash function should be designed so that it distributes the index values of inserted objects uniformly across the table.
- An efficient collision resolution algorithm should be designed so that it computes an alternative index for a key whose hash index corresponds to a location previously inserted in the hash table.
- We must choose a hash function that can be calculated quickly, returns values within the range of locations in our table, and minimizes collisions.

Characteristics of Good Hash Functions

A good hash function should have the following characteristics:

- Minimize collision.
- Be easy and quick to compute.
- Distribute key values evenly in the hash table.
- Use all the information provided in the key.
- Have a high load factor for a given set of keys.

Load Factor

The load factor of a non-empty hash table is the number of items stored in the table divided by the size of the table. This is the decision parameter used when we want to rehash or expand the existing hash table entries. This also helps us in determining the efficiency of the hashing functions. That means, it tells whether the hash function is distributing uniformly or not.

Load factor = Number of elements in the hash table/ Hash table size

Collisions

Hash functions are used to map each key to different address spaces but practically it is not possible to create such a hash function and the problem is called a collision. Collision is the condition where two records are stored in the same location.

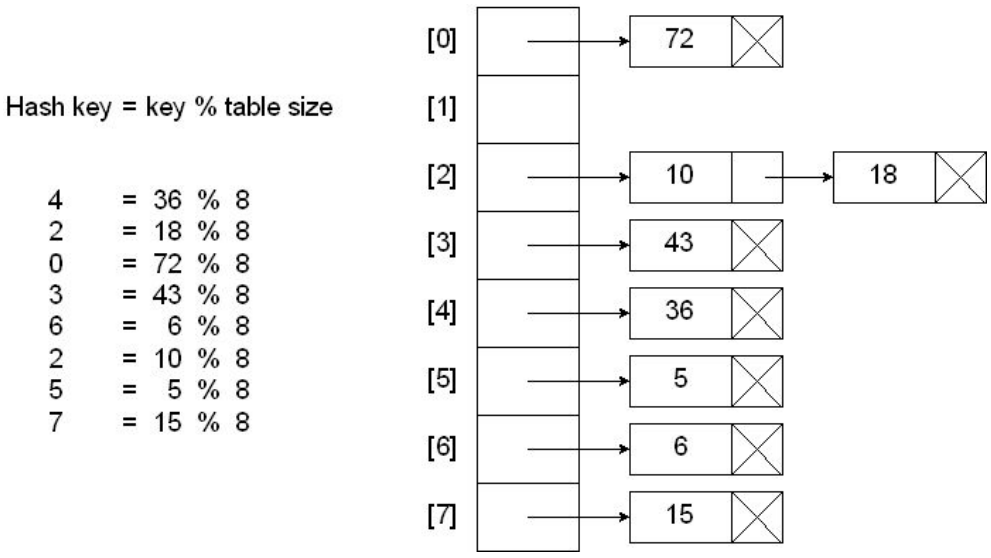
Collision Resolution Techniques

The process of finding an alternate location is called collision resolution. Even though hash tables are having collision problems, they are more efficient in many cases to all other data structures like search trees. There are many collision resolution techniques. For the purpose of this class we will study 2 important collision resolution techniques:

- Direct Chaining
  - Separate Chaining
- Open Addressing
  - Linear probing

Direct Chaining(Separate Chaining)

Collision resolution by chaining combines linked representation with a hash table. When two or more records hash to the same location, these records are constituted into a singly-linked list called a chain. For example, in the figure below we are trying to insert few integers into a hash table using the given hash function, and we can see how collision is resolved when two or more integers are hashed to the same index.



Open Addressing (Linear Probing)

In open addressing, all keys are stored in the hash table itself. This approach is also known as closed hashing. This procedure is based on probing. A collision is resolved by probing.

Linear Probing

The interval between probes is fixed at 1. In linear probing, we search the hash table sequentially starting from the original hash location. If a location is occupied, we check the next location. We wrap around from the last table location to the first table location if necessary. The function for rehashing is the following:

```
rehash(key) = (n+1)% tablesize
```

One of the problems with linear probing is that the table items tend to cluster together in the hash table. This means that table contains groups of consequently occupied locations that are called clustering.

Clusters can get close to one another, and merge into a larger cluster. Thus, one part of the table might be quite dense, even though another part has relatively few items. Clustering causes long probe searches and therefore decreases the overall efficiency.

The next location to be probed is determined by the step-size, where other step-sizes are possible. The step-size should be relatively prime to the table size., i.e their greatest common divisor should be equal to 1. If we choose the table size to be a prime number, then any step-size is relatively prime to the table size. Clustering cannot be avoided by larger step-sizes.

CSCI 2270 – Data Structures

Recitation 6, Summer 2020

Dijkstra's Algorithm and Hash Tables

Exercise

A. Silver Badge Problem (Mandatory)

1. Download the Lab6 **zipped** folder from Moodle.
2. Follow the TODOs in Graph.cpp and complete the **isBridge()** function.
3. This function finds if an edge is a bridge of the graph:
   A bridge is an edge of a graph whose deletion increases its number of connected components. You will be using DFS (Depth First Search) to solve this problem. Please refer to the previous recitation document for DFS).