

ESC190 Midterm Review:

Data Structures and Algorithms

QiLin Xue

March 9, 2021

Contents

1	Data Structures	1
1.1	Linked List	1
1.2	Stacks	2
1.3	Queues	2
1.4	Priority Queues	2
1.5	Heaps	2
1.5.1	Manual Implementation	4
1.5.2	Runtime	4
1.5.3	Heapsort	4
2	Graph Theory	4
2.1	Background	4
2.2	Representing Graphs	5
2.2.1	Adjacency Matrix	5
2.2.2	Adjacency List	5
2.3	Traversal Algorithms	6
2.3.1	Breadth First Search	6
2.3.2	Depth First Search	7
3	Shortest Path Algorithm	7
3.1	Dijkstra's Algorithm	8
3.2	Using Priority Queue	10
3.3	Greedy Best First Search	10
3.4	A*	12

1 Data Structures

1.1 Linked List

A linked list uses *pointers* to link from one object to another. In *C*, this is a practical way of describing lists that allow for inserting and deleting new entries, which we will call nodes. This is because we do not need to allocate this memory beforehand, which is especially handy if we do not know how large our list will be. You should be familiar with how to implement the following functions in *C* related to linked lists:

- Creating a linked list
- Deleting the linked list
- Appending a node at a certain position
- Deleting a node at a certain position
- Finding a node which has a specific value
- Looking to see if there is a loop

The last node of a linked list should always point to the `null` pointer.

1.2 Stacks

Stacks retrieve information in a *last in, first out* order.

```
stack = []

# Add in entries
stack.append('a')
stack.append('b')
stack.append('c')

# The last entry added was 'c' so the first entry that comes out will be 'c'
out = stack.pop()
print(out) # 'c'
```

1.3 Queues

Queues retrieve information in a *first in, first out* order

```
queue = []

# Add in entries
queue.append('a')
queue.append('b')
queue.append('c')

# The first entry added was 'a' so the first entry that comes out will be 'a'
out = queue.pop(0)
print(out) # 'a'
```

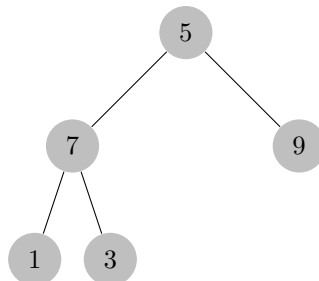
1.4 Priority Queues

Priority queues are similar to regular queues, except all entries are flagged with a specific weight. There are three primary operations:

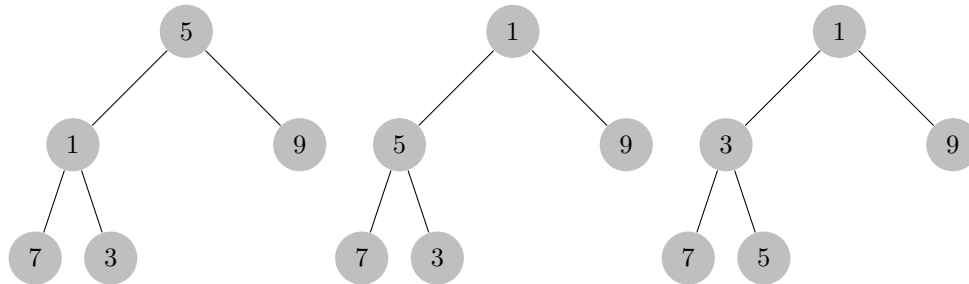
- Inserting an entry and its weight into a priority queue.
- Find the entry with the minimum / maximum weight
- Retrieve the entry with the minimum / maximum weight

1.5 Heaps

A heap is used to implement a priority queue. It takes the form of a binary tree, which can be read as a list. The only other condition for a heap is that the parent is smaller than both children. For example, `[1, 3, 9, 7, 5]` refers to the following tree:



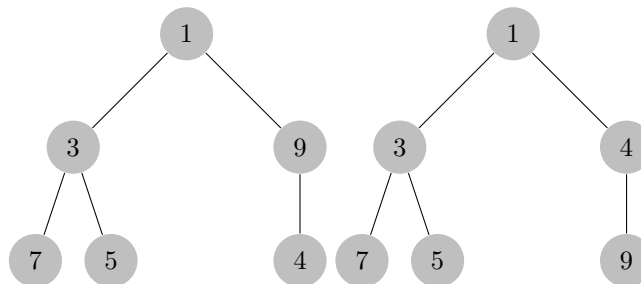
We can denote nodes that do not have any children as **leaves**. To turn this into a heap, we start at the topmost parent, and we see if it is smaller than both its children. Since this is true, we move on to its first child **7**, which is bigger than both its children. As a result, we swap it with the smallest of its two children **1**, and we start over the same process, going back to **5**. Both BFS (see 2.3.1) and DFS (see 2.3.2) can be used. The following illustrates the step by step tree diagram representations of the heap.



We can use Python to verify this:

```
import heapq
heap = [5, 7, 9, 1, 3]
heapq.heapify(heap)
print(heap) # [1, 3, 9, 7, 5]
```

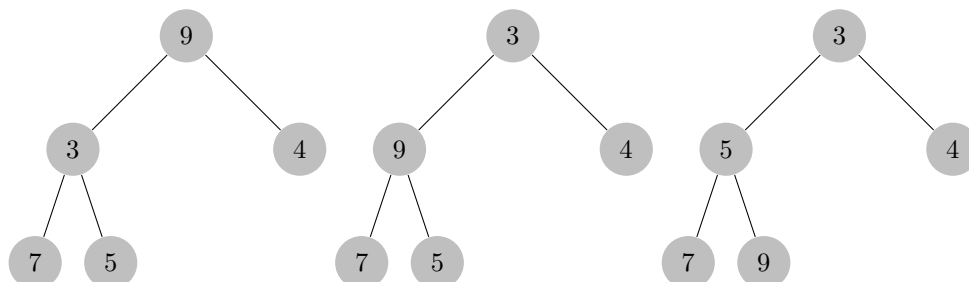
To add an element, we first place it at the next position of the tree (or the last position of the heap). We then “percolate” it up, comparing it with the parent until it is in the right spot. For example, suppose we wish to add **4**. Then the process will look like:



In Python, we can represent this as:

```
heapq.heappush(heap, 4)
print(heap) # [1, 3, 4, 7, 5, 9]
```

Since this is a priority queue, when we remove an element, we remove the smallest element which will always be at the very top. We can accomplish this by swapping this with the last element, and bubbling that element down. For example, suppose we wish to remove the smallest element **1**:



Using Python, we have:

```
heapq.heappop(heap)
print(heap) # [3, 5, 4, 7, 9]
```

1.5.1 Manual Implementation

Using this Python module makes working with heaps trivial and allows us to easily use a list. However, it is also possible to implement this manually with the following mathematical operations. Let the index of an element be i . Then:

- Index of parent: $\text{floor}\left(\frac{i}{2}\right)$
- Index of left child: $2i$
- Index of right child: $2i + 1$

For convention, let $i = 0$ correspond to a `Null` element. The manual implementation of heapify using Python is shown below:

```
# TBA
```

1.5.2 Runtime

If there are n entries, then we will percolate downwards on average $n/2$ times. The time complexity of percolating downwards is $\mathcal{O}(\log n)$, so the time complexity of heapifying a list is $\mathcal{O}(n \log n)$.

However, it is possible to gain a better upper bound. We know that there are at least $n/2$ leaves in the heap. There are $n/4$ nodes at height 1 (let the bottommost layer be $h = 0$). There are $n/8$ nodes at height 2, and so forth. Therefore at a height h , we perform:

$$N_h \propto h \cdot \frac{n}{2^{h+1}}$$

operations such that the total number of operations is proportional to:

$$N = \sum_{h=1}^{\log_2 n} h \cdot \frac{n}{2^{h+1}} \propto n$$

so the total runtime is $\mathcal{O}(n)$.

1.5.3 Heapsort

We can implement a heap to sort a list with a time complexity of $\mathcal{O}(n \log n)$. We perform the following steps:

1. Create a heap from the list.
2. Extract the minimum a total of n times, and placing the elements into a list in the original order.

2 Graph Theory

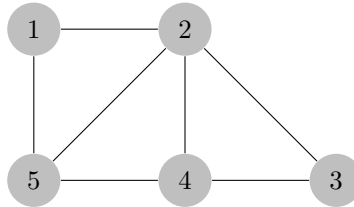
2.1 Background

- We can represent a graph as $G = (V, E)$ where V is the set of nodes $V = \{v_1, v_2, \dots, v_n\}$ and E is the set of edges $E = \{e_1, e_2, \dots, e_m\}$.
- Each edge can be written as $e_k = (v_i, v_j)$.
- Directed graphs, or **digraphs** have edges with directions associated with them.
- Weighted graphs have a weight associated with each edge.
- Vertex v_1 is **adjacent** to vertex v_2 if an edge connects them.
- A *path* is a sequence of vertices in which each vertex is adjacent to the next one. The length of the path is the number of edges in it.
- A cycle in a path is a sequence (v_1, \dots, v_n) such that $(v_i, v_{i+1}) \in E$ and $(v_n, v_1) \in E$.
- A graph with no cycles is known as **acyclic**.

- A directed graph which is acyclic is known as a **DAG**.
- A **simple path** and a **simple cycle** has no repeated vertices.
- Two vertices are **connected** if there is a path between them.
- A subset of vertices is a connected component of a graph G if each pair of vertices in the subset are connected.
- The **degree** of vertex v is the number of edges associated with v .

2.2 Representing Graphs

Suppose we take the following graph:



2.2.1 Adjacency Matrix

For the above graph, the adjacency matrix will look like:

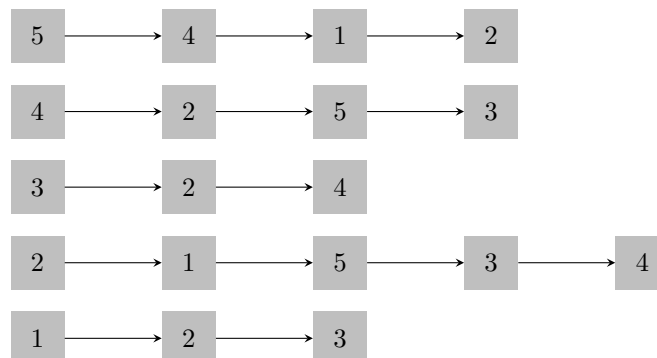
$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \end{matrix}$$

To look up if **1** is connected to **2**, we just need to look up $A[2,1] = A[1,2]$ (or in terms of Python notation, $A[2][1]$ or $A[1][2]$). Note that adjacency matrices are symmetric (e.g. $A^T = A$) for undirected graphs. It has the following complexities:

- Edge lookup: $\mathcal{O}(1)$
- Finding all vertices adjacent to vertex: $\mathcal{O}(|V|)$
- Space complexity: $\mathcal{O}(|V|^2)$

2.2.2 Adjacency List

An adjacency list uses linked lists (see sec. 1.1) to represent the graph:

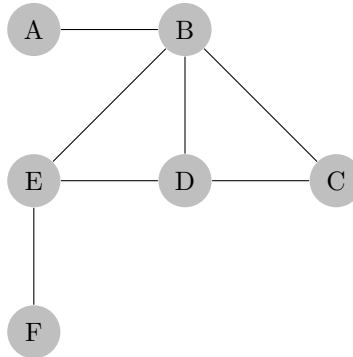


This has the following complexities:

- Edge lookup: $\mathcal{O}(d)$ where d is the maximum degree in the graph
- Finding all vertices adjacent to vertex: $\mathcal{O}(d)$
- Space complexity: $\mathcal{O}(|V| + |E|)$

2.3 Traversal Algorithms

We will apply both DFS and BFS to traverse the following graph:



We can represent this graph using Python:

```

class Node:
    def __init__(self, name):
        self.name = name
        self.connections = []
        self.visited = False

def connect(node1, node2):
    node1.connections.append(node2)
    node2.connections.append(node1)

A = Node("A")
B = Node("B")
C = Node("C")
D = Node("D")
E = Node("E")
F = Node("F")

connect(A,B)
connect(E,F)
connect(B,E)
connect(E,D)
connect(B,D)
connect(B,C)
connect(D,C)
  
```

2.3.1 Breadth First Search

To implement BFS, we use a queue. Suppose we start our search from C . We would explore the nodes in the following order:

$$C \rightarrow B \rightarrow D \rightarrow A \rightarrow E \rightarrow F$$

We visit nodes in the order of the queue and every time a node is visited, the unvisited neighbours of that node is added to the back of the queue. See this Python implementation:

```

def BFS(node):
    q = [node]
    node.visited = True
    while len(q) > 0:
        cur = q.pop(0) # remove q[0] from q and put it in cur
        print(cur.name)
        for con in cur.connections:
            if not con.visited:
  
```

```

        q.append(con)
        con.visited = True
BFS(C) # C B D A E F

```

2.3.2 Depth First Search

In a depth first search, we make use of a stack instead. Nodes are visited in the order of the stack, and every time a node is visited, its unvisited neighbours are added to the stack. Starting from C , the nodes are explored in the following order:

$$C \rightarrow D \rightarrow E \rightarrow F \rightarrow B \rightarrow A$$

which can be shown using Python:

```

def DFS(node):
    q = [node]
    node.visited = True
    while len(q) > 0:
        cur = q.pop() # remove last element from q and put it in cur
        print(cur.name)
        for con in cur.connections:
            if not con.visited:
                q.append(con)
                con.visited = True
DFS(C) # C D E F B A

```

Alternatively, we can use recursion:

```

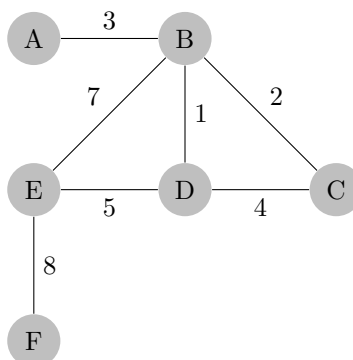
def DFS_rec(node):
    '''Print out the names of all nodes connected to node using a
    recursive version of DFS'''
    print(node.name)
    node.visited = True
    for con in node.connections:
        if not con.visited:
            DFS_rec(con)
DFS_rec(C) # C B A E F D

```

Note that the nodes are traversed in a different order, since the implementation is slightly different. However, it is still DFS.

3 Shortest Path Algorithm

The general problem here is that we want to find the shortest path between two points through a graph. To do this, we can create a weighted graph, such as below:



The shortest path from C to F is $C \rightarrow B \rightarrow D \rightarrow E \rightarrow F$, but there are several paths. A weighted graph can be represented using Python as:

```

class Node:
    def __init__(self, name):
        self.name = name
        self.connections = []
        self.visited = False

def connect(node1, node2, weight):
    node1.connections.append({"node": node2, "weight": weight})
    node2.connections.append({"node": node1, "weight": weight})

A = Node("A")
B = Node("B")
C = Node("C")
D = Node("D")
E = Node("E")
F = Node("F")

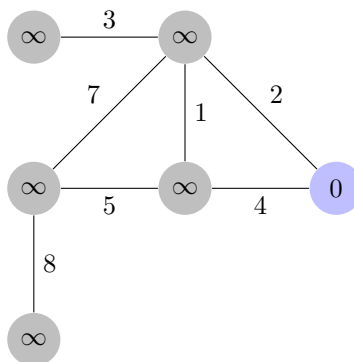
connect(A,B,3)
connect(E,F,8)
connect(B,E,7)
connect(E,D,5)
connect(B,D,1)
connect(B,C,2)
connect(D,C,4)

```

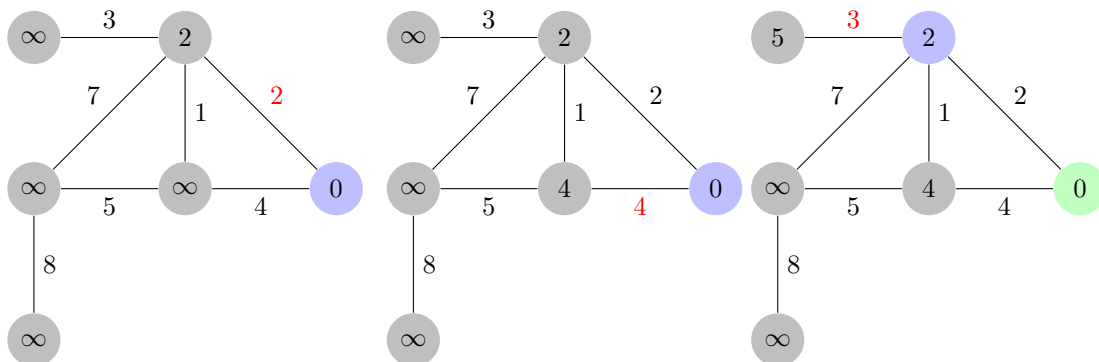
Search functions (e.g. BFS and DFS) can be slightly modified for this slightly different data type.

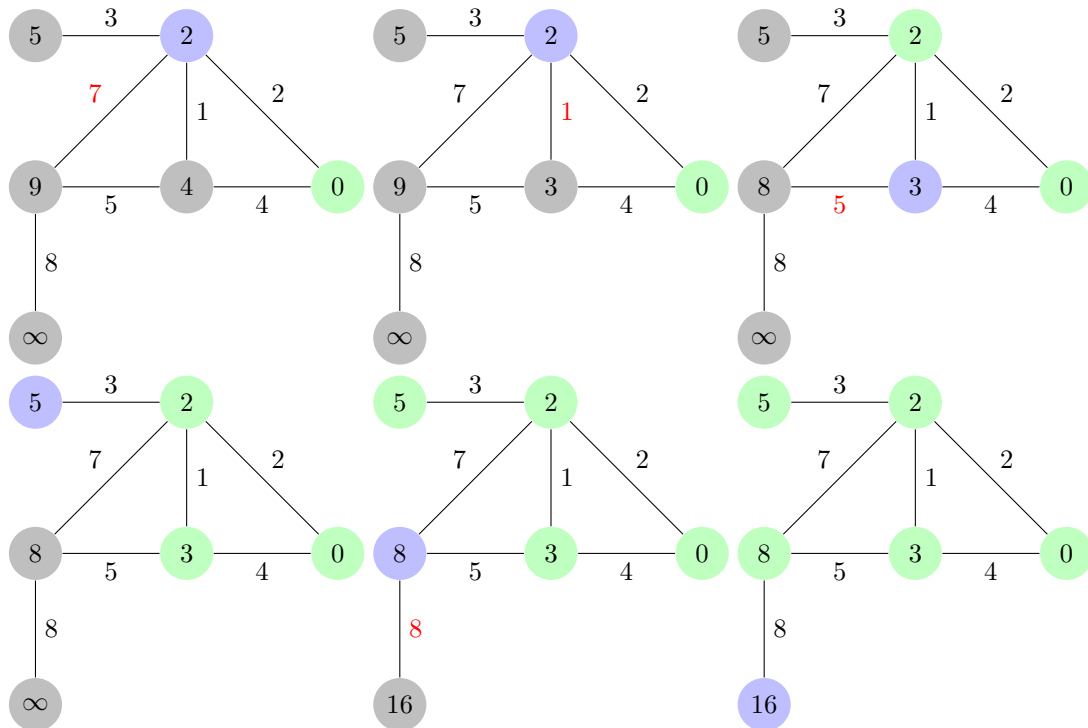
3.1 Dijkstra's Algorithm

The simplest method is to apply Dijkstra's Algorithm. There are several optimizations but we will look at the simplest. The overall idea is to apply to traverse through each unexplored node. For each unexplored node, the shortest distance to each of its unexplored neighbours are set. Initially, the distance from each node to the starting node (except the starting node) is infinite:



Here, a blue node represents the node we are currently exploring and nodes that we have already explored will be drawn as green.





We can show this below via Python:

```
def get_all_nodes(node):
    connections = []

    q = [node]
    node.visited = True
    while len(q) > 0:
        cur = q.pop(0) # remove q[0] from q and put it in cur
        connections.append(cur)
        for con in cur.connections:
            if not con["node"].visited:
                q.append(con["node"])
                con["node"].visited = True

    return connections

def dijkstra(node):

    S = [node] # Visited Nodes
    d = {node.name: 0} # Dictionary of distances to nodes
    prev = {}

    unexplored = get_all_nodes(node) # Unexplored nodes (S + unexplored = all nodes)

    # Set all nodes to infinity
    for n in unexplored:
        if n.name not in d:
            d[n.name] = 999999

    # Continue until everything is explored
    while len(unexplored) > 0:
        cur = unexplored.pop(0)
        for con in cur.connections:
            if con["node"] not in S:
                if con["weight"] + d[cur.name] < d[con["node"].name]:
                    d[con["node"].name] = con["weight"] + d[cur.name]

        S.append(cur)

    return d

print(dijkstra(C)) # {'C': 0, 'B': 2, 'D': 3, 'A': 5, 'E': 8, 'F': 16}
```

To obtain the path, we can implement a dictionary `prev` and whenever the path to a certain node improves, we call: `prev[con["node"].name] = cur.name`.

To add one vertex to S , we must search through all possible vertices so the runtime complexity is $\mathcal{O}(|V|^2)$.

3.2 Using Priority Queue

It is possible to improve the runtime to $\mathcal{O}(|E| \log |V|)$ using a priority queue:

```
import heapq

class Node:
    def __init__(self, name):
        self.name = name
        self.connections = []
        self.visited = False
        self.distance = 999999

    def __lt__(self, other):
        # Comparison function for priority queue
        return self.distance < other.distance

A = Node("A"); B = Node("B"); C = Node("C"); D = Node("D"); E = Node("E"); F = Node("F")

connect(A,B,3); connect(E,F,8); connect(B,E,7); connect(E,D,5)
connect(B,D,1); connect(B,C,2); connect(D,C,4)

def dijkstra_pq(node):
    node.distance = 0

    S = [] # Visited Nodes
    pq = [node]
    heapq.heapify(pq)

    d = {node.name: 0} # Dictionary of distances to nodes

    # Continue until everything is explored
    while len(pq) > 0:
        cur = heapq.heappop(pq)

        if cur in S:
            continue

        d[cur.name] = cur.distance

        for con in cur.connections:
            con["node"].distance = cur.distance + con["weight"]
            heapq.heappush(pq, con["node"])

        S.append(cur)



    return d

print(dijkstra_pq(C)) # {'C': 0, 'B': 2, 'D': 3, 'A': 5, 'E': 8, 'F': 16}
```

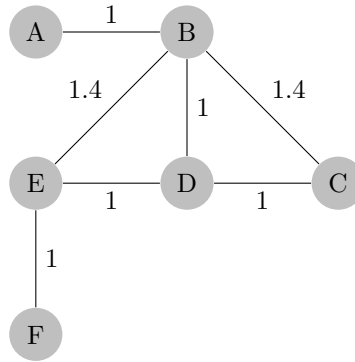
This algorithm is essentially the same, but instead of traversing explored nodes via BFS, we traverse it by sorting the nodes by their distance from the origin. The upper bound on the number of times a node is pushed is $2|E|$ and popping/pushing into `pq` has a time complexity of $\mathcal{O}(\log |V|)$. The time complexity of this algorithm is thus $\mathcal{O}(|E| \log |V|)$.

3.3 Greedy Best First Search

The general theme of search algorithms is to select a way to traverse explored nodes. In the simple method, it was via BFS. In the priority queue method, it was prioritizing nodes with a smaller distance. In the greedy best first search, it is by using a heuristic function $h(v)$ that gives a rough estimate of the distance between a node and the destination.

For example, take the below graph and suppose we wish to move from  to . Unlike the previous examples, the length of each path corresponds to their weights, so we can imagine these as physical points in space. We can let our

heuristic function be the *Manhattan distance* $\Delta x + \Delta y$:



We will start from **C** and look at the neighbours. The neighbour that minimizes the Manhattan distance is **B**, which directly leads to **A**. As you can expect, this is *really bad*. It doesn't even bother checking if going to **D** is faster, but if we have a good heuristic, this isn't necessary. We can implement it in Python below:

```

class Node:
    def __init__(self, name, x, y):
        self.name = name
        self.connections = []
        self.visited = False
        self.x = x
        self.y = y
        self.distance = 999999

def connect(node1, node2, weight):
    node1.connections.append({"node": node2, "weight": weight})
    node2.connections.append({"node": node1, "weight": weight})

A = Node("A", 0, 0)
B = Node("B", 1, 0)
C = Node("C", 2, -1)
D = Node("D", 1, -1)
E = Node("E", 0, -1)
F = Node("F", 0, -2)

connect(A,B,1); connect(E,F,1); connect(E,D,1)
connect(B,D,1); connect(D,C,1); connect(B,C,3); connect(B,E,1.4)

def h(con):
    v = con["node"]
    return abs(v.x)+abs(v.y)

def greedy(source, dest):
    v = source
    S = [] # Visited Nodes
    d = {v.name: 0}

    # Continue until destination is reached
    while not v == dest:
        con = sorted(v.connections, key=h)[0]
        con["node"].distance = v.distance + con["weight"]
        d[con["node"].name] = con["weight"] + d[v.name]

        v = con["node"]

    S.append(v)

    return d

print(greedy(C, A)) # {'C': 0, 'B': 1.4, 'A': 2.4}

```

3.4 A*

The A* algorithm is more robust, and combines the advantages of using a heuristic function with a priority queue. It is extremely similar to the optimized version of dijkstra's, except the priority queue is sorted by the heuristic function. We can also implement it in Python:

```
import heapq

class Node:
    def __init__(self, name, x, y):
        self.name = name
        self.connections = []
        self.visited = False
        self.x = x
        self.y = y
        self.distance = 999999
        self.estimate = abs(self.x) + abs(self.y)

    def __lt__(self, other):
        # Comparison function for priority queue
        return self.estimate < other.estimate

def connect(node1, node2, weight):
    node1.connections.append({"node": node2, "weight": weight})
    node2.connections.append({"node": node1, "weight": weight})

A = Node("A", 0, 0)
B = Node("B", 1, 0)
C = Node("C", 2, -1)
D = Node("D", 1, -1)
E = Node("E", 0, -1)
F = Node("F", 0, -2)

connect(A,B,1); connect(E,F,1); connect(E,D,1)
connect(B,D,1); connect(D,C,1); connect(B,C,1.4); connect(B,E,1.4)

def h(v1, v2):
    return abs(v1.x - v2.x)+abs(v1.y - v2.y)

def greedy(source, dest):
    source.distance = 0

    S = [] # Visited Nodes
    pq = [source]
    heapq.heapify(pq)

    d = {source.name: 0} # Dictionary of distances to nodes

    # Continue until destination is reached
    while len(pq) > 0:
        cur = heapq.heappop(pq)

        if cur in S:
            continue

        d[cur.name] = cur.distance

        for con in cur.connections:
            con["node"].distance = cur.distance + con["weight"]
            con["node"].estimate = h(con["node"], dest) + con["node"].distance
            heapq.heappush(pq, con["node"])

        S.append(cur)

    return d

print(greedy(C, E)) # {'C': 0, 'B': 1.4, 'A': 2.4}
```