# Discrete State Search for Robotics

CS354
Nathan Sprague

# Generic Graph Search Algorithm (without weighted edges)

From our book:

```
Procedure GraphSearch(start, goal)
    OPEN := {start}
    CLOSED := {}
    found := False
    while (OPEN not empty) and (not found)
        Select a node n from OPEN.
        OPEN := OPEN – {n}
        CLOSED := CLOSED U {n}
        if n  ∈ goal then
            found := True
        else
            Let M be the set of all states
            directly accessible from n which
            are not in CLOSED.
            OPEN := OPEN U M
```

Determines the order that states are searched.

Depends on the problem

# Depth First Search

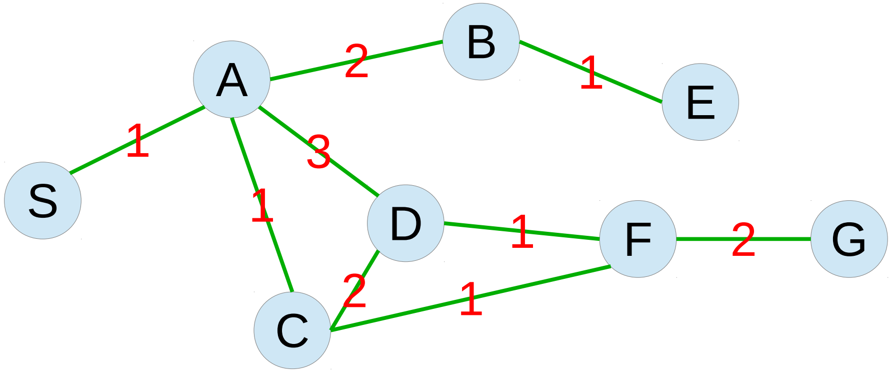
```
Procedure DFS(start, goal)
    OPEN := An empty stack.
    OPEN.push(start)
    CLOSED := {}
    found := False
    while (OPEN not empty) and (not found)
        n = OPEN.pop()
        CLOSED := CLOSED U {n}
        if n  ∈ goal then
            found := True
        else
            for each state m accessible from n
                if m ∉  CLOSED and m ∉  OPEN
                   OPEN.push(m)
```

Determines the order that states are searched.

Depends on the problem

# Depth First Search

```
Procedure DFS(start, goal)
    OPEN := An empty stack.
    OPEN.push(start)
    CLOSED := {}
    found := False
    while (OPEN not empty) and (not found)
        n = OPEN.pop()
        CLOSED := CLOSED U {n}
        if n ∈ goal then
            found := True
        else
            for each state m accessible from n
                if m ∉ CLOSED and m ∉ OPEN
                   OPEN.push(m)
```



# “Correct” version of Figure 6.1

| Chosen | Open | Closed |
|---|---|---|
| - | S | - |
| S | A | S |
| A | B, C, D | A, S |
| D | B, C, F | A, S, D |
| F | B, C, G | A, S, D, F |
| G | B, C | A, S, D, G, F |

# Breadth First Search

```
Procedure BFS(start, goal)
    OPEN := An empty Queue.
    OPEN.enqueue(start)
    CLOSED := {}
    found := False
    while (OPEN not empty) and (not found)
        n = OPEN.dequeue()
        CLOSED := CLOSED U {n}
        if n  ∈ goal then
            found := True
        else
            for each state m accessible from n
                if m ∉  CLOSED and m ∉ OPEN
                   OPEN.enqueue(m)
```

Determines the order that states are searched.

Depends on the problem

# Search Nodes

```
Type SearchNode
    State   state
    Node    parent_node
    Number  path_cost


Function CreateSearchNode(State state, Node parent,
                          number step_cost)
    Return a search node with
        state = state
        parent_node = parent
        path_cost = step_cost + parent.path_cost
```

# Generic Graph Search With Search Nodes

```
Function GraphSearch(start, goal)
    OPEN := { CreateSearchNode(start, NONE, 0) }
    CLOSED := {}
    found := False
    while (OPEN not empty) and (not found)
        Select a search node n from OPEN.
        OPEN := OPEN – {n}
        CLOSED := CLOSED U {n.state}
        if n.state ∈ goal then
            found := True
        else
            Let M be the set of all nodes
            directly accessible from n.state
            which are not in CLOSED.
            OPEN := OPEN U
                 {SearchNode(m, n, cost n->m) | m ∈ M}
    if found
        return a plan created by following
        parent links back from n
    else
        return FAILURE
```

Determines the order that states are searched.

Depends on the problem

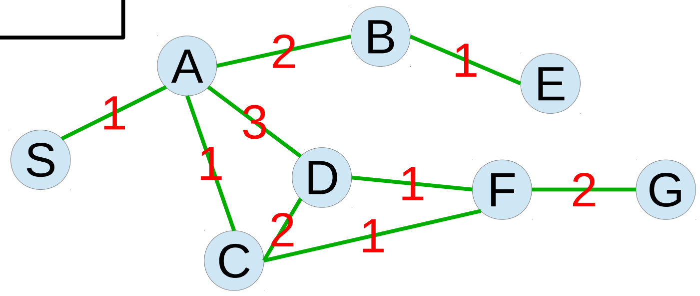
# Dijkstra's Algorithm

```
Procedure Dijkstra(start, goal)
    OPEN := An empty Priority Queue.
    n = SearchNode(start, NONE, 0)
    OPEN.enqueue(n, 0)
    CLOSED := {}
    found := False
    while (OPEN not empty) and (not found)
        n = OPEN.dequeue()
        CLOSED := CLOSED U {n.state}
        if n.state ∈ goal then
            found := True
        else
            for each node m accessible from n.state
                if m ∉ CLOSED and m ∉ any SearchNode in OPEN
                     m_node = SearchNode(m, n, cost of n->m)
                     OPEN.enqueue(m_node, m_node.path_cost)
    if found
        return a plan created by following
        parent links back from n
    else
        return FAILURE
```

(Missing detail: If m is already in a node in OPEN, then that node should be replaced if m_node has a lower cost.)

# Dijkstra's Algorithm

```
Procedure Dijkstra(start, goal)
    OPEN := An empty Priority Queue.
    n = SearchNode(start, NONE, 0)
    OPEN.enqueue(n, 0)
    CLOSED := {}
    found := False
    while (OPEN not empty) and (not found)
        n = OPEN.dequeue()
        CLOSED := CLOSED U {n.state}
        if n.state ∈ goal then
            found := True
        else
            for each node m accessible from n.state
                if m ∉ CLOSED and m ∉ any SearchNode in OPEN
                     m_node = SearchNode(m, n, cost of n->m)
                     OPEN.enqueue(m_node, m_node.path_cost)
    if found
        return a plan created by following
        parent links back from n
    else
        return FAILURE
```



# A*

- Exactly like Djikstra's, Except, priority is calculated as:

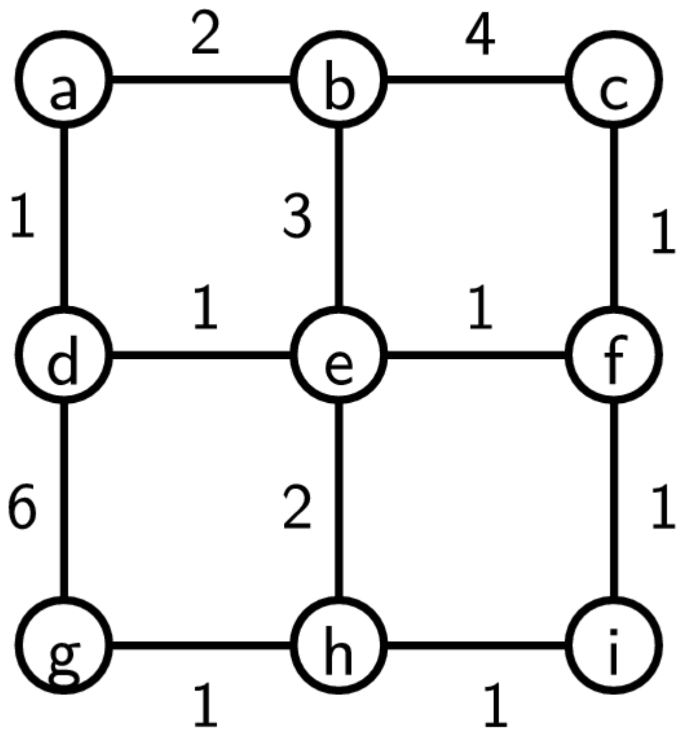
  $f(n) = g(n) + h(n)$

  $g(n) =$ Total path cost to that node

  $h(n) =$ Estimated cost to the goal

- As long as h(n) doesn't overestimate, A* is guaranteed to find an optimal path.

# Exercise



<u>A* Heuristic</u>

h(n) = Minimum number of edges between n and the goal.

For example (assuming the goal is a)
h(g) = 2
h(i) = 4

Since all weights are at least 1, this is guaranteed not to overestimate the path cost.

# Exercise

- Fill out Chosen/Open/Closed tables (like figures 6.1-6.3) and the final path for:
  - DFS: start=g, goal=a
  - BFS: start=g, goal=a
  - Djikstra: start=g, goal=a
  - A*: start=g, goal=a
- All “ties” should be broken by alphabetical order: State 'a' is selected before state 'b'
- For DFS and BFS, the Open column should be formatted as follows:

  (state, parent), e.g. ('d', 'g')
- For Dijkstra, the Open column should be formatted as follows:

  (state, parent, path_cost)
- For A*, the Open column should be formatted as follows:

  (state, parent, path_cost, f(n))