Best-First-Search Pseudocode Explanation (CSP)

Let's break down and analyze BEST-FIRST-SEARCH pseudocode as shown in your textbook. We have two functions there:

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Let's take a look at the first, main, one called BEST-FIRST-SEARCH. It accepts two arguments (think object references, etc.):

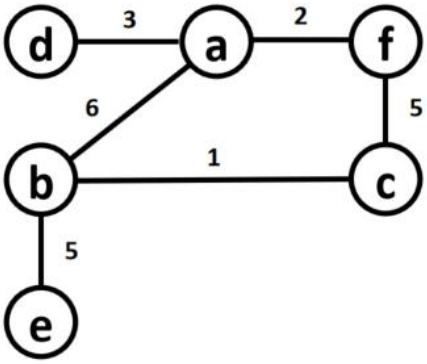
- *problem*, which is some sort of programming language representation of the **search problem**. Recall from your textbook, that it includes:
  - a set of possible states that the environment can be in, called a **state space**,
  - the **initial state**,
  - a set of one or more **goal states**,
  - a set of **actions** for each state in the state space,
  - a **transition model**, which describes what it actions does and what the resulting state is,
  - an **action cost function**, a function that defines what is the cost of an individual action that gets you from state A and B.
- *f*, which is some sort of programming language representation of the **evaluation function** for problem.

Naturally, the pseudocode above does not explicitly specify how to (efficiently) represent those two components in a programming language / as a data structure. This is left to the person implementing the algorithm (the only two structures that are explicitly defined in pseudocode are 'priority queue' and 'lookup table', but we will get to those later). The pseudocode is formulated with flexibility in mind and being programming language-agnostic.

It is entirely up to you how the *problem* will be represented, but there are more and less efficient ways. Let's quote your textbook here: 'The state space can be represented as a **graph** in which the vertices are the states and the directed edges between them are actions'. This is a good start.

By this time you should have a fairly good understanding on what a graph is and how can it be represented. Here's an example weighted **graph** (with edges being undirected, meaning actions are possible in both directions) represented with an **adjacency matrix**.

Weighted Graph G



A graph $G = \{V, E, w\}$**:**

- $N = |V|$ **vertices / nodes, and**
- $|E|$ **edges**
- $|E|$ **weights**

Adjacency Matrix for Graph G

	a	b	c	d	e	f
a	0	6	0	3	0	2
b	6	0	1	0	5	0
c	0	1	0	0	0	5
d	3	0	0	0	0	0
e	0	5	0	0	0	0
f	2	0	5	0	0	0

Adjacency matrix for graph G:

- **a 2D** N **x** N **array with:**
  - **weights indicating an edge / connection between two vertices**
  - **0s where there is no edge / connection between vertices**

Note how the graph itself resembles a map of some locations with edges representing connections between them. In our search problem context, locations would be **states** (labeled a, b, c, d, e, and f) and edges would be **actions**. In addition edge weights can represent individual action costs. For example, the **cost of transitioning** from state f to state c is 5. Adjacency matrix is a reliable structure that keeps all that information in one place and individual pieces of data can be extracted by traversing / accessing it.

So:

- an entire **state space** could be encoded using an adjacency matrix,
- a **set of possible actions** for a state is represented by all non-zero elements in a row (or column) corresponding to some state (for example for state f, possible actions are: toA, toC). Extracting that set is just a matter of correctly reading from adjacency matrix),
- similarly, a **transition model** is already 'built in' into the adjacency matrix,
- finally, **action cost function** is a mechanism of reading the cost (weight) corresponding to two states x and y (think: row and column) from the corresponding cell in the adjacency matrix.

In other word, a significant portion of the problem can be encoded using a graph represented with an adjacency matrix. You can think of search tree starting (rooted at) **initial state** as being overlaid on top of the graph to represent the search sequence.

Frequently, node labels will be replaced with indices (adjacency matrix row and column indices), but that is not always necessary.

In order to create a flexible and extendable *problem* representation in a programming language of your choice, object-oriented approach (where the entire *problem* is one single object) is appropriate, but not necessary to solve the problem.

Similarly, the evaluation function f, could be represented as an object, but it might be a simple function which makes the evaluation give the entire *problem* and **current state**. You have a lot of flexibility here, but your design choices will have numerous consequences: performance, debugging issues, etc.

Let's go back to the BEST-FIRST-SEARCH function pseudocode.

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

This function can return two results:

- a solution node, representing the final search state (it could be different than goal state if the algorithm is not optimal), or
- a *failure*, indicating that a goal state cannot be found if we started at the initial state.

Note that the term *node* in this pseudocode (according to your textbook) is a data structure that holds (encapsulates) the following information:

- node.STATE: the state to which the node belongs,
- node.PARENT: the node in the tree that **generated** that node,
- node.ACTION: the action that was applied to the PARENT's state to generate this node,
- node.PATH-COST: the **total cost** of the path from the initial state to this node.

Going backward from the solution node and using PARENT pointers can recreate the entire path from the initial node to the solution node. This would be a **linked-list** data structure holding the path.

The first step in the BEST-FIRST-SEARCH algorithm is setting up our starting point, the first node that will be looked at:

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

Here, the algorithm uses the function NODE (not shown / described here) to **generate** a search tree node from based on the initial state of the *problem* (*problem*.INITIAL). It means creating an object (or some other representation that contains information that all nodes should contain (see above):

- node.STATE: the state chosen to be the **initial state**,
- node.PARENT: this is our starting point, so there is no PARENT value (think: null),
- node.ACTION: this is our starting point, so there is no ACTION value (think: null),
- node.PATH-COST: this is our starting point, our **total cost** is zero.

Variable *node* will keep track of the **currently examined** search tree node.

Once this starting / search tree root *node* is set up it is being added to the *frontier* **priority queue**, where all nodes to be expanded / "looked at" are stored in order of their "value" to the search algorithm (as defined by **evaluation function** *f*).

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

The *frontier* now has a single element (the search tree root node, starting point) inside of it. It will be the first to analyze later. The "creation" of *node* means also that we **reached** that node. In order to avoid examining this node again in the future (and to avoid potential loops), let's add it to the *reached* table (pseudocode suggests a lookup table: a dictionary, a hash table or a similar data structure. The key would be the state label and the value the node itself as an object):

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

Now, we are ready to do the search. The search here is an iterative process that continues while there are nodes to examine left (in other words, there are elements left in the priority queue *frontier*; nodes that were not examined and expanded yet).

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

Recall, that the nodes to be examined / expanded (or re-examined) are stored in the *frontier* priority queue (prioritized according to their evaluation function f value). Meaning that the first element in *frontier* structure will always be the most "interesting" ("valuable") to examine. POPping that node from the queue means removing it from it to be examined (POP is going to be a native function of a priority queue data structure). The algorithm will now POP the next node of interest off the queue and make it the **current** *node*.

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

If the currently examined *node* happens to be the **goal state**, we return it and the search is over.

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

If it is not, we need to expand it (look at **all** its "children": all the *child* states with label s connected to the *node*.STATE in our state space graph) using the EXPAND function (see below):

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

Note that we are only interested in children states s that

- EITHER have NOT been reached (not in the *reached* lookup table) yet,
- OR were **reached before**, but now they were reached through a path with **lower** PATH-COST than previously.

All the children states of interest with label s of interest should be marked (or their entry in *reached* lookup table updated) as reached and added to *frontier* priority queue as nodes of interest:

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

If the goal state can be found it will be found and we will eventually reach it. If it is not reached, but the *frontier* queue ends up being empty (we have seen them all!), it means the goal state is not reachable and we need to report *failure*:

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

Let's take a look at the EXPAND function now:

```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

It accepts two arguments (think: object references, etc.):

- *problem*: see above,
- *node*: a node to analyze and expand.

And returns **a list of nodes** that are generated through the process of expansion.

```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

The **structure representing a list of nodes will have to created first** here. Linked list is a viable choice.

First it will need to extract the state label s from the *node* argument object:

```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Next, it will need to iterate through all possible actions (stored here in the *problem* representation) for a state with label s).

```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

For each action, the following steps will be taken:

- a **transition model** is used to decide what the resulting state will be after applying *action* to state with label *s*:

```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

- the action cost function will be used to determine the path cost to state with label *s* through state with label *s'* after applying *action*:

```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

- now, when we have all the information for a new node structure / object, we can create it using the NODE function and add it to the list of nodes (**yield**):

```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Eventually, after all *action*s are iterated through, the nodes list is ready and can be returned back to the BEST-FIRST-SEARCH function.