Search

Let's explore a particular (example) implementation of breadth-first search. The approach here uses an agenda of future nodes or paths to try. We keep track of nodes already seen, so we don't re-visit unsuccessful nodes.

A little later, we will reimplement our search using different queue abstractions, and then experiment with those to see the difference between breadth-first and depth-first search.

```python
from table import notebook_table #visualization

def search(start, is_goal, successors):
    """ Search for and return a node satisfying a goal

    start: the starting node
    is_goal(node): returns True if node satisfies the goal
    successors(node): a sequence of successor nodes to node

    Uses a list to keep track of an agenda of nodes to try
    ""
    agenda = [start]

    seen = {start}
    print_in_table = notebook_table('seen', 'agenda', 'node ='
    while agenda:
        print_in_table(seen)
        print_in_table(agenda)
        node = agenda.pop(0)
        print_in_table(node)
        if is_goal(node):
            return node
        for s in successors(node):
            if s not in seen:
                agenda.append(s)
                seen.add(s)

# Search for a node that has a particular value
#
def example_1():
    def is_goal(node):
        return node == 5
    def successors(node):
        if node < 100:
            return [node+1, node+3] #try different things here...
        return []

    start = 0
    res = search(start, is_goal, successors)
    notebook_table.display()
    print("search: start =", start, "; result =", res)

def example_1()
```
def example_2():
    x = 3;
    y = 4
    def is_goal(node):
        return node != 0 and node % x == 0 and node % y == 0
    def successors(node):
        if node < 100:
            return [node+1] # try different things here
    return []
    start = 0
    res = search(start, is_goal, successors)
    notebook_table.display()
    print("search: start =", start, "; result =", res)

def search(start, is_goal, successors):
    """ Search for a node that satisfies a goal. ""
    Uses a queue, implemented as a group of functions:
    make_queue, queue_empty, queue_next, queue_add
    """
    agenda = make_queue(start) ##
    seen = {start}
    print_in_table = notebook_table('seen', 'queue_elts(agenda)', 'node =')
    while not queue_empty(agenda):
        ##
        print_in_table(seen)
        print_in_table(queue_elts(agenda)) ##
        node = queue_next(agenda) ##
        print_in_table(node)
        if is_goal(node):
            return node
        for s in successors(node):
            if s not in seen:
                queue_add(agenda, s) ##
                seen.add(s)
Change implementation of queue
In [ ]: # FIFO queue as a dictionary
# We'll fill items in the dict with an integer
# index as key and the element as value, with
# the smallest index being the oldest. Will `del`
# dict entry once returned.
#
def make_queue(e):
    return {'oldest': 0,
            'newest': 0,
            0: e}

def queue_empty(q):
    return q['newest']-q['oldest'] < 0

def queue_add(q, elt):
    q[q['newest']+1] = elt
    q['newest'] += 1

def queue_elts(q):
    return [q[pos] for pos in range(q['oldest'], q['newest']+1)]

def fifo_queue_next(q):
    """ FIFO -- First In, First Out: pull from oldest end of queue """
    c = q['oldest']
    val = q[c]
    del q[c]
    q['oldest'] += 1
    return val

def lifo_queue_next(q):
    """ LIFO -- Last In, First Out: pull from newest end of queue """
    c = q['newest']
    val = q[c]
    del q[c]
    q['newest'] -= 1
    return val

queue_next = fifo_queue_next
#queue_next = lifo_queue_next

In [ ]: example_1()

A message-passing queue implementation (using closures!)

In [ ]: # Example of this 'message-passing' interface:
def test_dict_queue():
    q = make_queue(1, lifo=True)
    for e in [2, 3, 4, 2]:
        q('add', e)
    while not q('empty'):
        print("q elts:", q('elts'), "; next:", q('next'))
New search implementation using message-passing queue
**Example directed graph**

```
def search(start, is_goal, successors, dfs=False):
    """ Search for a node that satisfies a goal. """

    Internal use of a message-passing queue:
    q = make_queue(start, lifo=False)  # fifo by default
    q('empty'), q('next'), q('add', elt), q('elts')
    """

    agenda = make_queue(start, lifo=dfs)
    seen = {start}
    print_in_table = notebook_table('seen', 'agenda("elts")', 'node =')
    while not agenda('empty'):
        print_in_table(seen)
        print_in_table(agenda('elts'))
        node = agenda('next')
        print_in_table(node)
        if is_goal(node):
            return node
        for s in successors(node):
            if s not in seen:
                agenda('add', s)
                seen.add(s)
```

```
in [ ]: example_1()

in [ ]: example_2()

```

```python
def graph1:
    'root': [13, ['A', 'B', 'C']],
    'A': [77, ['D', 'E']],
    'B': [42, []],
    'C': [0, ['G']],
    'D': [-32, ['F']],
    'E': [42, ['F']],
    'F': [215, []],
    'G': [8, []],
```
Consider a search_path capability to find a path to a node that satisfies a goal.

```python
In [ ]: def search_path(start, is_goal, successors, dfs=False):
   """ Search for a path that satisfies a goal.

   start: the starting node
   is_goal(node): returns True if node satisfies the goal
   successors(node): a sequence of successor nodes to node
   """
   agenda = make_queue(start, lifo=dfs)
   seen = {start}
   print_in_table = notebook_table('seen', 'agenda("elts")', 'node =')
   while not agenda('empty'):
       print_in_table(seen)
       node = agenda('next')
       print_in_table(node)
       if is_goal(node):
           return node
       for s in successors(node):
           if s not in seen:
               agenda('add', s)
               seen.add(s)
```

```python
In [ ]: example_3(dfs=False)
```

```python
In [ ]: example_3(dfs=True)
```
An alternative search_paths

The search_paths function above is somewhat inefficient, in that it does a lot of copying of tuples to create new paths. An alternative is to create "nested" paths during the search, e.g., (s, path) rather than path + (s,), and then convert the nested result back to the "flat" path format once we've found a successful path. That alternative is left as an exercise for the reader.

What if we want all paths?

Our code above only finds one path to a node that satisfies the goal. How would we gather all paths? Assume that there are no cycles in the graph.

```python
# Search for all paths whose end satisfies a goal.
#
def search_all_paths(start, is_goal, successors, dfs=False):
    pass
```

```python
def example_5(dfs=False):
    start = 'root'

def is_goal(node):
    #return graph1[node][0] == 42 # node with value
    return graph1[node][0] > 0 and graph1[node][0] % 2 == 0 # node with positive even

def successors(node):
    return graph1[node][1]

res = search_all_paths(start, is_goal, successors, dfs=dfs)
notebook_table.display()
print('result: ', res)
```
example_5(dfs=False)

example_5(dfs=True)