Queues, Deques

Hsuan-Tien Lin

Dept. of CSIE, NTU

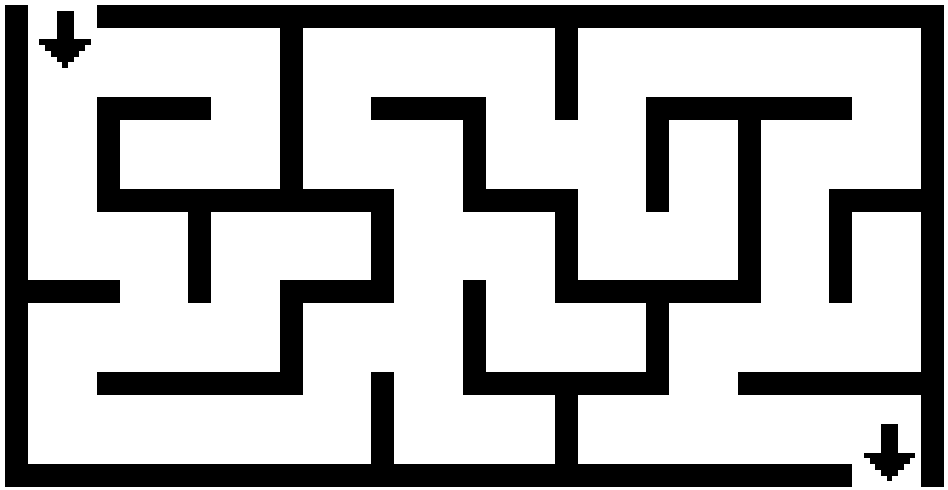
March 31, 2014

What We Have Done

algorithm	data structure
sequential search	array or (linked list)
selection sort	array or (linked list)
insertion sort	array or (linked list)
binary search	ordered array
polynomial “merge”	sparse array on array or (linked list)
parenthesis matching	stack
postfix evaluation	stack
infix to postifix	stack

next: another algorithm with stack (and more)

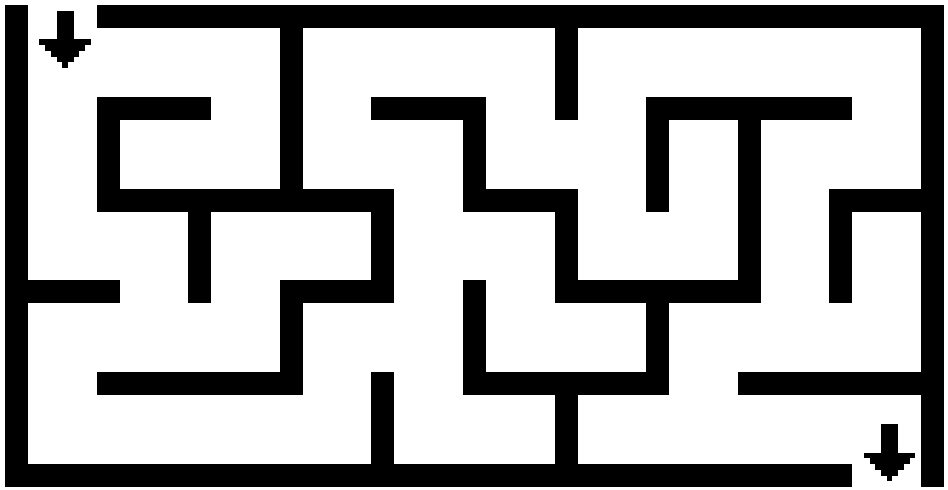
The Maze Problem



http://commons.wikimedia.org/wiki/File:Maze01-01.png

given a (2D) maze, is there a way out?

The Maze Problem



http://commons.wikimedia.org/wiki/File:Maze01-01.png

given a (2D) maze, is there a way out?

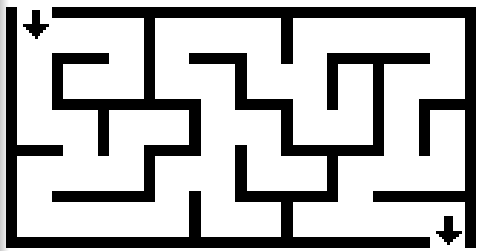
Recursive Algorithm

GET-OUT-RECURSIVE(m, $(0, 0)$)

Getting Out of Maze Recursively

GET-OUT-RECURSIVE(Maze m, Postion (i, j))

```
mark (i, j) as visited
for each unmarked (k, ℓ) reachable from (i, j) do
    if (k, ℓ) is an exit
        return TRUE
    end if
    if GET-OUT-RECURSIVE(m, (k, ℓ))
        return TRUE
    end if
end for
return FALSE
```



Recursion (Reading Assignment: Section 3.5)

- a function call to itself
- be ware of terminating conditions
- can represent programming intentions clearly
- at the expense of "space" (why?)

Recursion (Reading Assignment: Section 3.5)

- a function call to itself
- be ware of terminating conditions
- can represent programming intentions clearly
- at the expense of "space" (why?)

Recursion (Reading Assignment: Section 3.5)

- a function call to itself
- be ware of terminating conditions
- can represent programming intentions clearly
- at the expense of "space" (why?)

Recursion (Reading Assignment: Section 3.5)

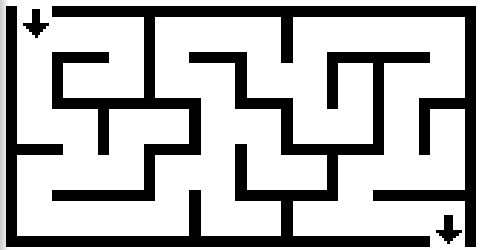
- a function call to itself
- be ware of terminating conditions
- can represent programming intentions clearly
- at the expense of "space" (why?)

From Recursion to Stack

Getting Out of Maze by Stack

GET-OUT-STACK(Maze m, Postion (i, j))

```
while stack not empty do
    (i, j) ← pop from stack
    mark (i, j) as visited
    for each unmarked (k, ℓ) reachable from (i, j) do
        if (k, ℓ) is an exit
            return TRUE
        end if
        push (k, ℓ) to stack [and mark (k, ℓ) as todo]
    end for
end while
return FALSE
```



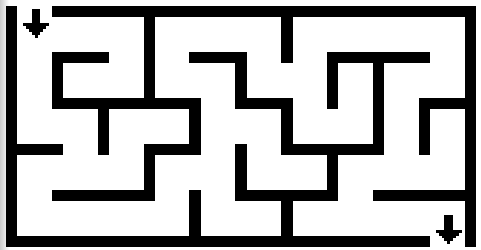
- similar result to recursive version, but conceptually different
  - recursive: one path on the system stack
  - stack: many positions-to-be-explored on the user stack

From Recursion to Stack

Getting Out of Maze by Stack

GET-OUT-STACK(Maze m, Postion (i, j))

```
while stack not empty do
    (i, j) ← pop from stack
    mark (i, j) as visited
    for each unmarked (k, ℓ) reachable from (i, j) do
        if (k, ℓ) is an exit
            return TRUE
        end if
        push (k, ℓ) to stack [and mark (k, ℓ) as todo]
    end for
end while
return FALSE
```



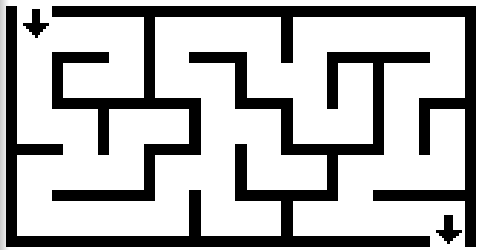
- similar result to recursive version, but conceptually different
  - recursive: one path on the system stack
  - stack: many positions-to-be-explored on the user stack

A General Maze Algorithm

Getting Out of Maze by Container

GET-OUT-CONTAINER(Maze m, Postion (i, j))

```
while container not empty do
    (i, j) ← remove from container
    mark (i, j) as visited
    for each unmarked (k, ℓ) reachable from (i, j) do
        if (k, ℓ) is an exit
            return TRUE
        end if
        insert (k, ℓ) to container [and mark (k, ℓ) as todo]
    end for
end while
return FALSE
```



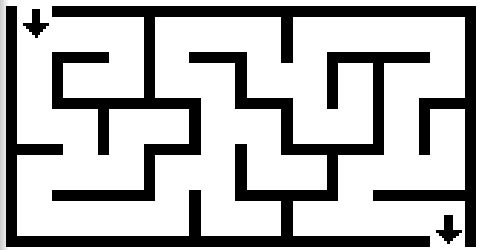
- if "random" remove from container: "random walk" to exit

A General Maze Algorithm

Getting Out of Maze by Container

GET-OUT-CONTAINER(Maze m, Postion (i, j))

```
while container not empty do
    (i, j) ← remove from container
    mark (i, j) as visited
    for each unmarked (k, ℓ) reachable from (i, j) do
        if (k, ℓ) is an exit
            return TRUE
        end if
        insert (k, ℓ) to container [and mark (k, ℓ) as todo]
    end for
end while
return FALSE
```



- if "random" remove from container: "random walk" to exit

Queues

Queue

- object: a container that holds some elements
- action: [constant-time] enqueue (to the rear), dequeue (from the front)

- first-in-first-out (FIFO): 買票，印表機
- also very restricted data structure, but also important for computers

Queues

Queue

- object: a container that holds some elements
- action: [constant-time] enqueue (to the rear), dequeue (from the front)

- first-in-first-out (FIFO): 買票，印表機
- also very restricted data structure, but also important for computers

Queues Implemented on Circular Array (5.2.4)

Reading Assignment

be sure to go ask the TAs or me if you are still confused

Queues Implemented on Circular List (5.2.5)

Reading Assignment

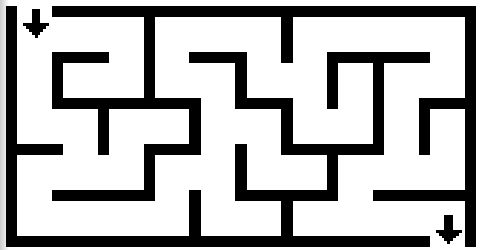
be sure to go ask the TAs or me if you are still confused

Maze From Stack to Queue

Getting Out of Maze by Queue

GET-OUT-QUEUE(Maze m, Postion (i, j))

```
while queue not empty do
    (i, j) ← dequeue from queue
    mark (i, j) as visited
    for each unmarked (k, ℓ) reachable from (i, j) do
        if (k, ℓ) is an exit
            return TRUE
        end if
        enqueue (k, ℓ) to queue [and mark (k, ℓ) as todo]
    end for
end while
return FALSE
```



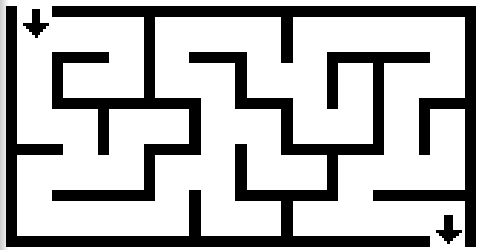
- use of stack/queue: store the yet-to-be-explored positions
- stack version : first (lexicographically) way out (explore deeply)
- queue version : shortest way out (explore broadly)

Maze From Stack to Queue

Getting Out of Maze by Queue

GET-OUT-QUEUE(Maze m, Postion (i, j))

```
while queue not empty do
    (i, j) ← dequeue from queue
    mark (i, j) as visited
    for each unmarked (k, ℓ) reachable from (i, j) do
        if (k, ℓ) is an exit
            return TRUE
        end if
        enqueue (k, ℓ) to queue [and mark (k, ℓ) as todo]
    end for
end while
return FALSE
```



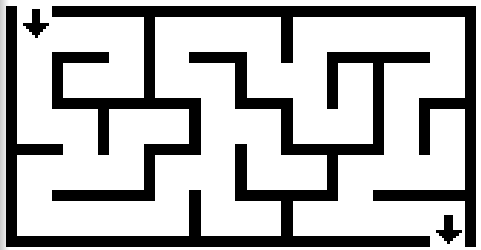
- use of stack/queue: store the yet-to-be-explored positions
- stack version : first (lexicographically) way out (explore deeply)
- queue version : shortest way out (explore broadly)

Maze From Stack to Queue

Getting Out of Maze by Queue

GET-OUT-QUEUE(Maze m, Postion (i, j))

```
while queue not empty do
    (i, j) ← dequeue from queue
    mark (i, j) as visited
    for each unmarked (k, ℓ) reachable from (i, j) do
        if (k, ℓ) is an exit
            return TRUE
        end if
        enqueue (k, ℓ) to queue [and mark (k, ℓ) as todo]
    end for
end while
return FALSE
```



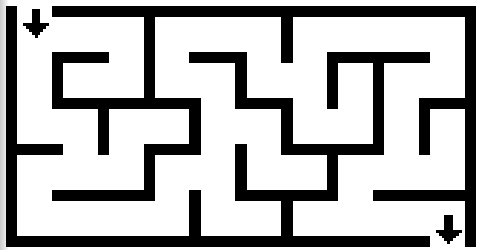
- use of stack/queue: store the yet-to-be-explored positions
- stack version : first (lexicographically) way out (explore deeply)
- queue version : shortest way out (explore broadly)

Maze From Stack to Queue

Getting Out of Maze by Queue

GET-OUT-QUEUE(Maze m, Postion (i, j))

```
while queue not empty do
    (i, j) ← dequeue from queue
    mark (i, j) as visited
    for each unmarked (k, ℓ) reachable from (i, j) do
        if (k, ℓ) is an exit
            return TRUE
        end if
        enqueue (k, ℓ) to queue [and mark (k, ℓ) as todo]
    end for
end while
return FALSE
```



- use of stack/queue: store the yet-to-be-explored positions
- stack version : first (lexicographically) way out (explore deeply)
- queue version : shortest way out (explore broadly)

Deques

Deque = Stack + Queue + push_front

- object: a container that holds some elements
- action: [constant-time] push_back (like push and enqueue), pop_back (like pop), pop_front (like dequeue), push_front

- application: job scheduling

Deques Implemented on Doubly-linked List (5.3.2)

Reading Assignment

be sure to go ask the TAs or me if you are still confused

Some Useful Implementations in C++

Standard Template Library (STL)

- container `vector`: dynamically growing dense array
- container `list`: doubly-linked list
- container `deque`: "chunked" linked-list implementation of deque
- container adapter `stack`: turning some container to a stack

```
1 template <typename T, typename Container = deque<T> >
2 class stack;
```

- container adapter `queue`: turning some container to a queue

```
1 template <typename T, typename Container = deque<T> >
2 class queue;
```

Some Useful Implementations in C++

Standard Template Library (STL)

- container `vector`: dynamically growing dense array
- container `list`: doubly-linked list
- container `deque`: "chunked" linked-list implementation of deque
- container adapter `stack`: turning some container to a stack

```
1 template <typename T, typename Container = deque<T> >
2 class stack;
```

- container adapter `queue`: turning some container to a queue

```
1 template <typename T, typename Container = deque<T> >
2 class queue;
```

Some Useful Implementations in C++

Standard Template Library (STL)

- container `vector`: dynamically growing dense array
- container `list`: doubly-linked list
- container `deque`: "chunked" linked-list implementation of deque
- container adapter `stack`: turning some container to a stack

```
1 template <typename T, typename Container = deque<T> >
2 class stack;
```

- container adapter `queue`: turning some container to a queue

```
1 template <typename T, typename Container = deque<T> >
2 class queue;
```

Some Useful Implementations in C++

Standard Template Library (STL)

- container `vector`: dynamically growing dense array
- container `list`: doubly-linked list
- container `deque`: "chunked" linked-list implementation of deque
- container adapter `stack`: turning some container to a stack

```
1 template <typename T, typename Container = deque<T> >
2 class stack;
```

- container adapter `queue`: turning some container to a queue

```
1 template <typename T, typename Container = deque<T> >
2 class queue;
```

Some Useful Implementations in C++

```
1   #include <vector>
2   #include <stack>
3   #include <queue>
4   using namespace std;
5   vector<int> intarray;
6   stack<char, vector<char> > charstackonvector;
7   queue<double> doublequeue;
8   intarray.resize(20); intarray[3] = 5;
9   charstack.push_back('(');
10  char c = charstack.pop_back();
11  doublequeue.push_back(3.14);
12  double d = doublequeue.pop_front();
```