**Battleplan**

- Correctness for Dijkstra
- More Reachability
- Breadth-first-search
- Depth-first-search

**More on Reachability**

Have to compute

\[ R(s) = \{ v \in V \mid \text{exists path } s \text{ to } v \}. \]

- Inductive approach: Build set \( R \) in stages.
  
  \[ \{ s \} = R_0 \subseteq R_1 \subseteq R_2 \subseteq \ldots \subseteq R_k = R(s) \]

Edge \((u, v)\) requires attention iff \( u \in R_i \) but \( v \notin R_i \).

Prototype algorithm:

\[ R = \{ s \}; \]

while( some edge \((u,v)\) requires attention )

add \( v \) to \( R \);

**Example: Games**

Consider some simple combinatorial puzzle such as placing Pentominoes.
- Given 12 pentominoes (5 unit squares each, 60 total).
- Place them on a chess board so that the center 4 squares are free.

Of course, there are no gaps, no overlaps, no saw is used, …

Surprisingly hard, there are just too many ways to place the pieces.

- Graph formulation:
  
  \( V \): All possible configurations of placements of some of the pentominoes, leaving 4 center squares free.
  
  \( E \): \((x, y)\) means that configuration \( x \) can be extended to configuration \( y \) by adding one more pentomino.

**Comments**

- Goal: Find a path from the empty configuration \( C_0 \), to any configuration with all 12 pieces.
  
  Or, show that there is no such path.

  Might also want to generate a list of some (many) of the solutions.

  Note: we are dealing with a DAG, there are no cycles.

  Also, length is irrelevant, all good paths will have length 12.

  But, the DAG is enormous. We do not want to explicitly have to write down all configurations.

  If we could, we would already have a solution anyways.

  The general problem of exponential size search spaces.
Depth First Search (DFS)

Ignore size problems for the moment, let’s just assume plain adjacency list implementation.

> Idea: Use stack to keep track of vertices in need of exploration.

```java
explore_stack( vertex s )
{
    S.push( s );
    mark s;     // put x into R

    while( !S.empty() ) {
        x = S.pop();
        forall (x,y) in E do
            if( y not marked ) {
                // requires attention
                S.push(y);
                mark y;     // put y into R
            }
    }
}
```

Upon completion of a call `dfs(s)`, exactly the vertices in `R(s)` are marked.

Recursive Version

Can avoid implementing stack explicitly: recursion.

```java
dfs( vertex x )
{
    mark x;
    for all (x,y) in E do
        if( y not marked ) dfs( y );
}
```

Running time: $O(n + m)$.

Additional space requirement: $n + O(n)$ for markings (array of size $n$) and recursion stack.

Implementation

For markings, use array of integers, say, initialized to 0.

```java
vector<int> mark(n,0);
cnt = 1;
...
if( mark[y] == 0 )
    mark[y] = cnt++;    // enter y into R
...
```

Upon completion of a call `dfs(s)`, the cardinality of $R(s)$ is $cnt$.

If necessary, could also add reached vertices to a container $R$.

`mark[x]` is called a DFS number (time stamp).

Breadth First Search (BFS)

Alternatively, use a queue (instead of stack/recursion):

```java
bfs( vertex s )
{
    Q.push( s );
    mark s;          // put x into R

    while( !Q.empty() ) {
        x = Q.pop();
        forall (x,y) in E do
            if( y not marked ) {
                Q.push(y);
                mark y;     // put y into R
            }
    }
}
```
Example

Graph: vertices: 10, edges: 17
1: 4 2 3
2: 4
3: 7 6
4: 3 5 7
5: 2
6: 9
7: 6 9 8
8: 10 5
9: 10
10:

reachable nodes (DFS numbers)
1: 1 2 3 4 5 6 7 8 9 10
2: 1 10 3 2 9 5 4 8 6 7
3: 0 2 4 3 1 6 5 9 7 8
4: 10:
5: 0 2 4 3 1 6 5 9 7 8
6: 9
7: 6 9 8
8: 10 5
9: 10
10:

reverse the adjacency lists
1: 1 9 2 10 8 3 6 7 4 5
2: 4
3: 7 6
4: 3 5 7
5: 2
6: 9
7: 6 9 8
8: 10 5
9: 10
10:

Spanning Trees

Consider all edges receiving attention ("traversed edges") in an exploration algorithm.

Can never traverse both \((u, v)\) and \((u', v): v \text{ already in } R\) after the first edge receives attention.

All edges lie on paths with source \(s\).

Hence the traversed edges form a tree with root \(s\).

The root is always \(s\), and the nodes in the tree are \(R(s)\), but the shape of the tree depends on the algorithm.

These trees are very important for other algorithms (e.g., finding strongly connected components).
Pop Quiz

What spanning trees would be produced by DFS and BFS on the following graph?

![Graph Diagram]

What’s the Difference?

DFS and BFS both compute $R(s)$, but in some applications there is no choice.

- Consider a game graph $G$, for example:
  - vertices: all possible chess configurations.
  - edges: legitimate chess moves.
  Cannot use DFS, need (truncated) form of BFS.

- Strongly connected component ($C \subseteq V$ where everybody can reach everybody, maximal such).
  Clever algorithm by Tarjan based on DFS, uses DFS numbers.

More on Shortest Paths

Suppose $t \in R(s)$.
If we assign a cost of 1 to each edge, we can still ask what is a shortest path from $s$ to $t$?

- Compute distance from $s$ to $v \in R(s)$:
  $\delta(v) =$ length of shortest path from $s$ to $v$.

- Determine actual path using predecessors.
  For each $v \in R(s)$ compute $\pi(v)$ so that $\pi(v)$ lies immediately before $v$ on a shortest path from $s$ to $v$ (say, $\pi(s) = s$).

Given predecessors $\pi(v)$ we can build path backwards:
$v, \pi(v), \pi(\pi(v)), \ldots, s$.

Augmented BFS

```plaintext
Q.push( s );
mark s;
dist[s] = 0;
pi[s] = s;

while( !Q.empty() )
{
x = Q.pop();
forall (x,y) in E do
  if( y not marked )
  {
    Q.push(y);
    mark y;
    dist[y] = dist[x] + 1;
    pi[y] = x;
  }
}
```
Correctness

Claim: Augmented BFS correctly computes the distances of predecessors for all \( v \in R(s) \).

Note that values for \( \text{dist}(v) \) and \( \pi(v) \) are set only once, then no other changes are made.

(As opposed to getting better and better approximations.)

Step 1: \( \text{dist}(v) \geq \delta(v) \).

Step 2: Let \( Q = (v_1, \ldots, v_k) \) be a snapshot of the queue.
Then
\[
\text{dist}(v_i) \leq \text{dist}(v_{i+1}) \quad \text{and} \\
\text{dist}(v_k) \leq \text{dist}(v_1) + 1.
\]

Step 3: Consider
\[ R_k := \{ v \in R(s) \mid \delta(v) = k \} \]
Show by induction on \( k \) that BFS works properly.

Clear for \( k = 0 \).

\( k > 0 \): By 2, vertices are inserted into queue in increasing \( \text{dist} \)-order.
By 1 and IH, must encounter \( v \in R_k \) after all \( u \in R_{i}, i < k \).
Consider first \( u \in R_{k-1} \) ever entered such that \( (u, v) \in E \).
When \( u \) is popped, \( v \) is still unmarked.
But then \( \text{dist}(v) = \text{dist}(u) + 1 = \delta(u) + 1 = \delta(v) \),
and \( \pi(v) = u \).

Back to Pentominoes

How about using DFS or BFS to solve the pentomino problem?

- BFS is a bad idea: explore all parts of DAG at level 1, 2, 3, \ldots, 12.

- DFS sounds more promising, in particular if only one solution is required.

But: we cannot store the whole adjacency list.

- Succinct Representations.
  - Can code a single configuration \( C \) as some data structure (simple in principle, tricky when efficiency is important).
  - Given \( C \), need to be able to enumerate the neighbors of \( C \).
In C++, ideally done by a function object.

Backtracking

Succinct representation alone still too cumbersome, we need to streamline the search. This is now very problem specific.

- Can fix the insertion order of the 12 pieces.
- Can start with a piece that cannot go in many places. Usually one starts with the X pentomino.
- Can shamelessly exploit symmetries of the square: rotations and reflections.
- Can truncate the search: uncovered connected regions must have size a multiple of 5.