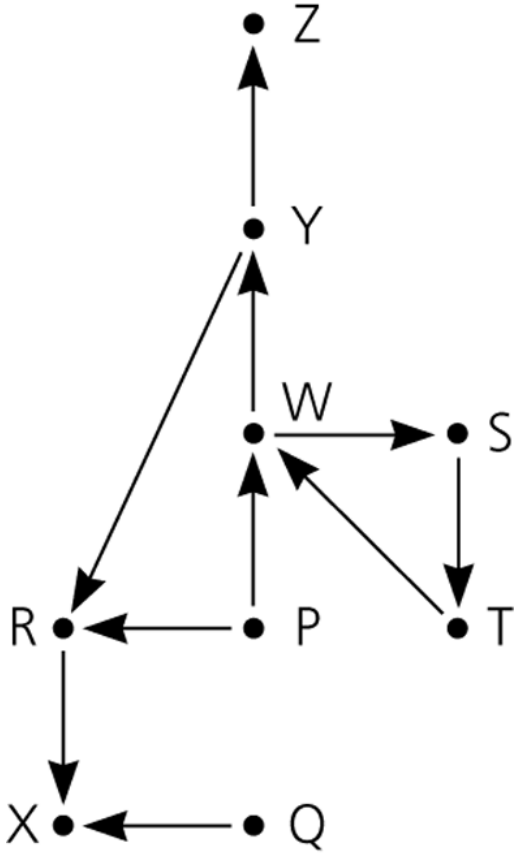
A Search Problem

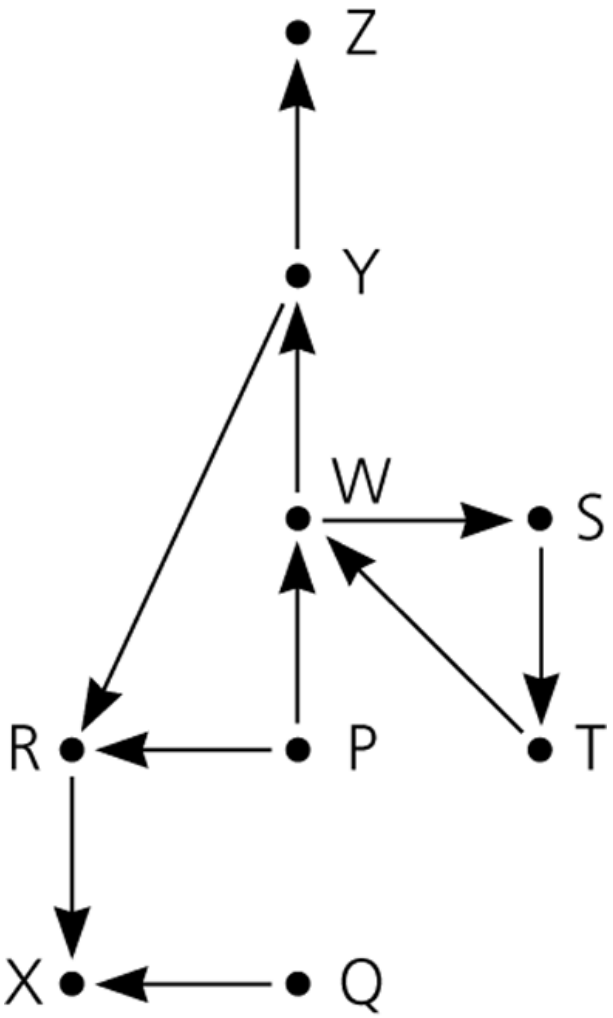
An airline company wants you to help them develop a program that generates flight itineraries for customer requests to fly from some origin city to some destination city.

Input data

- Cities
  - nodes in the graph
- Flight records
  - *178 Albuquerque Chicago 250*
  - Flight #, origin, destination, cost
  - Links in the graph
- Representation: directed graph



Flight map



To find flight itinerary origin to destination (Pseudo code)

```
aStack.create()
aStack.push(origin)
while(not found) {
    if(need to backtrack from the city on top of stack)
        aStack.pop()
    else {
        select a destination for a flight from city on top
        aStack.push(destination)
    }
}
```

The stack of cities as you travel

(a) from *P*;

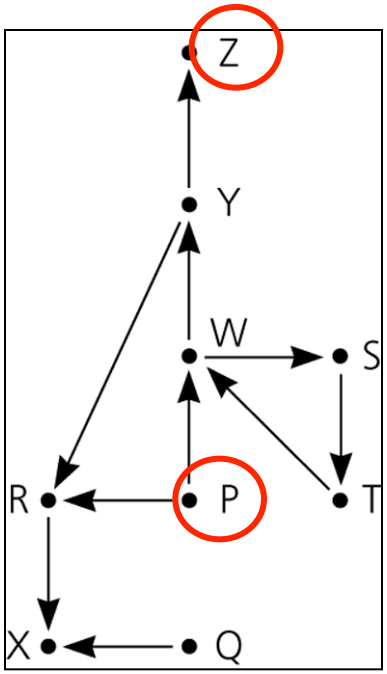
(b) to *R*;

(c) to *X*;

(d) back to *R*;

(e) back to *P*;

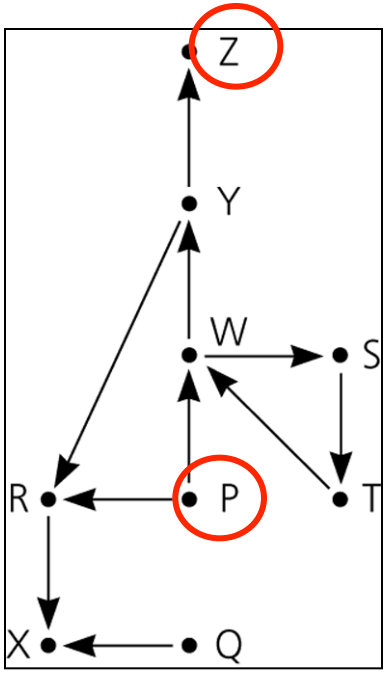
(f) to *W*

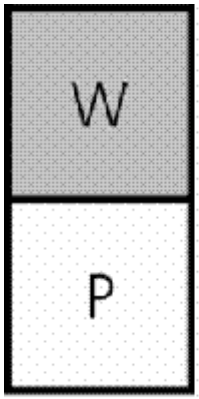


Three possible outcomes

- Eventually reach the destination city ☺
- Reach a city from which there are no departing flight ☹
- Go around in circles ☹ ☹

The stack of cities
(a)allowing revisits





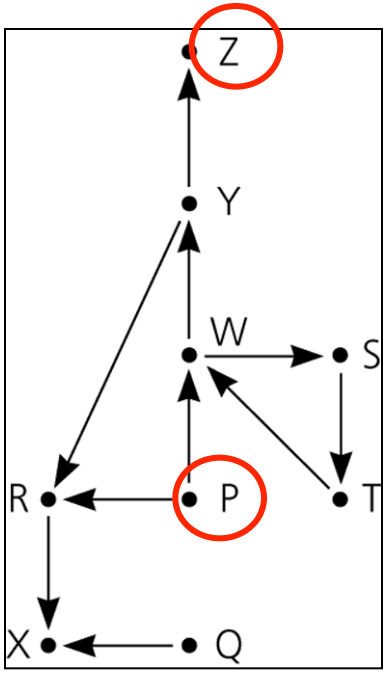
Solving the problem using a stack

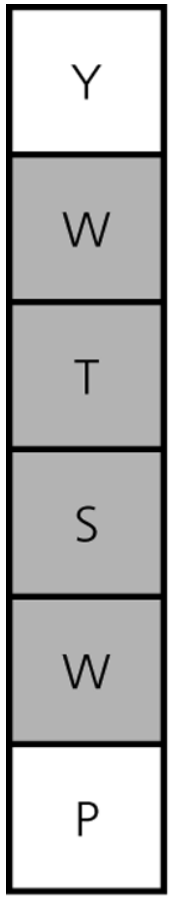
- What is in the stack?
  - Sequence of flights currently under consideration
  - Top of the stack is the currently visiting city
  - Bottom of the stack is the origin city
- How to find the path from the origin city to the destination city?
  - from the bottom to the top
- What do you do when you reach a dead-end?
  - No flight out of that city
- What happens if there is a cycle?

You never want to visit a city that the search has already visited

- Backtrack whenever there are no more unvisited cities to fly to
  - Visited city is still in the stack
  - Visited city is not in the stack because you backtracked from it
- Mark the visited city and choose the next city which is unmarked (not visited) and adjacent to the city on top of the stack

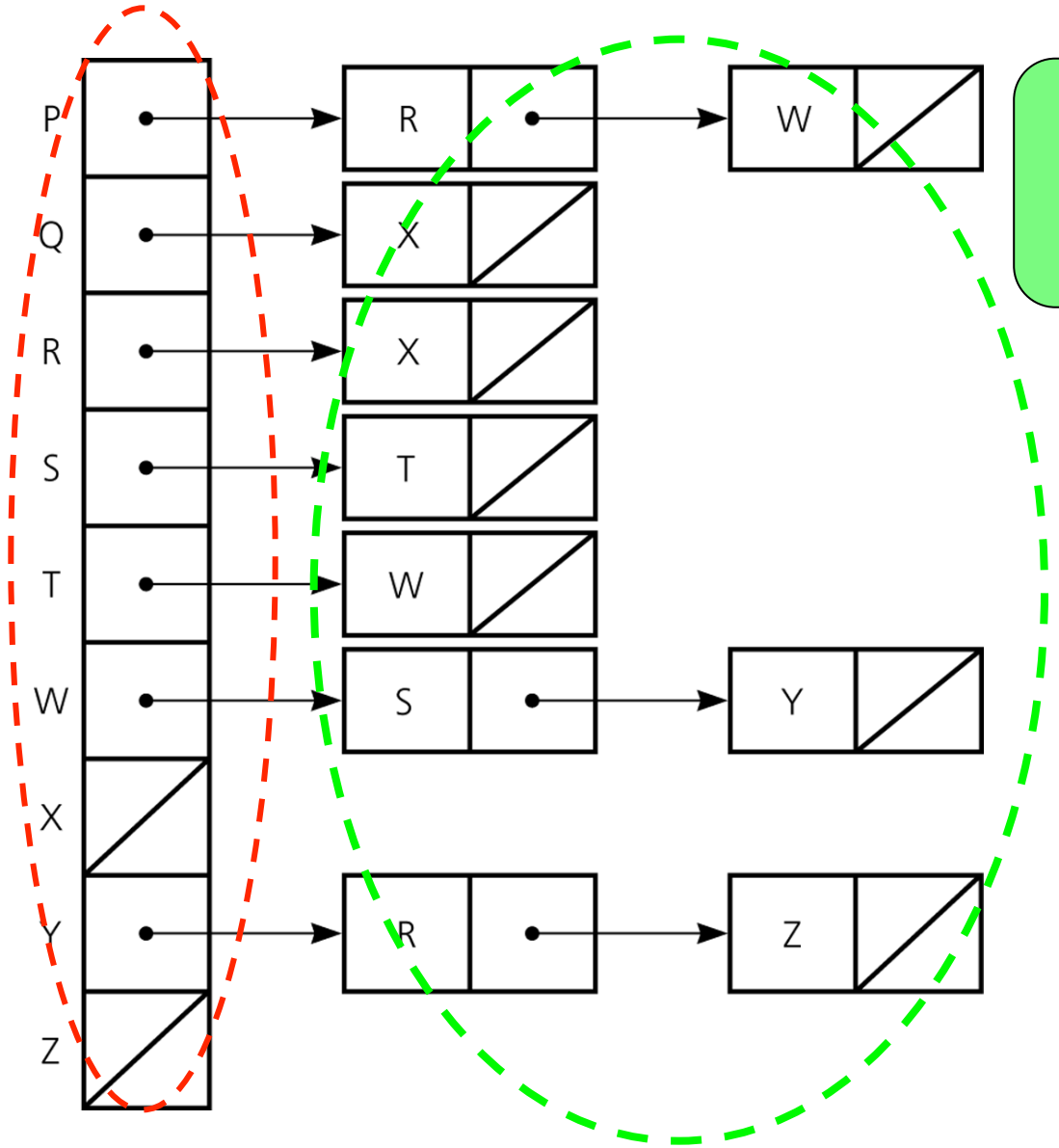
The stack of cities
(a)allowing revisits and
(b)after backtracking when revisits are not allowed





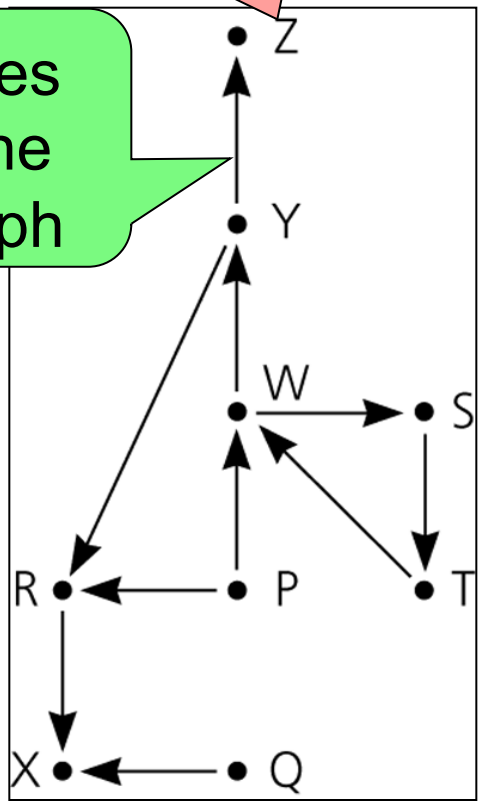
(a)

Adjacency list for the flight map



Nodes in the graph

edges in the graph



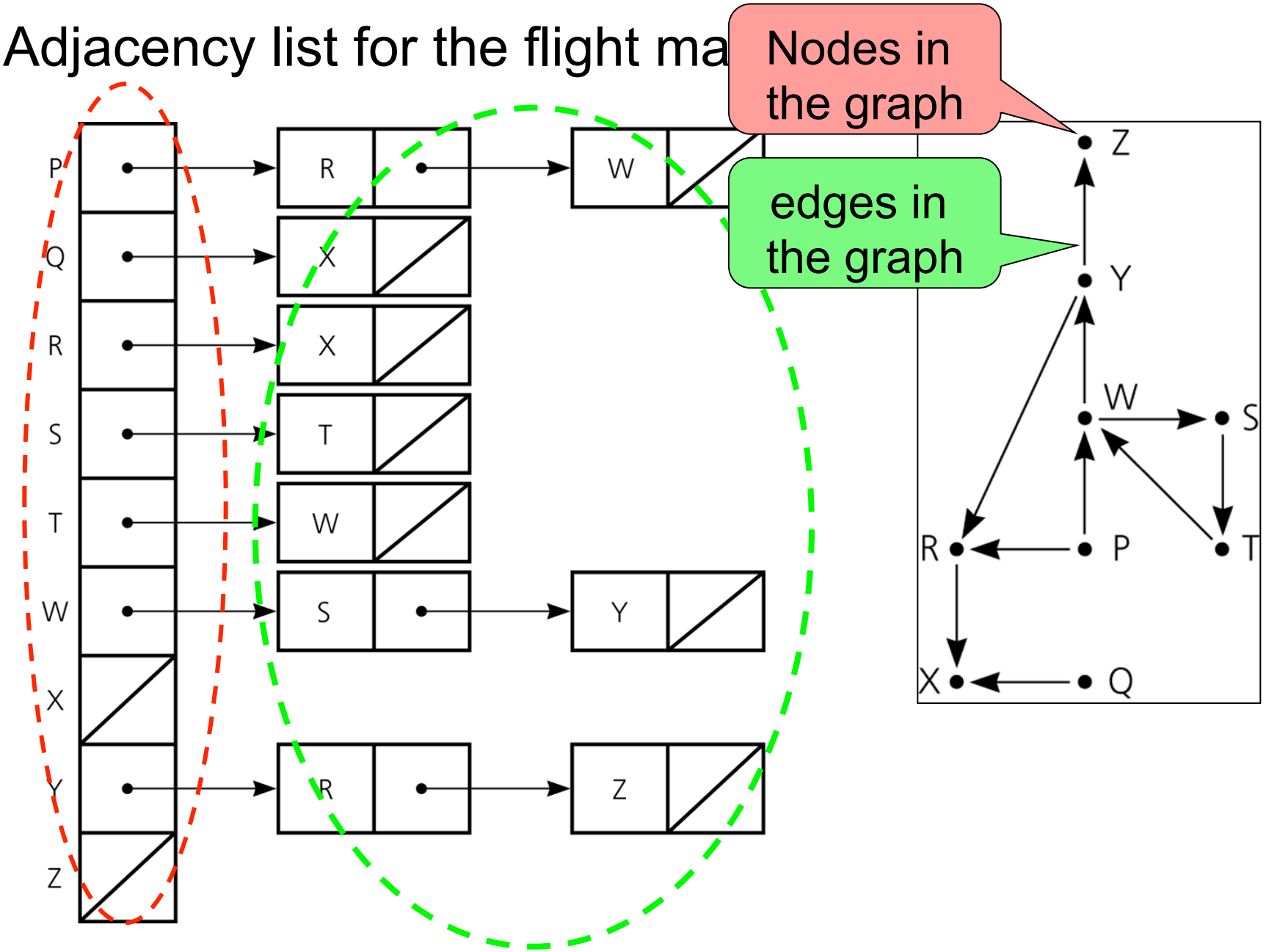
Revised search algorithm (pseudo code)

```
aStack.create()  and clear marks on all cities
aStack.push(origin) and mark origin as visited
while(!aStack.isEmpty( ) and destin != top of aStack) {
    if (no flight exist from city on top of aStack to unvisited cities)
        aStack.pop()
    else {
        select an unvisited city (C) for a flight from city on top of
        the stack
        aStack.push(C)
        mark C as visited
    }
}
if (aStack.isEmpty( ))          return false  // no path exist
else     return true  // path found
```

IsPath search algorithm in FlightMap class

```
bool FlightMap::isPath (string originCity, string destinationCity) {
  stack <string> aStack;
  int  topCity, nextCity;   bool  success;
  unvisitAll ( ); // clear marks on all cities
  aStack.push (originCity);
  markVisited (originCity);
  topCity=aStack.top();
  while (!aStack.isEmpty( ) && (topCity != destinationCity))   {
      success = getNextCity(topCity, nextCity);
      if (!success)
            aStack.pop(); // no city found; backtrack
      else  {  // visit city
           aStack.push(nextCity);
           markVisited(nextCity);
      } // end if
      topCity=aStack.top();
  } // end while
  return  !(aStack.empty())  // if stack is empty, no path exist
} // end isPath
```

Adjacency list for the flight ma



A trace of the search algorithm, given the flight map

Action	**Reason**	**Contents of Stack** (Bottom to top)
Push P	Initialize	P
Push R	Next unvisited adjacent city	P R
Push X	Next unvisited adjacent city	P R X
Pop X	No unvisited adjacent city	P R
Pop R	No unvisited adjacent city	P
Push W	Next unvisited adjacent city	P W
Push S	Next unvisited adjacent city	P W S
Push T	Next unvisited adjacent city	P W S T
Pop T	No unvisited adjacent city	P W S
Pop S	No unvisited adjacent city	P W
Push Y	Next unvisited adjacent city	P W Y
Push Z	Next unvisited adjacent city	P W Y Z