

Course Notes for
CS 1501
Algorithm Implementation

By
John C. Ramirez
Department of Computer Science
University of Pittsburgh



- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
 - Algorithms in C++ by Robert Sedgewick
 - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
 - Introduction to Algorithms, by Cormen, Leiserson and Rivest
 - Various Java and C++ textbooks
 - Various online resources (see notes for specifics)



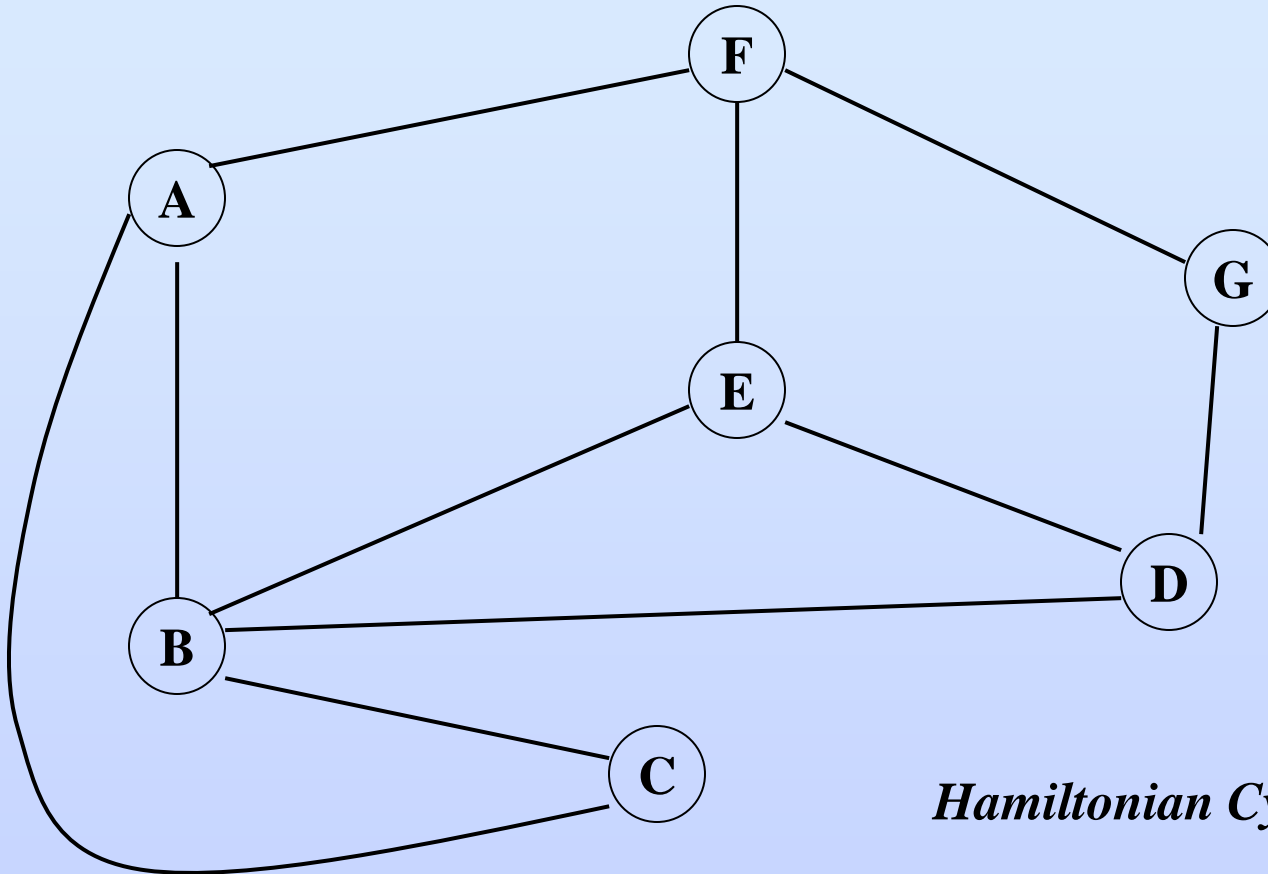
- Idea:
 - We find a solution to a problem by considering (possibly) **all potential solutions** and selecting the correct one.
- Run-time:
 - The run-time is **bounded by the number of possible solutions** to the problem.
 - If the number of potential solutions is exponential, then the run-time will be exponential.



- Example: Hamiltonian Cycle
 - ▶ A **Hamiltonian Cycle (HC)** in a graph is a cycle that **visits each node in the graph exactly one time**.
 - See example on next slide. →
 - ▶ Note that an HC is a permutation of the nodes in the graph (with a final edge back to the starting vertex).
 - Thus a fairly simple exhaustive search algorithm could be created to **try all permutations** of the vertices, checking each with the actual edges in the graph.



Exhaustive Search



Hamiltonian Cycle Problem

A solution is:

ACBEDGFA



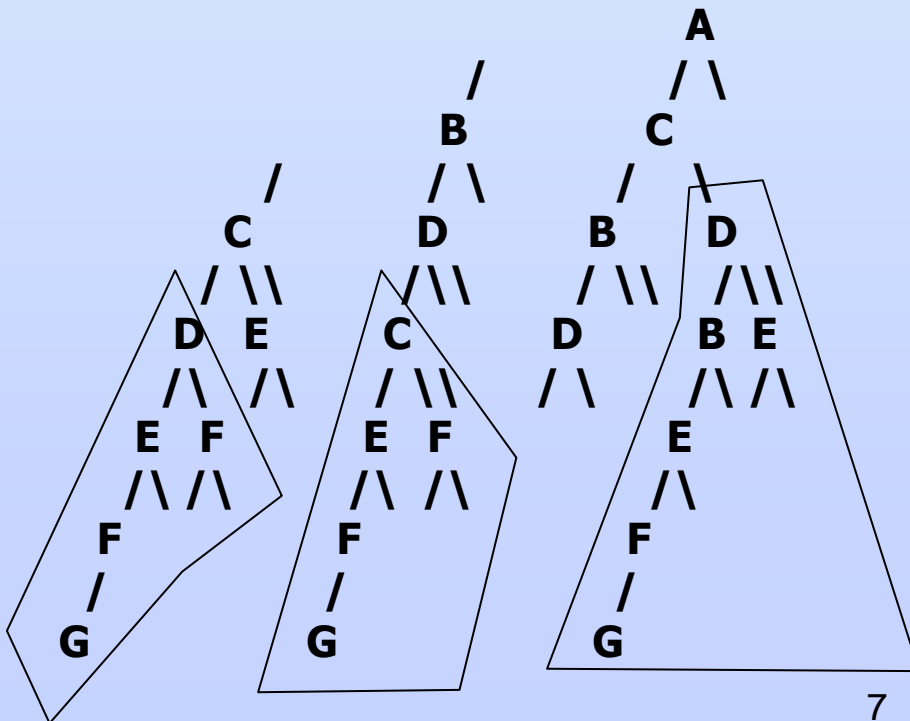
Exhaustive Search

- Unfortunately, for N vertices, there are $N!$ permutations, so the run-time here is poor.
- Pruning:
 - How can we improve our exhaustive search algorithms?
 - Think of the **execution** of our algorithm as a **tree**.
 - Each path from the root to a leaf is an attempt at a solution (i.e. one permutation).
 - The exhaustive search algorithm may try every path.
 - We'd like to eliminate some (many) of these execution paths if possible.
 - **If we can prune an entire subtree of solutions, we can greatly improve our algorithm runtime.**



Pruning and Branch and Bound

- In the example on slide 5:
 - Since edge (C, D) does not exist we know that no paths with (C, D) in them should be tried.
 - > If we start from A, this prunes several large branches from our execution tree and improves our run-time.
 - > Same for edge (A, E) and other edges too.



*Each permutation is a path
from A down to a leaf.
Many execution paths are
not shown.*

*Note: Many other pruned
branches not shown.*



► Important note:

- Pruning does **NOT improve** the algorithm **asymptotically**.
 - The worst case is still exponential in its run-time.
 - For a given (case-specific) problem we may in fact get a very good run-time.
 - > Extreme example: Graph is linear (resembling a linked list). In this case clearly there are only 2 permutations (going in each direction along the linear path) and the run-time is very good.
 - This cannot be guaranteed in general.
- However, **pruning can make the algorithm practically solvable for much larger values of N.**



- Exhaustive Search algorithms can often be effectively **implemented using recursion**.
 - ▶ Think again of the execution tree:
 - Each **recursive call progresses one node down the tree**.
 - BACKTRACKING – When a call terminates, control goes back to the previous call, which resumes execution.
 - The previous call then takes the next branch down the tree.
 - Execution continues until all paths have been tried.



- Idea of backtracking (review from CS 0445):
 - Proceed forward to a solution until it becomes apparent that no solution can be achieved along the current path.
 - At that point UNDO the solution (backtrack) to a point where we can again proceed forward.
 - Example: 8 Queens Problem
 - How can I place 8 queens on a chessboard such that no queen can take any other in the next move?
 - Recall that queens can move horizontally, vertically or diagonally for multiple spaces
 - See on next slide (and possibly from CS 0445). →



Recursion and Backtracking

	0	1	2	3	4	5	6	7
0	Q							
1				Q				
2		Q						
3					Q			
4			Q					
5								
6								
7								

- The Queen can move in any direction on the board.
- In the example shown we cannot place a Queen in column 5 because all of the locations are threatened by other Queens.
- The 8-Queens problem tries to find a placement of all 8 Queens such that none threaten each other.



► 8 Queens Exhaustive Search Solution:

- Try placing the queens on the board in every combination of 8 spots until we have found a solution (or not).
 - This solution will have an incredibly bad run-time.

$$\begin{aligned} - 64 \text{ C } 8 &= (64!)/[(8!)(64-8)!] \\ &= (64*63*62*61*60*59*58*57)/40320 \\ &= \mathbf{4,426,165,368 \text{ combinations}} \end{aligned}$$

(multiply by 8 for total queen placements)

- However, we can improve this solution by realizing that many possibilities should not even be tried, since no solution is possible.
- Ex: Any solution has exactly one queen in each column.



8 Queens Problem

- This would eliminate many combinations, but would still allow $8^8 = 16,777,216$ possible solutions ($\times 8 = 134,217,728$ total queen placements)
- If we further note that all queens must be in different rows, we reduce the possibilities even more.
 - Now the queen in the first column can be in any of the 8 rows, but the queen in the second column can only be in 7 rows, etc.
 - This reduces the possible solutions to $8! = 40320$ ($\times 8 = 322,560$ individual queen placements)
 - We can implement this in a recursive way.
- However, note that we can prune a lot of possibilities from even this solution execution tree by realizing early that some placements cannot lead to a solution.
 - Same idea as for Hamiltonian cycle – we are pruning the execution tree.



8 Queens Problem

- Ex: No queens on the same diagonal.
- Using this approach we come up with the solution as shown in 8-Queens handout.
 - See JRQueens.java
- Idea of solution:
 - Each recursive call attempts to place a queen in a specific column.
 - > A loop is used, since there are 8 squares in the column.
 - For a given call, the state of the board from previous placements is known (i.e. where are the other queens?).
 - If a placement within the column does not lead to a solution, the queen is removed and moved "down" the column.
 - > This is the backtracking step.



Recursion and Backtracking

	0	1	2	3	4	5	6	7
0	Q							
1				Q				
2		Q						
3					Q			
4			Q					
5								
6								
7					Q			

- The call at column 5 would try all rows and fail, backtracking to column 4.
- At column 4 we would move the queen down to the next legal row (7) and try again.
- The algorithm will proceed forward and backward until a solution is found.



8 Queens Problem


- When all rows in a column have been tried, the call terminates and backtracks to the previous call (in the previous column)
- If a queen cannot be placed into column i , do not even try to place one onto column $i+1$ – rather, backtrack to column $i-1$ and move the queen that had been placed there
- Using this approach we can reduce the number of potential solutions even more
- Run the handout JRQueens.java
 - > We will also briefly look at the code and run it during the interactive session



Finding Words in a Boggle Board

- ▶ Another example: finding words in a Boggle Board
 - Idea: form words from letters on mixed up printed cubes
 - Cubes are arranged in two-d array, as shown below
 - Words are formed by starting at any location in the cube and proceeding to adjacent cubes horizontally, vertically or diagonally
 - Any cube may appear at most one time in a word
- Ex: FRIENDLY and FROSTED are legal words in the board to the right
- Ex: FROO is not legal (not a word)
- Ex: DREAD is not legal (uses D twice)

F	R	O	O
Y	I	E	S
L	D	N	T
A	E	R	E



Finding Words in a Boggle Board

- This problem is very different from 8 Queens
- However, many of the ideas are similar
 - Each recursive call adds a letter to the word
 - Before a call terminates, the letter is removed
- But now the calls are in (up to) eight different directions:
 - For letter $[i][j]$ we can recurse to
 - > letter $[i+1][j]$ letter $[i-1][j]$
 - > letter $[i][j+1]$ letter $[i][j-1]$
 - > letter $[i+1][j+1]$ letter $[i+1][j-1]$
 - > letter $[i-1][j+1]$ letter $[i-1][j-1]$
- If we consider all possible calls, the runtime for this is **enormous!**
 - Has an approx. upper bound of $16! \approx 2.1 \times 10^{13}$



Finding Words in a Boggle Board

- Naturally, not all recursive calls may be possible
 - We cannot go back to the previous letter since it cannot be reused
 - > Note if we could words could have infinite length
 - We cannot go past the edge of the board
 - **We cannot go to any letter that does not yield a valid prefix to a word**
 - > Practically speaking, this will give us the greatest savings (i.e. the most pruning)
 - > For example, in the board shown (based on our dictionary), no words begin with FY, so we would not bother proceeding further from that prefix
 - Execution tree is pruned here as well



- Since a focus of this course is implementing algorithms, it is good to look at some implementation issues
 - Consider building / unbuilding strings that are considered in the Boggle game
 - Forward move adds a new character to the end of a string
 - Backtrack removes the most recent character from the end of the string
 - In effect our string is being used as a Stack – pushing for a new letter and popping to remove a letter



Implementation Note

- We know that Stack operations push and pop can both be done in $\Theta(1)$ time
 - Unless we need to resize, which would make a push linear for that operation (still amortized $\Theta(1)$ – why?)
- Unfortunately, the String data type in Java stores a **constant** string – it cannot be mutated
 - So how do we “push” a character onto the end?
 - In fact we must create a new String which is the previous string with one additional character
 - This has the overhead of allocating and initializing a new object for each “push”, with a similar overhead for a “pop”
 - Thus, push and pop have become $\Theta(N)$ operations, where N is the length of the string
 - > Very inefficient!

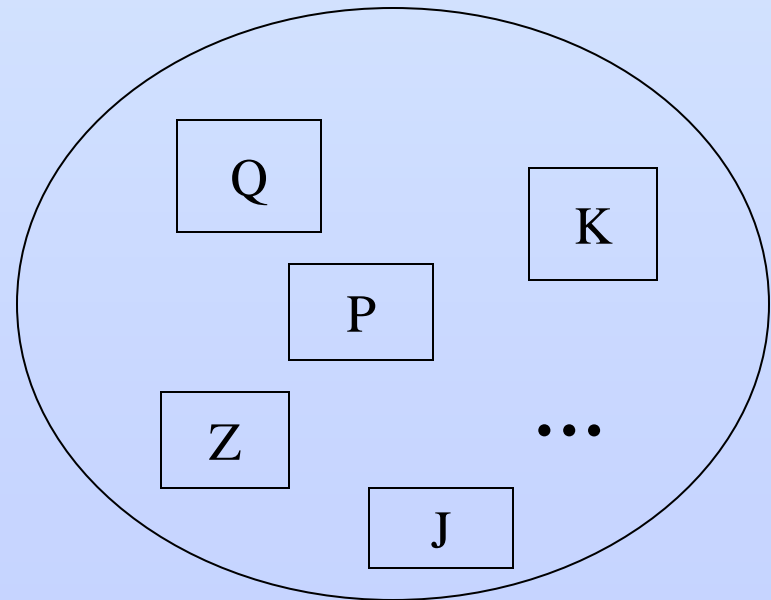


- For example:
S = new String(“ReallyBogusLongStrin”);
S = S + “g”;
- Consider now a program which does many thousands of these operations – you can see why it is not a preferred way to do it
- To make the “push” and “pop” more efficient ($\Theta(1)$) we could instead use a StringBuilder
 - append() method adds to the end without creating a new object
 - Reallocates memory only when needed
 - However, if we size the object correctly, reallocation need never be done
 - > Ex: For Boggle (4x4 square)
S = new StringBuilder(16);



- Consider the task of **searching** for an item within a collection
 - Given some collection C and some key value K, find/retrieve the object whose key matches K

K



- How do we know how to search so far?
 - ▶ Well let's first think of the collections that we know how to search
 - **Array/Vector**
 - Unsorted
 - > How to search? Run-time?
 - Sorted
 - > How to search? Run-time?
 - **Linked List**
 - Unsorted
 - > How to search? Run-time?
 - Sorted
 - > How to search? Run-time?



Review of Searching Methods

- **Binary Search Tree**
 - Slightly more complicated data structure
 - Run-time?
 - > Are average and worst case the same?
- ▶ So far binary search of a sorted array and a BST search are the best we have
 - Both are pretty good, giving $O(\log_2 N)$ search time
- ▶ Can we possibly do any better?
 - Perhaps if we use a very different approach



- **Symbol tables** are abstract structures that associate a **value** with a **key**
 - We use the key to search a data structure for the value
 - For a given application we may need only the keys or only the values or both
 - Text defines this as a class, but it would be more appropriate as an **interface**
 - Idea is that the symbol table specification does not require any specific implementation
 - In fact there are many different ways to implement a symbol table



Symbol Tables

- So far (from CS 0445) and previous slides we have seen how a symbol table can be implemented using an **array**, a **linked list**, or a **binary search tree**.
- Ex: See (briefly) BinarySearchST.java for a sorted array implementation
- Ex: See (briefly) BST.java for a binary search tree implementation
 - > We will look over these also in the interactive lecture
- ▶ Both of these implementations are similar in that the basic search involves **direct comparisons of keys**
 - In other words, to find a target key, K , we must **compare K** to one or more keys that are present in the data structure



Searching In a Different Way

- If we change our basic approach perhaps we can get an improvement
- In the next two topics we will look at different approaches to implementing a symbol table
 - In both cases we are able to improve over the $\Theta(\lg N)$ search time that is achieved by the sorted array and the BST
- First we will look at **Multiway Tries**
 - And a variation: **DLB**
- Then we will look at **Hash Tables**

