Parallel Bubblesort

In order for me to give back your midterms, please form a line and sort yourselves in alphabetical order, from A to Z.

There is traditionally a strong correlation between the midterm grades and the number of daily problems attempted:

daily: 0, sum: 134, count: 3, avg: 44.67

daily: 1, sum: 0, count: 2, avg: 0.00

daily: 2, sum: 63, count: 1, avg: 63.00

daily: 3, sum: 194, count: 3, avg: 64.67

daily: 4, sum: 335, count: 5, avg: 67.00

daily: 5, sum: 489, count: 8, avg: 61.12

daily: 6, sum: 381, count: 6, avg: 63.50

daily: 7, sum: 432, count: 6, avg: 72.00

daily: 8, sum: 217, count: 3, avg: 72.33

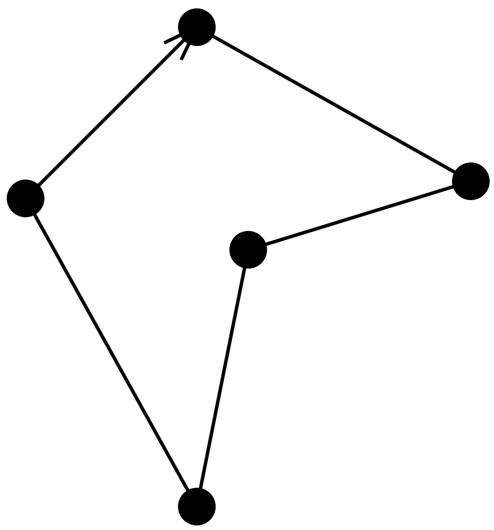
daily: 9, sum: 293, count: 4, avg: 73.25

Combinatorial Search

We have seen how clever algorithms can reduce sorting from $O(n^2)$ to $O(n \log n)$. However, the stakes are even higher for combinatorially explosive problems:

The Travelling Salesman Problem

Given a weighted graph, find the shortest cycle which visits each vertex once.



Applications include minimizing plotter movement, printed-circuit board wiring, transportation problems, etc.

There is no known polynomial time algorithm (ie. $O(n^k)$ for some fixed k) for this problem, so search-based algorithms are the only way to go if you need an optional solution.

But I want to use a Supercomputer

Moving to a faster computer can only buy you a relatively small improvement:

- Hardware clock rates on the fastest computers only improved by a factor of 6 from 1976 to 1989, from 12ns to 2ns.

- Moving to a machine with 100 processors can only give you a factor of 100 speedup, even if your job can be perfectly parallelized (but of course it can't).

- The fast Fourier algorithm (FFT) reduced computation from $O(n^2)$ to $O(n \lg n)$. This is a speedup of 340 times on $n = 4096$ and revolutionized the field of image processing.

- The fast multipole method for n-particle interaction reduced the computation from $O(n^2)$ to $O(n)$. This is a speedup of 4000 times on $n = 4096$.

Can Eight Pieces Cover a Chess Board?

Consider the 8 main pieces in chess (king, queen, two rooks, two bishops, two knights). Can they be positioned on a chessboard so every square is threatened?

			N	B			R
			N				
R							
	B						
		Q			K		

Only 63 square are threatened in this configuration. Since 1849, no one had been able to find an arrangement with bishops on different colors to cover all squares.

Of course, this is not an important problem, but we will use it as an example of how to attack a combinatorial search problem.

How many positions to test?

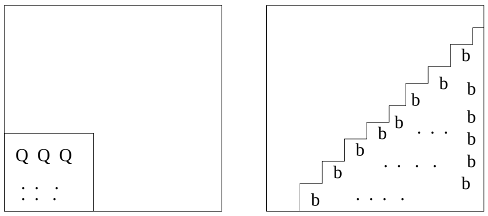
Picking a square for each piece gives us the bound:

$$64!/(64-8)! = 178,462,987,637,760 \approx 10^{15}$$

Anything much larger than 10^8 is unreasonable to search on a modest computer in a modest amount of time.

However, we can exploit symmetry to save work. With reflections along horizontal, vertical, and diagonal axis, the queen can go in only 10 non-equivallent positions.

Even better, we can restrict the white bishop to 16 spots and the queen to 16, while being certain that we get all distinct configurations.



$$16\times16\times32\times64\times2080\times2080 = 2,268,279,603,200 \approx 10^{12}$$

Backtracking

Backtracking is a systematic way to go through all the possible configurations of a search space.

In the general case, we assume our solution is a vector $v = (a_1, a_2, ..., a_n)$ where each element a_i is selected from a finite ordered set S_i,

We build from a partial solution of length k $v = (a_1, a_2, ..., a_k)$ and try to extend it by adding another element. After extending it, we will test whether what we have so far is still possible as a partial solution.

If it is still a candidate solution, great. If not, we delete a_k and try the next element from S_k:

```
Compute S_1, the set of candidate first elements of v.
k = 1
While k > 0 do
        While S_k ≠ ∅ do (*advance*)
                a_k = an element in S_k
                S_k ← S_k − a_k
                if (a_1, a_2, ..., a_k) is solution, print!
                k = k + 1
                compute S_k, the candidate kth elements given v.
        k = k − 1 (*backtrack*)
```

Recursive Backtracking

Recursion can be used for elegant and easy implementation of backtracking.

```
Backtrack(a, k)
if a is a solution, print(a)
else {
        k = k + 1
        compute S_k
        while S_k ≠ ∅ do
                a_k = an element in S_k
                S_k = S_k − a_k
                Backtrack(a, k)
}
```

Backtracking can easily be used to iterate through all subsets or permutations of a set.

Backtracking ensures correctness by enumerating all possibilities.

For backtracking to be efficient, we must prune the search space.

Constructing all Subsets

How many subsets are there of an n-element set?

To construct all 2^n subsets, set up an array/vector of n cells, where the value of a_i is either true or false, signifying whether the ith item is or is not in the subset.

To use the notation of the general backtrack algorithm, $S_k = (true, false)$, and v is a solution whenever $k \geq n$.

What order will this generate the subsets of $\{1, 2, 3\}$?

$$(1) \rightarrow (1,2) \rightarrow (1,2,3)* \rightarrow$$

$$(1,2,-)* \rightarrow (1,-) \rightarrow (1,-,3)* \rightarrow$$

$$(1,-,-)* \rightarrow (1,-) \rightarrow (1) \rightarrow$$

$$(-) \rightarrow (-,2) \rightarrow (-,2,3)* \rightarrow$$

$$(-,2,-)* \rightarrow (-,-) \rightarrow (-,-,3)* \rightarrow$$

$$(-,-,-)* \rightarrow (-,-) \rightarrow (-) \rightarrow ()$$

Constructing all Permutations

How many permutations are there of an n-element set?

To construct all $n!$ permutations, set up an array/vector of n cells, where the value of a_i is an integer from 1 to n which has not appeared thus far in the vector, corresponding to the ith element of the permutation.

To use the notation of the general backtrack algorithm, $S_k = (1, \ldots, n) - v$, and v is a solution whenever $k \geq n$.

$$
\begin{array}{rcl}
(1) & \rightarrow & (1,2) \rightarrow (1,2,3)* \rightarrow (1,2) \rightarrow (1) \rightarrow (1,3) \rightarrow \\
(1,3,2)* & \rightarrow & (1,3) \rightarrow (1) \rightarrow () \rightarrow (2) \rightarrow (2,1) \rightarrow \\
(2,1,3)* & \rightarrow & (2,1) \rightarrow (2) \rightarrow (2,3) \rightarrow (2,3,1)* \rightarrow (2,3) \rightarrow () \\
(2) & \rightarrow & () \rightarrow (3) \rightarrow (3,1)(3,1,2)* \rightarrow (3,1) \rightarrow (3) \rightarrow (3,2 \\
(3,2,1)* & \rightarrow & (3,2) \rightarrow (3) \rightarrow ()
\end{array}
$$

The n-Queens Problem

INCLUDE PICTURES OF THE EIGHT QUEENS

The first use of pruning to deal with the combinatorial explosion was by the king who rewarded the fellow who discovered chess!

In the eight Queens, we prune whenever one queen threatens another.

Covering the Chess Board

In covering the chess board, we prune whenever we find there is a square which we *cannot* cover given the initial configuration!

Specifically, each piece can threaten a certain maximum number of squares (queen 27, king 8, rook 14, etc.) Whenever the number of unthreated squares exceeds the sum of the maximum number of coverage remaining in unplaced squares, we can *prune*.

As implemented by a graduate student project, this backtrack search eliminates 95% of the search space, when the pieces are ordered by decreasing mobility.

With precomputing the list of possible moves, this program could search 1,000 positions per second. But this is too slow!

$$10^{12}/10^{3} = 10^{9} \text{ seconds } > 1000 \text{ days}$$

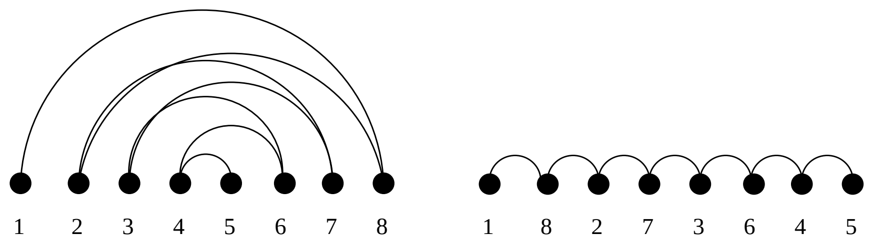
Although we might further speed the program by an order of magnitude, we need to prune more nodes!

By using a more clever algorithm, we eventually were able to prove no solution existed, in less than one day's worth of computing.

You too can fight the combinatorial explosion!

The Backtracking Contest: Bandwidth

The *bandwidth problem* takes as input a graph G, with n vertices and m edges (ie. pairs of vertices). The goal is to find a permutation of the vertices on the line which minimizes the maximum length of any edge.



The bandwidth problem has a variety of applications, including circuit layout, linear algebra, and optimizing memory usage in hypertext documents.

The problem is NP-complete, meaning that it is *exceedingly* unlikely that you will be able to find an algorithm with polynomial worst-case running time. It remains NP-complete even for restricted classes of trees.

Since the goal of the problem is to find a permutation, a backtracking program which iterates through all the $n!$ possible permutations and computes the length of the longest edge for each gives an easy $O(n! \cdot m)$ algorithm. But the goal of this assignment is to find as practically good an algorithm as possible.

Rules of the Game

1. Everyone must do this assignment separately. Just this once, you are not allowed to work with your partner. The idea is to think about the problem from scratch.

2. If you do not completely understand what the bandwidth of a graph is, you don't have the *slightest* chance of producing a working program. *Don't be afraid to ask for a clarification or explanation!!!!!*

3. There will be a variety of different data files of different sizes. Test on the smaller files first. Do not be afraid to create your own test files to help debug your program.

4. The data files are available via the course WWW page.

5. You will be graded on how fast and clever your program is, not on style. No credit will be given for incorrect programs.

6. The programs are to run on the whatever computer you have access to, although it must be vanilla enough that I can run the program on something I have access to.

7. You are to turn in a listing of your program, along with a brief description of your algorithm and any

interesting optimizations, sample runs, and the time it takes on sample data files. Report the largest test file your program could handle in one minute or less of wall clock time.

8. The top five self-reported times / largest sizes will be collected and tested by me to determine the winner.

Producing Efficient Programs

1. **Don't optimize prematurely:** Worrying about recursion vs. iteration is counter-productive until you have worked out the best way to prune the tree. That is where the money is.

2. **Choose your data structures for a reason:** What operations will you be doing? Is case of insertion/deletion more crucial than fast retrieval?

   When in doubt, keep it simple, stupid (KISS).

3. **Let the profiler determine where to do final tuning:** Your program is probably spending time where you don't expect.