Knight's Tour

- Given a natural number, n, describe how a Knight should be moved on an $n \times n$ chessboard so that it visits every square exactly once and ends on its starting square.
- Implicit graph in this problem has n^2 nodes corresponding to each position the Knight must occupy.
- There is an edge between two of these nodes if the corresponding positions are **one move apart**.
- The sub-graph that defines a solution is a cycle which contains each node of the implicit graph.

Knight's Moves

	5		7	
3				1
4				2
	6		8	

A Complete Tour

1	14	19	8	25
6	9	2	13	18
15	20	7	24	3
10	5	22	17	12
21	16	11	4	23

Knight's Tour

- The algorithm below, recursively searches the graph, labeling each square (i.e. node) in the order in which it is visited.
- In this algorithm:
  - An $n \times n$ representation of the board; initiated to 0.
  - (x, y) are the coordinates (row, column) of the current square.
  - `move` is the number of squares visited so far.
  - `ok` is a Boolean indicating success or failure.

Knight's Tour

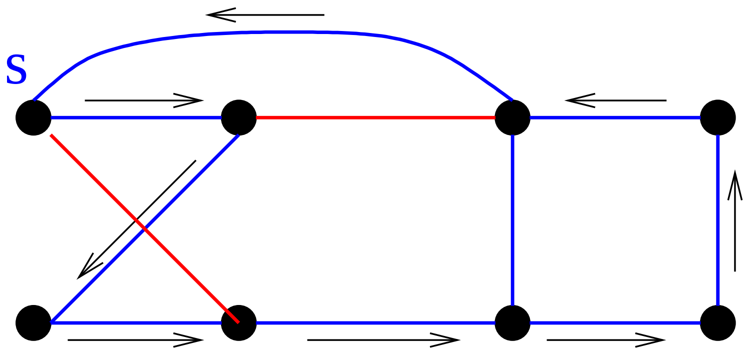
```
move(x, y, m) {
// (x, y) are location on the board
// m is a move number
   if ((x < 0) || (x ≥ n)
         || (y < 0) || (y ≥ n))
      return false;
   if (visited[x,y])
      return false;
   if (m == M) {
      //M = n^2-1, board is n × n.
      print "Solution is found";
      print "x, y";
      visited[x,y] = true;
      return true;
   }
```

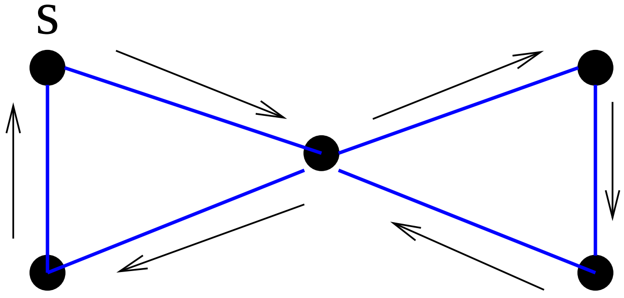

Knight's Tour

```
    else {
        visited[x, y] = true;
        result = false;
        result ||= move( x+2, y+1, m+1 )
        result ||= move( x+2, y-1, m+1 )
        result ||= move( x-2, y+1, m+1 )
        result ||= move( x-2, y-1, m+1 )
        result ||= move( x+1, y+2, m+1 )
        result ||= move( x+1, y-2, m+1 )
        result ||= move( x-1, y+2, m+1 )
        result ||= move( x-1, y-2, m+1 )
    }
        if (result) {
            print "x, y";
            return true;
        }
        else {
            visited[x, y] = false;
            return false;
        }
}
```

Hamiltonian Cycle

- It is cyclic tour of $|V|$ edges such that every vertex of the given graph $G = (V, E)$ is visited exactly once and returns back to starting vertex.
- Two examples are provided below.
  - First has a Hamiltonian cycle. The second one does not have one.





Hamiltonian Cycle

- Solution vector is represented as $(x[1], x[2], \ldots, x[n])$, where $x[i]$ represents ith vector in the cycle.
- Given partial solution vector $(x[1], x[2], \ldots, x[k-1])$, the backtrack solution has to identify the set of all possible vertices which can be added to extend this partial vector.
- If $k = 1$ then any of the n vertices of the graph can be candidates for $x[1]$.
- To avoid repeated discovery of the same cycle, we start with $x[1] = 1$.
- If $1 < k < n$ then $x[k]$ can be any of the vertices which is distinct from vertices $x[i]$, for $1 \leq i \leq k-1$.
- Also that $(x[k-1], x[k]) \in E$
- $x[n]$ can be a vertex such that $(x[n-1], x[n]), (x[1], x[n]) \in E$.

Hamiltonian Cycle

```
nextVertex(k, A) {
   while (1) {
      // Get next vertex.
      x[k] = (x[k]+1) mod (n+1);
      // If not found, we reached
      // a deadend.
      if (x[k] == 0) return;
      // Check for cycle edge.
      if (A(x[k-1],x[k]))
         // Check distinctness
         for (j=1; j<k; j++)
            if (x[j]==x[k]) break;
         if (j==k)
            if ((k<n)||((k==n) &&
               A(x[n],1)))
               return;
   }
}
```

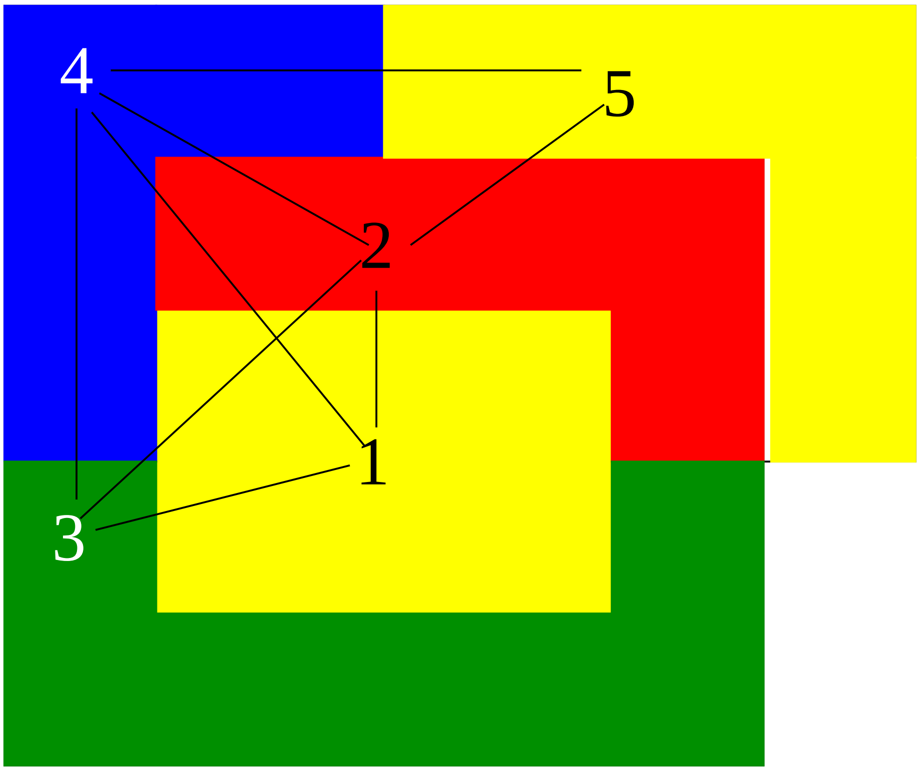
Hamiltonian Cycle

```
hamitonianCycle(k, A) {
   while (1) {
      // Find the next vertex.
      nextVertex(k);
      if (x[k]==0)
         // No cycle possible.
         return;
      if (k==n)
         // Found one, print it.
         print(x,'1');
      else
         // Extend the current.
         hamitonianCycle(k+1);
   }
}
```

Initial Requirements

- Initially all $x[i]$ for $1 < i \leq n$ should be set to 0.
- $x[1]$ set to 1.
- Then call `hamiltonianCycle(2)`.
- Travelling salesman problem is a Hamiltonian tour.
- So finding a minimum cost Hamiltonian tour is the TSP problem.

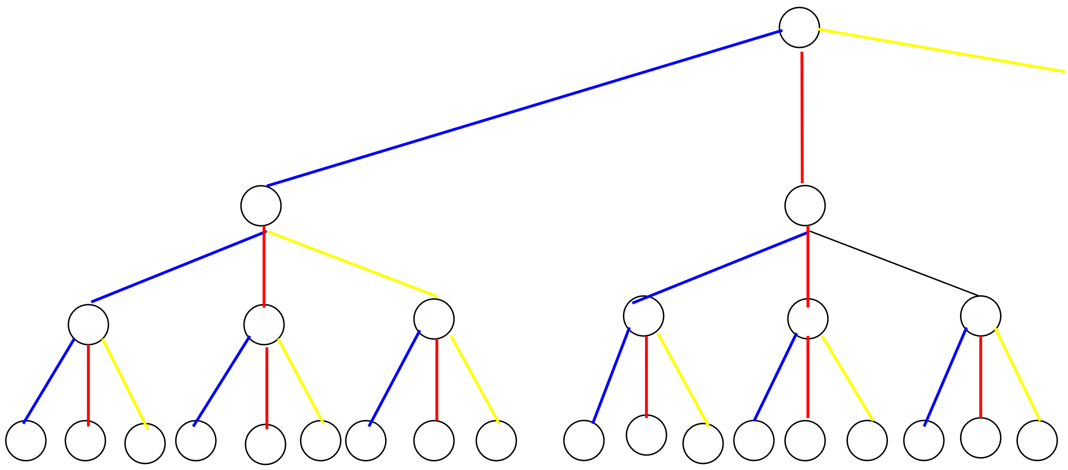
Graph Coloring



Graph Coloring

- Colors are represented by integers 1, 2, $\ldots, m$.
- Solution is in form of n-vectors $x[1], x[2], \ldots, x[n]$ where, $x[i] = j$ for some $1 \leq j \leq m$.
- The state space for solution is a tree of degree m and height $n + 1$.
- Each node at level i has m children corresponding to m possible assignments to $x[i]$ for $1 \leq i \leq n$.
- Nodes at level $n + 1$ are leaf nodes.

State Space for Coloring Problem



m=3, n=3

Graph Coloring

- Initialization: set all $x[1 \ldots n] = 0$.
- Invoke `mColoring(1)`
- Base case: assigning color 1 to first vertex.
- Assuming colors of $x[1 \ldots k-1]$ are assigned try to assign smallest available color to $x[k]$ and check if a coloring can done.
- Call `mColoring` recursively.

Graph Coloring

```
mColoring(k) {
   while (1) {
      // Find next color.
      nextColor(k);
      // Exit if no color can be
      // found.  No solution.
      if (x[k]==0) exit();
      // Solution found, print it.
      if (k==n) print(x[1], ..., x[n]);
      else
         mColor(k+1);
   }
}
```

Graph Coloring

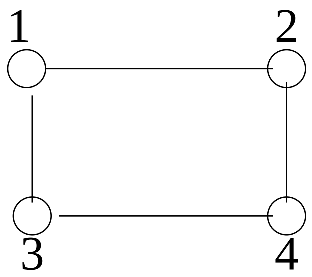
```
nextColor(k) {
   while (1) {
      // Find next color.
      x[k] = (x[k]+1) mod (m+1);
      // All colors checked.
      if (x[k] == 0) return;
      for (j=1; j≤n; j++)
         // Check distinctness of
         // adjacent colors.
         if (A[k, j] && x[k]==x[j])
            break;
      // New color found.
      if (j==n+1) return;
   }
}
```

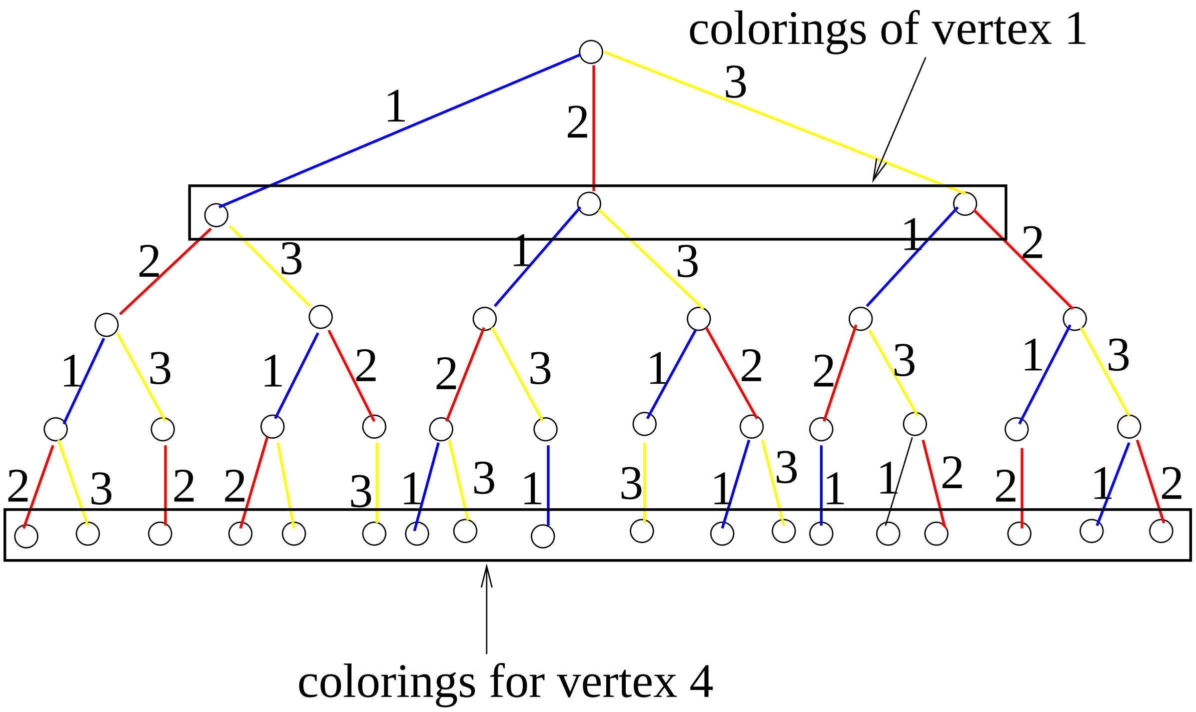
Running Time

- The state space as indicated has $\sum_{i=0}^{n-1} m^i$ nodes.
- Each time `nextColor` is called $O(mn)$ time is spent.
- So time is bounded by

$$\begin{aligned}\left(\sum_{i=0}^{n-1} m^i\right) \times mn &= \sum_{i=1}^{n} m^i n \\ &= \frac{n(m^{m+1}-1)}{(m-1)} \\ &= O(nm^n)\end{aligned}$$

3 Coloring of a Graph





Summary of Backtracking

- It is a search technique for finding optimal solution satisfying a set of constraint.
- Desired solution is expressed as an n-tuple $x_1, x_2, \ldots, x_n$.
- x_i is chosen from some finite set S_i.
- It differs from exhaustive search by arranging the search to progress incrementally.
- The major advantage in this approach is backtrack can predict if the partial vector representing partial solution will not lead to a full solution.
- In order to predict this it uses a bounding function.
- Therefore backtrack is able to avoid many unnecessary search paths.

Summary of Backtracking

- In other words, the implicit search tree representing the traversals of solution space in backtrack is comparatively small in size as opposed to that generated by unconstrained search.
- Domain knowledge must be taken into account to develop the bounding function.
- Bounding function can greatly influence running time.
- A good bounding function is the one that restricts the size of the implicit search tree substantially.
- At the same time such a bounding function would itself require substantial running time.
- So there is treadoff due to bounding function.

Summary of Backtracking

- Efficiency of backtrack depends on four factors:
  - Time to generate next element to be appended to the current partial solution (extension of partial solution).
  - The number of possible extensions.
  - The number of extensions satisfying the constraints of bounding function.
  - Time for bounding function.
- We looked at following examples.
  - Turnpike reconstruction problem.
  - Eight queens problem.
  - Hamiltonian cycles problem.
  - Graph coloring problem.

