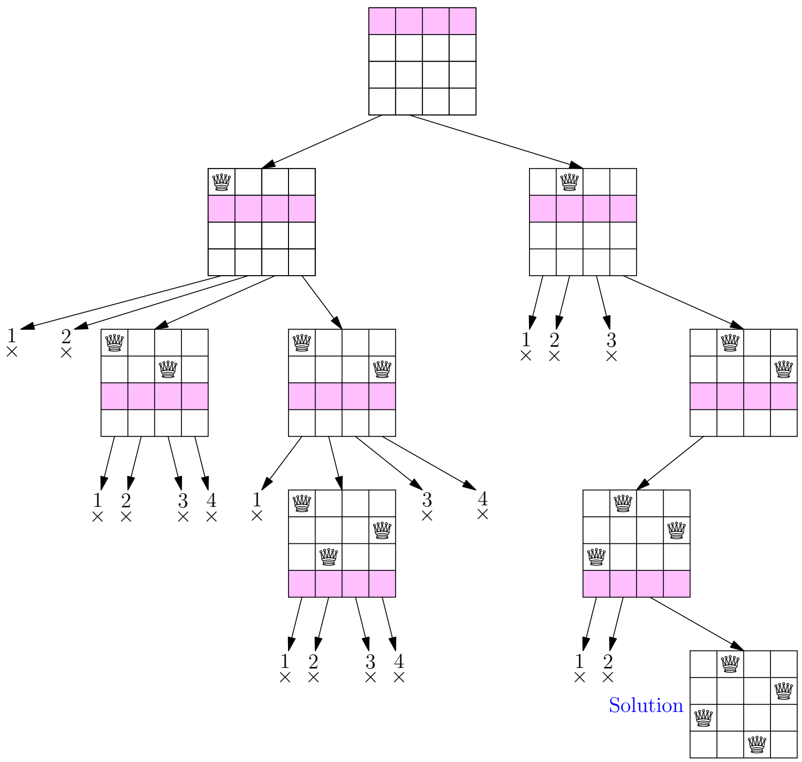
Algorithms and Analysis

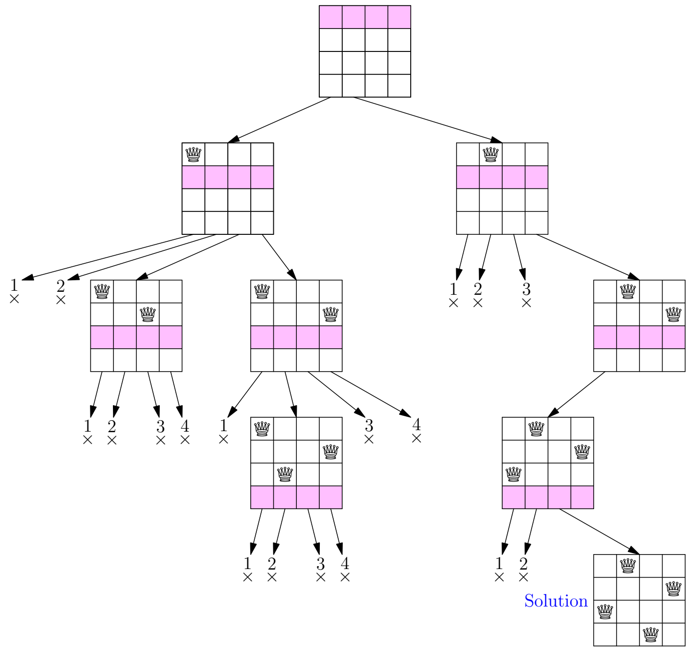
Lesson 22: *Know how to Search*



Backtracking, Branch and Bound

Outline

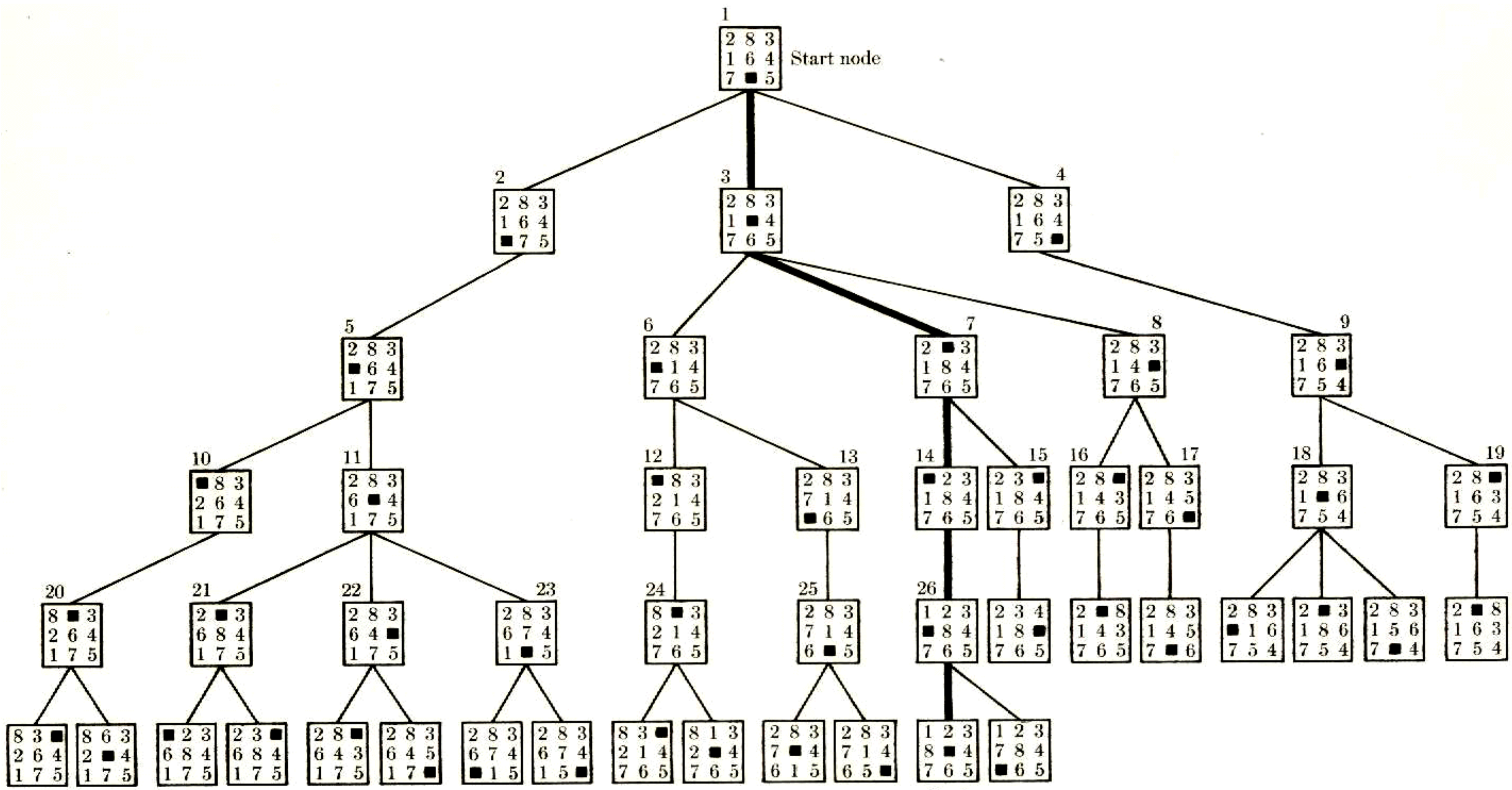
1. **Search Trees**
2. Backtracking
3. Branch and Bound
4. Search in AI



State Space Representation

- Many real world problems involve taking a series of actions to manipulate the state of the system
- This is the area of planning and search which sits within the domain of artificial intelligence
- One of the key props to help us develop algorithms is to think of the states as nodes of a graph which are linked if there exists an action taking us from one state to another
- This provides a **state space representation** of the problem (we saw this before when we derived a low bound on sorting)

8-Puzzle Example

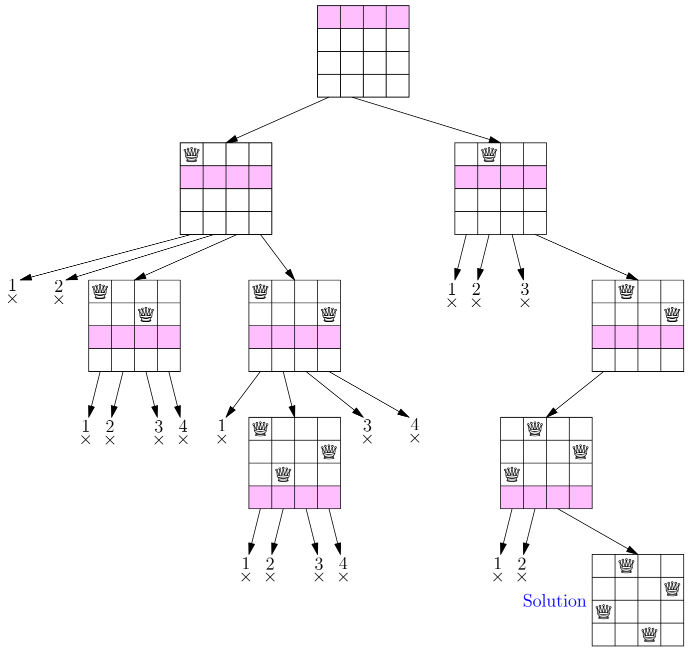


Large State Spaces

- The search space typically increase exponentially with the problem size
- We can find the quickest solution to the 8-puzzle (and the 15 puzzle) using breadth first search, but larger puzzles soon become intractable
- Nevertheless, a lot of important problems involve very large state spaces and we have to find algorithms to explore them

Outline

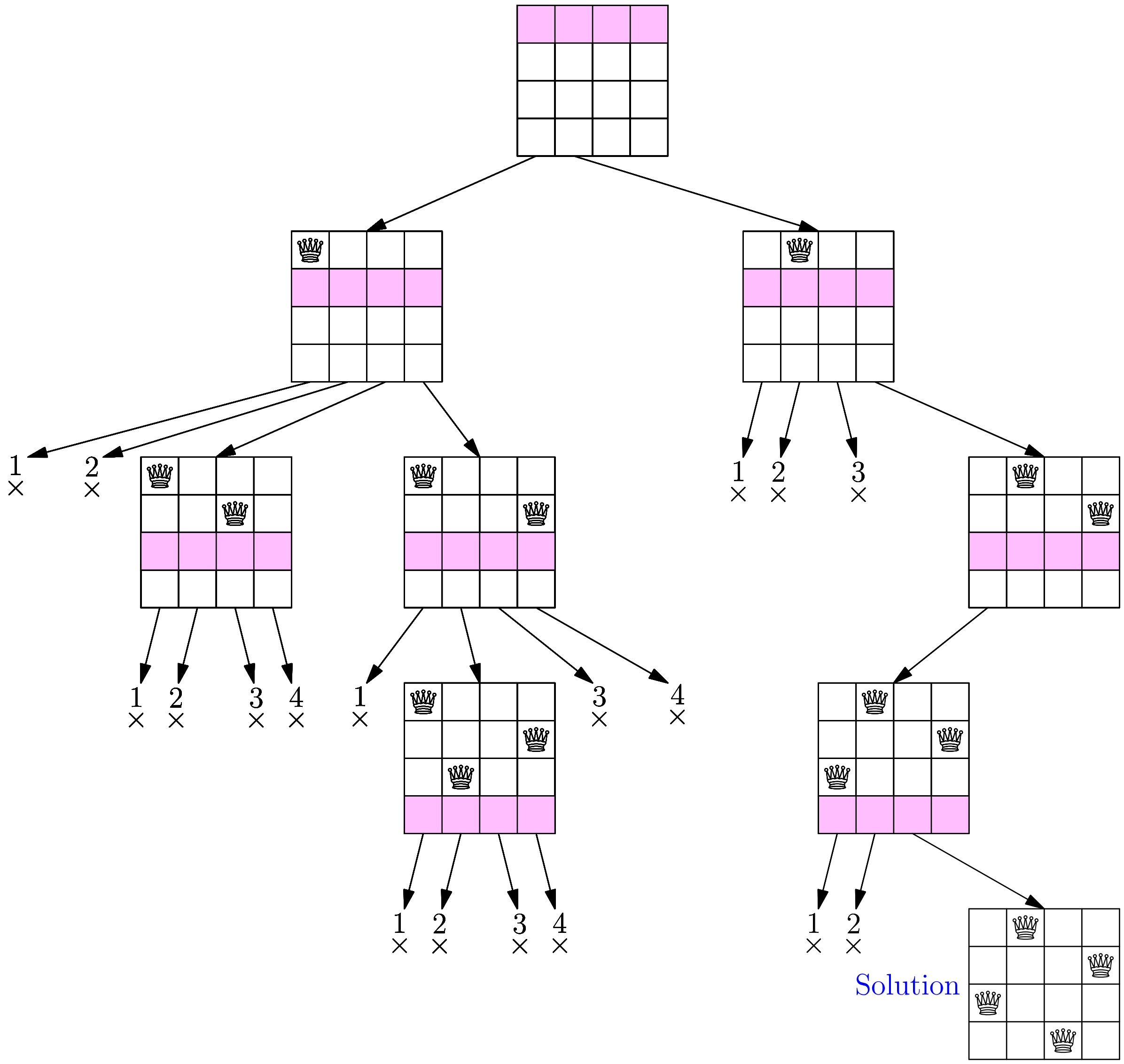
1. Search Trees
2. **Backtracking**
3. Branch and Bound
4. Search in AI



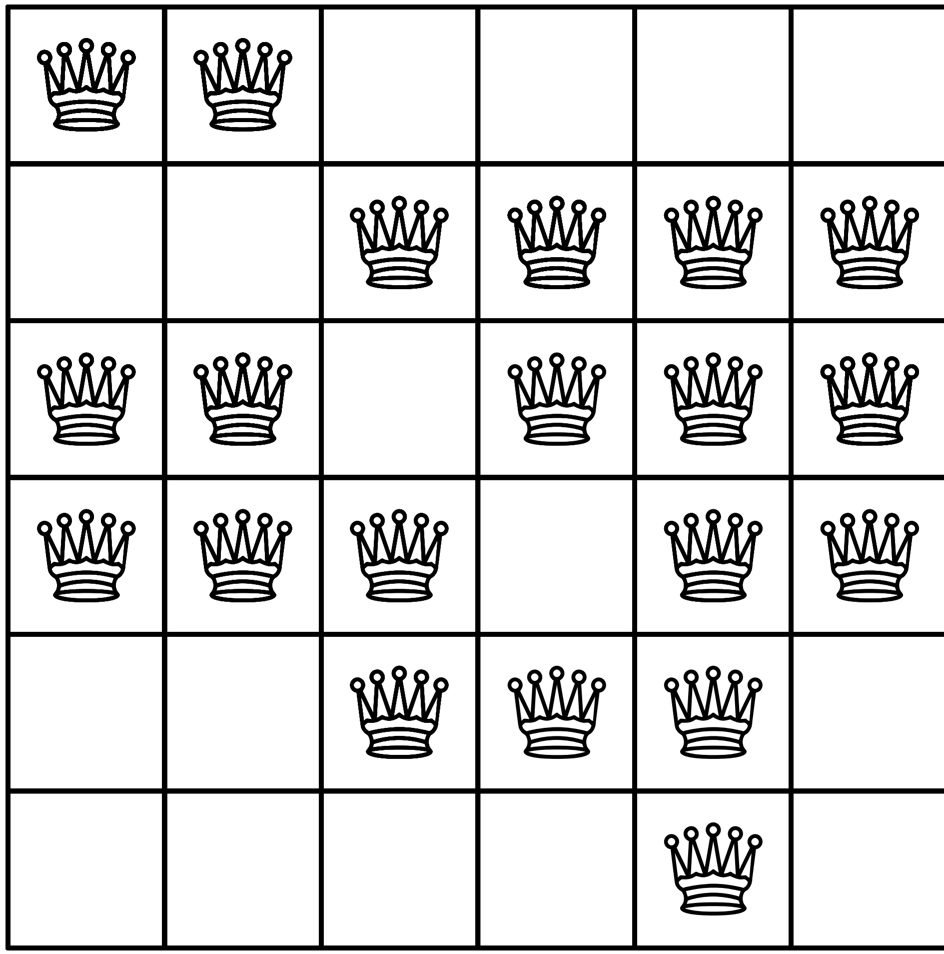
Backtracking

- Backtracking is used to find feasible solutions in large state spaces
- E.g. solving sudoku
- It works by growing partial solutions until either
  - ⋆ a feasible solution is found when we can finish
  - ⋆ no feasible solution is found when we backtrack
- We often search the state space using depth first search

4-Queens Problem



6-Queens Problem



Implementing n-Queens

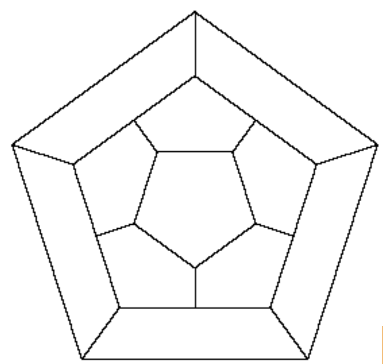
- Implementing backtracking is easily done using recursion
- Recall depth-first search is easily implemented using recursion
- We just need a recursive function `next(n, row, sol)` which for a n-Queens problem searches new solutions in `row` given queens in previous rows given in `sol`
- Run: `List sol = nextRow(6, 0, new List());`

Code

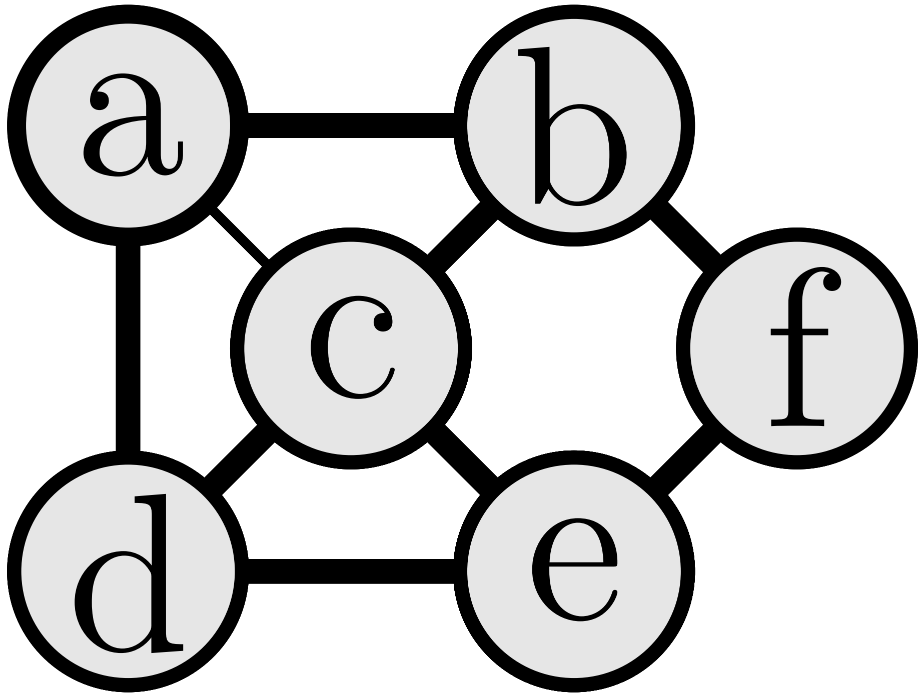
```
List nextRow(int noRows, int row, List queenPositions) {
  if (row==noRows) {return queenPositions;}
  for (int col=0; col<noRows; ++col) {
    if (legalQueen(col, row, queenPositions)) {
      queenPositions.add(col);
      List solution = nextRow(noRows, row+1, queenPositions);
      if (solution!=null)
        return solution;
    }
  }
  return null;
}

bool LegalQueen(int col, int row, List sol) {
  for(int r=0; r<row: ++r) {
    rf (sol[r] == col || sol[r]-row+r == col || sol[r]+row-r==col) {
      return false;
    }
  }
  return true;
}
```

Hamiltonian Circuit

- A Hamiltonian cycle is a tour through a graph which visits every vertex once only and returns to the start
- It is a hard problem in that there are no known algorithms that are guaranteed to find a Hamiltonian cycle in polynomial time
- For many graphs it is not too hard



Hamiltonian Circuit Example

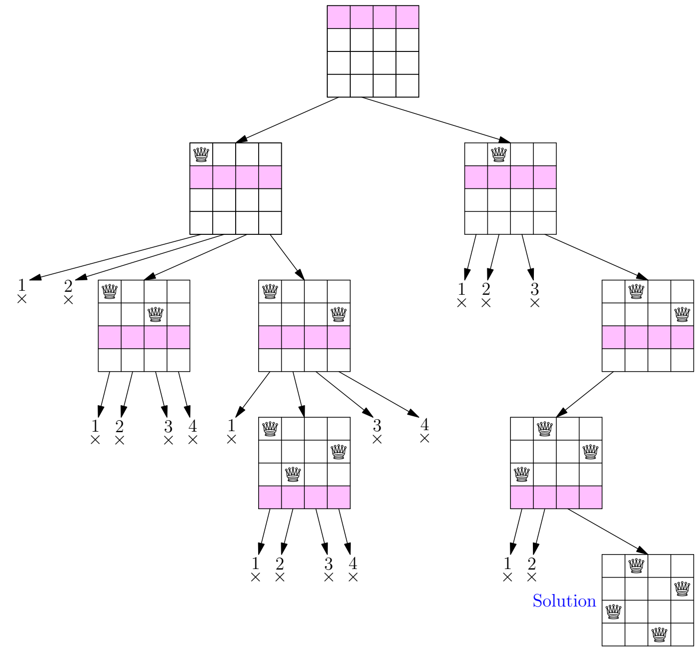


Backtracking

- Backtracking is a standard algorithm for solving constraint problems with large search spaces
- It can take exponential amount of time, however with many constraints it will often find solutions relatively quickly
- A backtracking algorithm does not solve, for example, sudoku in the same way as a human—it uses speed rather than brains
- We can often speed up backtracking by adding more constraints (although, this can make writing the program longer)

Outline

1. Search Trees
2. Backtracking
3. **Branch and Bound**
4. Search in AI



Optimisation Problems

- In many optimisation problems (TSP, Graph-colouring, etc.) we again have a huge search space ($n!$, k^n)
- However, we don't have hard constraints
- If we are interested in finding the optimal then we can use the cost as a constraint

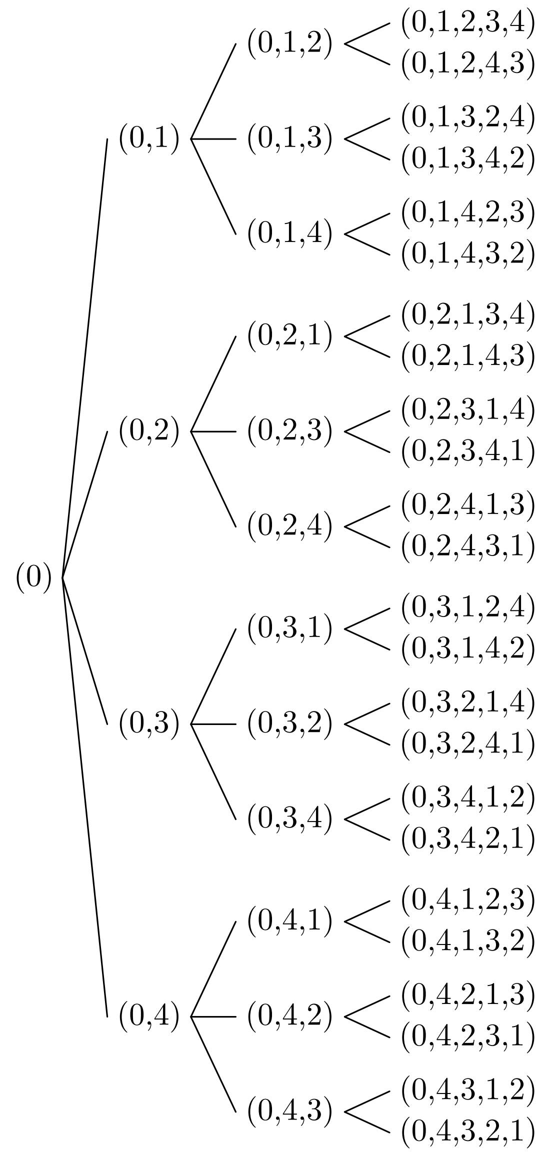
  any partial solution has to have a lower cost than the best solution we have found so far
- This allows us to develop a backtracking strategy known as branch and bound

Branch and Bound

- Branch and bound is used on optimisation problems where efficient strategies just don't work
- It beats exhaustive enumeration by eliminate many possible solutions without having to enumerate them all
- Branch and bound can be slow as the constraints aren't necessarily very strong
- By working harder we can sometimes strengthen the constraints thus eliminating much of the search space
- This strategy works quite well on smallish problems, but usually fails on large problems

Cutting the Search Tree

- We can think of exact enumeration as exploring a giant search tree
- If we know a partial solution is worse than our bound we cut the search tree
- The earlier we cut the tree the more we can save



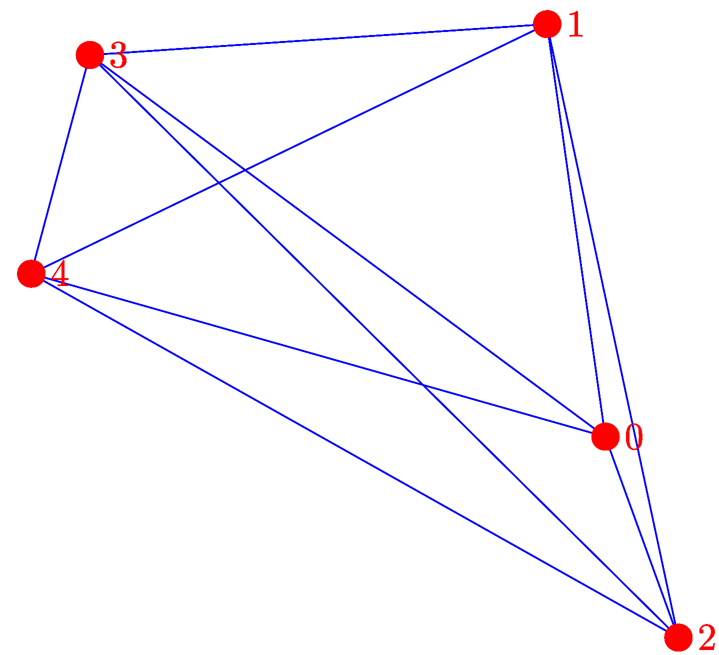
Branch and Bound in Action

(0)

- (0,1)
  - (0,1,2) < (0,1,2,3) — (0,1,2,3,4)*; (0,1,2,4) — (0,1,2,4,3)*
  - (0,1,3) < (0,1,3,2) — (0,1,3,2,4); (0,1,3,4) — (0,1,3,4,2)*
  - (0,1,4) < (0,1,4,2) — (0,1,4,2,3); (0,1,4,3) — (0,1,4,3,2)
- (0,2)
  - (0,2,1) < (0,2,1,3) — (0,2,1,3,4); (0,2,1,4) — (0,2,1,4,3)
  - (0,2,3) < (0,2,3,1) — (0,2,3,1,4); (0,2,3,4) — (0,2,3,4,1)
  - (0,2,4) < (0,2,4,1) — (0,2,4,1,3); (0,2,4,3) — (0,2,4,3,1)*
- (0,3)
  - (0,3,1) < (0,3,1,2) — (0,3,1,2,4); (0,3,1,4) — (0,3,1,4,2)
  - (0,3,2) < (0,3,2,1); (0,3,2,4)
  - (0,3,4) < (0,3,4,1) — (0,3,4,1,2); (0,3,4,2) — (0,3,4,2,1)
- (0,4)
  - (0,4,1) < (0,4,1,2) — (0,4,1,2,3); (0,4,1,3) — (0,4,1,3,2)
  - (0,4,2) < (0,4,2,1) — (0,4,2,1,3); (0,4,2,3)
  - (0,4,3) < (0,4,3,1) — (0,4,3,1,2); (0,4,3,2) — (0,4,3,2,1)

bound =

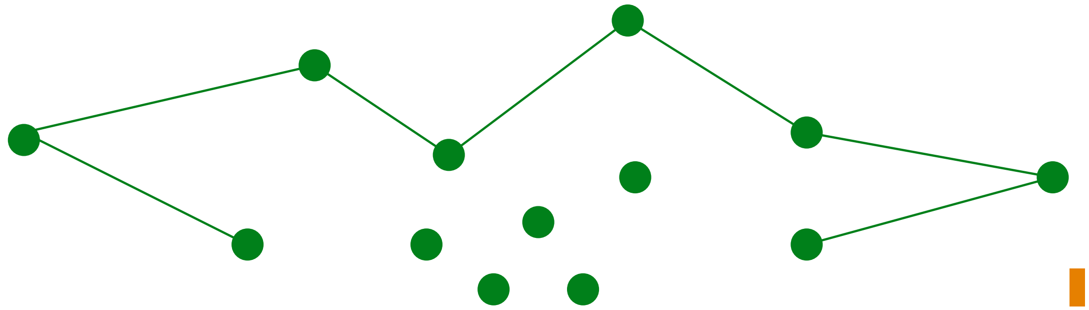
length =



(0,

Bound on Partial Solution

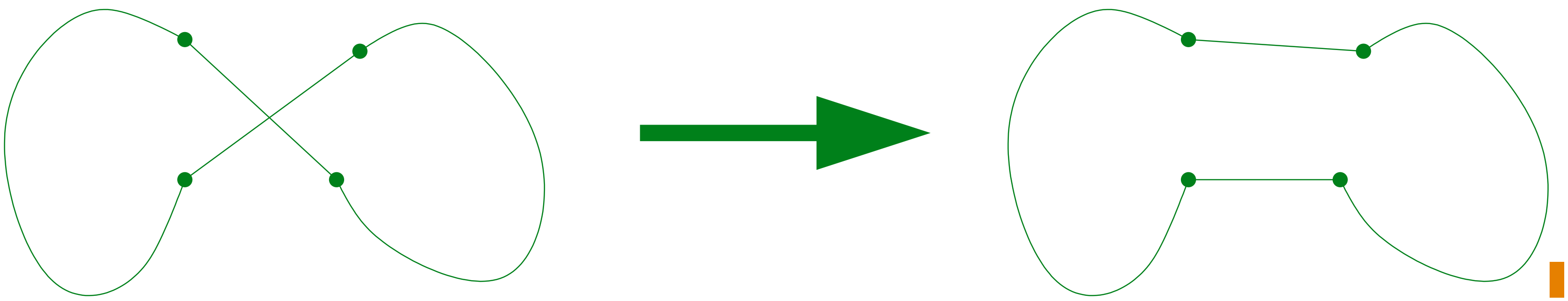
- We know that the partial solution has to include all the remaining cities



- We can use this to obtain a lower bound on the partial solution
- We know the remaining tour will go through each of the unvisited cities and the two edge cities
- In fact the remaining part of the tour is a spanning tree of these vertices (it connects all the vertices once and has no cycles)
- But we know a lower bound for this—**the minimum spanning tree**

Other Cuts

- For 2-D Euclidean TSPs edges should never cross

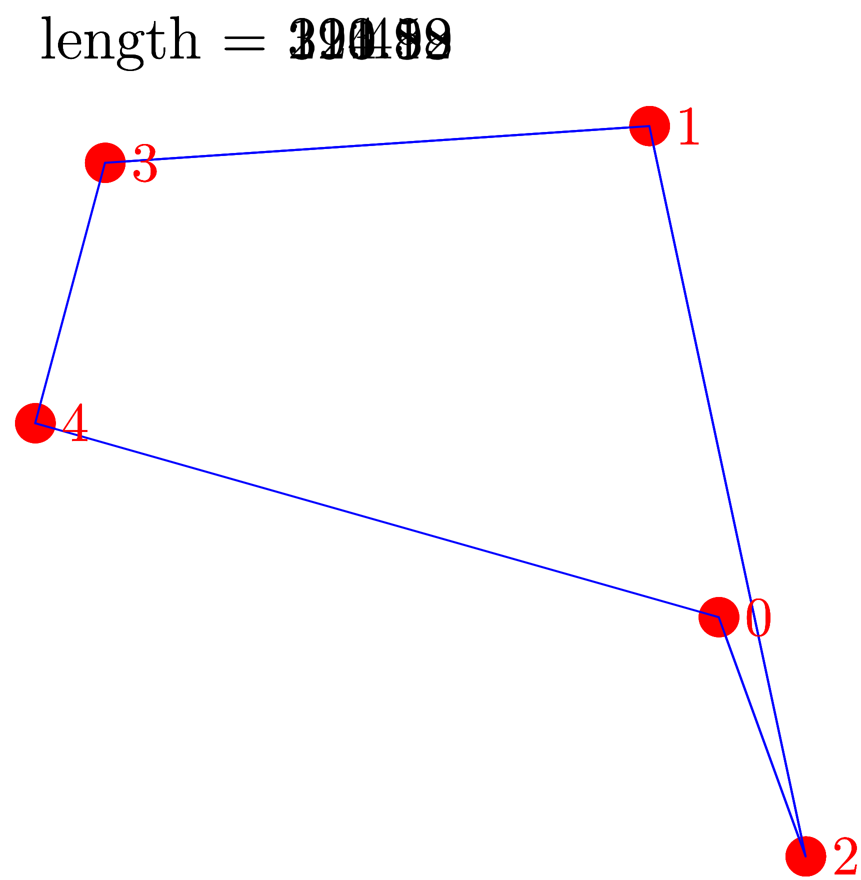


- In fact we can check that we cannot perform a 2-opt move

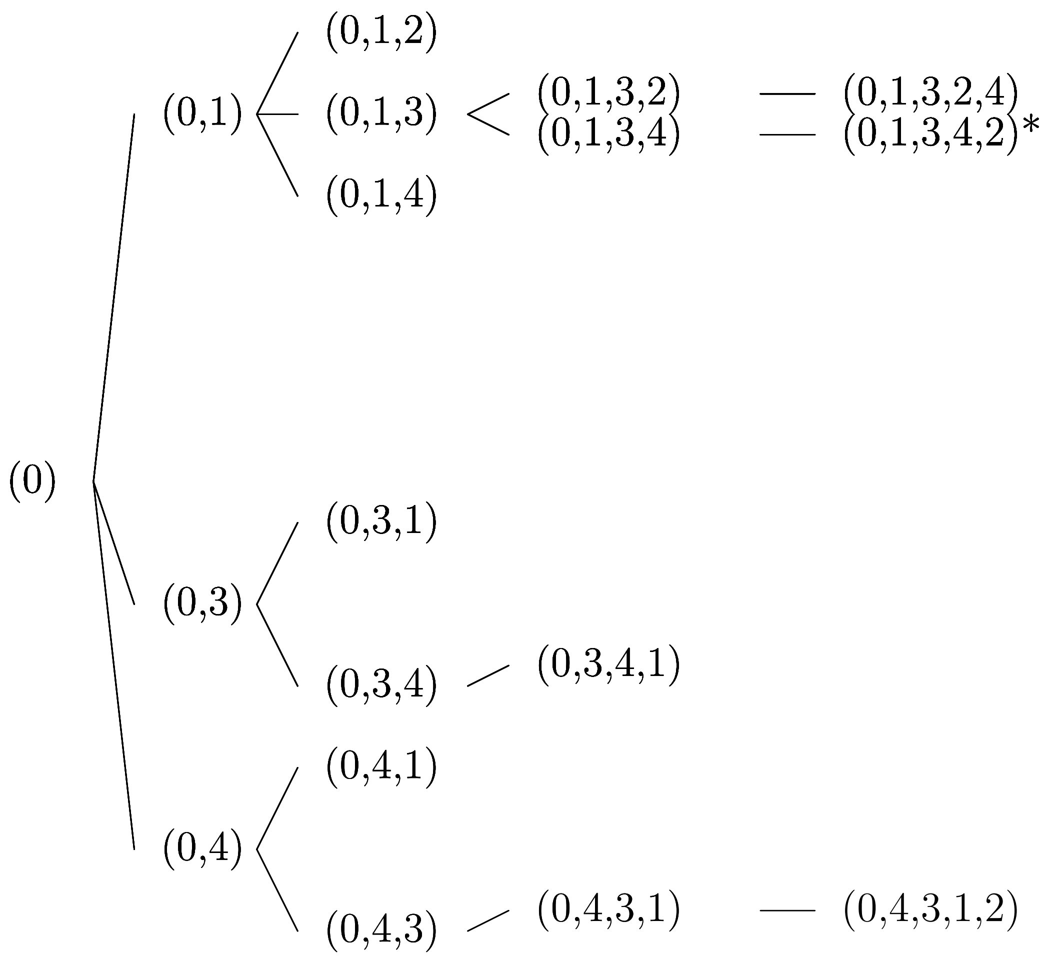
- We can also halve the search by considering only one direction—for example, by insisting we visit city 1 before city 2

Good Starting Bound

- It helps to start with a good bound
- We can use an *incomplete heuristic algorithm* to find a good solution which will act as a starting bound
- One very simple heuristic is a greedy algorithm

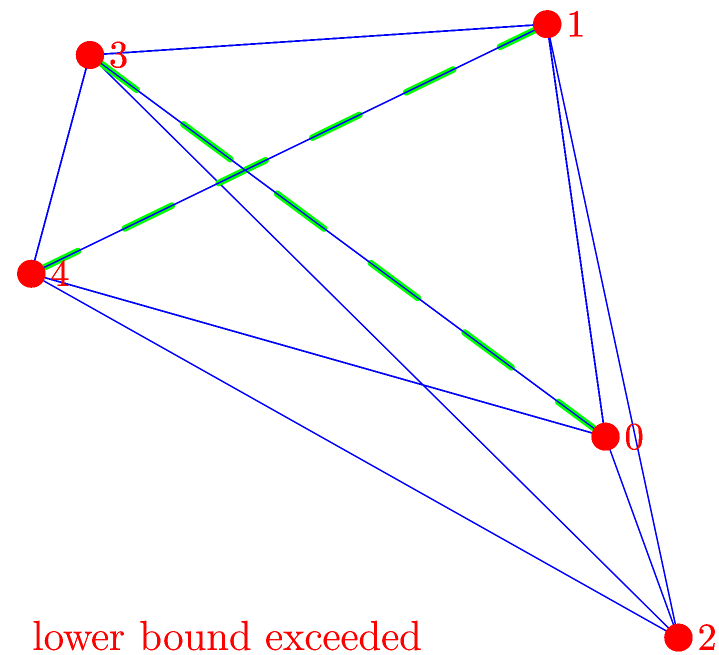


Branch and Bound after Pruning



bound = 301.88

length = [illegible]



lower bound exceeded

Not 2-opt

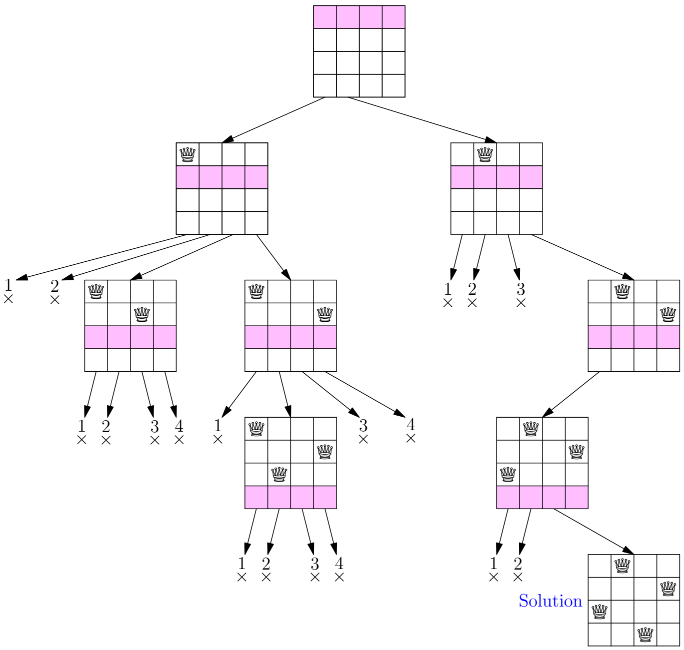
(0,[illegible])

Applications of Branch and Bound

- Branch and bound works for many optimisation problems
- It's drawback is that you often end up still searching an exponentially large search space even though it might be massively faster than exhaustive enumeration
- To make it work well requires considerable work
- This is not an instantaneous algorithm, you may be waiting hours before you find a solution
- For really large problems branch and bound might be too slow

Outline

1. Search Trees
2. Backtracking
3. Branch and Bound
4. **Search in AI**



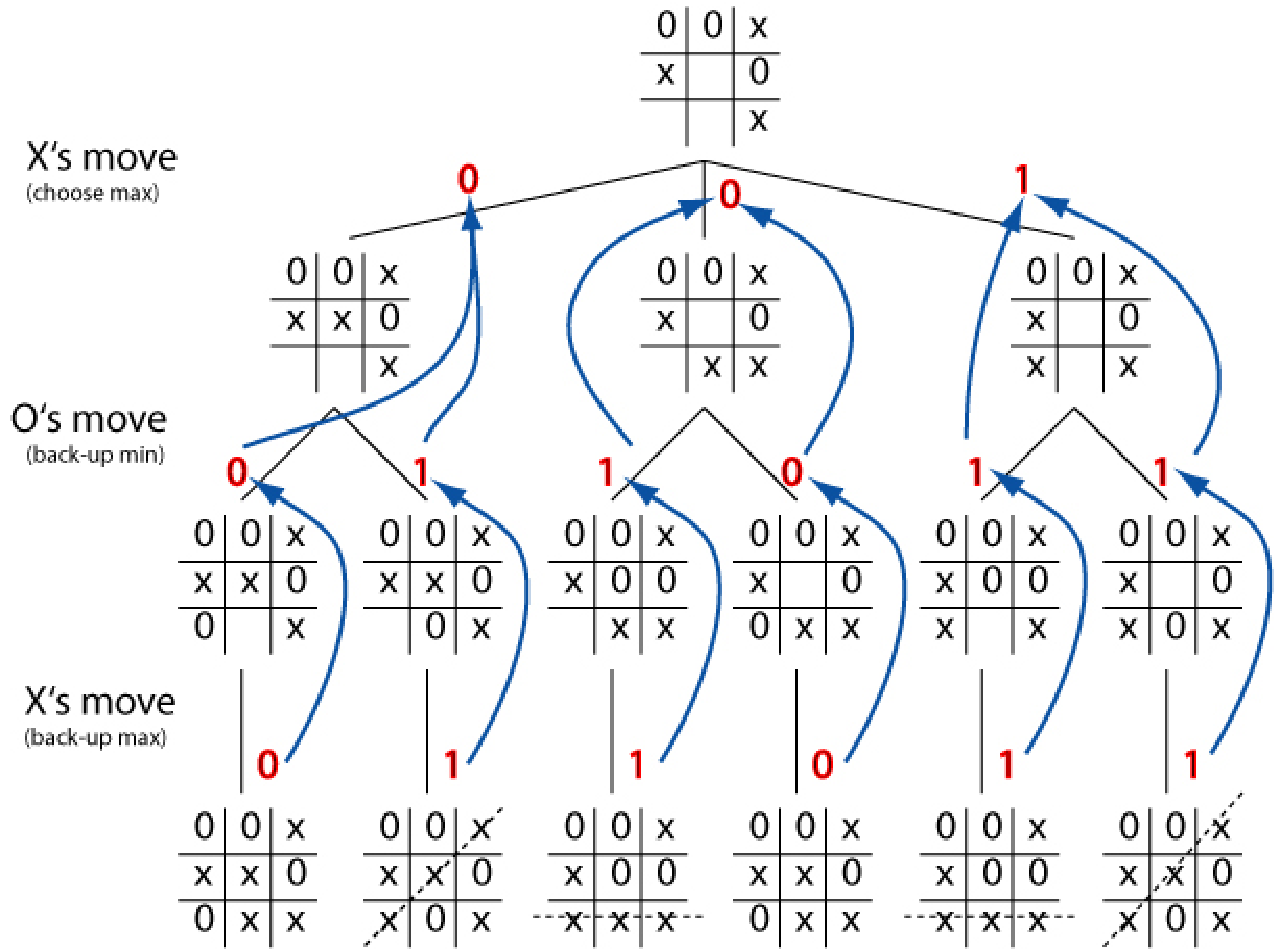
Other Search Strategies

- Search is a big topic in AI
- The algorithms used depends on the information available
- A classic search scenario is when there is "heuristic" information which provides a hint as to where an optimal solution lies
- Algorithms such as A^* exist which will finds the best route given an (admissible) heuristic as efficiently as possible
- You should learn about this next year in AI

Planning and Game Paying

- Search is also used to find the best action to take in planning problems and game playing (e.g. computer chess)
- Again it is useful to think in terms of a search tree
- Searching all paths on the search tree is usually infeasible
- Look for ways of pruning the search tree to focus on good moves
- Strategies include *minimax* and *alpha-beta pruning*

Minimax with Alpha-Beta Prunning



Lessons

- Search has many applications
- It is helpful to consider the search space as a tree whose branch corresponds to possible actions
- Backtracking is useful in search trees with constraints
- For optimisation problems branch and bound uses backtracking and costs of partial solutions as constraints
- Widely applicable, but can take too long