Coping with the Limitations of Algorithm Power

12.1 - 12.2

First, a brief homework review

5.1: 1abcd, 2abc 5.2: 5ab 6.1: 1, 4 6.2: 1, 3 6.6: 1, 11 7: Given n friends, each one can remain single or can be paired up with some other friend. Each friend can be paired only once. Find out the total number of ways in which friends can remain single or can be paired up.	9.1: 9, 11 9.2: 1, 4 9.3: 2 9.4: 1 10.1: 2 10.2: 7 11.1: #8 and 11.2: #6

Backtracking and Branch-and-Bound

Sections 12.1 and 12.2 introduce two algorithm design techniques—backtracking and branch-and-bound—that often make it possible to solve at least some large instances of difficult combinatorial problems.

Backtracking

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

Branch-and-Bound

In general, given an NP-Hard problem (problem for which we cannot prove that a polynomial time solution exists), a branch and bound algorithm explores the entire search space of possible solutions and provides an optimal solution.

Backtracking and Branch-and-Bound vs Brute Force

Unlike exhaustive search, they construct candidate solutions one component at a time and evaluate the partially constructed solutions: if no potential values of the remaining components can lead to a solution, the remaining components are not generated at all. This approach makes it possible to solve some large instances of difficult combinatorial problems, though, in the worst case, we still face the same curse of exponential explosion encountered in exhaustive search.

Backtracking and Branch-and-Bound

Both backtracking and branch-and-bound are based on the construction of a state-space tree whose nodes reflect specific choices made for a solution’s components.

Both techniques terminate a node as soon as it can be guaranteed that no solution to the problem can be obtained by considering choices that correspond to the node’s descendants.

Application

Branch-and-bound is applicable only to optimization problems because it is based on computing a bound on possible values of the problem’s objective function.

Backtracking is not constrained by this demand, but more often than not, it applies to non-optimization problems.

Backtracking

Backtracking is a more intelligent variation of a brute force approach. The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.

Backtracking

Backtracking is a more intelligent variation of a brute force approach. The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. If a partially constructed solution can be developed further without violating the problem’s constraints, it is done by taking the first remaining legitimate option for the next component.

Backtracking

Backtracking is a more intelligent variation of a brute force approach. The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. If a partially constructed solution can be developed further without violating the problem’s constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered.

Backtracking

Backtracking is a more intelligent variation of a brute force approach. The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. If a partially constructed solution can be developed further without violating the problem’s constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

Backtracking- example

For example, consider the SudoKo solving Problem, we try filling digits one by one. Whenever we find that current digit cannot lead to a solution, we remove it (backtrack) and try next digit.

This is better than naive approach (generating all possible combinations of digits and then trying every combination one by one)

The State-Space Tree

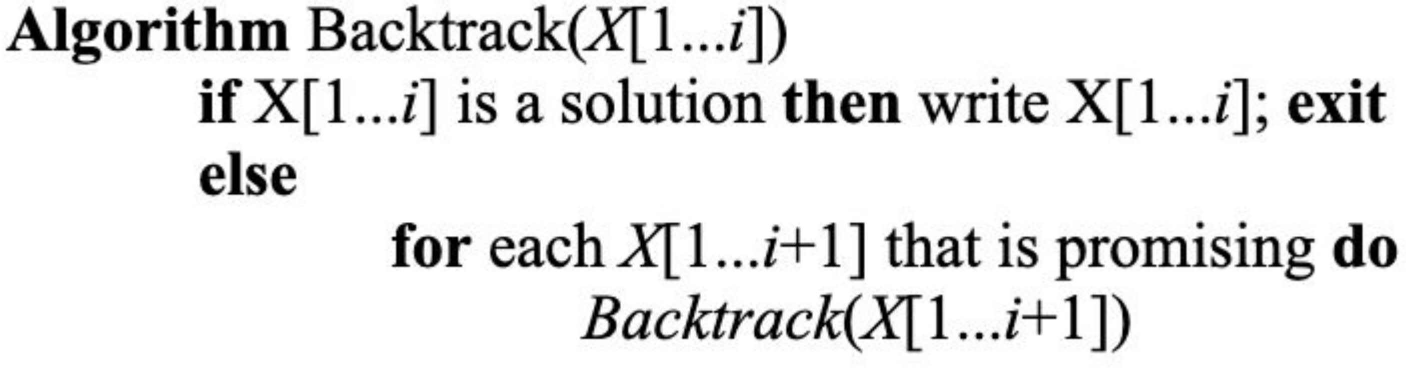
It is convenient to implement this kind of processing by constructing a tree of choices being made, called the state-space tree. Its root represents an initial state before the search for a solution begins.

The nodes of the first level in the tree represent the choices made for the first component of a solution, the nodes of the second level represent the choices for the second component, and so on.

Nodes

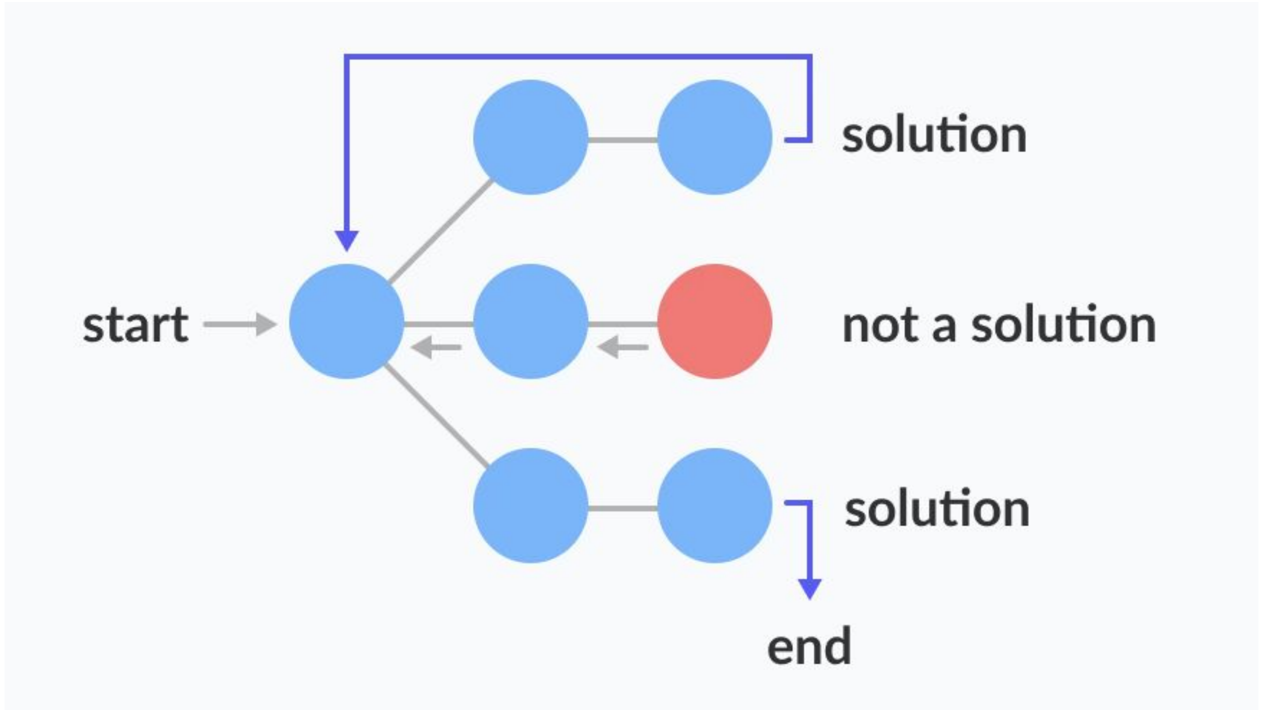
A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called non-promising.

Some general backtracking pseudocode



```
Algorithm Backtrack($X[1...i]$)
    if $X[1...i]$ is a solution then write $X[1...i]$; exit
    else
        for each $X[1...i+1]$ that is promising do
            Backtrack($X[1...i+1]$)
```

State Space Tree



In Class Example

Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches.

Constraint: Girl should not be on the middle bench.

In Class Example

Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches.

Constraint: Girl should not be on the middle bench.

Please take a moment to work on this

In Class Example

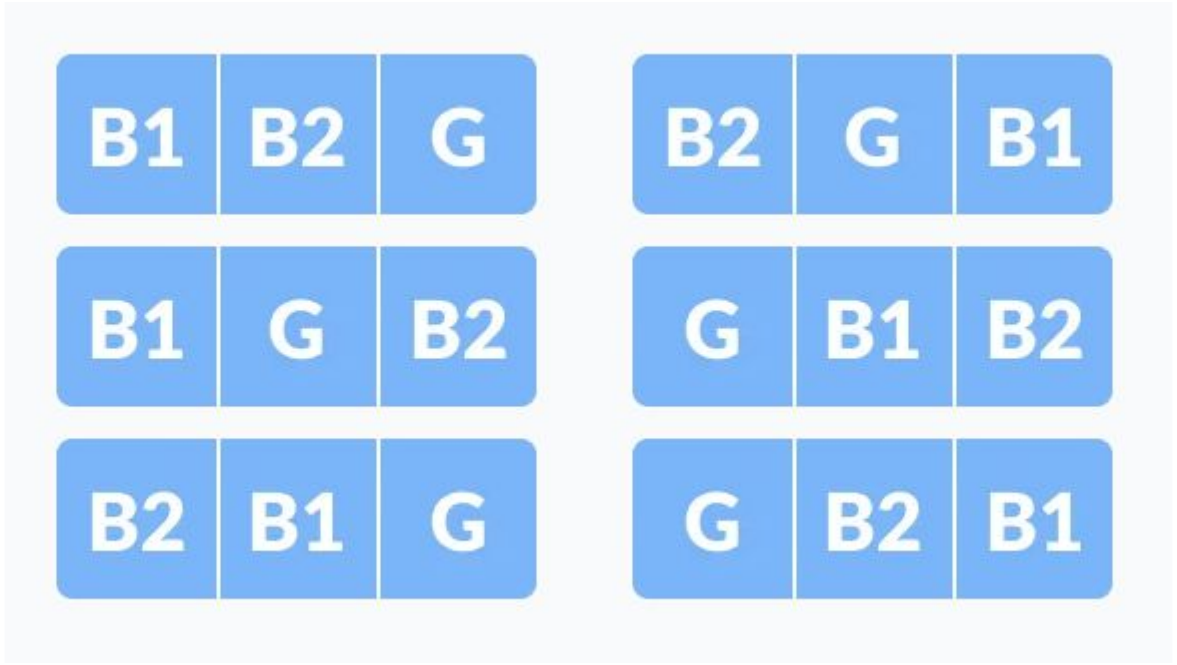
Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches.

Constraint: Girl should not be on the middle bench.

Solution: There are a total of 3! = 6 possibilities.

In Class Example

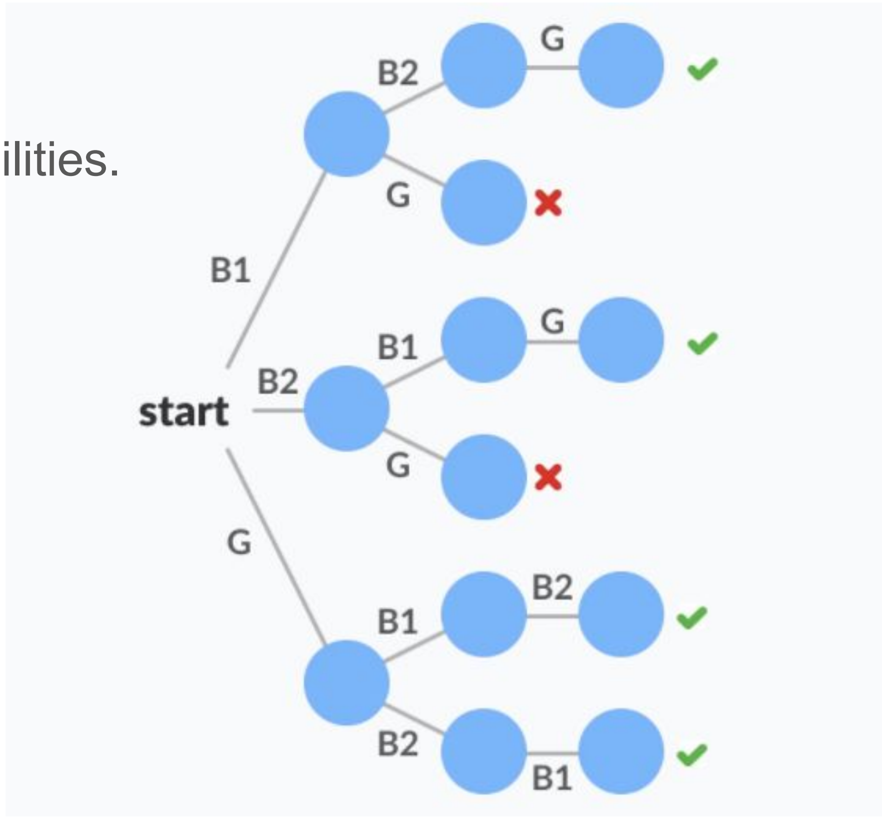
Solution: There are a total of 3! = 6 possibilities.



In Class Example

Solution: There are a total of 3! = 6 possibilities.

Here is the state space tree:



N Queens- example

The problem is to place n queens on an n × n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

N Queens- example

1) Start in the leftmost column

2) If all queens are placed return true

3) Try all rows in the current column.

Do following for every tried row.

a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.

b) If placing the queen in [row, column] leads to a solution then return true.

c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.

4) If all rows have been tried and nothing worked,

return false to trigger backtracking.

N Queens- example

The problem is to place n queens on an n × n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

https://www.youtube.com/watch?v=wGbuCyNpxIg

(skip the first 30 seconds)

So how efficient is backtracking?

In the worst case, it may have to generate all possible candidates in an exponentially (or faster) growing state space of the problem at hand. The hope, of course, is that a backtracking algorithm will be able to prune enough branches of its state-space tree before running out of time or memory or both. The success of this strategy is known to vary widely, not only from problem to problem but also from one instance to another of the same problem.

In summary

Backtracking constructs its state-space tree in the depth-first-search fashion in the majority of its applications.

If the sequence of choices represented by a current node of the state-space tree can be developed further without violating the problem's constraints, it is done by considering the first remaining legitimate option for the next component.

Otherwise, the method backtracks by undoing the last component of the partially built solution and replaces it by the next alternative.

Homework problems!

Please complete 12.1: #3

Branch and Bound

Recall that the central idea of backtracking is to cut off a branch of the problem’s state-space tree as soon as we can deduce that it cannot lead to a solution.

This idea can be strengthened further if we deal with an optimization problem. An optimization problem seeks to minimize or maximize some objective function.

Branch and Bound

Compared to backtracking, branch-and-bound requires two additional items:

1) a way to provide, for every node of a state-space tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
2) the value of the best solution seen so far

Branch and Bound

With that info, we can terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:

1. The value of the node’s bound is not better than the value of the best solution seen so far.

Branch and Bound

With that info, we can terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:

1. The value of the node’s bound is not better than the value of the best solution seen so far.
2. The node represents no feasible solutions because the constraints of the problem are already violated.

Branch and Bound

With that info, we can terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:

1. The value of the node’s bound is not better than the value of the best solution seen so far.
2. The node represents no feasible solutions because the constraints of the problem are already violated.
3. The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

N workers, N jobs- example

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment.

It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

N workers, N jobs- example

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment.

It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

https://www.youtube.com/watch?v=BV2MIZna6PI

In summary

Branch-and-bound is an algorithm design technique that enhances the idea of generating a state-space tree with the idea of estimating the best value obtainable from a current node of the decision tree: if such an estimate is not superior to the best solution seen up to that point in the processing, the node is eliminated from further consideration.

Compare/Contrast

Backtracking is used to find all possible solutions available to a problem. When it realises that it has made a bad choice, it undoes the last choice by backing it up. It searches the state space tree until it has found a solution for the problem.

Branch-and-Bound is used to solve optimisation problems. When it realises that it already has a better optimal solution that the pre-solution leads to, it abandons that pre-solution. It completely searches the state space tree to get optimal solution.