Coping with NP-Hardness

- Strategies to deal with hard problems
- Algorithms for Special Cases
- Fixed Parameter Tractability
- Intelligent Exhaustive Search
  - Backtracking
  - Branch and Bound
- Dynamic Programming based pseudo polynomial algorithm TSP

Imdad ullah Khan

Coping with NP-Hardness

Approaches to tackle hard problems

1. Special Cases: Relevant structure on which the problem is easy
   - Exact results in poly-time only for special cases or a range of parameters
2. Intelligent Exhaustive Search: Exponential time in worst case
   - The base and/or exponent are usually smaller
   - could be efficient on typical more realistic instances
   - Backtracking, Brand-and-Bound
3. Nearly exact solutions: Output is ‘close’ to exact (optimal) solution
   - Approximation Algorithms: Solutions of guaranteed quality in poly-time
   - Heuristic Algorithms: Solutions hopefully good in poly-time
4. Randomized Algorithms: Use coin flips for making decisions
   - Typically used for approximation, also used for easy problems

Intelligent Exhaustive Search

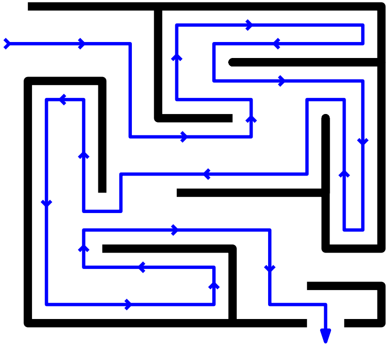
- Specific structure in instances is helpful sometimes
  - e.g. IND-SET(G, k) for trees is easy
  - 2-SAT is easy
- Sometime even a well-characterized special structure does not help
  - IND-SET(G, k) is NP-HARD even for planar graphs
  - 3-SAT is NP-HARD
- In many cases, we cannot neatly characterize the particular cases
- Can still avoid exp-time exhaustively searching with clever methods
- These algorithms are still exp time in the worst case
  - With the right ideas they are efficient on typical (likely) instances

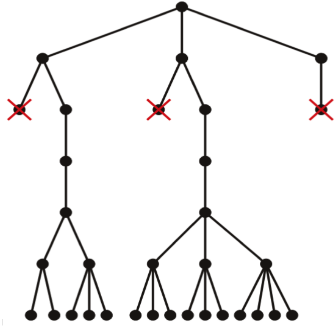
Backtracking

- Often solution to a problem can be made with a series of choices
- Each choice represents a partial solution
- These partial solutions form a tree (or DAG)
- Backtracking refers to a brute force solution where only feasible partial solutions are considered
- Feasibility and in-feasibility of partial solutions are determined given the specific problem in hand
- The idea in backtracking: **many partial solutions can be rejected quickly without completing it**

Backtracking

Finding path in a maze - backtrack when you reach a dead-end





Exhaustive Search for SAT

- Given a CNF formula f on n variables and m clauses
- The brute force algorithm
  - Check all 2^n possible assignments to the n variables
  - Determine in $O(m+n)$ whether an assignment is satisfying
  - Running time is $O(2^n(n+m))$
- Visualize it as a complete full binary tree
  - Root of the tree correspond to variable x_1
  - Left and right branches of root correspond to values of 1 and 0 for x_1
  - Left and right subtrees are all possibilities for variables $x_2, \ldots, x_n$

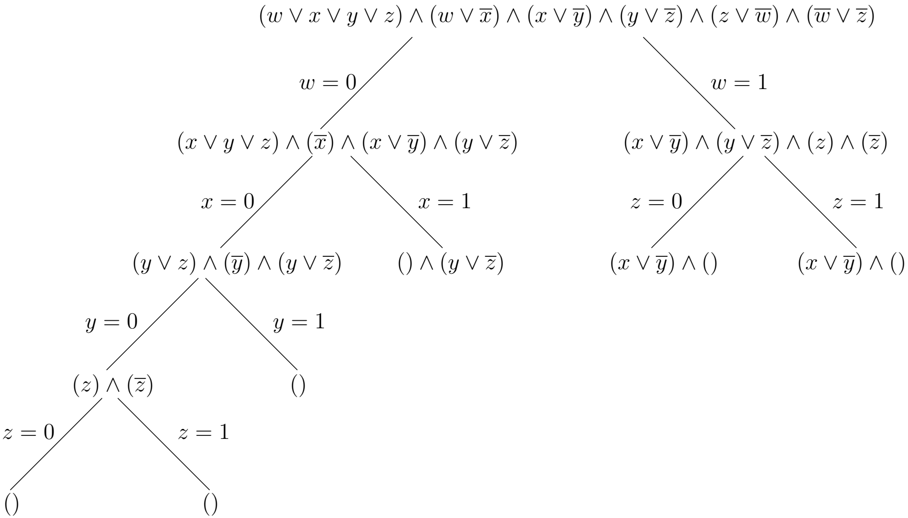
Intelligent Exhaustive Search for SAT

- Do not consider all 2^n branches of the binary tree (solution space)
- Carefully track each branch
- Stop when "get" a dead branch (cannot be extended to a solution)
  - $f = (\cdots) \wedge \cdots \wedge (x_6) \wedge \cdots (\cdots)$
  - Reject all solutions $(x_1, \ldots, x_n) \in \{0,1\}^n$ with $x_6 = 0$
  - Saves a lot- out of the 2^n sized search space, we eliminated 2^{n-1}
- A more elaborate example follows

Intelligent Exhaustive Search for SAT

- Do not consider all 2^n branches of the binary tree (solution space)
- Carefully track each branch
- Stop when "get" a dead branch (cannot be extended to a solution)
- When a literal in a clause is 1, the clause is satisfied we remove it
- When a literal in a clause is 0, the clause depends on other literals in it we remove the variable from it
- A partial assignment cannot satisfy the formula if there is an empty clause (no literal is 1)

Intelligent Exhaustive Search for SAT



General Backtracking Procedure

A backtracking algorithm requires a test that looks at a subproblem and quickly declares one of three outcomes:

- FAILURE: the subproblem has no solution
- SUCCESS: a solution to the subproblem is found
- UNCERTAINTY: not yet clear if it is either - need to explore further

Backtracking algorithm for problem P

Algorithm Backtracking procedure for Problem P, Instance I_0

$\mathcal{S} \leftarrow \{I_0\}$
while $\mathcal{S} \neq \emptyset$ **do**
  Choose a subproblem instance $I \in \mathcal{S}$
  $\mathcal{S} \leftarrow \mathcal{S} \setminus \{I\}$
  EXPAND I into $\{I_1, I_2, \ldots, I_k\}$
  for each I_j **do**
    if TEST(P_j) = SUCCESS **then**
      return the current solution
    else if TEST(P_j) = FAILURE **then**
      return NF
    else
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{I_j\}$
return NF

Backtracking for 3-SAT

- Exhaustive search takes $O(2^n \cdot (n + m))$ for a 3-CNF formula f on n variables and m clauses
- The previous approach was more variable centric
- Consider a more cluase centric approach
- View a 3-CNF formula f as $(\ell_1 \vee \ell_2 \vee \ell_3) \wedge (f')$ (unless f is empty)
- f' too is a (possibly empty) 3-CNF formula
- By the distributive law we get

$$f = (\ell_1 \vee \ell_2 \vee \ell_3) \wedge (f')$$

$$\implies f = (\ell_1 \wedge f') \vee (\ell_2 \wedge f') \vee (\ell_3 \wedge f')$$

Backtracking for 3-SAT

$$f = (\ell_1 \vee \ell_2 \vee \ell_3) \wedge (f') \implies f = (\ell_1 \wedge f') \vee (\ell_2 \wedge f') \vee (\ell_3 \wedge f')$$

- $f[x = \textbf{true}]$ (f with the value of x plugged in as **true**)

Algorithm Backtracking for 3-SAT

```
function CHECK-SAT(f)
  if f is empty then
    return true
  else
    Let f = (ℓ1 ∨ ℓ2 ∨ ℓ3) ∧ (f')
    if CHECK-SAT(f'[ℓ1 = true]) then        ▷ implies l1 ∧ f' = true
      return true
    if CHECK-SAT(f'[ℓ2 = true]) then
      return true
    if CHECK-SAT(f'[ℓ3 = true]) then
      return true
    return false
```

Backtracking for 3-SAT

$T(n)$: runtime of this algorithm for a f on n variables with m clauses

$$T(n) = \begin{cases} 3T(n-1) + O(poly(n,m)) & \text{if } n \geq 1 \\ 1 & \text{otherwise} \end{cases}$$

$T(n) = O(3^n \cdot poly(n,m))$ ▷ Simple recursion tree expansion

- Even worse that the variable centric brute-force search
- Observe the overlap in the subproblems - unnecessary repetitions

Backtracking for 3 − SAT

- Need to make these subproblems mutually exclusive
- Every satisfying assignment this algorithm finds (since it satisfies the clause $(\ell_1 \vee \ell_2 \vee \ell_3)$) must be exactly one of the following types
  - $\ell_1 =$ **true**
  - $\ell_1 =$ **false** $\wedge\ \ell_2 =$ **true**
  - $\ell_1 =$ **false** $\wedge\ \ell_2 =$ **false** $\wedge\ \ell_3 =$ **true**
- We can pinpoint any of of these three types of satisfying assignments to three literals in exactly one of the recursive calls

Backtracking for 3 − SAT

- Here is the clause centric algorithm based on this idea

Algorithm Backtracking for 3-SAT

function CHECK-SAT(f)
  if f is empty **then**
    return true
  else
    Let $f = (\ell_1 \vee \ell_2 \vee \ell_3) \wedge (f')$
    if CHECK-SAT($f'[\ell_1 =$ **true**$]$) **then**
      return true
    if CHECK-SAT($f'[\ell_1 =$ **false** $\wedge\ \ell_2 =$ **true**$]$) **then**
      return true
    if CHECK-SAT($f'[\ell_1 =$ **false** $\wedge\ \ell_2 =$ **false** $\wedge\ \ell_2 =$ **true**$]$) **then**
      return true
    return false

Intelligent Exhaustive Search for 3 − SAT

Fixing values of k literals reduced number of variables in f by k

$$T(n) = \begin{cases} T(n-1) + T(n-2) + T(n-3) + O(poly(n,m)) & n \geq 1 \\ 1 & \text{else} \end{cases}$$

Closed form of this recurrence is $T(n) = O(1.84^n)$

- This is substantially faster than the $O(2^n)$ algorithm
- Even for $n \sim 100$ this is more than 4180 times faster