

## SPECIAL CASES

---

- Some times we do not really need to solve the original problem
- A less general variant might do (and might be easy)
- Examples include:
  - 2-SAT versus the full-blown SAT
  - most problems on graphs become easy when the graph is a tree

## APPROXIMATION ALGORITHMS

---

- Some times we do not need a perfect solution
- A “good enough” solution will do instead
- $\varepsilon$ -approximation algorithm  $A$ :

$$\frac{|opt(x) - A(x)|}{opt(x)} \leq \varepsilon$$

- $\mathcal{NP}$ -complete problems can be
  - **fully approximable** when they have  $\varepsilon$ -approximation algorithms for arbitrarily small values of  $\varepsilon$
  - **partly approximable** when  $\varepsilon$ -approximation algorithms exist for some  $\varepsilon$  but not all the way to 0
  - **inapproximable** when no  $\varepsilon$ -approximation algorithm exists

## BACKTRACKING

---

- We produce algorithms that are exponentially time bounded in general but often do much better
- One of the most efficient way to do this in practice is using **backtracking**:

**Algorithm** BACKTRACKING( $S_0$ : problem)

1.  $A \leftarrow \{S_0\}$
  2. **while**  $A$  is not empty **do**
    - (a) choose a sub-problem  $S$  from  $A$  and remove it from  $A$
    - (b) choose a way of branching out  $S$  into sub-problems  $S_1, S_2, \dots, S_n$
    - (c) **foreach**  $S_i \in \{S_1, \dots, S_n\}$  **do**
      - i. **if** TEST( $S_i$ ) = “found” **then** halt
      - ii. **else** TEST( $S_i$ ) = “unknown” **then** add  $S_i$  to  $A$
  3. **return** “no solution”
- Varied strategies of traversing sub-problems (each with advantages/disadvantages)
    - How do we add the sub-problems  $S_1, S_2, \dots, S_n$  back to  $A$ ?

## BACKTRACKING (CONT'D)

---

- Basic backtracking also has a straightforward **recursive definition**

**Algorithm** BACKTRACKING( $S_0$ : problem)

1. **if** TEST( $S_0$ ) = “found” **then return** solution for  $S$
  2. **else**
    - (a) Choose a way of branching out  $S$  into sub-problems  $S_1, S_2, \dots, S_n$
    - (b) Combine BACKTRACKING( $S_1$ ),  $\dots$ , BACKTRACKING( $S_n$ ) and return the result
- Issues specific to every particular problem:
    - How to split into sub-problems
    - How to test for elementary solutions

## BRANCH AND BOUND

---

- Backtracking is especially efficient for binary (yes/no) problems
- For more complex (namely, optimization) problems we can do better:

**Algorithm** BRANCH-AND-BOUND( $S_0$ : problem)

1.  $A \leftarrow \{S_0\}$ ,  $bestsofar = \infty$
  2. **while**  $A$  is not empty **do**
    - (a) choose a sub-problem  $S$  from  $A$  and remove it from  $A$
    - (b) choose a way of branching out  $S$  into sub-problems  $S_1, S_2, \dots, S_n$
    - (c) **foreach**  $S_i \in \{S_1, \dots, S_n\}$  **do**
      - i. **if**  $S_i$  is a complete solution **then** update  $bestsofar$
      - ii. **else if** LOWERBOUND( $S_i$ ) <  $bestsofar$  **then** add  $S_i$  to  $A$
  3. **return** solution associated with  $bestsofar$
- Same design issues, plus how to compute LOWERBOUND
  - Other methods include heuristics, local improvements
    - Really the realm of **artificial intelligence**