Introduction to Artificial Intelligence

Backtracking

Andres Mendez-Vazquez

April 23, 2019

Outline

Outline

In the Beginning

James Bernoulli 17th Century [1]

- He successfully used the principle to solve the “Tot tibi sunt dotes”

Basically a Combinatoric Problem

- How many ways exist to write such words by permuting them?

tibi sunt tot dotes

sunt tibi tot dotes

...

His notes with the traces exist

- They look as a classic backtracking algorithm...

In the Beginning

James Bernoulli 17th Century [1]

- He successfully used the principle to solve the “Tot tibi sunt dotes”

Basically a Combinatoric Problem

- How many ways exist to write such words by permuting them?

tibi sunt tot dotes

sunt tibi tot dotes

...

His notes with the traces exist

- They look as a classic backtracking algorithm...

In the Beginning

James Bernoulli 17th Century [1]

- He successfully used the principle to solve the "Tot tibi sunt dotes"

Basically a Combinatoric Problem

- How many ways exist to write such words by permuting them?

tibi sunt tot dotes

sunt tibi tot dotes

...

His notes with the traces exist

- They look as a classic backtracking algorithm...

Further

Around 1882

- Edourdad Lucas credited his student Tremaux
  - About the use of depth-first search in walk of a tree...

The problem of the 8 queens was first proposed in 1948-1950

- By Max Bezzel and Franz Nauck

An here comes Gauss, the prince in mathematics

- He saw the publications by Franz Nauck and wrote several letters to his friend H.C. Schumacher...

Further

Around 1882

- Edourdad Lucas credited his student Tremaux
  - About the use of depth-first search in walk of a tree...

The problem of the 8 queens was first proposed in 1948-1950

- By Max Bezzel and Franz Nauck

An here comes Gauss, the prince in mathematics

- He saw the publications by Franz Nauck and wrote several letters to his friend H.C. Schumacher...

Further

Around 1882

- Edourdad Lucas credited his student Tremaux
  - About the use of depth-first search in walk of a tree...

The problem of the 8 queens was first proposed in 1948-1950

- By Max Bezzel and Franz Nauck

An here comes Gauss, the prince in mathematics

- He saw the publications by Franz Nauck and wrote several letters to his friend H.C. Schumacher...

And in the letter dated 27 of September 1850

Gauss explained how to find all the solutions by Backtracking

- He called the procedure “Tatonniren” from French “to feel one’s way.”

Finally

Computer arrived finally 100 years latter

- And the technique was fully described by Robert J. Walker

He introduced a full description of Backtracking

- We will review it latter on...

Finally

Computer arrived finally 100 years latter

- And the technique was fully described by Robert J. Walker

He introduced a full description of Backtracking

- We will review it latter on...

Outline

n Chess Queen Problem

A Puzzle

- Given a $n \times n$ chessboard, How to position n queens such that they cannot attack each other?

Remember

- The Queen attacks any piece in the same row, column or either diagonal.

n Chess Queen Problem

A Puzzle

- Given a $n \times n$ chessboard, How to position n queens such that they cannot attack each other?

Remember

- The Queen attacks any piece in the same row, column or either diagonal.

Outline

The First Step

Starting from an empty board

- Place a Queen in the first column... then the second row... and so on

Then after doing so

- We try in the next column and proceed recursively

The First Step

Starting from an empty board

- Place a Queen in the first column... then the second row... and so on

Then after doing so

- We try in the next column and proceed recursively

The Backtracking Step

If we get stuck at some column k

- Then, we **backtrack** to the previous column $k - 1$

Observe the tree is constructed in Preorder

- We do calculations before we move to any possible options...

The Backtracking Step

If we get stuck at some column k

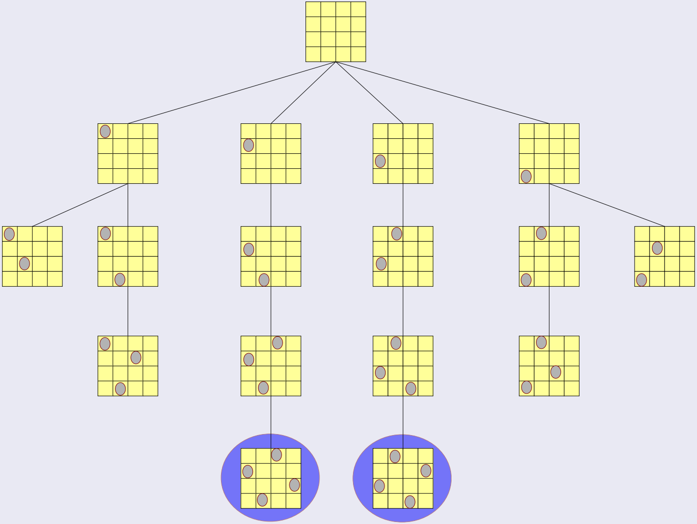
- Then, we **backtrack** to the previous column $k-1$

Observe the tree is constructed in Preorder

- We do calculations before we move to any possible options...

Example

Something Notable



Outline

Codification

How do we get compact and easy to use?

Position Column	1	2	3	4
Row Position	2	4	1	3

A simple vector

2	4	1	3

What about?

- We need to have a way to know when a queen can attack another.

Codification

How do we get compact and easy to use?

Position Column	1	2	3	4
Row Position	2	4	1	3

A simple vector

2	4	1	3

What about?

- We need to have a way to know when a queen can attack another.

Codification

How do we get compact and easy to use?

Position Column	1	2	3	4
Row Position	2	4	1	3

A simple vector

2	4	1	3

What about?

- We need to have a way to know when a queen can attack another.

We can have three boolean arrays

Array A

- It indicates if a row does not contain a queen.

Array B

- It indicates if a front diagonal does not contain a queen
  - Indexation is the sum of row and columns.

Array C

- It indicates if a back diagonal does not contain a queen.
  - Indexation is the difference of row and columns.

We can have three boolean arrays

Array A

- It indicates if a row does not contain a queen.

Array B

- It indicates if a front diagonal does not contain a queen
  - Indexation is the sum of row and columns.

Array C

- It indicates if a back diagonal does not contain a queen.
  - Indexation is the difference of row and columns.

We can have three boolean arrays

Array A

- It indicates if a row does not contain a queen.

Array B

- It indicates if a front diagonal does not contain a queen
  - Indexation is the sum of row and columns.

Array C

- It indicates if a back diagonal does not contain a queen.
  - Indexation is the difference of row and columns.

With the following indexes

For the array A

- The indexing goes from 1 to 4.

For the array B

- The indexing goes from 2 to 8.

For the array C

- The indexing goes from -3 to 3.

With the following indexes

For the array A

- The indexing goes from 1 to 4.

For the array B

- The indexing goes from 2 to 8.

For the array C

- The indexing goes from -3 to 3.

With the following indexes

For the array A

- The indexing goes from 1 to 4.

For the array B

- The indexing goes from 2 to 8.

For the array C

- The indexing goes from -3 to 3.

We have something quite interesting

Something Notable

- The sum of the row and column indices is constant along diagonals
- The difference of the row and column indices is constant along diagonals

We can see that in the 4 Queen chessboard

- Take a look at the board...

We have something quite interesting

Something Notable

- The sum of the row and column indices is constant along diagonals
- The difference of the row and column indices is constant along diagonals

We can see that in the 4 Queen chessboard

- Take a lock at the board...

Code for the 8 Queens

Queen$(col : \mathbb{N})$:

local $row : \mathbb{N}$

Init a, b, c to true

for $row = 1$ **to** 8

    if $a[row]$ **and** $b[row + col]$ **and** $c[row - col]$ **then**

        $x[col] = row$

        $a[row] = b[row + col] = c[row - col] = False$

        if $col < 8$ **then Queen**$(col + 1)$ **else Print** x

        $a[row] = b[row + col] = c[row - col] = True$

Outline

Some Notes

Properties

- It uses Depth-First Search.
- It takes a sequence $V = \{x_1, x_2, ..., x_n\}$ of variables of X to be instantiated (Initially X including all the variables).
- An initially empty instantiation I as arguments.

Some Notes

Properties

- It uses Depth-First Search.
- It takes a sequence $V = \{x_1, x_2, ..., x_n\}$ of variables of X to be instantiated (Initially X including all the variables).
- An initially empty instantiation I as arguments.

Some Notes

Properties

- It uses Depth-First Search.
- It takes a sequence $V = \{x_1, x_2, ..., x_n\}$ of variables of X to be instantiated (Initially X including all the variables).
- An initially empty instantiation I as arguments.

Outline

Here, we can use something quite elegant

The use of recursion

- Here, we can use a recursion algorithm that at a certain node with assignaimets

$$p = \langle x_1 = a_1, x_2 = a_2, ..., x_j = a_j \rangle$$

Then, a new variable is added and a search in the possible values done

$$p' = \langle x_1 = a_1, x_2 = a_2, ..., x_j = a_j, x_{j+1} \rangle$$

We can see this

- I the naive backtracking algorithm.

Here, we can use something quite elegant

The use of recursion

- Here, we can use a recursion algorithm that at a certain node with assignaimets

$$p = \langle x_1 = a_1, x_2 = a_2, ..., x_j = a_j \rangle$$

Then, a new variable is added and a search in the possible values done

$$p' = \langle x_1 = a_1, x_2 = a_2, ..., x_j = a_j, x_{j+1} \rangle$$

We can see this

- I the naive backtracking algorithm.

Here, we can use something quite elegant

The use of recursion

- Here, we can use a recursion algorithm that at a certain node with assignaimets

$$p = \langle x_1 = a_1, x_2 = a_2, ..., x_j = a_j \rangle$$

Then, a new variable is added and a search in the possible values done

$$p' = \langle x_1 = a_1, x_2 = a_2, ..., x_j = a_j, x_{j+1} \rangle$$

We can see this

- I the naive backtracking algorithm.

Recursive Backtracking Algorithm

BackTracking(V, I)

```
If V = ∅ then
    I is a solution
else
    Let x_i ∈ V
    for each v ∈ D_{x_i} do
        If I ∪ {(x_i, v)} is consistent then
            BackTracking (V − {x_i}, I ∪ (x_i, v))
```

Outline

Clearly

Any Recursive Version of the Algorithm

- It can be converted into an iterative version by means...

Extra memory

- Of an stack to simulate the depth-first search of the software stack...

Clearly

Any Recursive Version of the Algorithm

- It can be converted into an iterative version by means...

Extra memory

- Of an stack to simulate the depth-first search of the software stack...

Now, we have an iterative version

Here, we have the following set

- $S_k = \langle s = x_1, x_2, ..., x_k \rangle$ a sequence of states

A always a Boolean statement

$$P\left(x_1, x_2, ..., x_k\right)$$

- That it turns TRUE when reaching a feasible solution

Now, we have an iterative version

Here, we have the following set

- $S_k = \langle s = x_1, x_2, ..., x_k \rangle$ a sequence of states

A always a Boolean statement

$$P(x_1, x_2, ..., x_k)$$

- That it turns TRUE when reaching a feasible solution

Iterative Backtracking Algorithm

BackTracking

$k = 1$

Generate S_1 a stack with the initial states

while $k > 0$:

  while $S_k \neq \emptyset$

    $\triangleright$ Advance to next position

    $x_k = pop(S)$

    $p_k = \langle x_1, x_2, ..., x_{k-1} \rangle \circ x_k$

    If $P(p_k)$ then return p_k

    $T =$ Generate the new expansion from x_k

    $k = k + 1$

    Generate new stack S_k

    $push(T, S_k)$

  $k = k - 1$

Outline

Backtracking

Something Notable

Backtracking is based on that it is often possible to reject a solution by looking at just a small portion of it.

Example

If an instance of SAT contains the clause $C_i = (x_1 \vee x_2)$, then all assignments with $x_1 = x_2 = 0$ can be instantly eliminated.

Backtracking

Something Notable

Backtracking is based on that it is often possible to reject a solution by looking at just a small portion of it.

Example

If an instance of SAT contains the clause $C_i = (x_1 \vee x_2)$, then all assignments with $x_1 = x_2 = 0$ can be instantly eliminated.

Example

Pruning Example

Given the possible values that you can give to two literals:

x_1	x_2
1	1
1	0
0	1
0	**0**

It is possible to prune a quarter of the entire search space... Can this be systematically exploited?

An example of exploiting this idea in SAT solvers

Consider the following Boolean formula $\phi(w,x,y,z)$

$(w \lor x \lor y \lor z) \land (w \lor \neg x) \land (x \lor \neg y) \land (y \lor \neg z) \land (z \lor \neg w) \land (\neg w \lor \neg z)$

We start branching in one variable, we can choose w

Note: This selection does not violate any of the clauses of $\phi(w,x,y,z)$

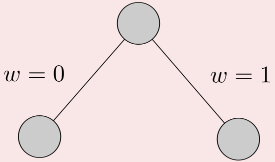
An example of exploiting this idea in SAT solvers

Consider the following Boolean formula $\phi(w, x, y, z)$

$(w \vee x \vee y \vee z) \wedge (w \vee \neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg w) \wedge (\neg w \vee \neg z)$

We start branching in one variable, we can choose w

Initial formula ϕ

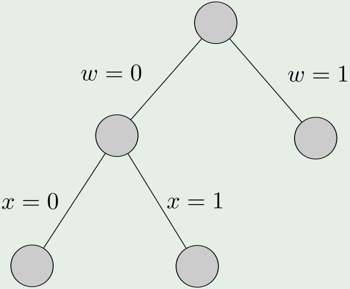


Note: This selection does not violate any of the clauses of $\phi(w, x, y, z)$

Now

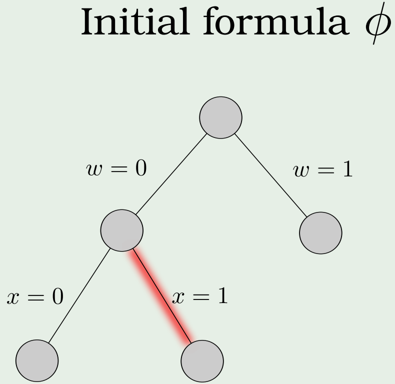
The partial assignment $w = 0, x = 1$ violates the clause $(w \vee \neg x)$

Initial formula ϕ



Now

Then, we prune that branch



In addition

What if $w = 0, x = 0$

Then, the following clauses are satisfied

1. $\neg w = 1$
2. $\neg x = 1$

Thus, we have the following left

1. Before
   1. $(w \vee x \vee y \vee z) \wedge (w \vee \neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg w) \wedge (\neg w \vee \neg z)$
2. After
   1. $(0 \vee 0 \vee y \vee z) \wedge (0 \vee 1) \wedge (0 \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee 1) \wedge (1 \vee \neg z)$

In addition

What if $w = 0, x = 0$

Then, the following clauses are satisfied

1. $\neg w = 1$
2. $\neg x = 1$

Thus, we have the following left

1. Before
   1. $(w \vee x \vee y \vee z) \wedge (w \vee \neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg w) \wedge (\neg w \vee \neg z)$
2. After
   1. $(0 \vee 0 \vee y \vee z) \wedge (0 \vee 1) \wedge (0 \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee 1) \wedge (1 \vee \neg z)$

Finally

We have the following reduced number of equations

$$(y \vee z), (1), (\neg y), (y \vee \neg z), (1), (1) \Leftrightarrow \boldsymbol{(y \vee z), (\neg y), (y \vee \neg z)}$$

What if $w = 0, x = 1$

1. Before
   1. $(w \vee x \vee y \vee z) \wedge (w \vee \neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg w) \wedge (\neg w \vee \neg z)$
2. After
   1. $(1) \wedge (0) \wedge (1) \wedge (y \vee \neg z) \wedge (1) \wedge (1)$

Finally

We have the following reduced number of equations

$(y \vee z), (1), (\neg y), (y \vee \neg z), (1), (1) \Leftrightarrow \boldsymbol{(y \vee z), (\neg y), (y \vee \neg z)}$

What if $w = 0, x = 1$

1. Before
   1. $(w \vee x \vee y \vee z) \wedge (w \vee \neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg w) \wedge (\neg w \vee \neg z)$
2. After
   1. $(1) \wedge (0) \wedge (1) \wedge (y \vee \neg z) \wedge (1) \wedge (1)$

Thus

We have something no satisfiable

$$(1) \wedge (0) \wedge (1) \wedge (y \vee \neg z) \wedge (1) \wedge (1) \Leftrightarrow (), (y \vee \neg z)$$

Clearly

We prune that part of the search tree.

Note: we use "()≡(0)" to point out to a "empty clause" ruling out satisfiability.

Thus

We have something no satisfiable

$$(1) \wedge (0) \wedge (1) \wedge (y \vee \neg z) \wedge (1) \wedge (1) \Leftrightarrow (), (y \vee \neg z)$$

Clearly

We prune that part of the search tree.

Note: we use "$() \equiv (0)$" to point out to a "empty clause" ruling out satisfiability.

The decisions we need to make in backtracking

First

Which subproblem to expand next.

Second

Which branching variable to use.

Remark

The benefit of backtracking lies in its ability to eliminate portions of the search space.

The decisions we need to make in backtracking

First

Which subproblem to expand next.

Second

Which branching variable to use.

Remark

The benefit of backtracking lies in its ability to eliminate portions of the search space.

The decisions we need to make in backtracking

First

Which subproblem to expand next.

Second

Which branching variable to use.

Remark

The benefit of backtracking lies in its ability to eliminate portions of the search space.

Choosing

Something Notable

A classic strategy:

- You choose the subproblem that contains the smallest clause.
- Then, you branch on a variable in that clause.

Choosing

Something Notable

A classic strategy:

- You choose the subproblem that contains the smallest clause.
- Then, you branch on a variable in that clause.

Then

If the clause is a singleton then at least one of the resulting branches will be terminated.

Choosing

Something Notable

A classic strategy:

- You choose the subproblem that contains the smallest clause.
- Then, you branch on a variable in that clause.

Then

If the clause is a singleton then at least one of the resulting branches will be terminated.

Choosing

Something Notable

A classic strategy:

- You choose the subproblem that contains the smallest clause.
- Then, you branch on a variable in that clause.

Then

If the clause is a singleton then at least one of the resulting branches will be terminated.

The Backtracking Test

The test needs to look at the subproblem to declare quickly if

1. **Failure**: the subproblem has no solution.
2. **Success**: a solution to the subproblem is found.
3. **Uncertainty**.

The Backtracking Test

The test needs to look at the subproblem to declare quickly if

1. **Failure**: the subproblem has no solution.
2. **Success**: a solution to the subproblem is found.
3. **Uncertainty**.

What about SAT

- The test declares failure if there is an empty clause
- The test declares success if there are no clauses
- Uncertainty Otherwise.

The Backtracking Test

The test needs to look at the subproblem to declare quickly if

1. **Failure**: the subproblem has no solution.
2. **Success**: a solution to the subproblem is found.
3. **Uncertainty**.

What about SAT

- The test declares failure if there is an empty clause
- The test declares success if there are no clauses
- Uncertainty Otherwise.

The Backtracking Test

The test needs to look at the subproblem to declare quickly if

1. **Failure**: the subproblem has no solution.
2. **Success**: a solution to the subproblem is found.
3. **Uncertainty**.

What about SAT

- The test declares failure if there is an empty clause
- The test declares success if there are no clauses
- Uncertainty Otherwise.

The Backtracking Test

The test needs to look at the subproblem to declare quickly if

1. **Failure**: the subproblem has no solution.
2. **Success**: a solution to the subproblem is found.
3. **Uncertainty**.

What about SAT

- The test declares failure if there is an empty clause
- The test declares success if there are no clauses
- Uncertainty Otherwise.

The Backtracking Test

The test needs to look at the subproblem to declare quickly if

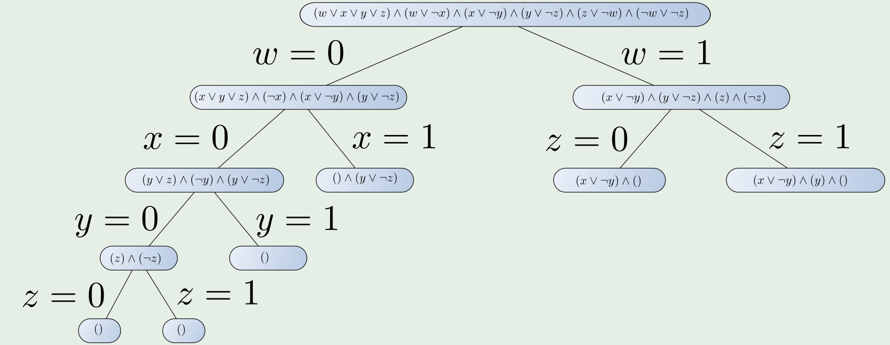
1. **Failure**: the subproblem has no solution.
2. **Success**: a solution to the subproblem is found.
3. **Uncertainty**.

What about SAT

- The test declares failure if there is an empty clause
- The test declares success if there are no clauses
- Uncertainty Otherwise.

Example

We have the following



Outline

Pseudo-code for Backtracking

We have

BACKTRACKING(P_0)

1. Start with some problem P_0
2. Let $\mathcal{S} = \{P_0\}$, the set if active subproblems
3. While $\mathcal{S} \neq \emptyset$
4. **choose** a subproblem $P \in \mathcal{S}$ and remove it from $\mathcal{S}$
5. **expand** it into smaller subproblems $P_1, P_2, ..., P_k$
6. For each P_i
7. if test(P_i) succeeds: halt and return the branch solution
8. if test(P_i) fails: discard P_i
9. Otherwise: add P_i to $\mathcal{S}$
10. return there is no solution.

Pseudo-code for Backtracking

We have

BACKTRACKING(P_0)

1. Start with some problem P_0
2. Let $\mathcal{S} = \{P_0\}$, the set if active subproblems
3. While $\mathcal{S} \neq \emptyset$
4.   **choose** a subproblem $P \in \mathcal{S}$ and remove it from $\mathcal{S}$
5.   **expand** it into smaller subproblems $P_1, P_2, ..., P_k$
6.   For each P_i
7.   if test(P_i) succeeds: halt and return the branch solution
8.   if test(P_i) fails: discard P_i
9.   Otherwise: add P_i to $\mathcal{S}$
10. return there is no solution.

Pseudo-code for Backtracking

We have

BACKTRACKING(P_0)

1. Start with some problem P_0
2. Let $\mathcal{S} = \{P_0\}$, the set if active subproblems
3. While $\mathcal{S} \neq \emptyset$
4.   **choose** a subproblem $P \in \mathcal{S}$ and remove it from $\mathcal{S}$
5.   **expand** it into smaller subproblems $P_1, P_2, ..., P_k$
6.   For each P_i
7.   if test(P_i) succeeds: halt and return the branch solution
8.   if test(P_i) fails: discard P_i
9.   Otherwise: add P_i to $\mathcal{S}$
10. return there is no solution.

Pseudo-code for Backtracking

We have

BACKTRACKING(P_0)

1. Start with some problem P_0
2. Let $\mathcal{S} = \{P_0\}$, the set if active subproblems
3. While $\mathcal{S} \neq \emptyset$
4.   **choose** a subproblem $P \in \mathcal{S}$ and remove it from $\mathcal{S}$
5.   **expand** it into smaller subproblems $P_1, P_2, ..., P_k$
6.   For each P_i
7.   if test(P_i) succeeds: halt and return the branch solution
8.   if test(P_i) fails: discard P_i
9.   Otherwise: add P_i to $\mathcal{S}$
10. return there is no solution.

Outline

Choose and Expand

For SAT

1. The choose procedure picks a clause,
2. Expand picks a variable within that clause.

There has been already

A discussion on how to make such choices.

Choose and Expand

For SAT

1. The choose procedure picks a clause,
2. Expand picks a variable within that clause.

There has been already

A discussion on how to make such choices.

Notes

With the right test, expand, and choose routines

- Backtracking can be remarkably effective in practice

Further

- The backtracking algorithm we showed for SAT is the basis of many successful satisfiability programs

For Example, 2-SAT problems

- It is a conjunction (a Boolean and operation) of clauses,
- Where each clause is a disjunction (a Boolean or operation) of two variables or negated variables.

Notes

With the right test, expand, and choose routines

- Backtracking can be remarkably effective in practice

Further

- The backtracking algorithm we showed for SAT is the basis of many successful satisfiability programs

For Example, 2-SAT problems

- It is a conjunction (a Boolean and operation) of clauses,
- Where each clause is a disjunction (a Boolean or operation) of two variables or negated variables.

Notes

With the right test, expand, and choose routines

- Backtracking can be remarkably effective in practice

Further

- The backtracking algorithm we showed for SAT is the basis of many successful satisfiability programs

For Example, 2-SAT problems

- It is a conjunction (a Boolean and operation) of clauses,
- Where each clause is a disjunction (a Boolean or operation) of two variables or negated variables.

Then

Backtracking

- If presented with a 2-SAT instance,
  - it will always find a satisfying assignment, if one exists, in polynomial time!!!

Something Notable

- Therefore, we depend on the constraints!!!

These problems are known as

- Constraint Satisfaction Problems!!!

Then

Backtracking

- If presented with a 2-SAT instance,
  - it will always find a satisfying assignment, if one exists, in polynomial time!!!

Something Notable

- Therefore, we depend on the constraints!!!

These problems are known as

- Constraint Satisfaction Problems!!!

Then

Backtracking

- If presented with a 2-SAT instance,
  - it will always find a satisfying assignment, if one exists, in polynomial time!!!

Something Notable

- Therefore, we depend on the constraints!!!

These problems are known as

- Constraint Satisfaction Problems!!!

Bibliography

D. E. Knuth, *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 2.*
Pearson Education India, 2011.