Quantum speedup for backtracking algorithms

Mathys Rennela

(LIACS, Leiden University)

Recap on classical backtracking

(1) What is a Constraint Satisfaction Problem?
(2) Example of CSP: the k-SAT problem
(3) Unstructured classical brute force search
(4) What is a backtracking algorithm?

What is a Constraint Satisfaction Problem?

Definition: A constraint satisfaction problem (CSP) is a problem defined on *n* variables and specified by a set of constraints which must be satisfied by all variables.

Example: Map coloring, Sudoku, Crosswords, …

Remark: The best algorithms for CSPs tend to have an **exponential** runtime, when the problem is taken in all its generality.

Example of CSP: the k-SAT problem

3-SAT: Given a Boolean formula *F* in 3-conjunction normal form on *n* variables, is there a bit string *y* such that f(y)=1?

Such a bitstring y is called a satisfying assignment.

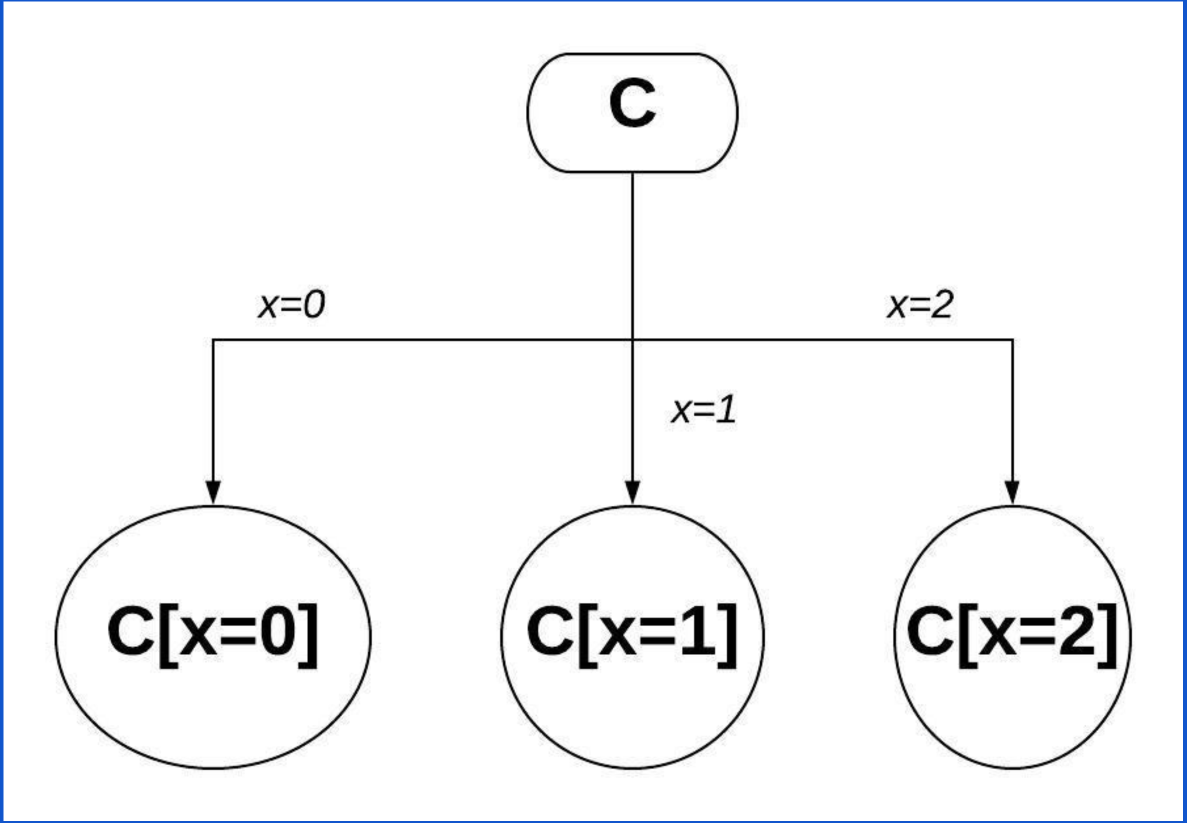
$$f : \{0,1\}^n \to \{0,1\}$$

$$f(x_1, \ldots, x_n) = (x_1 \vee x_8 \vee \overline{x_9}) \wedge (x_2 \vee \overline{x_7}) \wedge (\overline{x_6} \vee x_4 \vee \overline{x_3})$$

Unstructured classical backtracking

Backtracking is a common strategy to solve constraint satisfaction problems.

Given a CSP defined on *n* variables take values in a finite set of size *s*. Let's say that s=3, so that each variable takes value 0,1 or 2. A bruteforce backtracking strategy is to explore all possibilities.



Bruteforce backtracking of k-SAT

A **free variable** is a variable whose value has not been assigned yet.

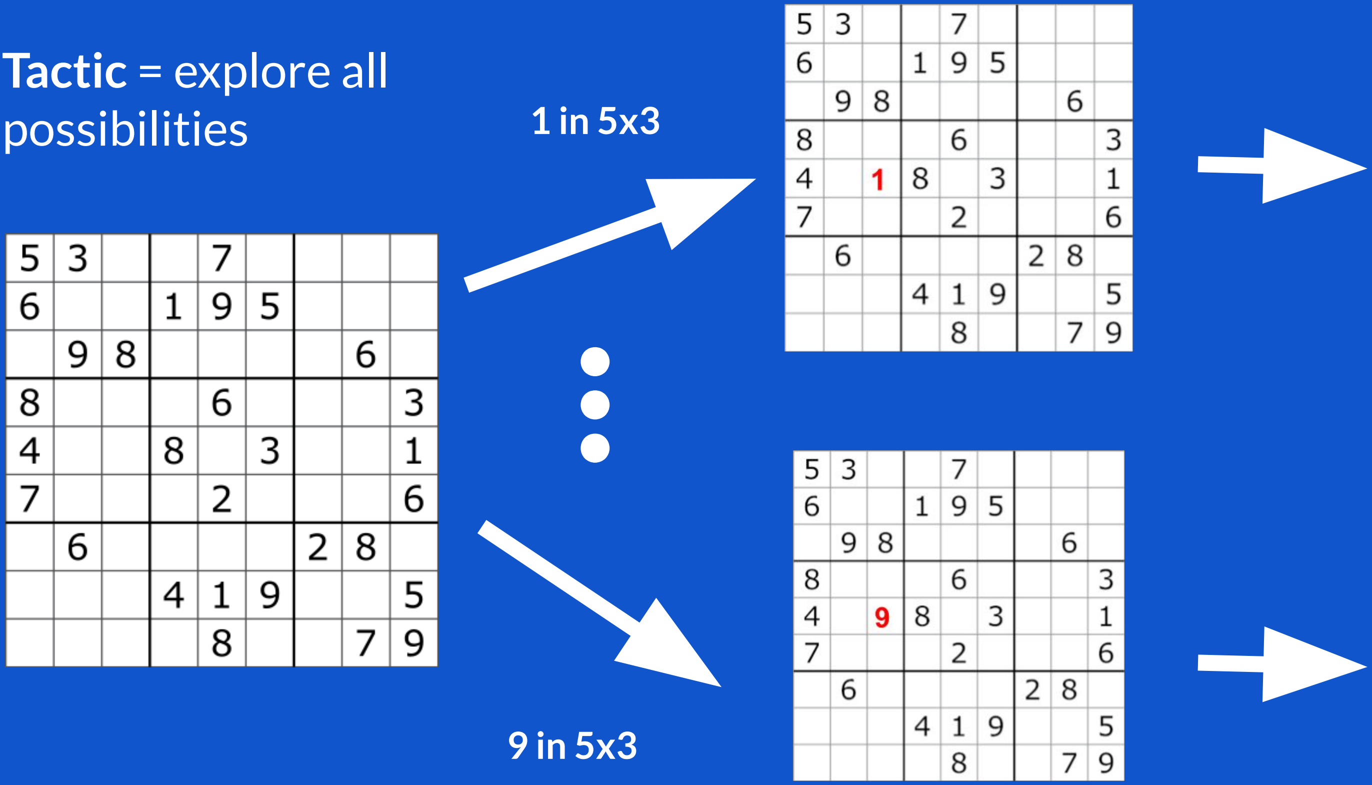
We fix a strategy to decide which variable needs to be considered next by the algorithm. **Example:** take the most significant variable, randomly pick a variable.

For the next free variable, we assign to each variable all its possible values: here 0 and 1.

Assigning a value to a variable simplifies the formula. We recursively check whether the simplified formula is satisfiable.

Bruteforce backtracking of Sudoku

Tactic = explore all possibilities



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

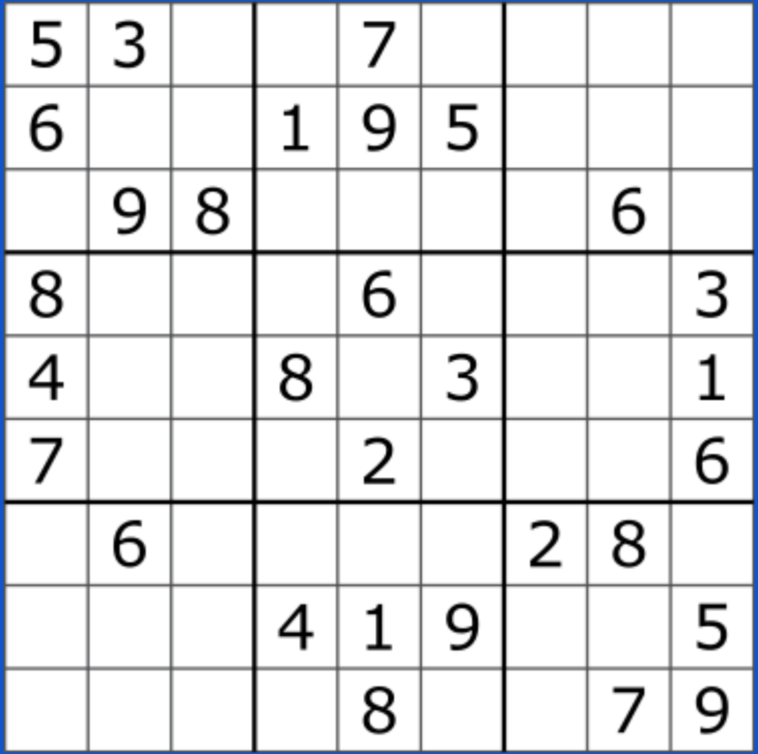
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4		1	8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4		9	8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

CONTRADICTION: backtrack to the previous configuration

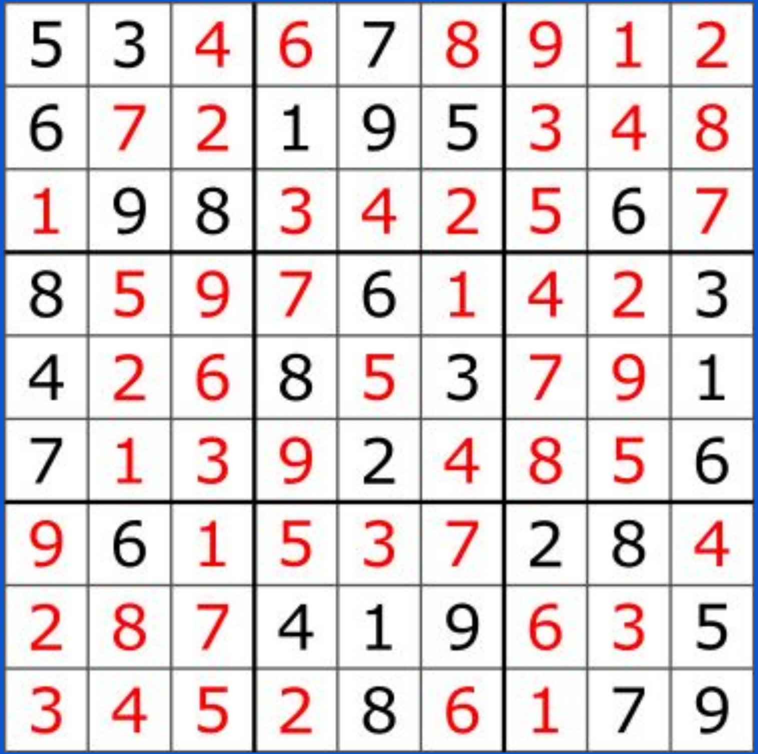
NO CONTRADICTION: Continue the exploration of the grid

Bruteforce backtracking of Sudoku



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

$\longrightarrow$



5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Unstructured classical backtracking

Consider a CSP *C* with *n* variables which take values in {0,...,*s*-1}.

A *partial assignment a*:{1,...,n}⟶ {0,...,*s*-1,*} is a function which associates each variable x to a value v or marks it as undefined (*).

Example: a partial assignment for k-SAT associates the value 0, 1 or ‘undefined’ to each variable of the Boolean formula

```
ExhaustiveSearch(CSP C, partial assignment a):

    If $a$ is a solution for C: return True
    If $a$ is a counter-example for C: return False

    $b \leftarrow$ False

    For every variable $x$ in C:
        For every value $v$ in [$s$]:
            $a[x] \leftarrow v$
            $b \leftarrow$ ExhaustiveSearch($C$,$a$)
            If $b$ = True: return $b$

    return False
```

Unstructured classical backtracking

Given a CSP whose *n* variables take values in a finite set of size *s*, there is a classical backtracking algorithm which finds the solution in time $O(s^n)$. This is a bruteforce search algorithm works by an exhaustive search of all solutions.

```
ExhaustiveSearch(CSP C, partial assignment a):

    If $a$ is a solution for $C$: return True
    If $a$ is a counter-example for $C$: return False

    $b \leftarrow$ False

    For every variable $x$ in C:
        For every value $v$ in [$s$]:
            $a$[x] $\leftarrow v$
            $b \leftarrow$ExhaustiveSearch($C$,$a$)
            If $b$ = True: return $b$

    return False
```

What is a backtracking algorithm?

Consider a CSP on *n* variables whose values are taken in [s] = {0,...,s-1}. Call V the set of variables. Call *A* the set of all partial assignments *V* → {0,1,*undef*}

A **predicate** is a function P : A → {True,False,Undetermined} which determines whether a partial solution satisfies the problem.

A **heuristics** is a function h: A → [n] which determines the next variable to be considered.

The algorithm Backtrack determines if the CSP is satisfiable (and if yes, gives an answer) or not, starting from an indeterminate entry a:x↦*undef*.

```
Backtrack(a)
    If P(a) = True then return a
    If P(a) = False or a has no free bit then return FAIL
    j <- h(a)
    For v = 0..s-1:
        Backtrack(a[$\mathbf{x}_j$ -> v])
    return P(x)
```

Case study: DPLL for k-SAT

Consider a k-CNF formula F on n variables whose values are taken in $\{0,1\}$. Call V the set of variables.

A **partial assignment** is a function $V \rightarrow \{0,1,\mathit{undef}\}$
Call A the set of all partial assignments.

Various choices of heuristics h: $A \rightarrow [n]$.
Example: most significant variable, random, …

Starting from F and the partial assignment a:$x \mapsto \mathit{undef}$, DPLL tries to build a satisfying assignment of the formula.

```
DPLL(F,a)
    If a satisfies F then return True
    If a does not satisfy F or a has no free bit then return False

    F' <- F with all the variables set to the value they have in a (unless that value is undefined).

    j <- h(a)
    If F' contains {x_j} then
        return DPLL(F',a[x_j → 1])
    If F' contains {~x_j} then
        return DPLL(F',a[x_j → 0])
    return DPLL(F',a[x_j → 0]) or DPLL(F',a[x_j → 1])
```

Speeding up search algorithms via Grover

(1) Balanced vs unbalanced trees
(2) Using Grover on balanced trees
(3) Grover isn’t enough for unbalanced trees

Balanced vs unbalanced trees

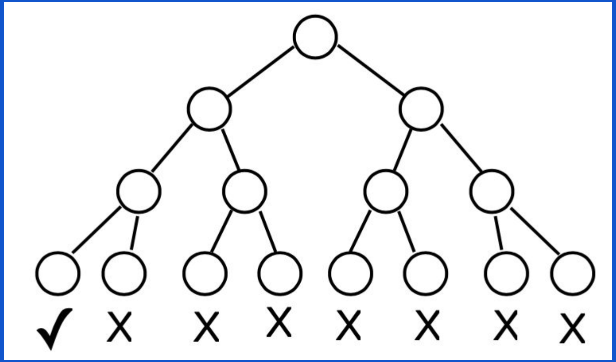
Full tree: each node has 0 or 2 children

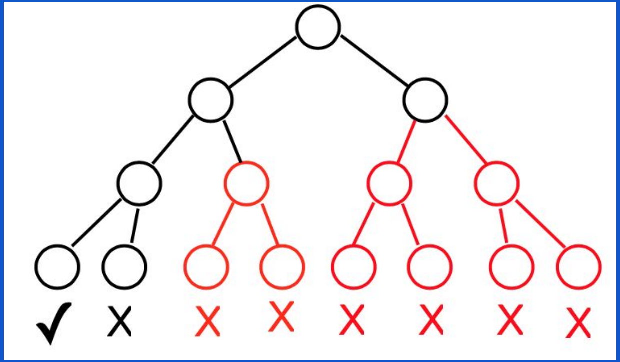
Balanced tree: subtrees of a node differ in height by at most 1.

Perfect tree: full tree + all leaves at the same depth

An efficient backtracking algorithm tries to find a solution by exploring the ***most promising*** branches.

In which case: the subtree it explores is **unbalanced**





Using Grover on balanced trees (for k-SAT)

Consider the k-SAT problem for a given Boolean formula *F*.

Define *T* to be the size of the balanced tree $\mathcal{T}$ defined by a classical backtracking algorithm.

Given access to an oracle *f* : [*T*] -> {0,1} such that f(x)=1 if *x* is a satisfying assignment, i.e. *x* is satisfying all clauses of *F*.

$O(\sqrt{T})$ evaluations of *f* necessary to find *x* such that f(x)=1 with high probability (if it exists)

Grover assumes that the search space is **known**: all the possible solutions have been indexed **prior** to the search. This means that the search space is of size *T*.

Application: Grover can solve the *k*-SAT problem for a Boolean formula with n variables in time

$$O(2^{\frac{n}{2}})$$

whereas classical exhaustive search does it in time

$$O(2^n)$$

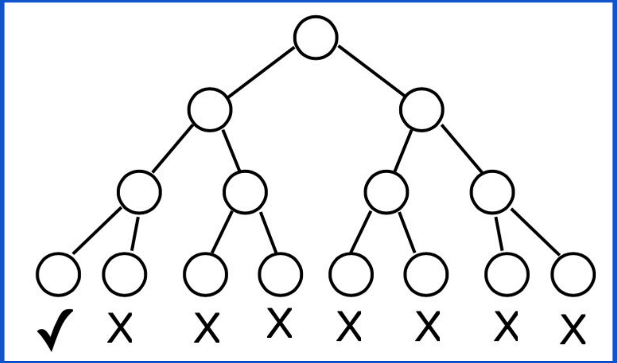
Grover isn't enough for unbalanced trees

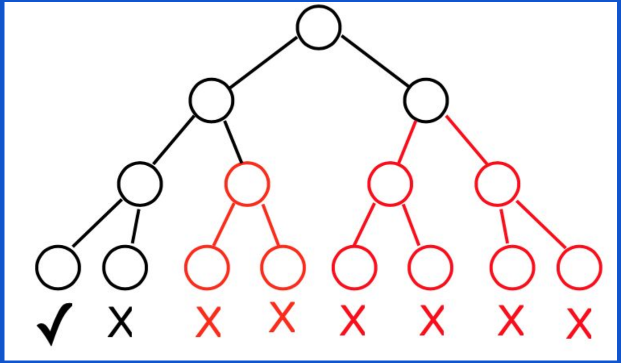
	Balanced	Unbalanced
Classical backtracking	$2^{\lambda n}$	$< 2^{\lambda n}$
Grover	$2^{\lambda n/2}$	$2^{\lambda n/2}$

Grover assumes that the search space is **known**: all the possible solutions have been indexed prior to the search.

We need an algorithm whose execution depends on the choice made by the classical backtracking algorithm.

We need **quantum backtracking**.





Quantum backtracking

(1) How to speedup classical backtracking
(2) How to construct a quantum walk operator
(3) Case study: quantum backtracking for DPLL
(4) How to detect marked vertices within a search tree?

How to speed up classical backtracking

Build a quantum algorithm which determines whether there is a marked vertex (for k-SAT, a satisfying assignment), given an upper bound *T* on the number of vertices of the search tree generated by a classical backtracking algorithm

Reference: Montanaro, 2015.

QUANTUM BACKTRACKING

Input: quantum walk operator W, number of vertices *T*, and an upper bound *d* on the depth of the tree

(1) Repeatedly apply Quantum Phase Estimation (**QPE**) to the operator *W* on a state representing the **root** of the tree, with finite precision. If the eigenvalue is 1 accept. Otherwise, reject.

(2) If the acceptance rate is above a pre-determined threshold, return YES. Else, return NO.

How to construct a quantum walk operator

$$|\varphi_x\rangle = \frac{1}{\sqrt{d_x}}\left(|x\rangle + \sum_{x\to y}|y\rangle\right)$$

$$|\varphi_r\rangle = \frac{1}{\sqrt{d_r n+1}}\left(|r\rangle + n\cdot\sum_{r\to y}|y\rangle\right)$$

$$R_A = \bigoplus_{x\in A} D_x$$

$$R_B = |r\rangle\langle r| + \bigoplus_{x\in A} D_x$$

$$W = R_B R_A$$

D_x is the ***diffusion operator*** for the node *x*, and represents the next moves of the backtracking algorithm, starting from *x*.

Given U an operator on m qubits with an eigenvector $|\varphi\rangle$such that $U|\varphi\rangle = e^{2\pi i\theta}|\varphi\rangle$ for $0 \leq \theta < 1$, quantum phase estimation finds the phase θ up to a finite level of precision

Implementing diffusion operators

$$|\varphi_x\rangle = \frac{1}{\sqrt{d_x}}\left(|x\rangle + \sum_{x\to y}|y\rangle\right)$$

$$|\varphi_r\rangle = \frac{1}{\sqrt{d_r n+1}}\left(|r\rangle + n\cdot\sum_{r\to y}|y\rangle\right)$$

D_x is the identity if *x* is marked

$$D_x = \mathrm{Id} - 2|\varphi_x\rangle\langle\varphi_x| \text{ if } x \text{ not marked}$$

$$R_A = \bigoplus_{x\in A} D_x$$

$$R_B = |r\rangle\langle r| + \bigoplus_{x\in A} D_x$$

$$W = R_B R_A$$

Each diffusion operator represents a set of ***moves*** of the quantum walk.

A diffusion operator on *x* is implemented with ***local knowledge***: only knowing *x* and the children of *x*.

A ***step*** of the quantum walk is an application of the walk operator.

The way we construct the state corresponding to *x* and all its children depends on how the backtracking algorithm chooses the next moves.

Case study: quantum backtracking for DPLL

$$|\varphi_x\rangle = \frac{1}{\sqrt{d_x}}\left(|x\rangle + \sum_{x\to y}|y\rangle\right)$$

$$V|x\rangle|0\rangle|0\rangle = V|x\rangle|x_1\rangle|x_2\rangle$$

$$C|x\rangle|y\rangle|z\rangle = \frac{1}{\sqrt{3}}|x\rangle|y\rangle|z\rangle\left(|x\rangle+|y\rangle+|z\rangle\right)$$

$$D_x = V^\dagger C V$$

$$R_A = \bigoplus_{x\in A} D_x$$

$$R_A|x\rangle|0\rangle = D_x|x\rangle|0\rangle = |x\rangle|\varphi_x\rangle$$

The unitary *V* determines the next moves, starting from an unmarked node *x*. This process is done in two operations:
(1) check whether there are unit clauses (only one literal) and set the corresponding variable to true
(2) Select the next free variable

This implements the ***branching*** of the DPLL algorithm.

Detecting a marked vertex

Algorithm: Applying phase estimation to a quantum walk (starting at the root) with precision $O\left(\frac{1}{\sqrt{Td}}\right)$, where *T* and *d* are respectively upper bounds on the number of vertices of the tree and the depth of the tree, a solution exists if the eigenvalue is 1, and there's no solution otherwise.

Theorem (Belovs, 2013): This algorithm succeeds with high probability.

Consequence: $\tilde{O}\left(\sqrt{Td}\right)$ rounds of this algorithm can detect the existence of a solution.

How a marked vertex is detected

Algorithm A: Applying phase estimation to a quantum walk (starting at the root) with precision

$$O\left(\frac{1}{\sqrt{Td}}\right)$$

where *T* and *d* are respectively upper bounds on the number of vertices of the tree and the depth of the tree, a solution exists if the eigenvalue is 1, and there's no solution otherwise.

Theorem (Belovs, 2013): Algorithm A succeeds with high probability.

Consequence: $\tilde{O}\left(\sqrt{Td}\right)$ rounds of this algorithm can detect the existence of a solution.

Algorithm A exploits an estimate of number of steps on the quantum walk necessary to encounter a marked vertex, starting from the root. It's the ***hitting time***.

It determines whether the set of marked vertices is non-empty by performing the quantum walk.

k-SAT: **marked vertex = satisfying assignment**

Property: if there is a marked vertex, then the state representing the root is close to an eigenvector of the walk operator *W* with eigenvalue 1.

Finding all satisfying assignments

Assume that every vertex in the search tree has a finite degree (at most s).

To find a marked vertex starting from any vertex *x* in the tree, it suffices to apply the detection algorithm on each subtree with a child of *x* as root, until we find a subtree with a marked vertex as root.

The search tree is of depth at most *d*, therefore we need to run the detection algorithm at most *O(d)* to reach a leaf which is a marked vertex. This process can be repeated to find all satisfying assignments, with an overall time complexity of

$$\tilde{O}\left(d^{\frac{3}{2}}\sqrt{T}\right)$$

Improvements on quantum backtracking

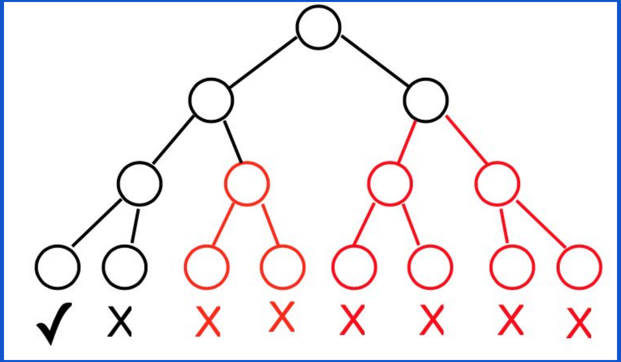
(1) Quantum tree size estimation
(2) Optimizing phase estimation

Quantum tree size estimation

Drawback of quantum backtracking: the runtime depends on the estimate of the size of the search tree (which is a parameter of the algorithm), and not on the size of the subtree that the classical backtracking algorithm explores.

The efficiency of classical backtracking algorithm relies on their capacity to explore the most promising branches first.

Problem: what if the classical algorithm finds a marked vertex after exploring *T'* vertices, with *T'* much smaller than *T*?



Quantum tree size estimation

Improvement on quantum backtracking: estimate the size of the tree explored by the classical algorithm instead of using a general upper bound.

Theorem: Consider a classical backtracking algorithm $\mathcal{A}$ which generates a search tree $\mathcal{T}$. There is a quantum algorithm which outputs 1 with high probability if *T* contains a marked vertex and 0 if it doesn't, with query complexity

$$\tilde{O}\left(n^{\frac{3}{2}}\sqrt{T'}\right)$$

where T' is the number of vertices actually explored by $\mathcal{A}$.

Main strategy: generate subtrees which contains the first **2^i** vertices explored by the classical backtracking algorithm, increasing *i* until a marked vertex is found, or the whole tree is searched.

Reference: Ambainis, Kokainis, 2017.

Quantum tree size estimation

Theorem: Consider a classical backtracking algorithm $\mathcal{A}$ which generates a search tree $\mathcal{T}$. There is a quantum algorithm which outputs 1 with high probability if $\mathcal{T}$ contains a marked vertex and 0 if it doesn't, with query complexity $\tilde{O}\left(n^{\frac{3}{2}}\sqrt{T'}\right)$

where *T'* is the number of vertices actually explored by $\mathcal{A}$.

Main strategy: generate subtrees which contains the first $\mathbf{2^i}$ vertices explored by the classical algorithm, increasing *i* until a marked vertex is found, or the whole tree is searched.

Algorithm:

```
    Let i = 1
    Repeat:
        Use tree size estimation to generate the subtree T' corresponding to the first 2^i vertices visited by the classical algorithm
        Run quantum backtracking on T', and if a marked vertex is found, return 1.
        i <- i+1
    Until T' contains the whole tree.

    return 0
```

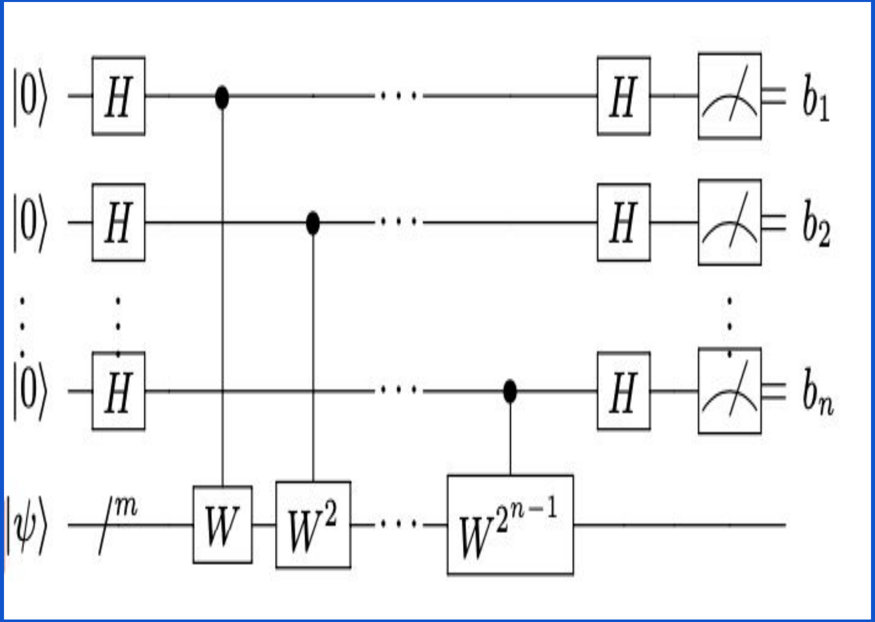
Optimizing phase estimation

Practical implementations of quantum backtracking require a crucial optimisation of the circuits involved in the quantum search

Key observation: quantum phase estimation is used to distinguish between eigenvalue 1 and eigenvalues which are really far from 1.

In practice: the quantum Fourier transformation of the quantum phase estimation can be replaced by Hadamard gates.

Reference: Campbell, Khurana, Montanaro, 2019.


$|0\rangle$
$|0\rangle$
$|0\rangle$
$|\psi\rangle$
H
m
W
W^2
$W^{2^{n-1}}$
b_1
b_2
b_n