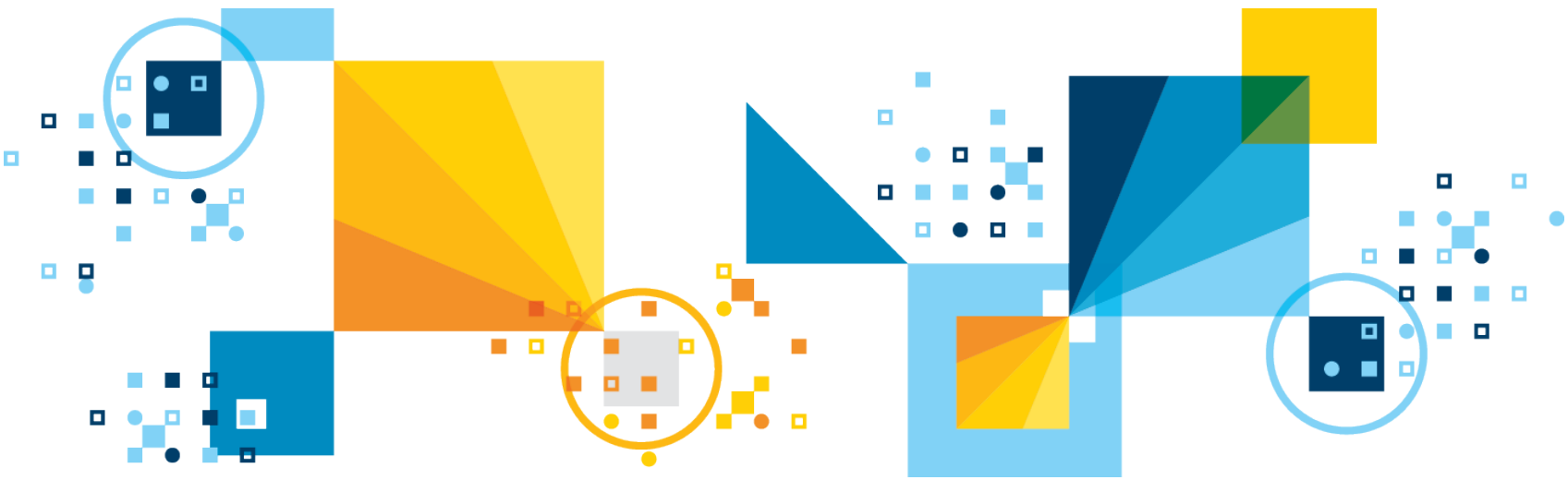
Failure-directed Search for Constraint-based Scheduling

Petr Vilím, Philippe Laborie, Paul Shaw

IBM

July 12, 2015



Outline

Why we developed it

Inspiration

How it works

Experimental results

Why it works well?

Why we developed failure-directed search

IBM

Automatic search in CP Optimizer for scheduling used to be:

- Portfolio based.
- Initial solution(s): portfolio of SetTimes searches.
- Optimization: Large Neighbourhood Search (LNS).
- Proof of optimality: almost nothing.

The problem:

- We were satisfied with performance of the LNS.
- But for small and medium sized problems, we needed something in the portfolio to finish the search by a proof of optimality.
- We needed a generic plan B when LNS is stuck.

Plan B approach

Start plan B one when LNS is not improving the solution any more.

Assumptions:

- There probably isn't any (better) solution.
- If there is one, it is very hard to find.
- It is necessary to explore the whole search space.

Consequences:

- Failure-directed search was tuned on infeasible problems.
- We gave up on leading the search towards a solution.
  - If a solution is found, it is by an accident.
- The search can perform badly when the gap is still big.

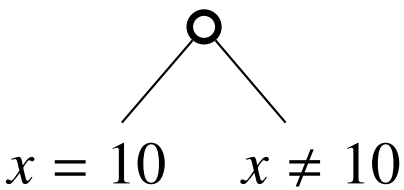
Inspiration from CP (and SAT)

State-of-the-art generic search in CP

- Impact-based search
- Weighted-degree heuristics
- Activity-based search

The main idea:

- Choose the most *interesting* variable
- Assign the most *promising* value to it



Other ingredients:

- Periodic restarts (geometric, luby, ...)
- Nogood learning (from restarts)

Impact-based search

Search space estimation:

$$P = \begin{cases} 0 & \text{if infeasible} \\ |D_{x_1}| \times \cdots \times |D_{x_n}| & \text{otherwise} \end{cases}$$

Impact of an assignment:

$$I(x_i = a) = 1 - \frac{P_{\text{after}}}{P_{\text{before}}} \qquad \text{bigger is better}$$

Average impact of an assignment:

$$\bar{I}(x_i = a) = \frac{1}{|K|} \sum_{k \in K} I_k(x = a)$$

Impact of a variable:

$$\bar{I}(x_i) = \sum_{a \in D_i} (1 - I\,(x_i = a))$$

Impact-based search

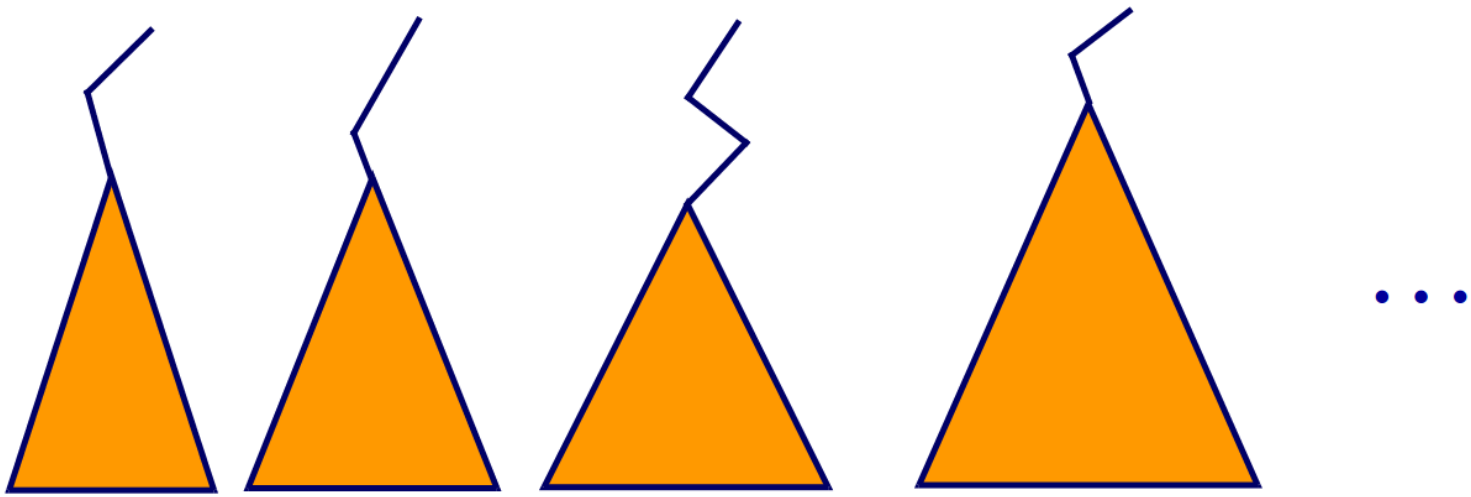
Choose variable

- with the maximum impact (depends on current domain)
  - to treat hard-to-assign variables first.

Choose value

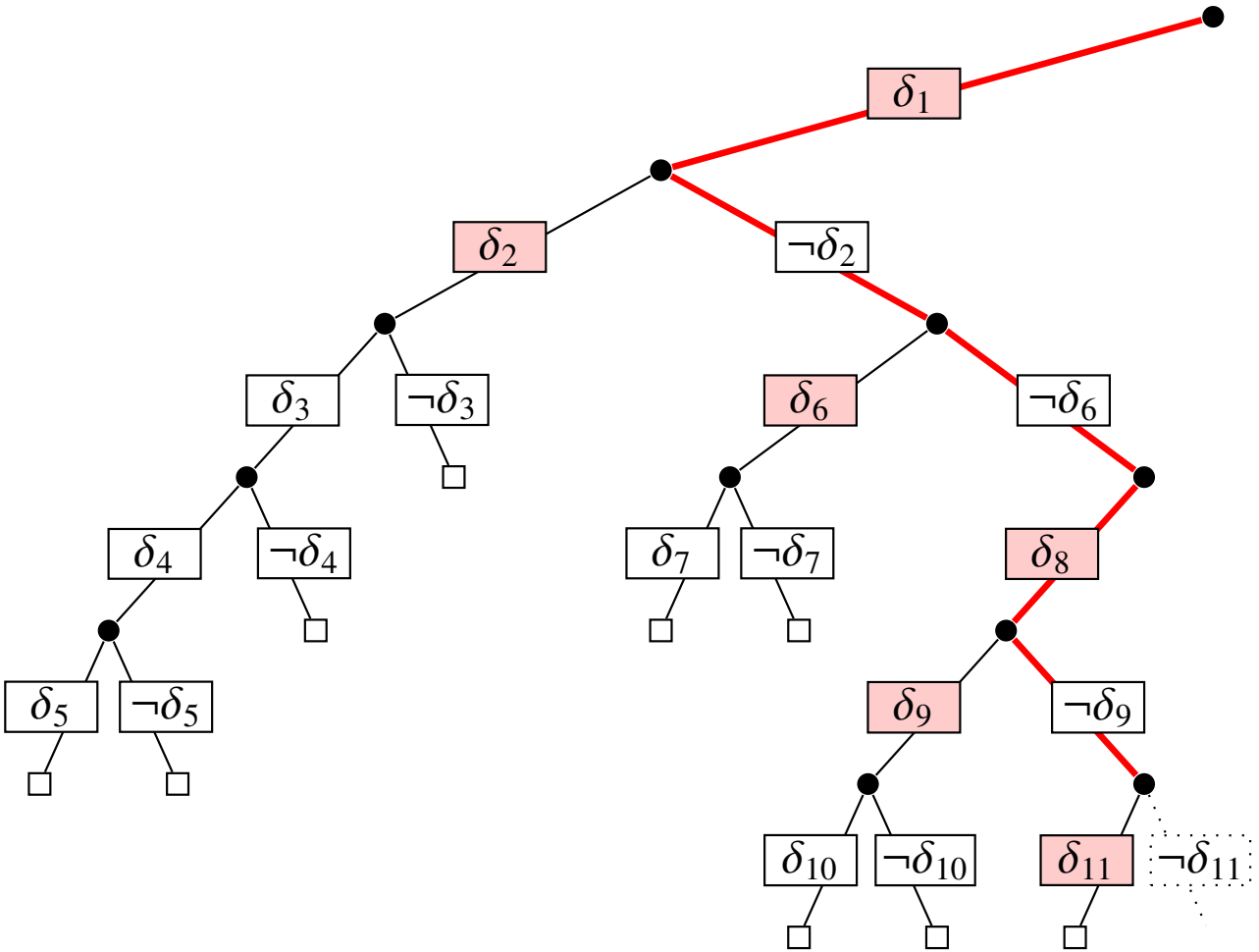
- with the smallest impact
  - to maximize chances of extending current partial assignment to a solution.

Restarts

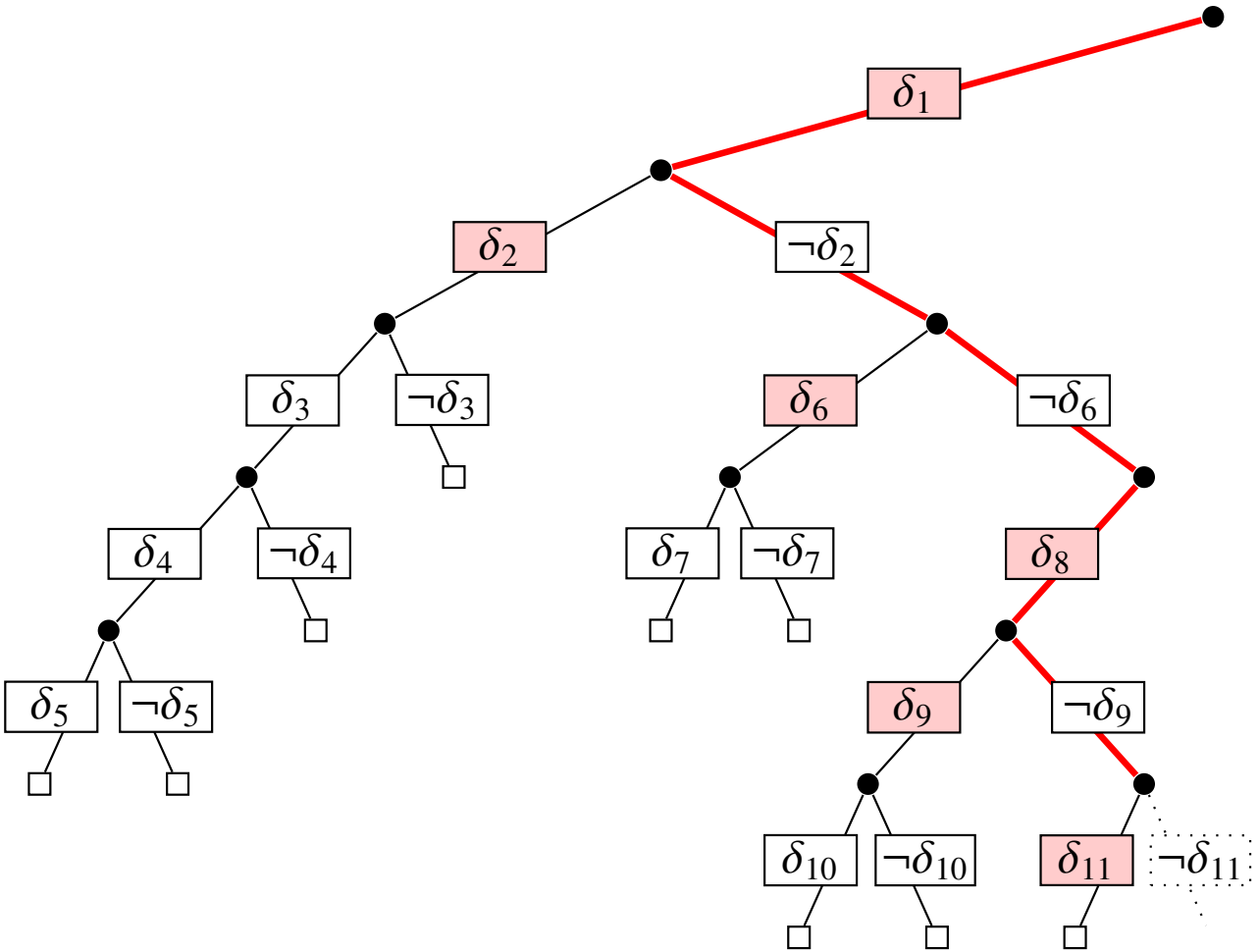


- Restarting allows to reconsider decisions made at the top of the search tree.
- Without nogood recording, the search may duplicate some work.

Nogood recording



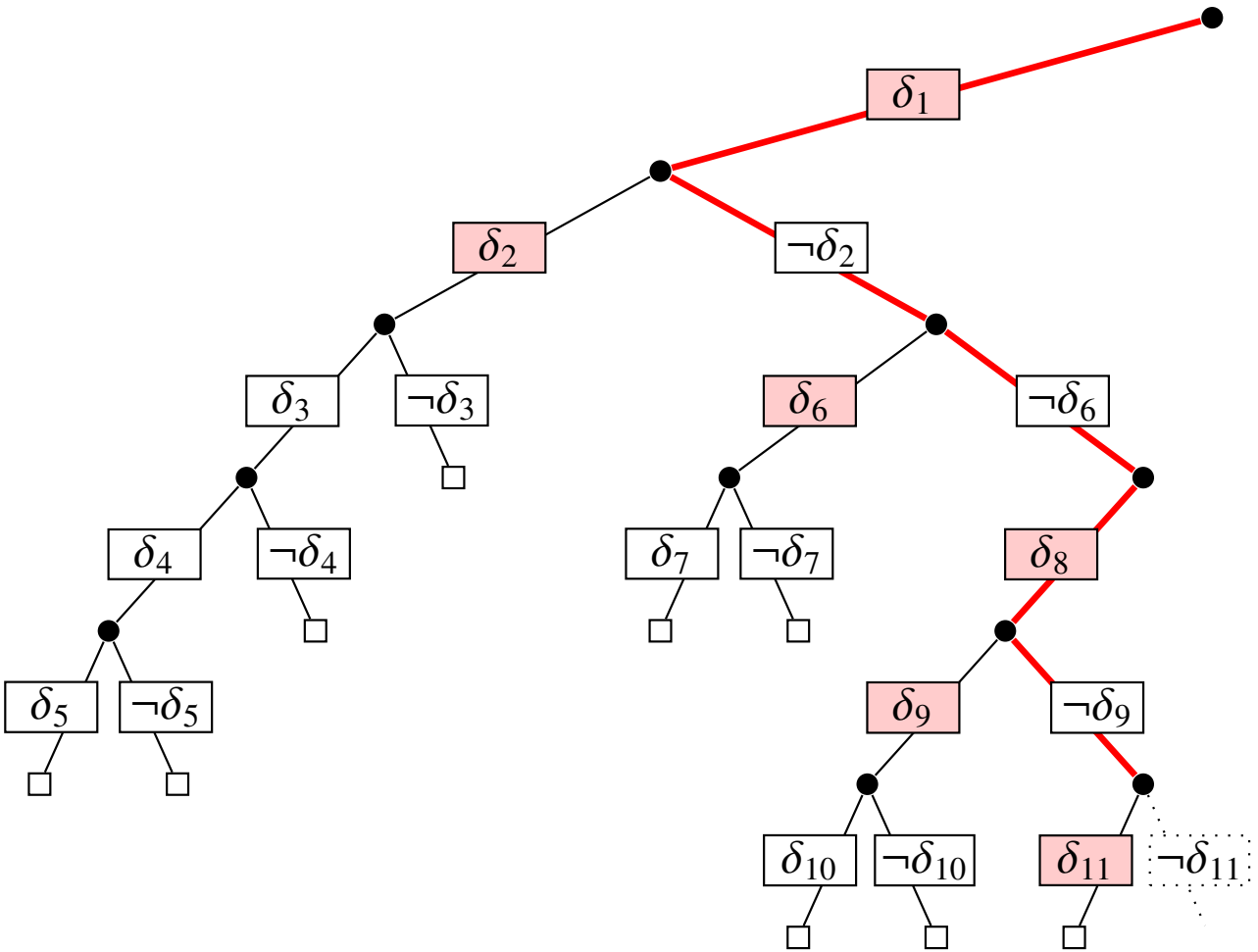
Nogood recording



Nogoods:

- $\neg\delta_1 \vee \neg\delta_2$

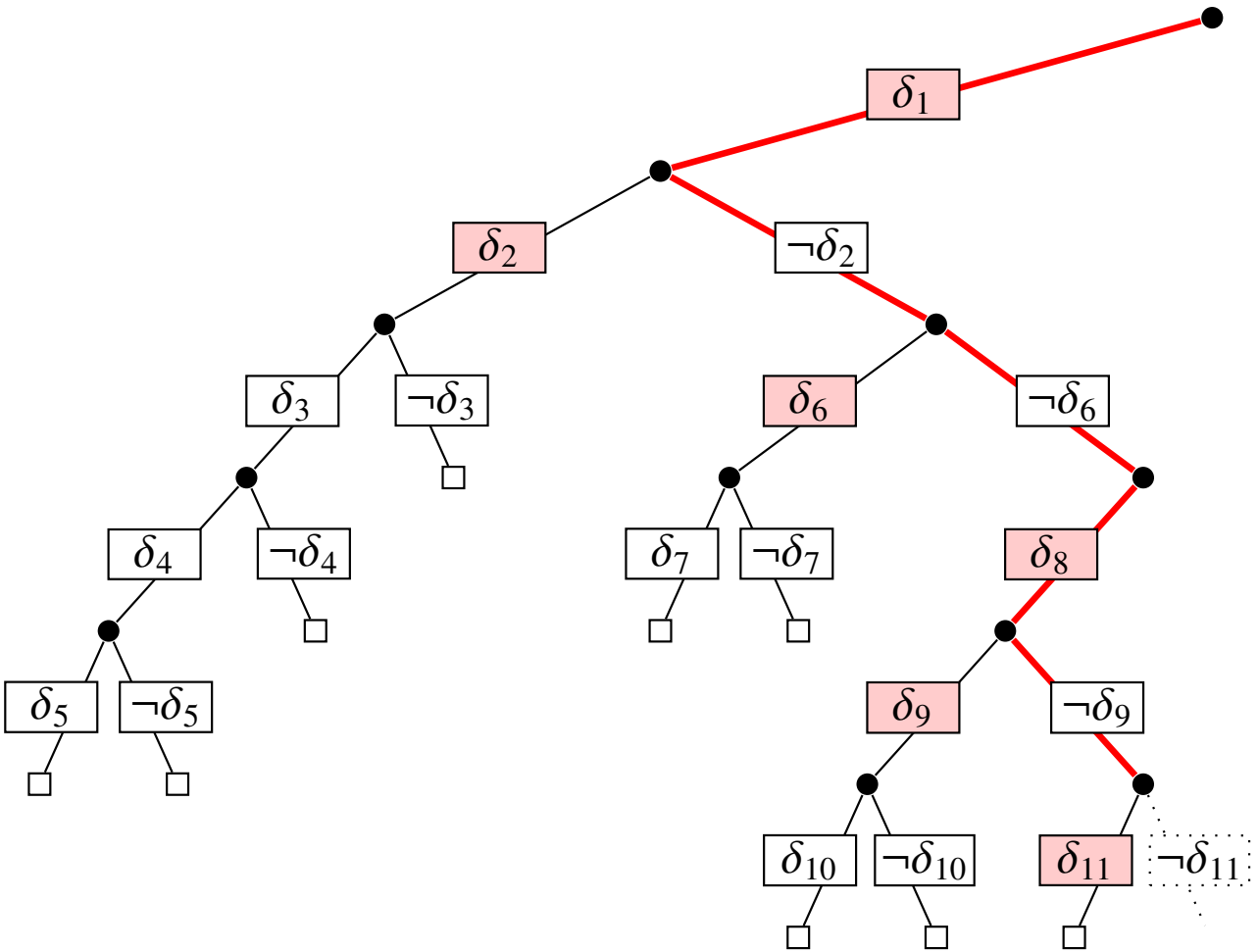
Nogood recording



Nogoods:

- $\neg\delta_1 \vee \neg\delta_2$
- $\neg\delta_1 \vee \neg\delta_6$

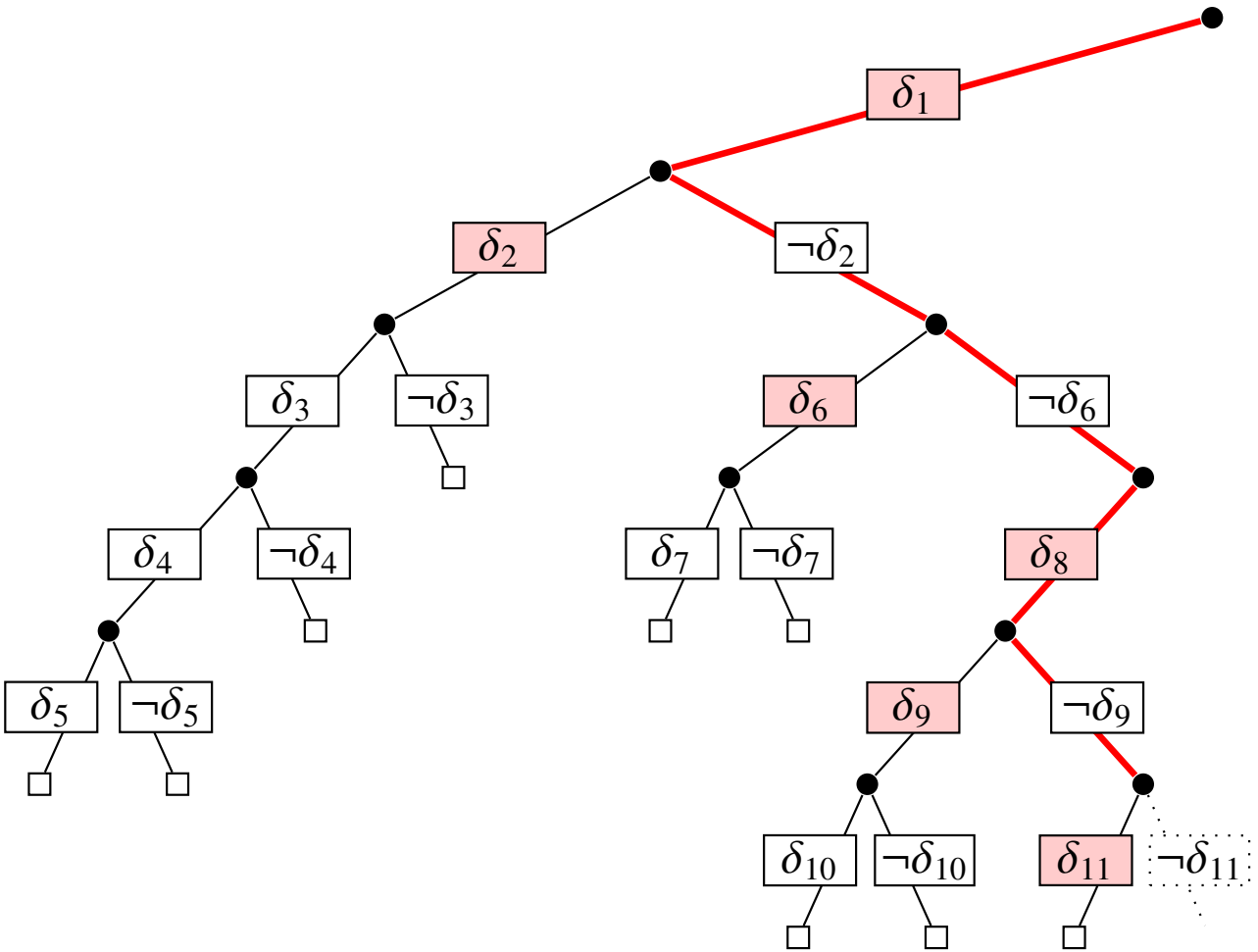
Nogood recording



Nogoods:

- $\neg\delta_1 \vee \neg\delta_2$
- $\neg\delta_1 \vee \neg\delta_6$
- $\neg\delta_1 \vee \neg\delta_8 \vee \neg\delta_9$

Nogood recording



Nogoods:

- $\neg\delta_1 \vee \neg\delta_2$
- $\neg\delta_1 \vee \neg\delta_6$
- $\neg\delta_1 \vee \neg\delta_8 \vee \neg\delta_9$
- $\neg\delta_1 \vee \neg\delta_8 \vee \neg\delta_{11}$

Nogood propagation

Nogoods are often long:

$$\neg\delta_1 \vee \neg\delta_2 \vee \neg\delta_3 \vee \neg\delta_4 \vee \neg\delta_5 \vee \neg\delta_6 \vee \neg\delta_7 \vee \neg\delta_8 \vee \neg\delta_9$$

Filtering is done only if many δ_i are true:

$$\delta_1 \wedge \delta_2 \wedge \cdots \wedge \delta_8 \;\Rightarrow\; \neg\delta_9$$

Constant re-evaluation of a nogood is

- costly
  - expression is long
  - there are many nogoods
- most of the time doesn't propagate anything
  - because too many δ_i are false or uknown.

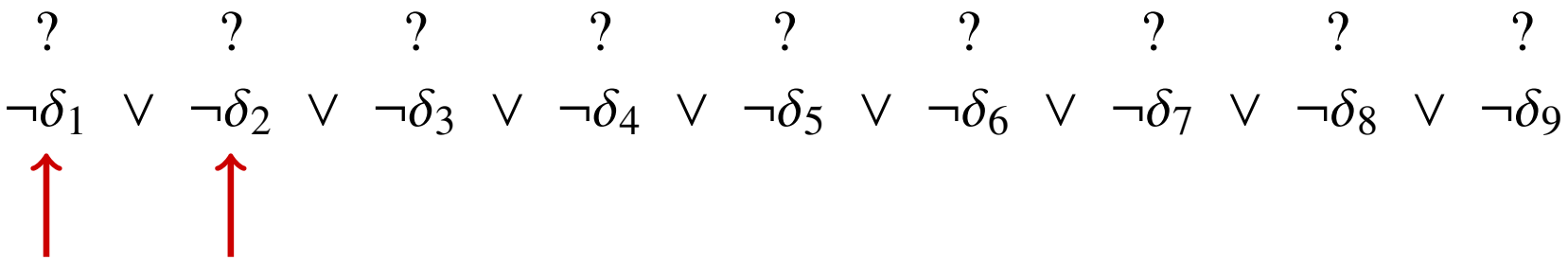
Watched literals

$$\begin{array}{ccccccccccccccccc} ? & & ? & & ? & & ? & & ? & & ? & & ? & & ? \\ \neg\delta_1 & \vee & \neg\delta_2 & \vee & \neg\delta_3 & \vee & \neg\delta_4 & \vee & \neg\delta_5 & \vee & \neg\delta_6 & \vee & \neg\delta_7 & \vee & \neg\delta_8 & \vee & \neg\delta_9 \end{array}$$

Key idea:

- As long as at least two δ_i are unknown, there's nothing to do.
- Watch only those two and ignore the rest.

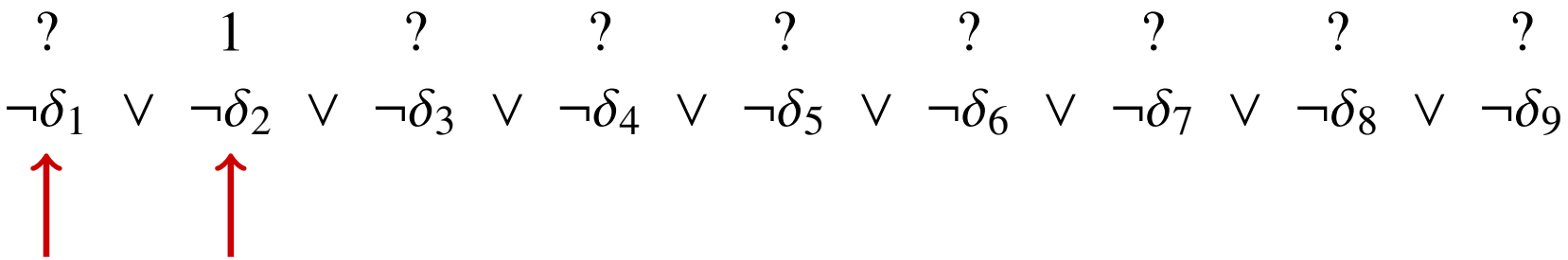
Watched literals



Key idea:

- As long as at least two δ_i are unknown, there's nothing to do.
- Watch only those two and ignore the rest.

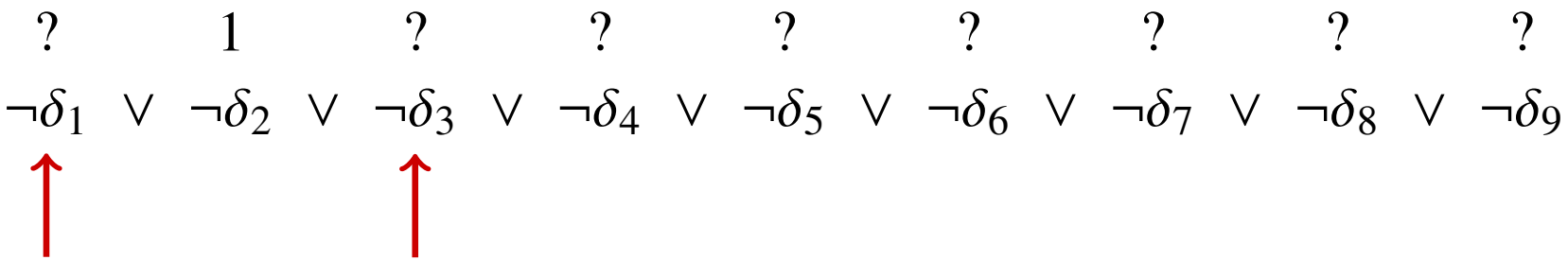
Watched literals



Key idea:

- As long as at least two δ_i are unknown, there's nothing to do.
- Watch only those two and ignore the rest.
- Move the watch when δ_i becomes true.

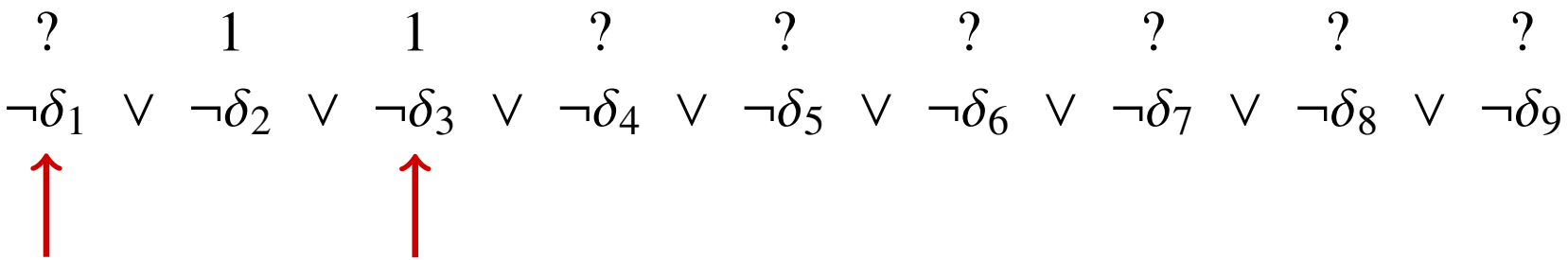
Watched literals



Key idea:

- As long as at least two δ_i are unknown, there's nothing to do.
- Watch only those two and ignore the rest.
- Move the watch when δ_i becomes true.

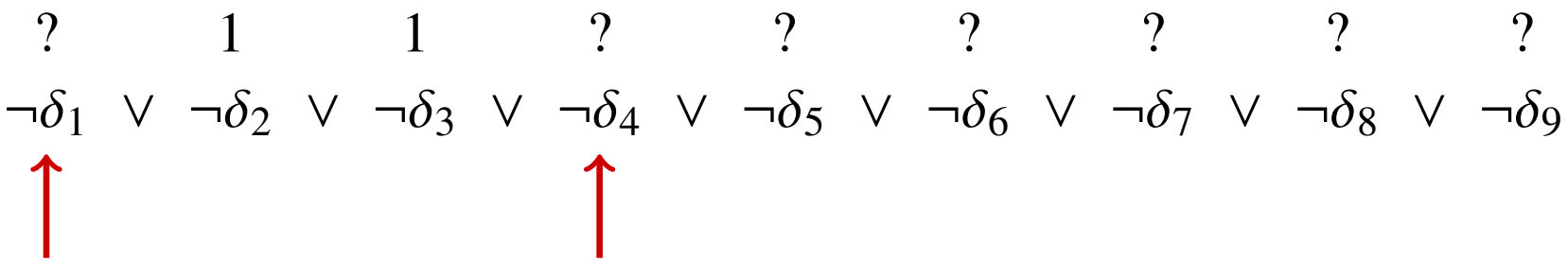
Watched literals



Key idea:

- As long as at least two δ_i are unknown, there's nothing to do.
- Watch only those two and ignore the rest.
- Move the watch when δ_i becomes true.

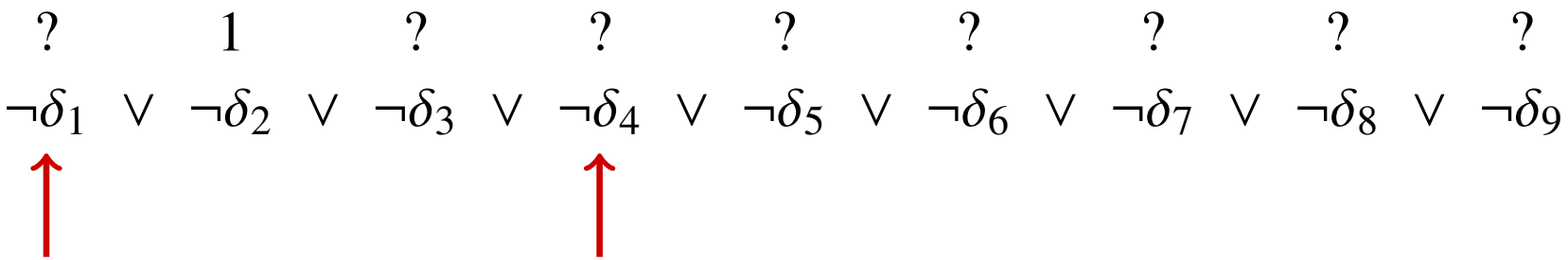
Watched literals



Key idea:

- As long as at least two δ_i are unknown, there's nothing to do.
- Watch only those two and ignore the rest.
- Move the watch when δ_i becomes true.

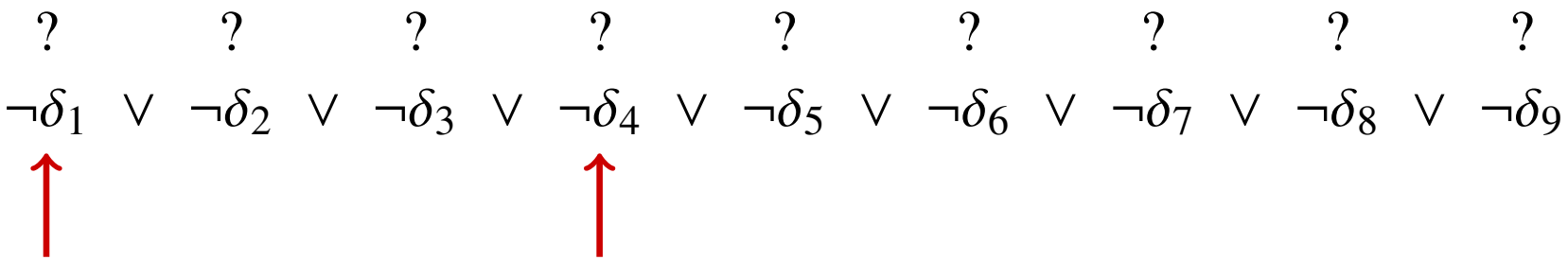
Watched literals



Key idea:

- As long as at least two δ_i are unknown, there's nothing to do.
- Watch only those two and ignore the rest.
- Move the watch when δ_i becomes true.
- The watch can stay in place after backtrack

Watched literals



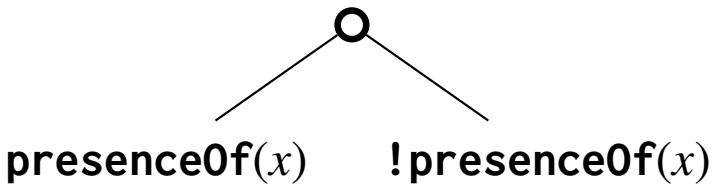
Key idea:

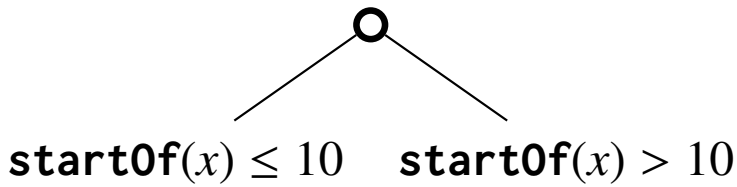
- As long as at least two δ_i are unknown, there's nothing to do.
- Watch only those two and ignore the rest.
- Move the watch when δ_i becomes true.
- The watch can stay in place after backtrack

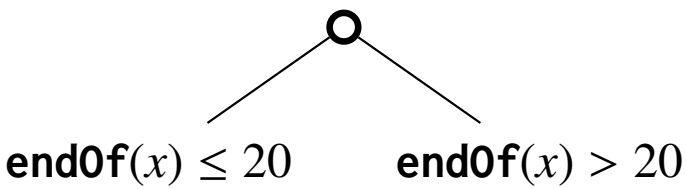
How it works

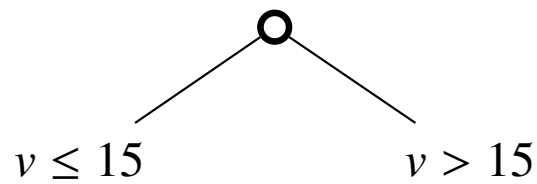
Branching on time, choices

Branch on ranges instead:



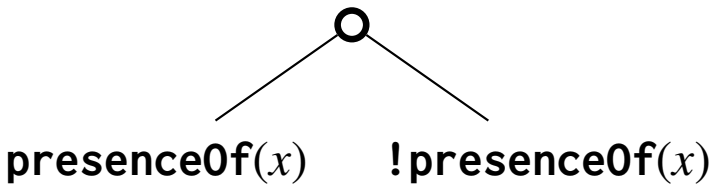


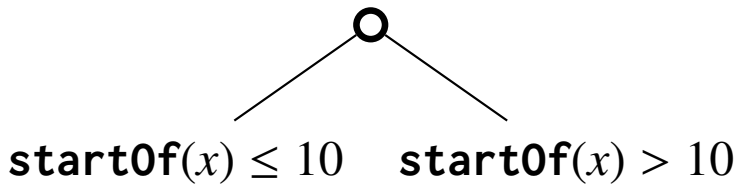


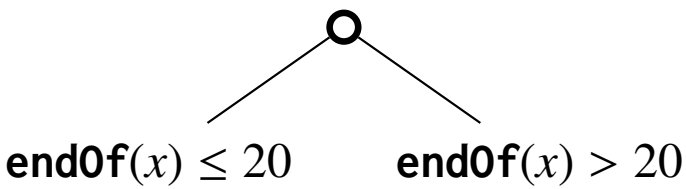


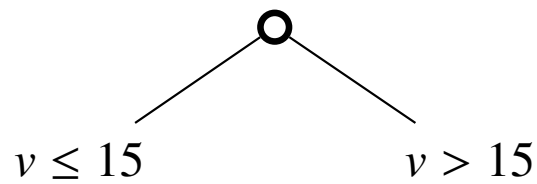
Branching on time, choices

Branch on ranges instead:





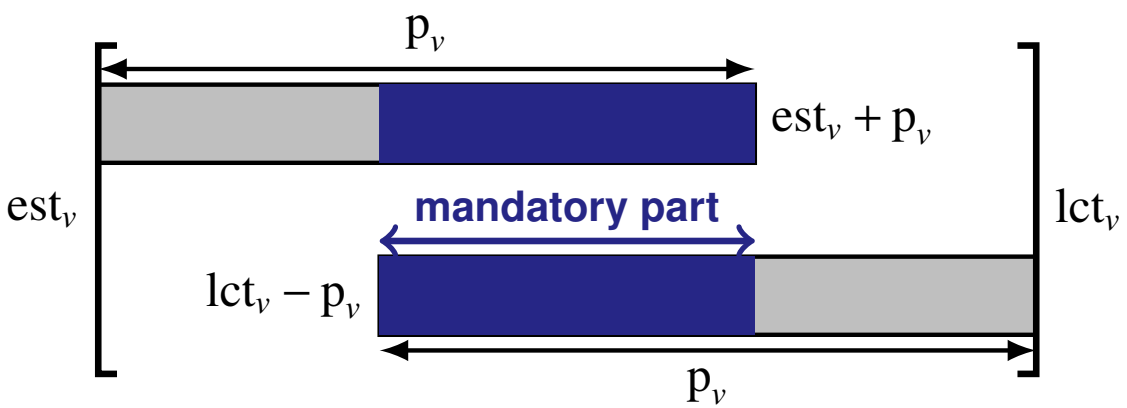




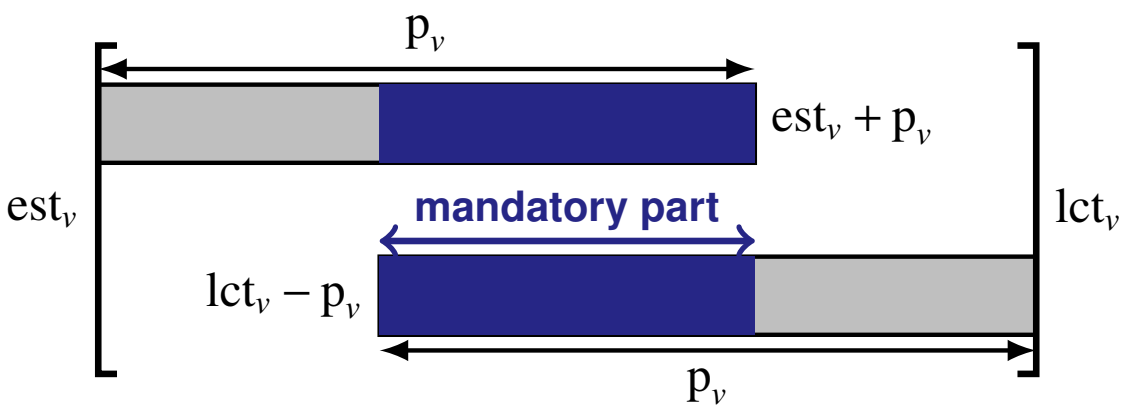
Choice:

- An abstraction of any kind of binary decision.
  - The decision doesn't have to fix variable value.
    - Multiple decisions on the same variable may be needed.
- The search doesn't know what the choice is doing.
- Choices are generated when the search starts.

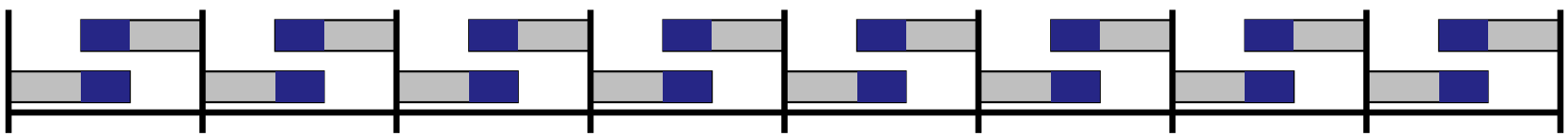
Stopping on mandatory parts



Stopping on mandatory parts



When all choices are fixed then there must be mandatory part:



Additional choices are generated when necessary.

Rating the choices

For each choice c we maintain:

- Rating of positive branch: $\text{rating}^+[c]$
- Rating of negative branch: $\text{rating}^-[c]$
- Rating of the choice: $\text{rating}[c] := \text{rating}^+[c] + \text{rating}^-[c]$

Design of the ratings

- Lower number means better rating.
- Ratings prefer choices that fails often or propagate a lot.

Computation of ratings

When a branch of a choice is taken:

$$\text{localRating} := \begin{cases} 0 & \text{if the branch fails immediately} \\ 1 + R & \text{otherwise} \end{cases}$$

Where $R \in (0, 1]$ is a measure of reduction done by propagation.

Computation of ratings

When a branch of a choice is taken:

$$\text{localRating} := \begin{cases} 0 & \text{if the branch fails immediately} \\ 1 + R & \text{otherwise} \end{cases}$$

Where $R \in (0, 1]$ is a measure of reduction done by propagation.

Update rating of choice branch:

$$\text{rating}^{+/-}[c] := \alpha \cdot \text{rating}^{+/-}[c] + (1 - \alpha) \cdot \frac{\text{localRating}}{\text{avgRating}[d]}$$

Where:

- $\alpha \in [0.9, 0.99]$ is a constant controlling the speed of decay.
- avgRating$[d]$ is average rating on current depth d.

Computation of ratings

When a branch of a choice is taken:

$$\text{localRating} := \begin{cases} 0 & \text{if the branch fails immediately} \\ 1 + R & \text{otherwise} \end{cases}$$

Where $R \in (0, 1]$ is a measure of reduction done by propagation.

Update rating of choice branch:

$$\text{rating}^{+/-}[c] := \alpha \cdot \text{rating}^{+/-}[c] + (1 - \alpha) \cdot \frac{\text{localRating}}{\text{avgRating}[d]}$$

Where:

- $\alpha \in [0.9, 0.99]$ is a constant controlling the speed of decay.
- avgRating$[d]$ is average rating on current depth d.

Update rating of the choice:

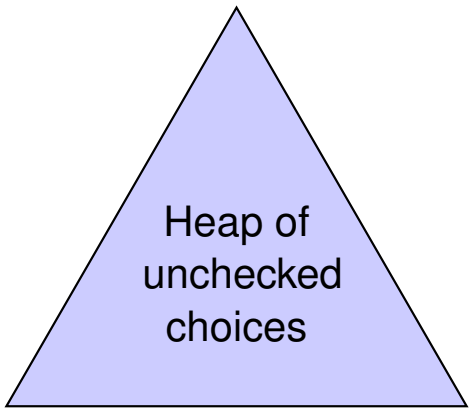
$$\text{rating}[c] := \text{rating}^{+}[c] + \text{rating}^{-}[c]$$

Search algorithm

DFS, Always decide the choice with the best rating.

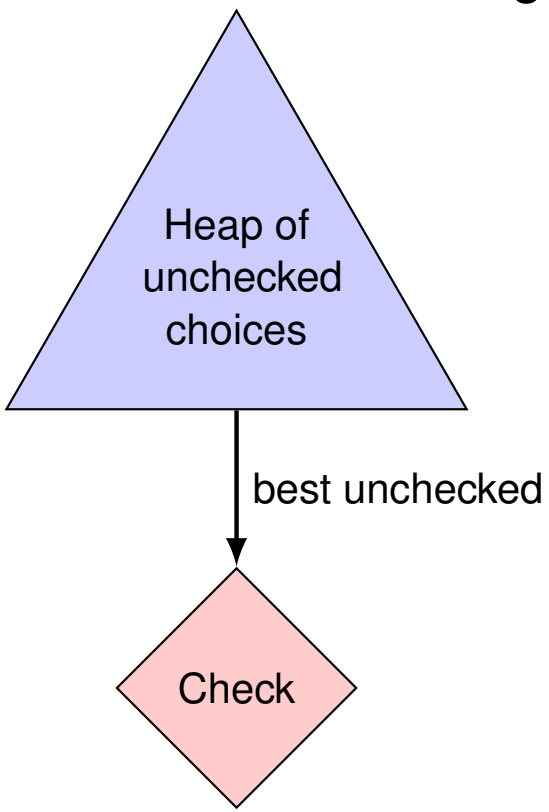
Search algorithm

DFS, Always decide the choice with the best rating.



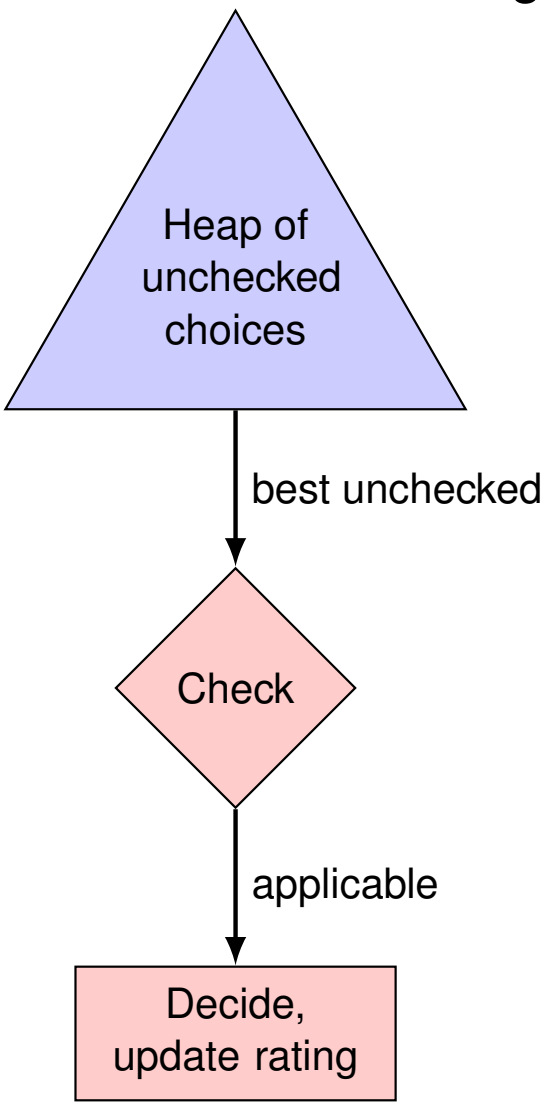
Search algorithm

DFS, Always decide the choice with the best rating.



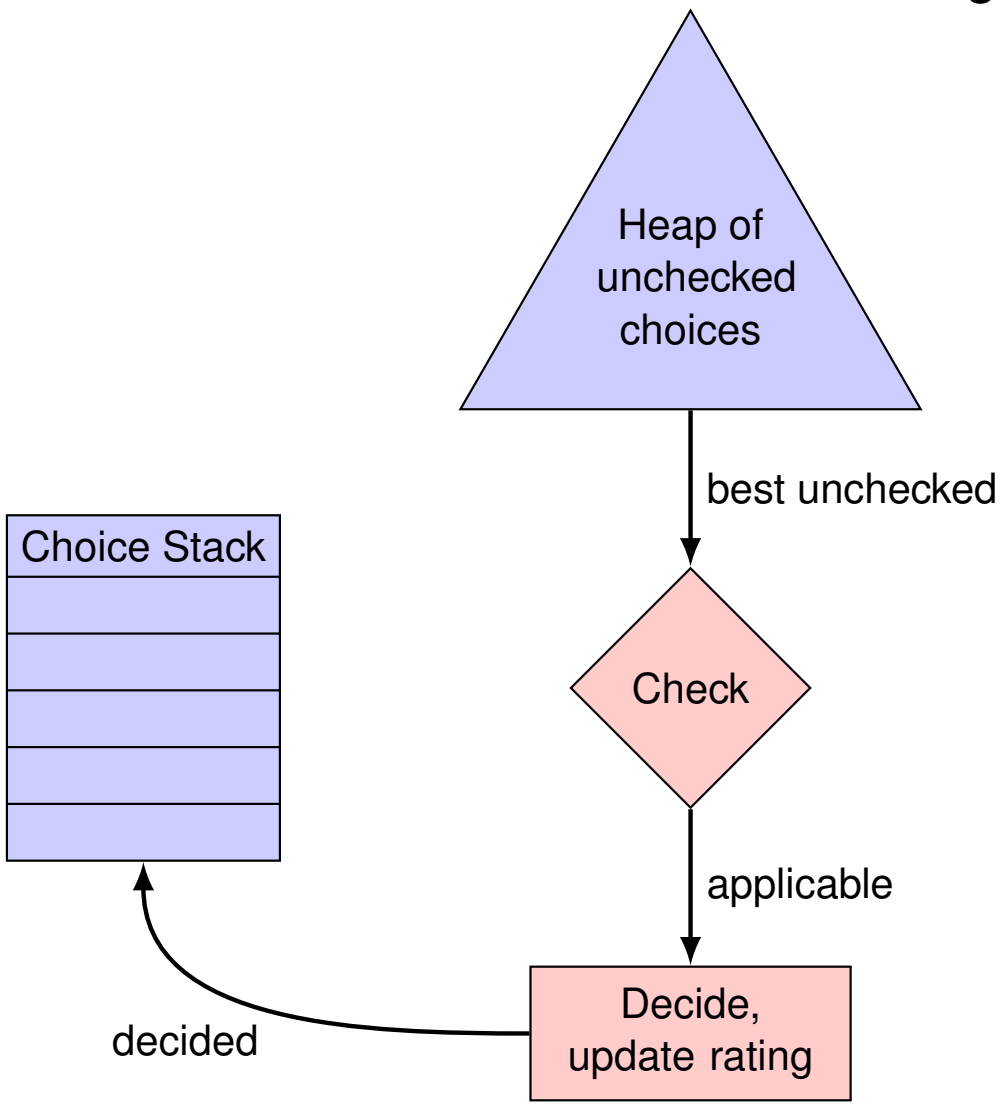
Search algorithm

DFS, Always decide the choice with the best rating.



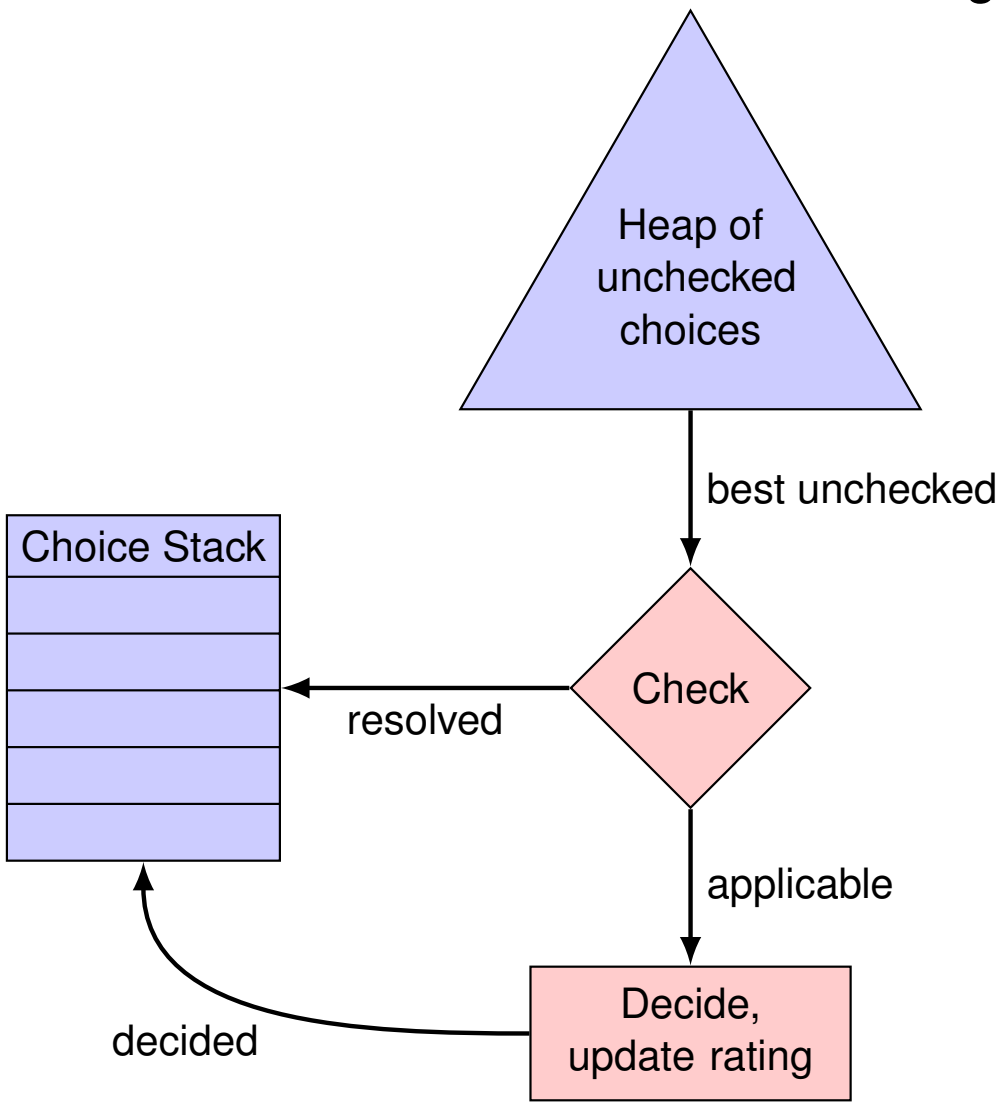
Search algorithm

DFS, Always decide the choice with the best rating.



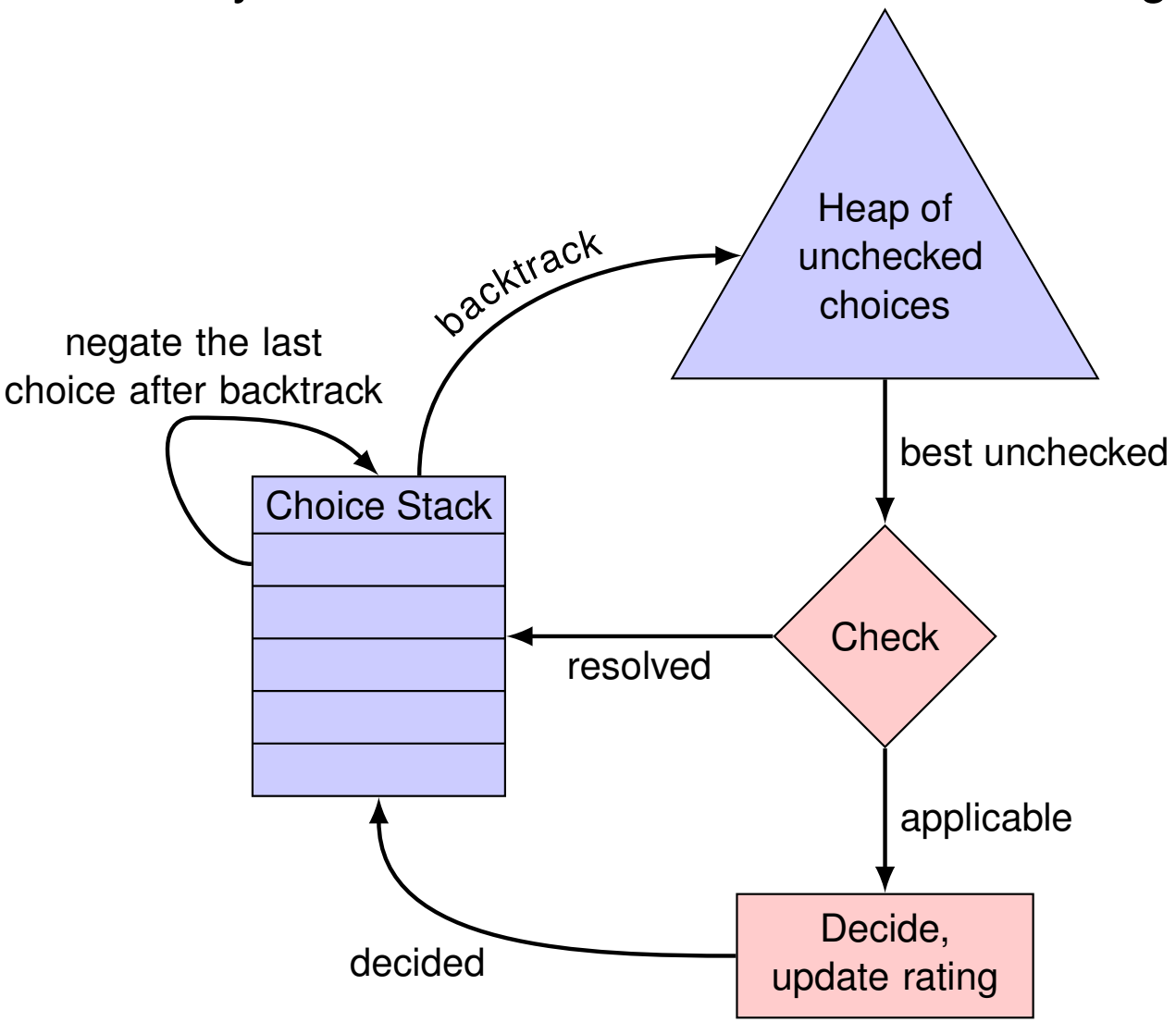
Search algorithm

DFS, Always decide the choice with the best rating.



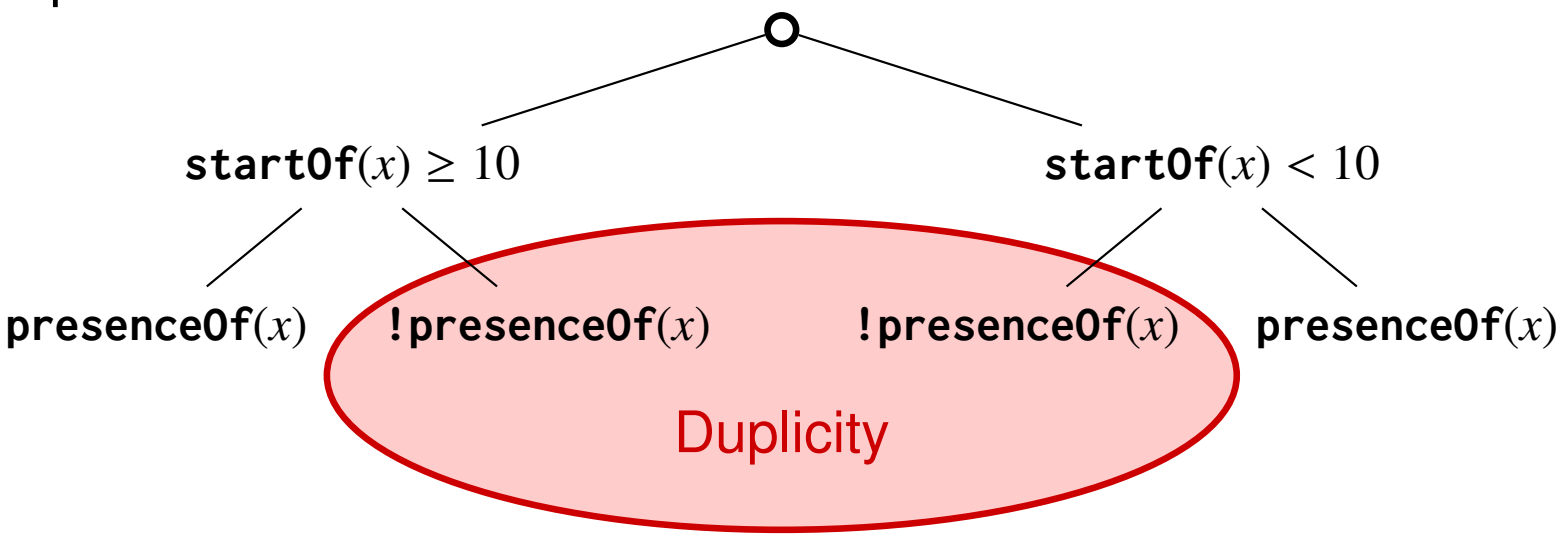
Search algorithm

DFS, Always decide the choice with the best rating.



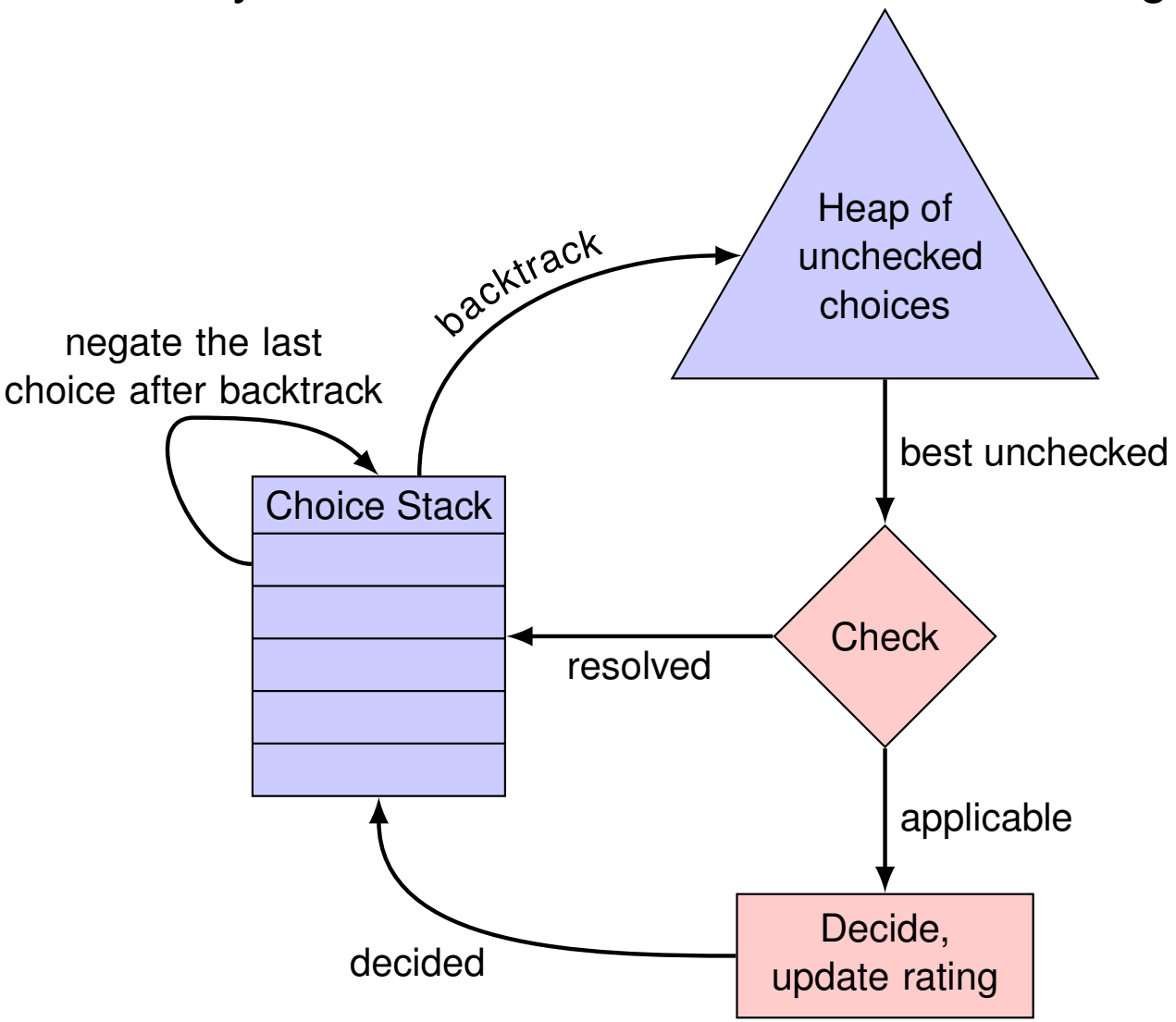
Decide presence status first

For optional interval variables, if we branch on time before branching on presence status then part of the search tree may do duplicate work:



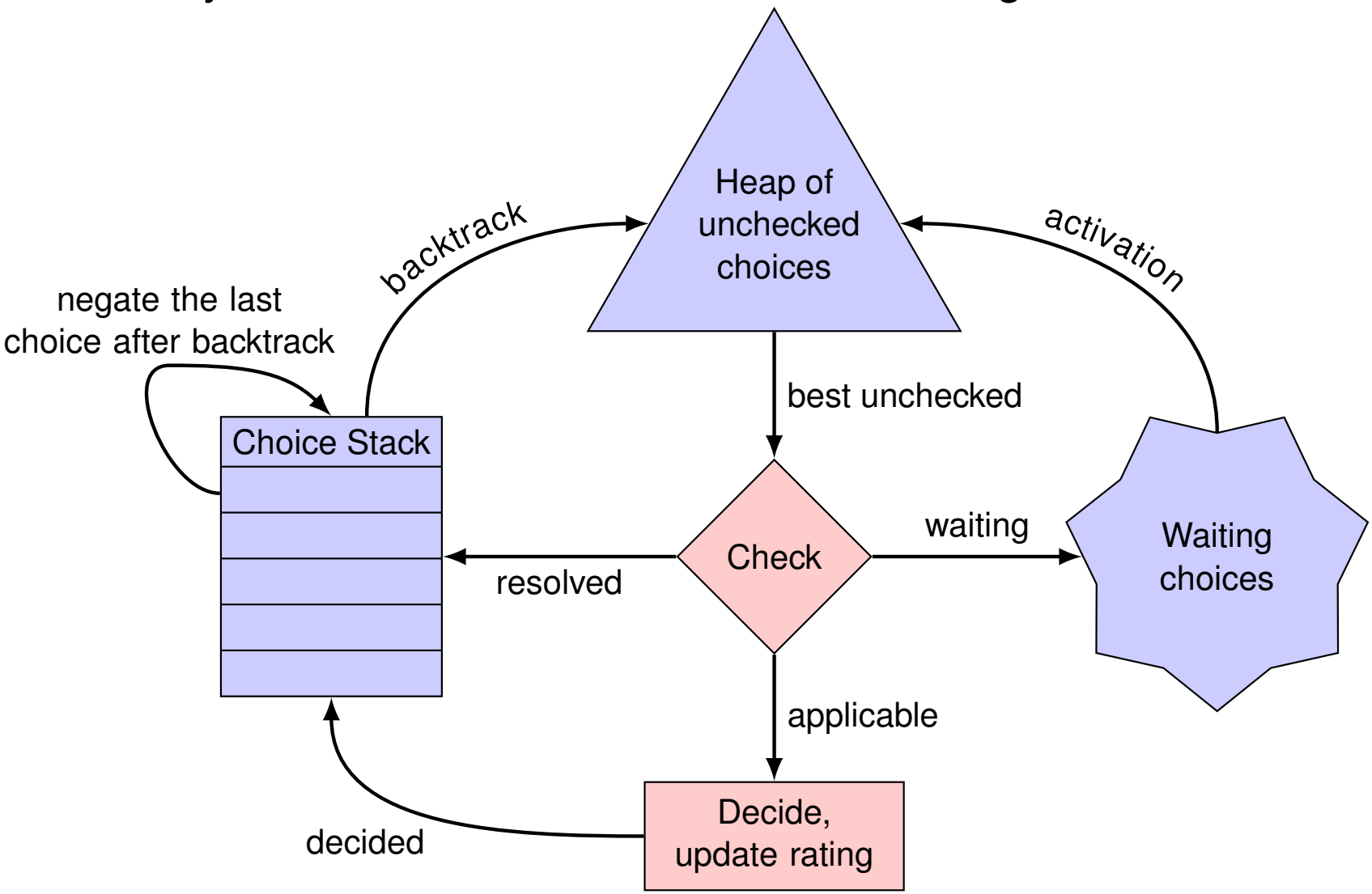
Search algorithm

DFS, Always decide the choice with the best rating.



Search algorithm

DFS, Always decide the choice with the best rating.



Which branch of the choice try first?

The one with better rating

- It is more likely to fail then to other branch.
- It heads towards failures and not to a solution.

Why?

- It is faster (on infeasible models).

And why?

- See later.

More ingredients

- Geometric restarts by 15% and nogoods.
- Initialize ratings by trying all choices in the root node.
- Strong branching and shaving.
  - In root node, evaluate multiple choices before committing to one.
  - Exploit failures and same propagations in both branches.
- Couple FDS with LNS.
- Maximum propagation available in CP Optimizer.
- Randomization.

Experimental results

Experimental setup

Our goal:

- Show that FDS is generic and works well as “plan B” LNS.
- Show that it is comparable with state of the art.
- $\Rightarrow$ evaluate on open benchmark instances (more than 1500).

Experimental setup

Our goal:

- Show that FDS is generic and works well as “plan B” LNS.
- Show that it is comparable with state of the art.
- $\Rightarrow$ evaluate on open benchmark instances (more than 1500).

Attack upper bounds:

- Solve the problem using two threads: LNS + FDS.
- Time limit 10 minutes up to 8 hours 20 minutes.

Experimental setup

Our goal:

- Show that FDS is generic and works well as “plan B” LNS.
- Show that it is comparable with state of the art.
- $\Rightarrow$ evaluate on open benchmark instances (more than 1500).

Attack upper bounds:

- Solve the problem using two threads: LNS + FDS.
- Time limit 10 minutes up to 8 hours 20 minutes.

Attack lower bounds:

- Use 1 thread running FDS only.
- Start with bestLB – 1 and continue up if infeasible.
- Time limit 5..10 minutes per step.

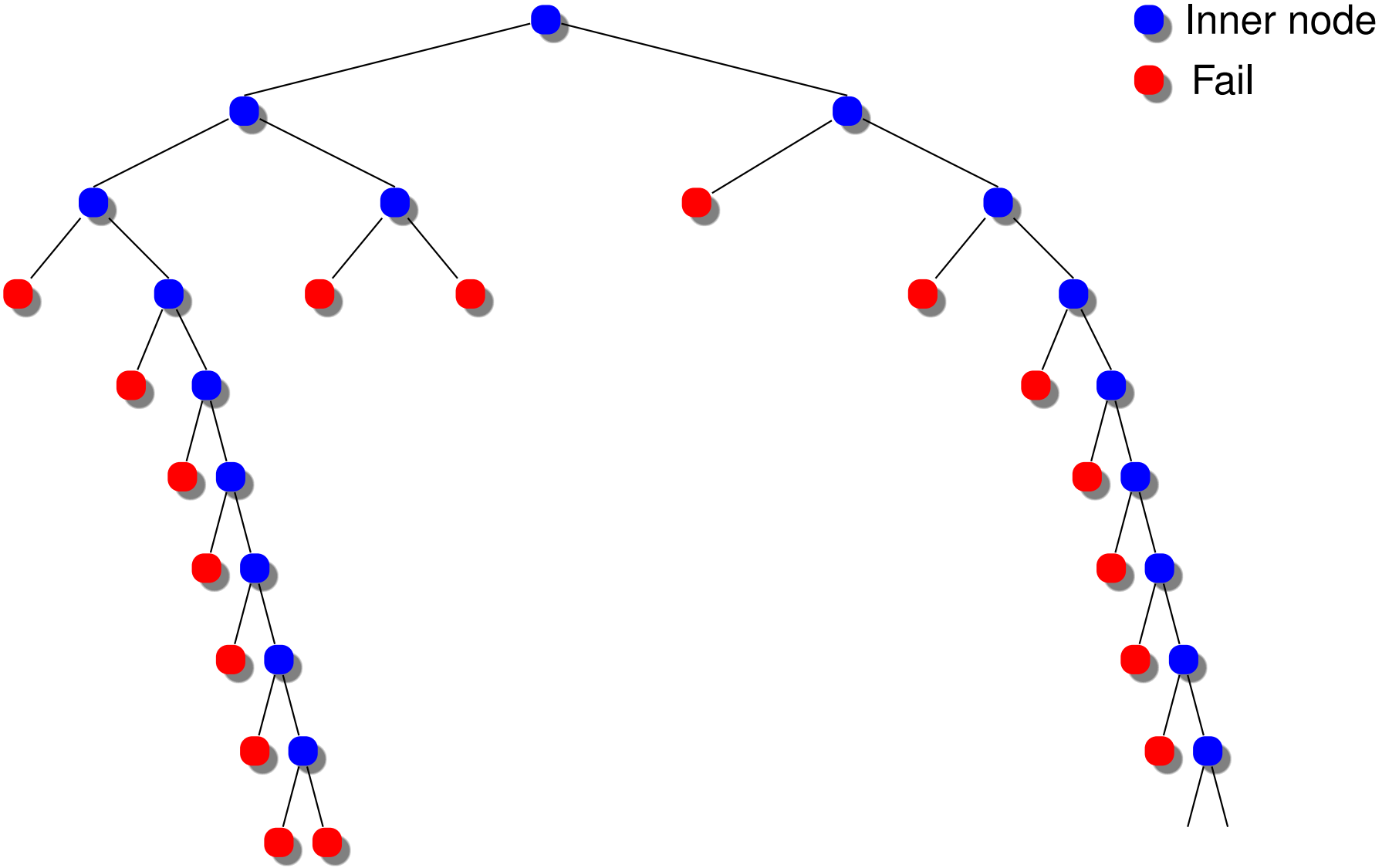
Experimental results

Benchmark set	# of instances	Lower bound improvements	Upper bound improvements	Closed instances
JobShop	48	40	3	15
JobShopOperators	222	107	215	208
FlexibleJobShop	107	67	39	74
RCPSP	472	52	1	0
RCPSPMax	58	51	23	1
MultiModeRCPSP (j30)	552	No reference	3	535
MultiModeRCPSPMax	85	84	77	85

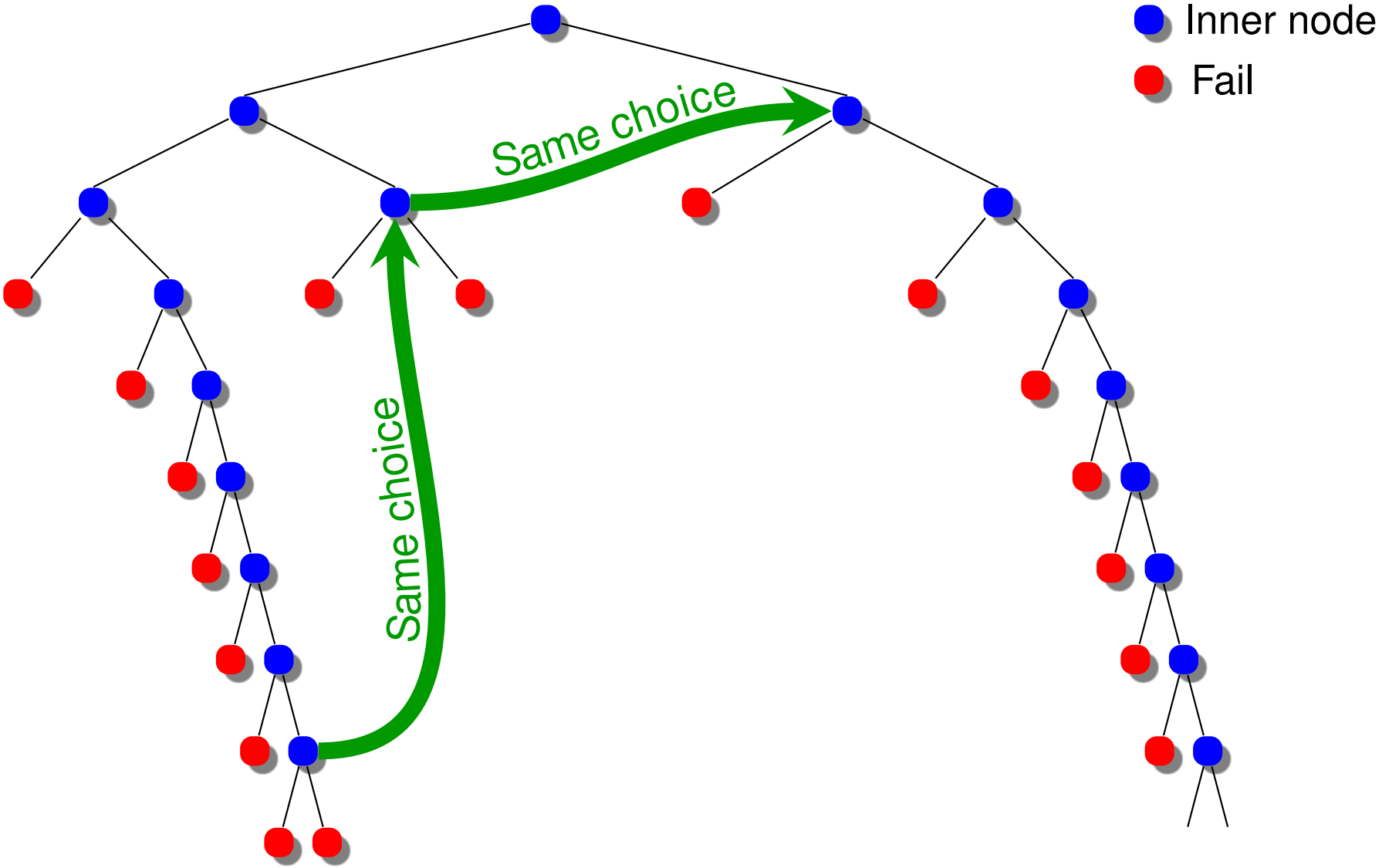
Why it works well?

Analyzing behavior of the search.

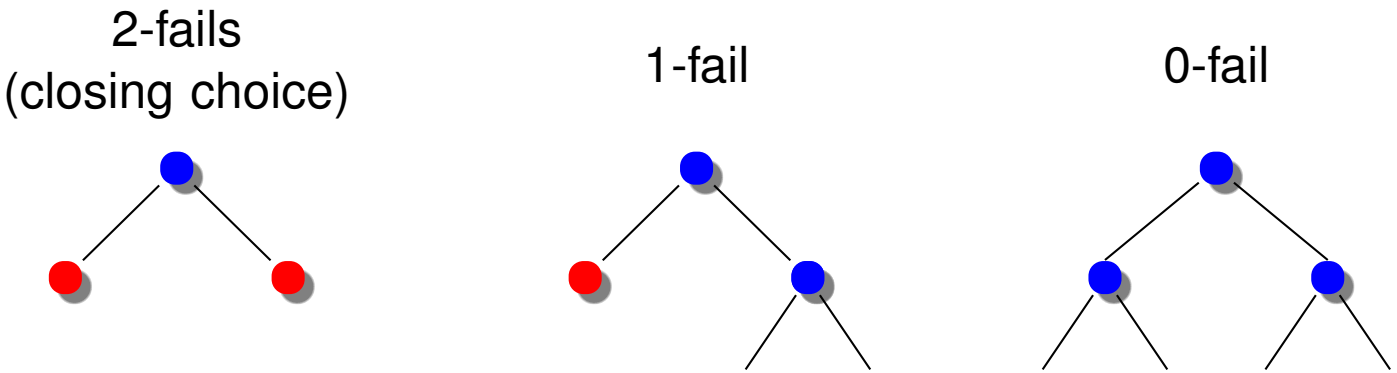
Search tree



Search tree



Why it (probably) works



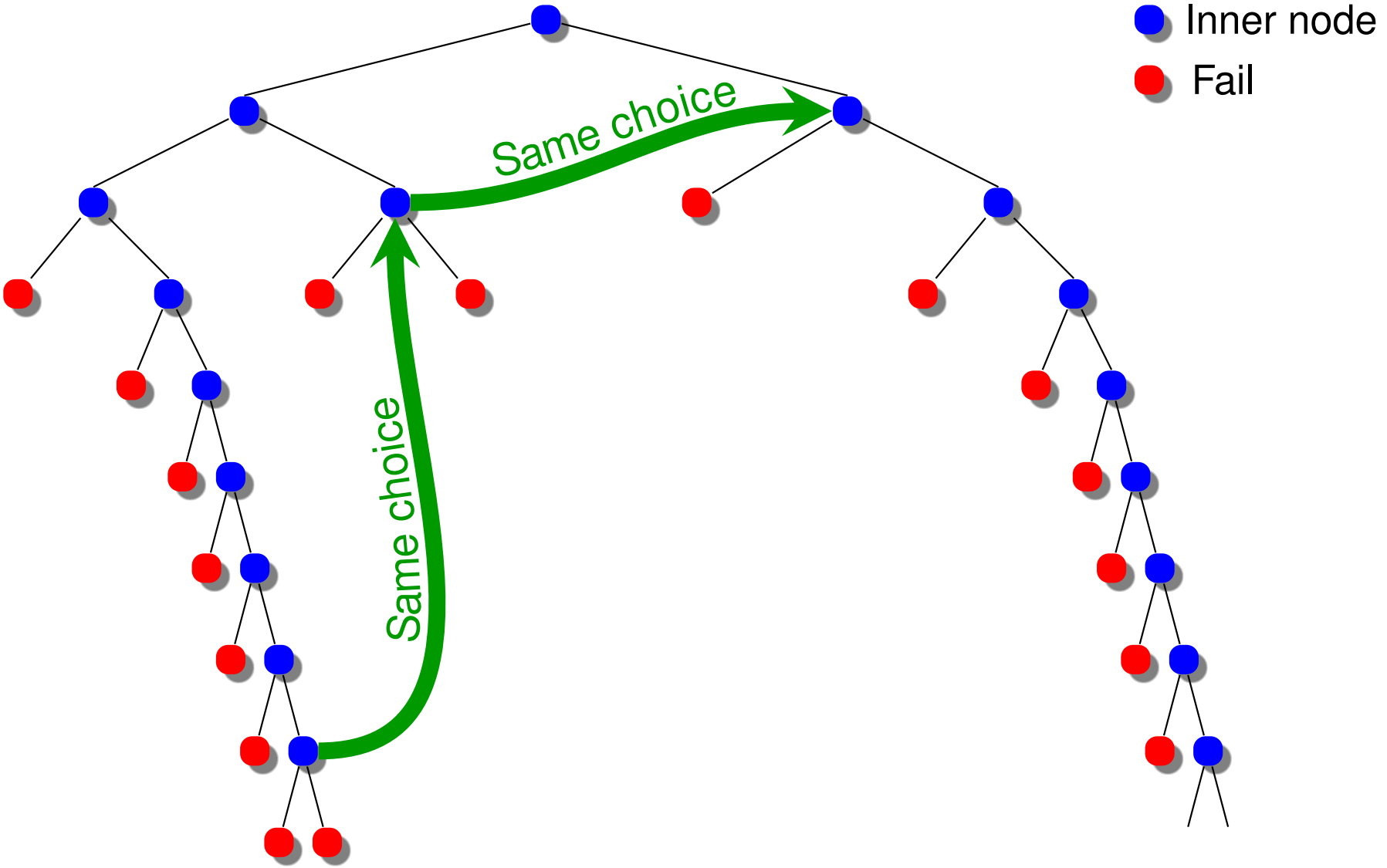
Closing choices:

- Mandatory to close a branch.
- Strongly preferred by rating system.
- Rating of a closing choice immediately improves.
  - Likely to be drawn again

1-fail choices:

- Does not increase number of open branches.
- 2-fails choices recruit from 1-fail choices.
- Not a big harm to repeat it.

Search tree

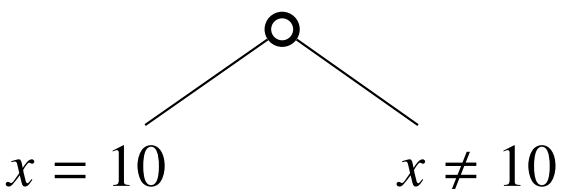


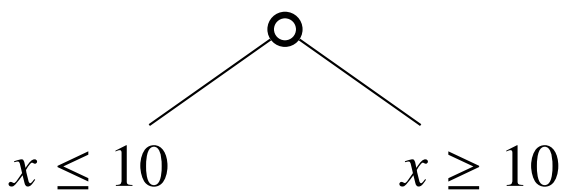
Why less promising branch first?

Assuming hard infeasible model:

- It is not likely that current restart finishes the proof.
- What remains after restart is a nogood constraint.
- Shorter nogoods are easier to apply.
  - Especially if left branch in root node was fully explored.
- Hope for cumulative effect of many short nogoods.

Traditional versus range branching on variable $x \in [0..1000]$:





Case study

jobshop `tail50` makespan ≤ 1832

- Infeasible, proof takes 465s.

Case study

jobshop `tail50` makespan ≤ 1832

- Infeasible, proof takes 465s.

Inverse branch order:

- Proof takes 1023s.

Case study

jobshop `tail50` makespan ≤ 1832

- Infeasible, proof takes 465s.

Inverse branch order:

- Proof takes 1023s.

Don't prefer failures:

$$\text{localRating} := \begin{cases} 0 & \text{if the branch fails immediately} \\ R \cancel{+1} & \text{otherwise} \end{cases}$$

- No proof in 24 hours.

Case study

jobshop `tai50` makespan ≤ 1832

- Infeasible, proof takes 465s.

Inverse branch order:

- Proof takes 1023s.

Don't prefer failures:

$$\text{localRating} := \begin{cases} 0 & \text{if the branch fails immediately} \\ R \cancel{+1} & \text{otherwise} \end{cases}$$

- No proof in 24 hours.

Parallelism (since version 12.6.2):

- 317s using 2 workers (32% faster).


thank you
danke
謝謝
ngiyabonga
teşekkür ederim
gracias
tapadh leat
mochchakkeram
go raibh maith agat
merci
dakujem
arigatô
grazie
kop khun krap
sukriya
terima kasih
obrigado
dziękuję
bedankt
спасибо
Баярлалаа
рахмат
dank je
mersi
barka
welalin
tack
blagodaram
misaotra
matondo
paldies
grazzi
mahalo
хвала
asante
manana
obrigada
tenki
murakoze
chokrane
mamnun
djiere dieuf
faafetai lava
kiitos
dankie
dhanyavad
hvala
maururu
koszonom
enkosi
sobodi
dêkuji
mèsi
sagolun
chnorakaloutioun
gratias ago
gracies
sulpay
taiku
tanemirt
rahmet
trugarez
мерси
shukriya
dhanyavadagalu
diolch
xiexie
감사합니다
ありがとう
najis tuke
kam sah hamnida
didi madloba
rahmat
ευχαριστώ

Questions?