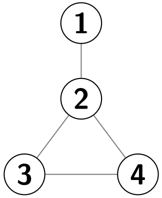
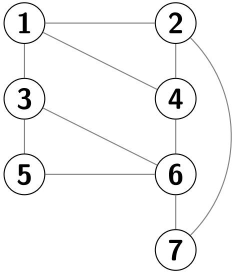
Parallel Search, Backjumping, and Brittle Skeletons

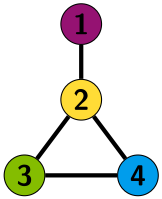
Ciaran McCreesh and Patrick Prosser

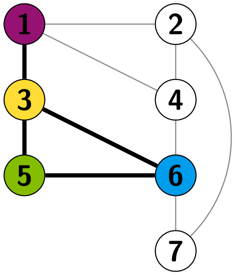
The Subgraph Isomorphism Problem



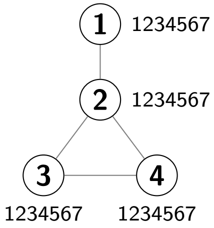


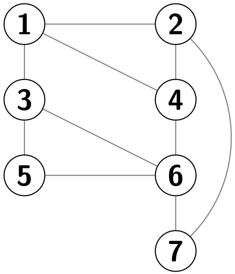
The Subgraph Isomorphism Problem



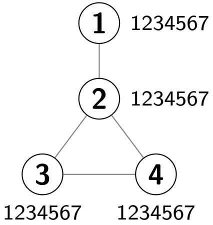


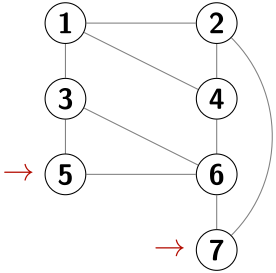
Filtering



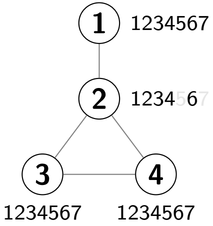


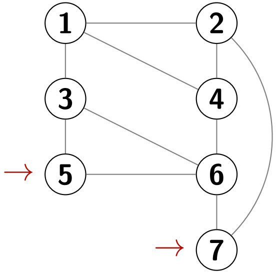
Filtering



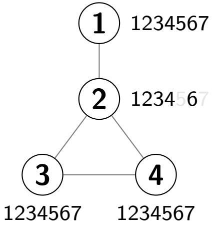


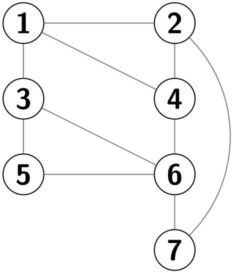
Filtering



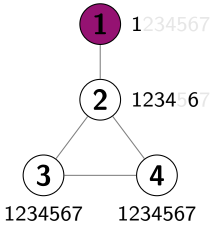


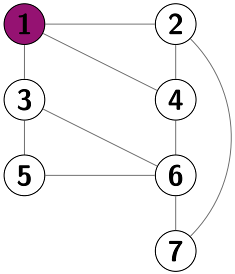
Filtering



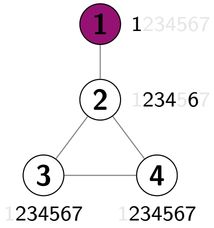


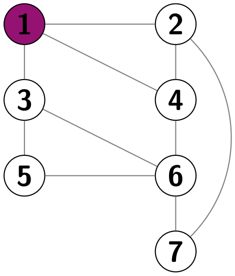
Backtracking Search



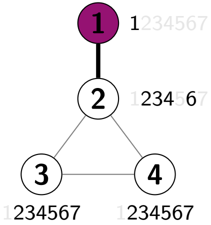


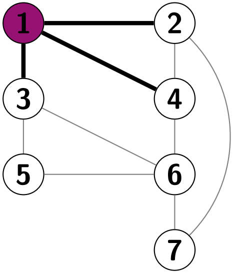
Backtracking Search



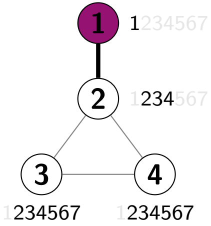


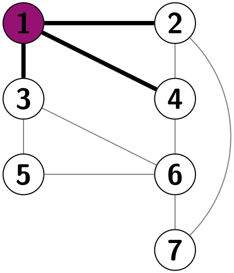
Backtracking Search



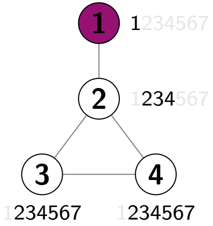


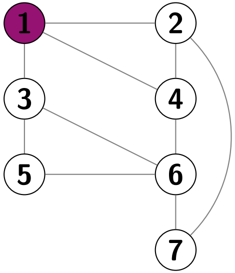
Backtracking Search



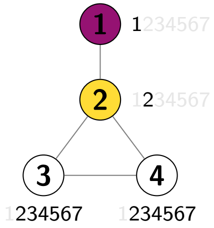


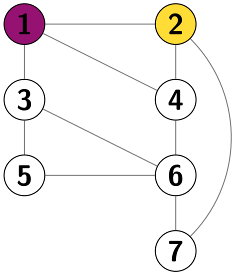
Backtracking Search



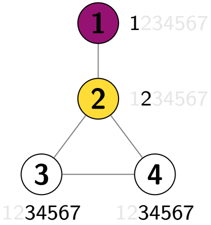


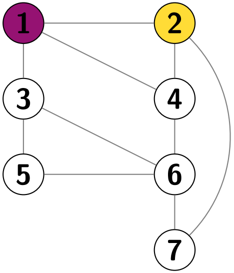
Backtracking Search



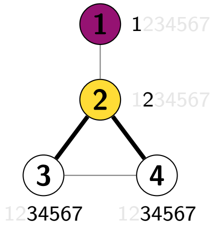


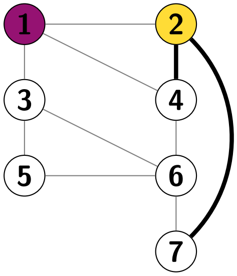
Backtracking Search



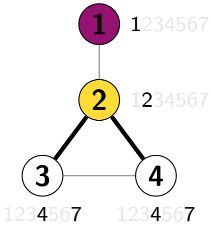


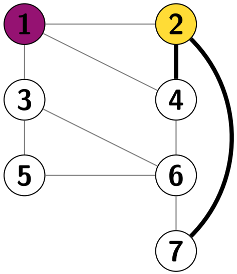
Backtracking Search



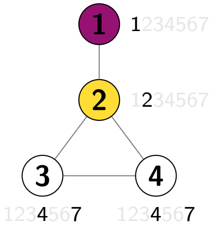


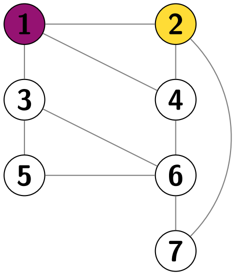
Backtracking Search



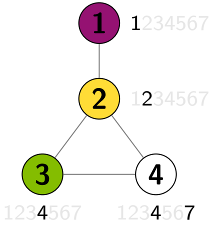


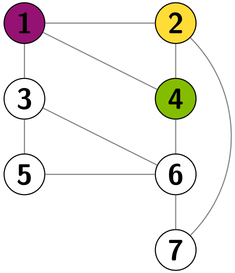
Backtracking Search



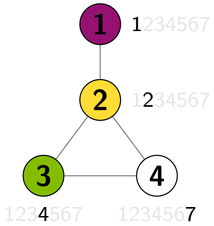


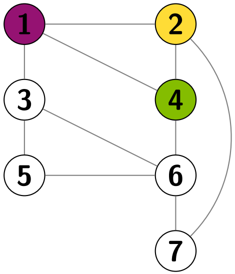
Backtracking Search



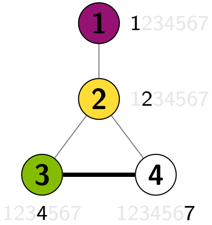


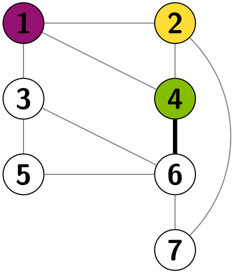
Backtracking Search



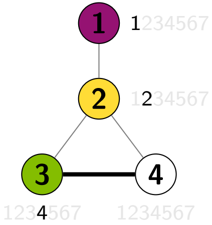


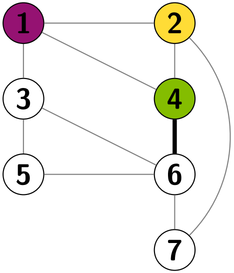
Backtracking Search



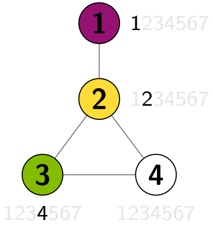


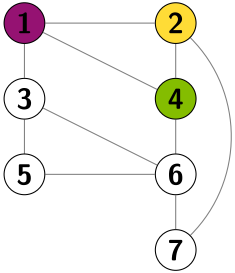
Backtracking Search



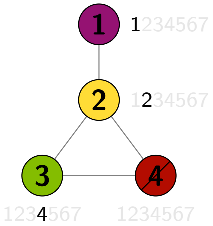


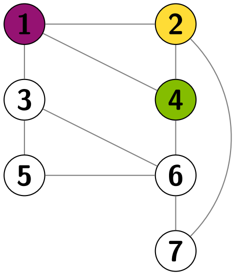
Backtracking Search



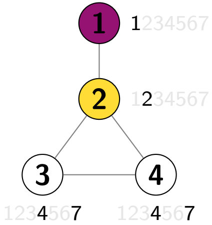


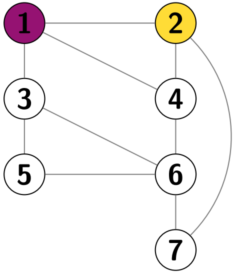
Backtracking Search



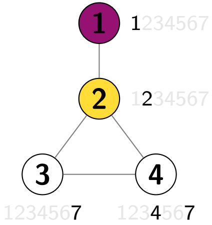


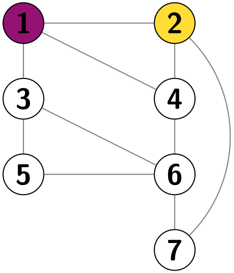
Backtracking Search



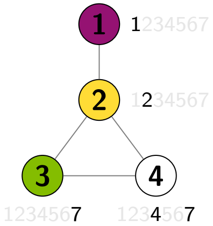


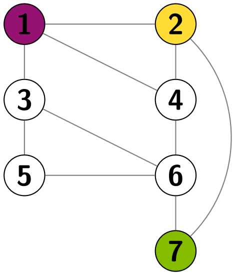
Backtracking Search



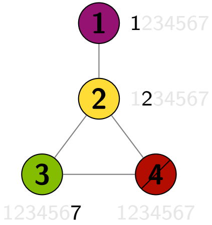


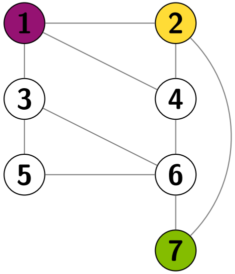
Backtracking Search



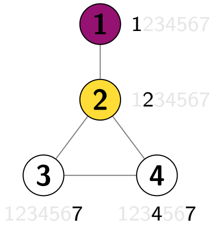


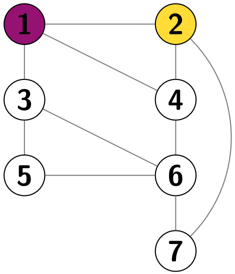
Backtracking Search



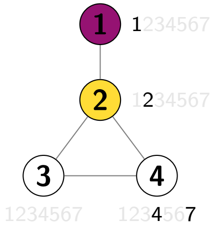


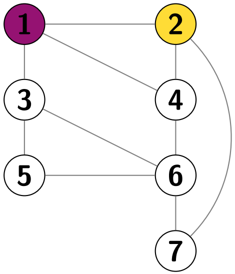
Backtracking Search



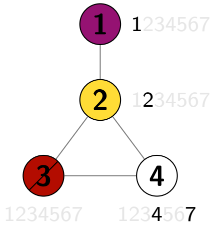


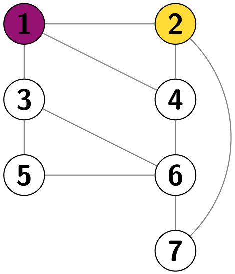
Backtracking Search



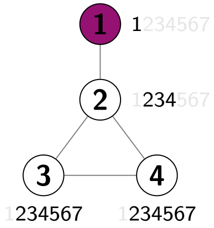


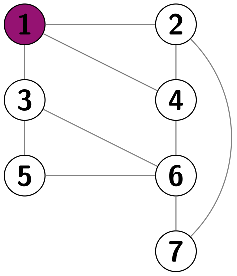
Backtracking Search



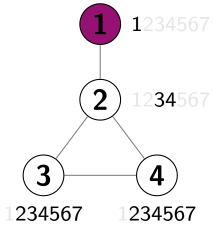


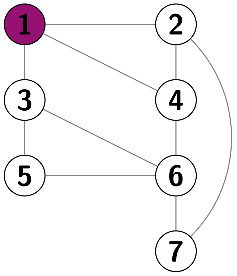
Backtracking Search



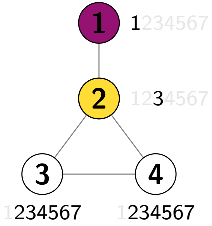


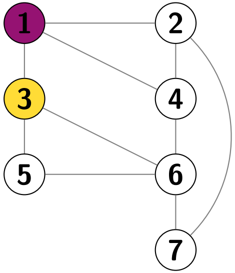
Backtracking Search



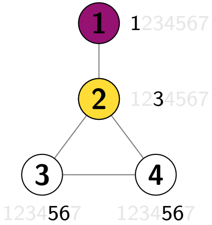


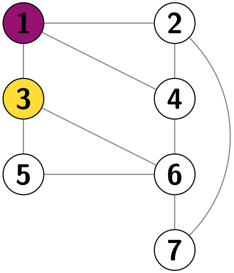
Backtracking Search



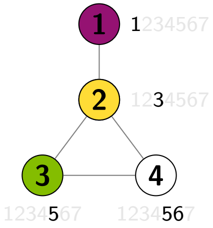


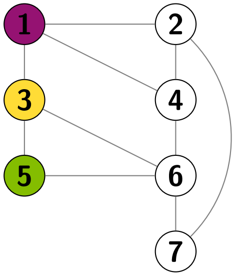
Backtracking Search



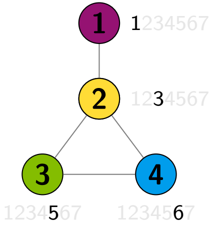


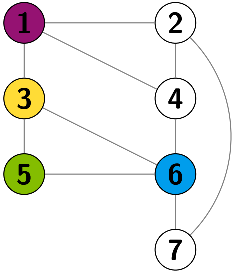
Backtracking Search





Backtracking Search

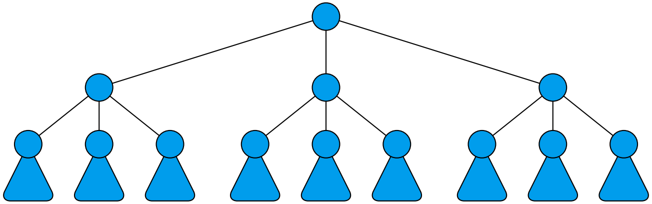




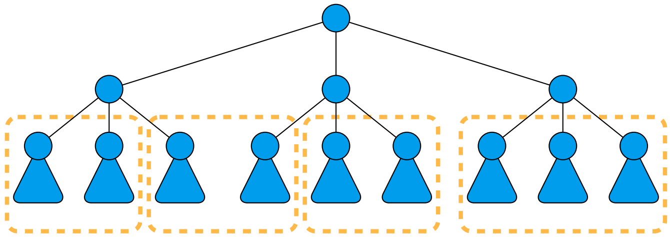
A Backtracking Algorithm

```
search (Domains $D$) $\rightarrow$ Fail **or** Success
**begin**
    **if** $D = \emptyset$ **then return** Success
    $D_v \leftarrow$ a domain in $D$ with minimum size
    **foreach** $v' \in D_v$ ordered by a heuristic **do**
        $D' \leftarrow$ clone($D$)
        **case** assign($D', v, v'$) **of**
            Fail **then** keep going
            Success **then**
                **case** search($D' - D_v$) **of**
                    Fail **then** keep going
                    Success **then return** Success
    **return** Fail
```

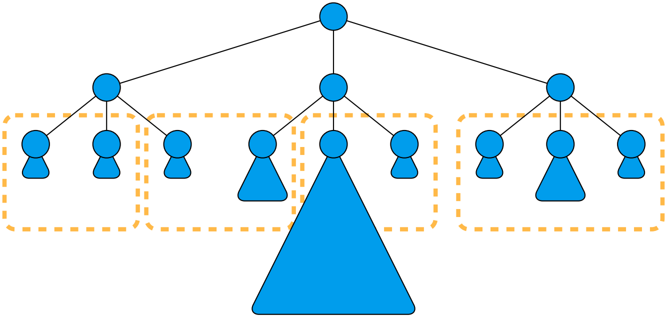
Search as a Tree



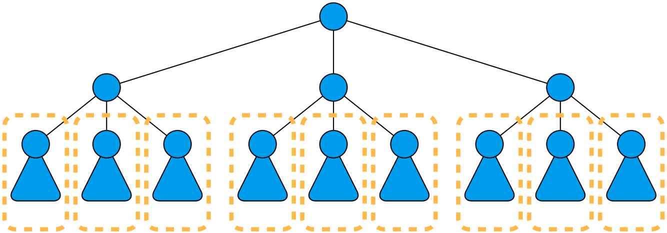
Parallel Search



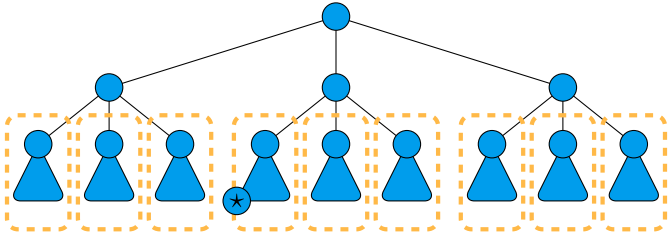
Parallel Search



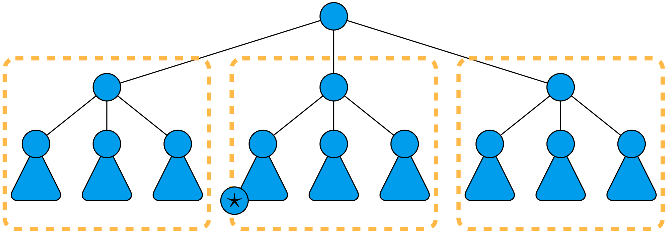
Parallel Search



Work-Stealing is Not Just About Balance



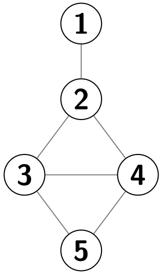
Work-Stealing is Not Just About Balance

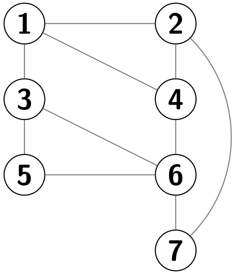


Preventing a Slowdown, Part 1

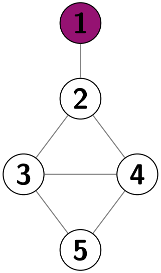
- At least one thread must preserve the “sequential” search order.
- If a solution is found, we must cancel all other workers immediately.

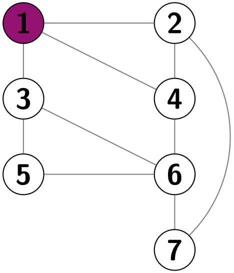
Backjumping



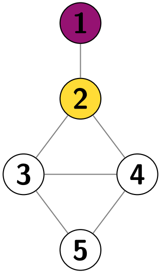


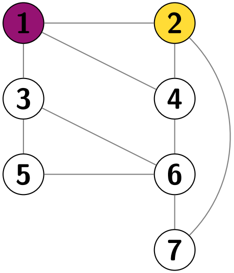
Backjumping



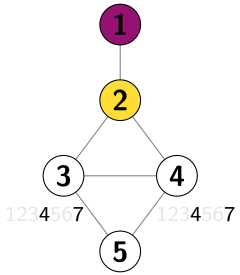


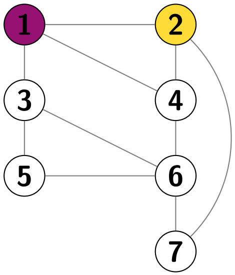
Backjumping



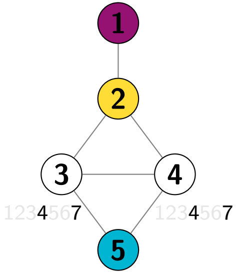


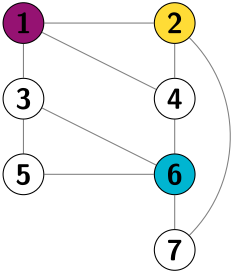
Backjumping



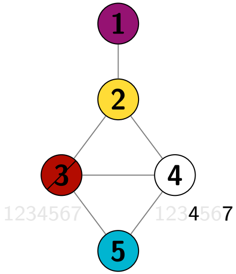


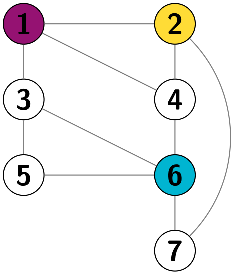
Backjumping



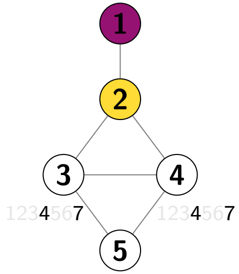


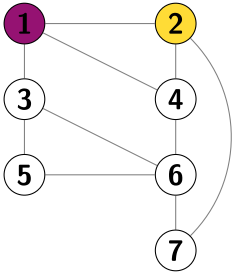
Backjumping



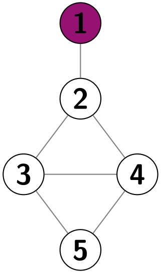


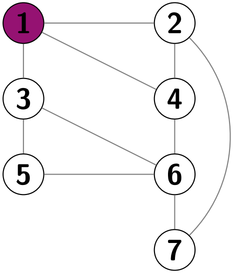
Backjumping



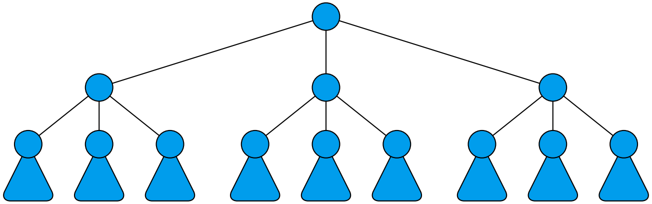


Backjumping

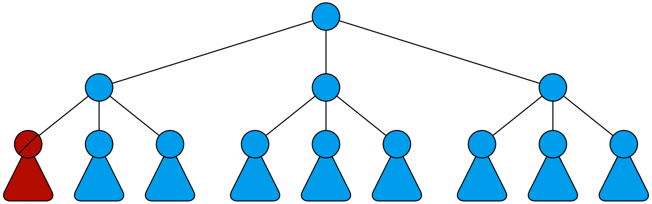




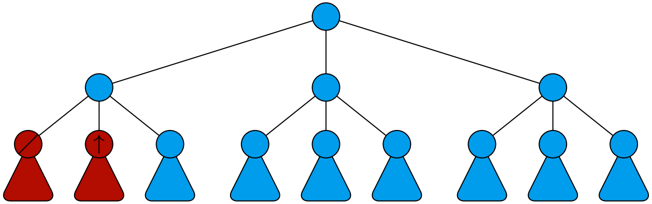
Backjumping as a Tree



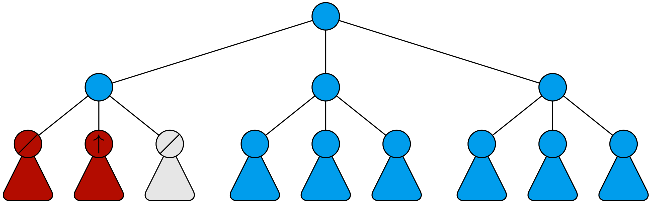
Backjumping as a Tree



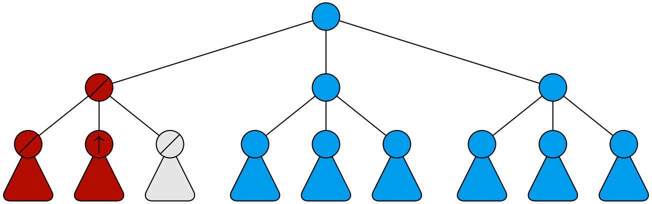
Backjumping as a Tree



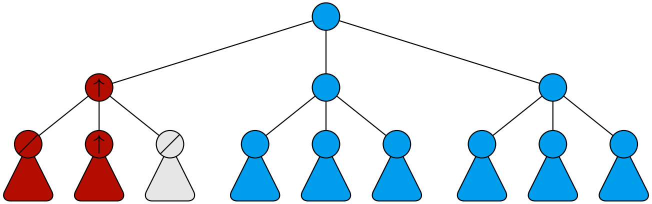
Backjumping as a Tree



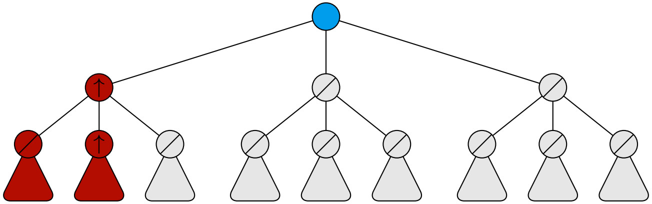
Backjumping as a Tree



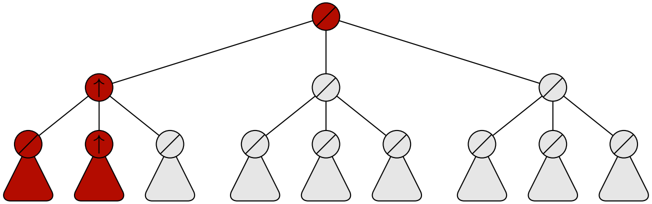
Backjumping as a Tree



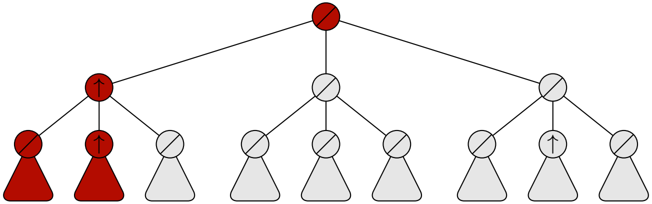
Backjumping as a Tree



Backjumping as a Tree



Backjumping as a Tree



A Backjumping Algorithm

```
search (Domains $D$) $\rightarrow$ Fail $F$ or Success
begin
    if $D = \emptyset$ then return Success
    $D_v \leftarrow$ a domain in $D$ with minimum size
    $F \leftarrow \{v\}$
    foreach $v' \in D_v$ ordered by a heuristic do
        $D' \leftarrow$ clone($D$)
        case assign($D', v, v'$) of
            Fail $F'$ then $F \leftarrow F \cup F'$
            Success then
                case search($D' - D_v$) of
                    Fail $F'$ then
                        if $\nexists w \in F'$ such that $D_w \neq D'_w$ then return Fail $F'$
                        $F \leftarrow F \cup F'$
                    Success then return Success
    return Fail $F$
```

A Backjumping Algorithm

```
search (Domains $D$) $\rightarrow$ Fail $F$ **or** Success
**begin**
    **if** $D = \emptyset$ **then return** Success
    $D_v \leftarrow$ a domain in $D$ with minimum size
    $F \leftarrow \{v\}$
    **foreach** $v' \in D_v$ ordered by a heuristic **do**
        $D' \leftarrow$ clone($D$)
        **case** assign($D', v, v'$) **of**
            Fail $F'$ **then** $F \leftarrow F \cup F'$
            Success **then**
                **case** search($D' - D_v$) **of**
                    Fail $F'$ **then**
                        **if** $\nexists w \in F'$ **such that** $D_w \neq D'_w$ **then return** Fail $F'$
                        $F \leftarrow F \cup F'$
                    Success **then return** Success
    **return** Fail $F$
```

A Backjumping Algorithm

```
search (Domains $D$) $\rightarrow$ Fail $F$ or Success
begin
    if $D = \emptyset$ then return Success
    $D_v \leftarrow$ a domain in $D$ with minimum size
    $F \leftarrow \{v\}$
    foreach $v' \in D_v$ ordered by a heuristic do
        $D' \leftarrow$ clone($D$)
        case assign($D', v, v'$) of
            Fail $F'$ then $F \leftarrow F \cup F'$
            Success then
                case search($D' - D_v$) of
                    Fail $F'$ then
                        if $\nexists w \in F'$ such that $D_w \neq D'_w$ then return Fail $F'$
                        $F \leftarrow F \cup F'$
                    Success then return Success
    return Fail $F$
```

A Backjumping Algorithm

```
search (Domains $D$) $\rightarrow$ Fail $F$ or Success
begin
    if $D = \emptyset$ then return Success
    $D_v \leftarrow$ a domain in $D$ with minimum size
    $F \leftarrow \{v\}$
    foreach $v' \in D_v$ ordered by a heuristic do
        $D' \leftarrow$ clone($D$)
        case assign($D', v, v'$) of
            Fail $F'$ then $F \leftarrow F \cup F'$
            Success then
                case search($D' - D_v$) of
                    Fail $F'$ then
                        if $\nexists w \in F'$ such that $D_w \neq D'_w$ then return Fail $F'$
                        $F \leftarrow F \cup F'$
                    Success then return Success
    return Fail $F$
```

A Backjumping Algorithm

```
search (Domains $D$) $\rightarrow$ Fail $F$ or Success
begin
    if $D = \emptyset$ then return Success
    $D_v \leftarrow$ a domain in $D$ with minimum size
    $F \leftarrow \{v\}$
    foreach $v' \in D_v$ ordered by a heuristic do
        $D' \leftarrow$ clone($D$)
        case assign($D', v, v'$) of
            Fail $F'$ then $F \leftarrow F \cup F'$
            Success then
                case search($D' - D_v$) of
                    Fail $F'$ then
                        if $\nexists w \in F'$ such that $D_w \neq D'_w$ then return Fail $F'$
                        $F \leftarrow F \cup F'$
                    Success then return Success
    return Fail $F$
```

A Backjumping Algorithm

```
search (Domains $D$) $\rightarrow$ Fail $F$ or Success
begin
    if $D = \emptyset$ then return Success
    $D_v \leftarrow$ a domain in $D$ with minimum size
    $F \leftarrow \{v\}$
    foreach $v' \in D_v$ ordered by a heuristic do
        $D' \leftarrow$ clone($D$)
        case assign($D', v, v'$) of
            Fail $F'$ then $F \leftarrow F \cup F'$
            Success then
                case search($D' - D_v$) of
                    Fail $F'$ then
                        if $\nexists w \in F'$ such that $D_w \neq D'_w$ then return Fail $F'$
                        $F \leftarrow F \cup F'$
                    Success then return Success
    return Fail $F$
```

Backjumping as a Lazy Fold

- Lazily map each subproblem to Jump F **or** Fail F **or** Success.
- Lazily fold, starting with Fail $\{v\}$, as follows:

$$\begin{aligned}
\_\_ \circledgtr \text{Success} &= \text{Success} \\
\_\_ \circledgtr \text{Jump } F &= \text{Jump } F \\
\text{Fail } F \circledgtr \text{Fail } G &= \text{Fail } (F \cup G)
\end{aligned}$$

- If a Jump F occurs to the left of a Success, we have a bug.

Folding Zero

- When multiplying, if any item is 0, the result is 0.

$$\begin{aligned} \_\_ \times 0 &= 0 \\ 0 \times \_\_ &= 0 \end{aligned}$$

- Here, if any item is Success, the result is Success, and we do not need to evaluate the rest of the map.

$$\_\_ \circledcirc \text{Success} = \text{Success}$$

- If any item is Jump F, the result is either Jump F, or some Jump G or Success that is further to the left. We do not need to evaluate any item to the right.

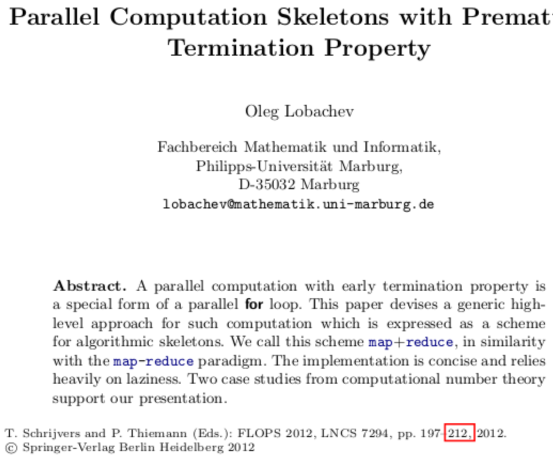
$$\_\_ \circledcirc \text{Jump } F = \text{Jump } F$$

Preventing a Slowdown, Part 2

- Any subproblem which we have shown will not be used, must be cancelled (recursively) immediately.
- When the result of a fold is known, the continuation must be executed immediately.

Some Grumpy Remarks about Brittle Skeletons

Some Grumpy Remarks about Brittle Skeletons



Parallel Computation Skeletons with Premature Termination Property

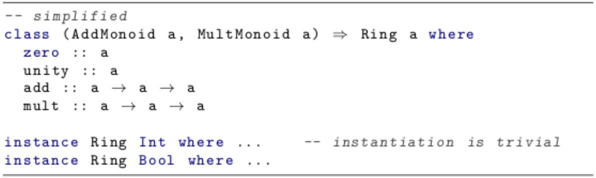
Oleg Lobachev

Fachbereich Mathematik und Informatik,
Philipps-Universität Marburg,
D-35032 Marburg
lobachev@mathematik.uni-marburg.de

Abstract. A parallel computation with early termination property is a special form of a parallel **for** loop. This paper devises a generic high-level approach for such computation which is expressed as a scheme for algorithmic skeletons. We call this scheme `map+reduce`, in similarity with the `map-reduce` paradigm. The implementation is concise and relies heavily on laziness. Two case studies from computational number theory support our presentation.

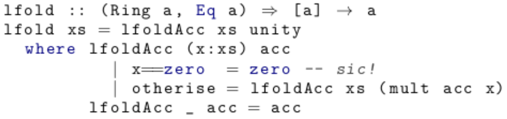
T. Schrijvers and P. Thiemann (Eds.): FLOPS 2012, LNCS 7294, pp. 197–$\boxed{212,}$ 2012.
© Springer-Verlag Berlin Heidelberg 2012

Some Grumpy Remarks about Brittle Skeletons



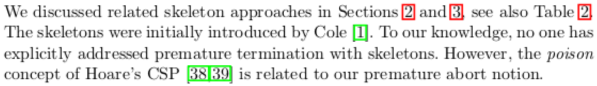
```
-- simplified
class (AddMonoid a, MultMonoid a) ⇒ Ring a where
  zero :: a
  unity :: a
  add :: a → a → a
  mult :: a → a → a

instance Ring Int where ...    -- instantiation is trivial
instance Ring Bool where ...
```

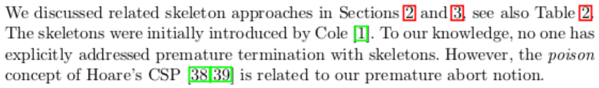


```
lfold :: (Ring a, Eq a) ⇒ [a] → a
lfold xs = lfoldAcc xs unity
  where lfoldAcc (x:xs) acc
          | x==zero  = zero -- sic!
          | otherise = lfoldAcc xs (mult acc x)
        lfoldAcc _ acc = acc
```

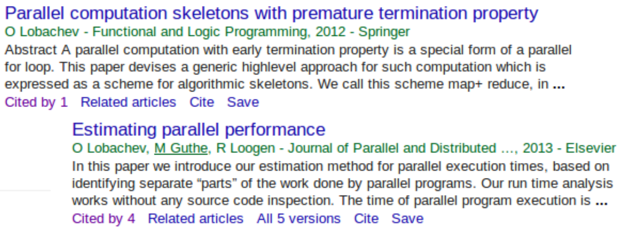
Some Grumpy Remarks about Brittle Skeletons


We discussed related skeleton approaches in Sections 2 and 3, see also Table 2. The skeletons were initially introduced by Cole [1]. To our knowledge, no one has explicitly addressed premature termination with skeletons. However, the *poison* concept of Hoare's CSP [38,39] is related to our premature abort notion.

Some Grumpy Remarks about Brittle Skeletons



We discussed related skeleton approaches in Sections [2] and [3], see also Table [2]. The skeletons were initially introduced by Cole [1]. To our knowledge, no one has explicitly addressed premature termination with skeletons. However, the *poison* concept of Hoare's CSP [38,39] is related to our premature abort notion.



Parallel computation skeletons with premature termination property
O Lobachev - Functional and Logic Programming, 2012 - Springer
Abstract A parallel computation with early termination property is a special form of a parallel for loop. This paper devises a generic highlevel approach for such computation which is expressed as a scheme for algorithmic skeletons. We call this scheme map+ reduce, in ...
Cited by 1 Related articles Cite Save

Estimating parallel performance
O Lobachev, M Guthe, R Loogen - Journal of Parallel and Distributed ..., 2013 - Elsevier
In this paper we introduce our estimation method for parallel execution times, based on identifying separate "parts" of the work done by parallel programs. Our run time analysis works without any source code inspection. The time of parallel program execution is ...
Cited by 4 Related articles All 5 versions Cite Save

What’s the Alternative?

What's the Alternative?

- Doing it by hand?
  - This works, but is painful and error-prone. . .
  - My current implementation works by keeping a "sequential" thread and "precomputing" using extra threads. This often leads to the sequential thread being idle and blocking.
  - Allowing the blocking thread to suspend and steal elsewhere could give an absolute slowdown.

What's the Alternative?

- Doing it by hand?
    - This works, but is painful and error-prone...
    - My current implementation works by keeping a "sequential" thread and "precomputing" using extra threads. This often leads to the sequential thread being idle and blocking.
    - Allowing the blocking thread to suspend and steal elsewhere could give an absolute slowdown.
- Better skeletons?
    - But they would need to be very domain-specific, which defeats the point of skeletons...

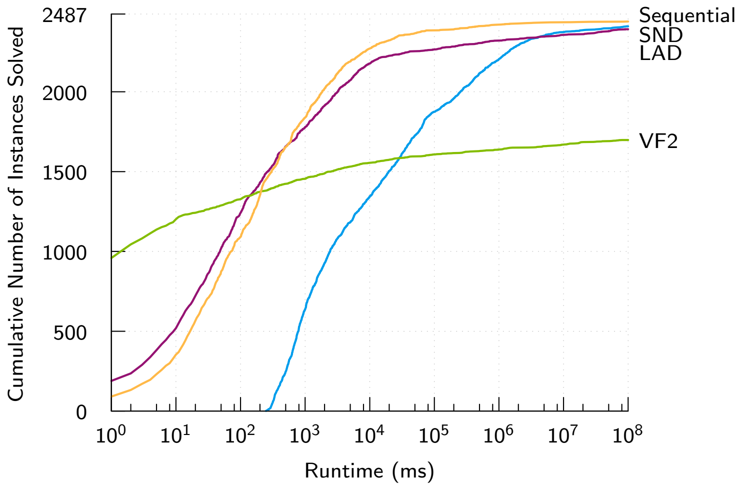
What's the Alternative?

- Doing it by hand?
  - This works, but is painful and error-prone...
  - My current implementation works by keeping a "sequential" thread and "precomputing" using extra threads. This often leads to the sequential thread being idle and blocking.
  - Allowing the blocking thread to suspend and steal elsewhere could give an absolute slowdown.
- Better skeletons?
  - But they would need to be very domain-specific, which defeats the point of skeletons...
- External descriptions of search?
  - I've yet to figure out why this will end up not being very good...

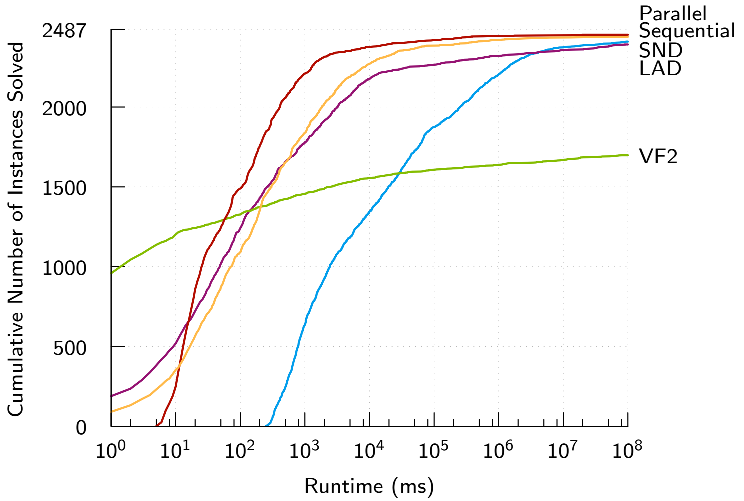
A Quick Look at Some Results

- 2,487 pattern / target pairs from 11 families of benchmark problems.
- 32 threads, 16 core HT system.
- A more complicated algorithm than the one I've described (all-different filtering, supplemental graphs, . . .).
- Some boring parallel preprocessing too.
- C++11 native threads.

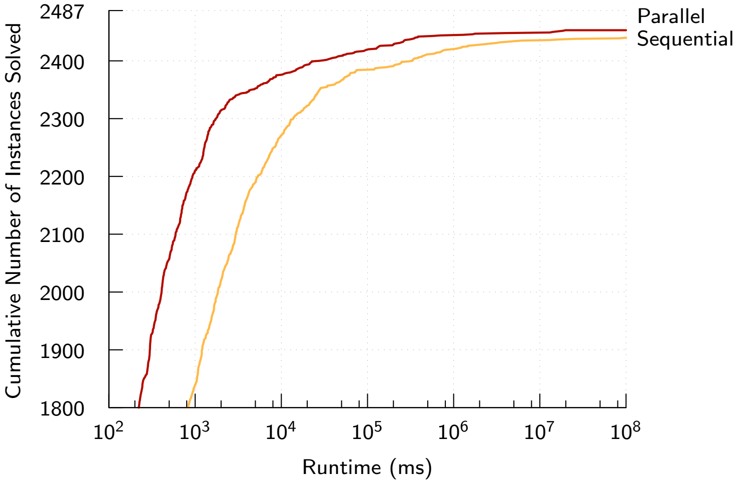
A Quick Look at Some Results



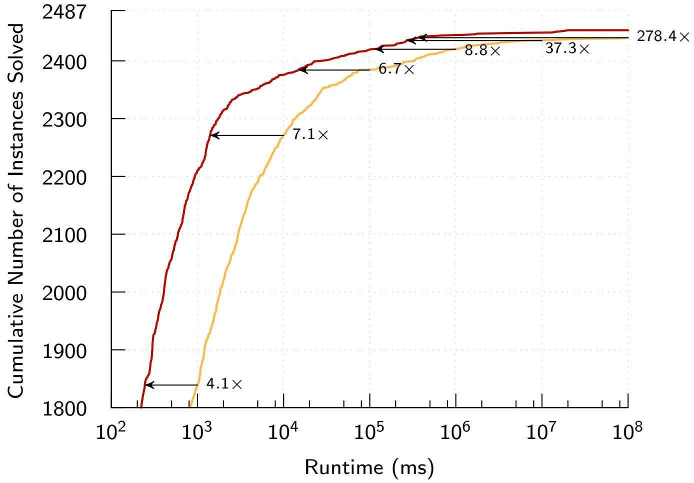
A Quick Look at Some Results



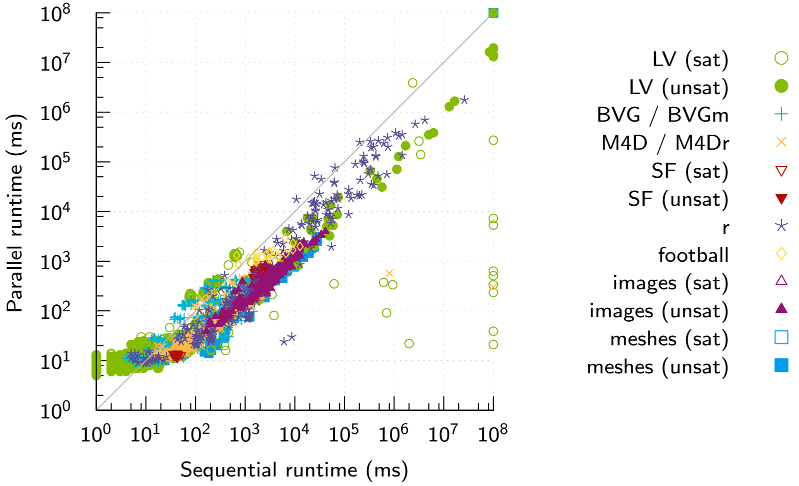
A Quick Look at Some Results



A Quick Look at Some Results



A Quick Look at Some Results



A Quick Look at Some Results

- 150 LOC for a suitable priority queue.
- Search function goes from 40 LOC to 120 LOC.
- Horribly intrusive, and making it distributed would be seriously painful.
- 7% slower when run with one thread, even when the queue for stealing is removed.

http://dcs.gla.ac.uk/~ciaran
c.mccreesh.1@research.gla.ac.uk