A Backtracking-Based Algorithm for Computing Hypertree-Decompositions

Georg Gottlob and Marko Samer
Draft

(Part 1)

Motivation: Solving Graphical Models

- 'Convert' reasoning problem to tree structure by decomposition:
  - Given a tree decomposition of width *w*, we can solve the reasoning problem in
    - time $O((r+m) \cdot deg \cdot k^{w+1})$
    - space $O(m \cdot k^{sep})$
  - Given a hypertree decomposition of width *hw*, we can solve the reasoning problem (absorbing rel. to 0) in
    - time $O(m \cdot deg \cdot hw \cdot log(t) \cdot t^{hw})$
    - space $O(t^{hw})$

Motivation: Solving Graphical Models

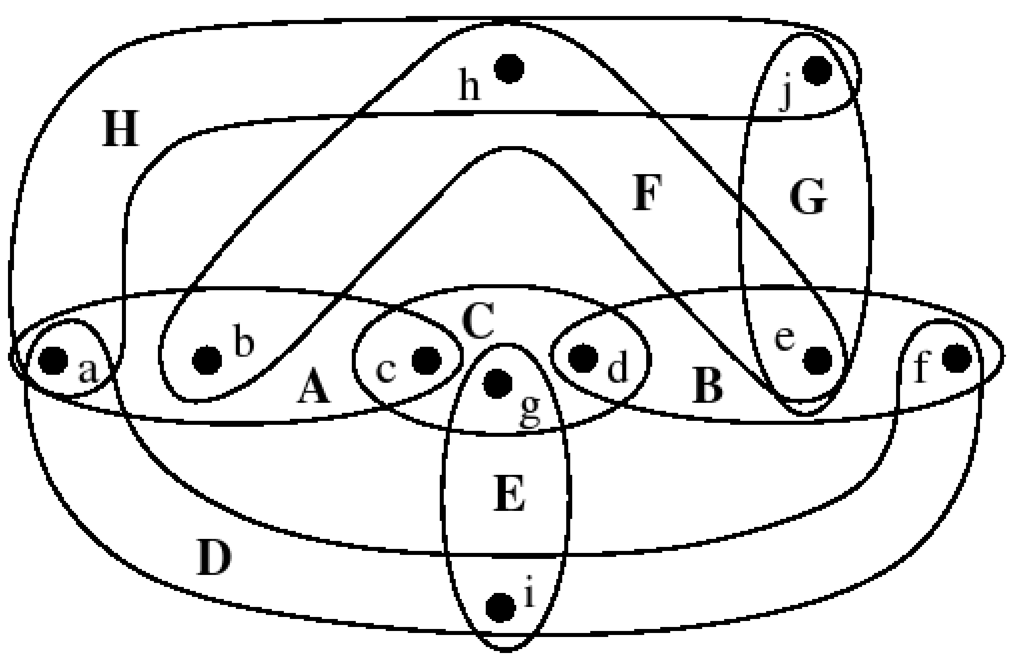
- 'Convert' reasoning problem to tree structure by decomposition:
  - Given a tree decomposition of width *w*, we can solve the reasoning problem in
    - time $O((r+m) \cdot deg \cdot k^{w+1})$
    - space $O(m \cdot k^{sep})$
  - Given a hypertree decomposition of width *hw*, we can solve the reasoning problem (absorbing rel. to 0) in
    - time $O(m \cdot deg \cdot hw \cdot log(t) \cdot t^{hw})$
    - space $O(t^{hw})$
- Question: How to compute hypertree decomposition?

Problem hardness

- Given a reasoning problem, finding a hypertree decomposition with minimal width is NP hard in general.
  - In their paper, the authors suggest an algorithm that, for a problem and given k , finds a hypertree decomposition of width at most k (if one exists) in polynomial time .
    - First step: Nondeterministic algorithm.
    - Second step: Introduce heuristic to achieve determinism.

Definitions

- View reasoning problem as its hypergraph *H = (V,E)*
  - Vertices *V* are the variables of the problem
  - Hyperedges *E* are the scopes of the problem's functions / relations, each one a subset of *V*.



Definitions (our way)

A tree decomposition of a reasoning problem with hypergraph $H = (V, E)$ is a triple (T, χ, ψ) where $T = (V_T, E_T)$ is a tree and $\chi\colon V_T \to 2^V$ and $\psi\colon V_T \to 2^E$ are labeling functions, satisfying the following:

1. For each hyperedge $X \in E$, there is exactly one vertex $v \in V_T$ such that $X \in \psi(v)$.
2. If $X \in \psi(v)$, then $X \subseteq \chi(v)$.
3. For each variable $x \in V$, the set $\{v \in V_T \mid x \in \chi(v)\}$ induces a connected subtree of T. This is also called the running intersection or the connectedness property.

The treewidh of a tree decomposition is $w = \max_{v \in V_T} |\chi(v)| - 1$. T is a hypertree decomposition if the following additional condition is satisfied:

4. For each $v \in V_T$: $\chi(v) \subseteq \bigcup \psi(v)$.

The hypertree width of a hypertree decomposition is then $hw = \max_{v \in V_T} |\psi(v)|$.

Definitions (their way)

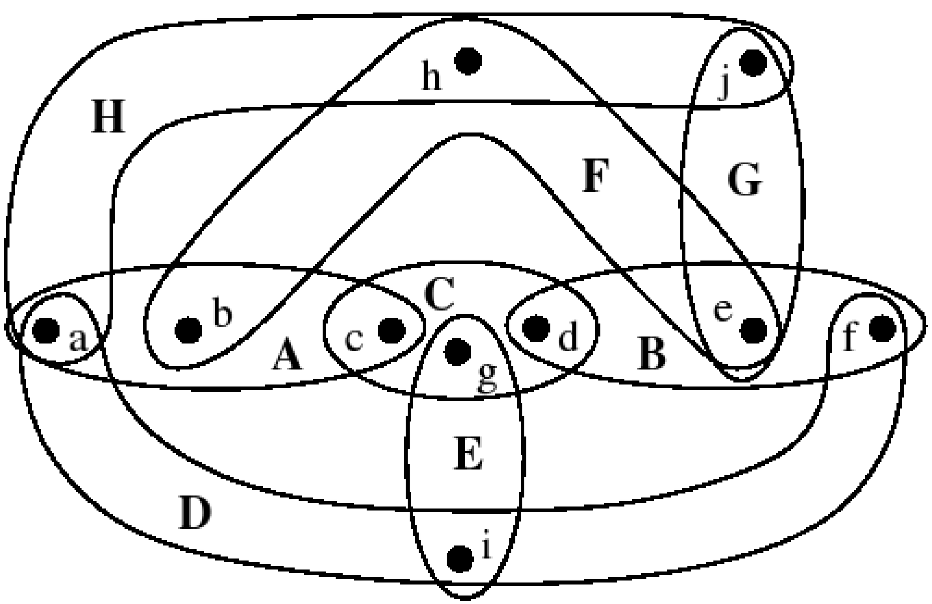
A tree decomposition of a reasoning problem with hypergraph $H = (V, E)$ is a triple (T, χ, ψ) where $T = (V_T, E_T)$ is a tree and $\chi\colon V_T \to 2^V$ and $\psi\colon V_T \to 2^E$ are labeling functions, satisfying the following:

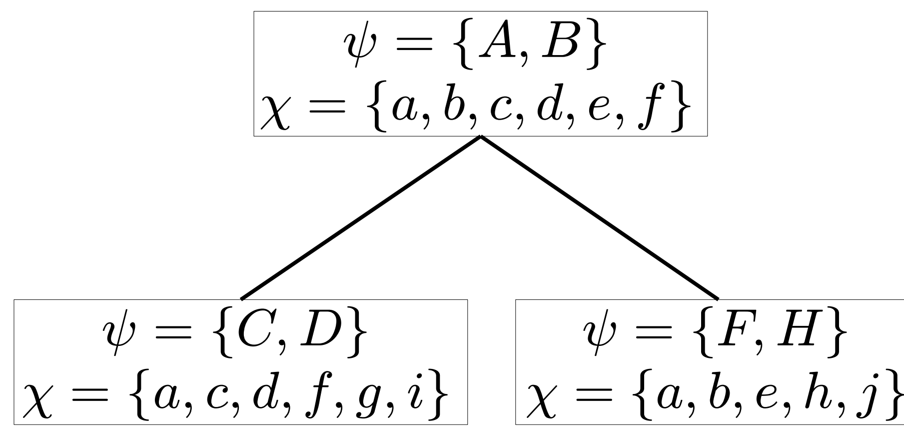
1. For each hyperedge $X \in E$, there is one vertex $v \in V_T$ such that $X \subseteq \chi(v)$.
2. For each variable $x \in V$, the set $\{v \in V_T \mid x \in \chi(v)\}$ induces a connected subtree of T.
3. For each $v \in V_T$: $\chi(v) \subseteq \bigcup \psi(v)$.
4. For each $v \in V_T$: $\bigcup \psi(v) \cap \chi(T_v) \subseteq \chi(v)$.

$\chi(T_v)$ here denotes all variables occuring in the nodes V'_T of the subtree rooted at v, formally $\bigcup_{v' \in V'_T} \chi(v')$.

Definitions

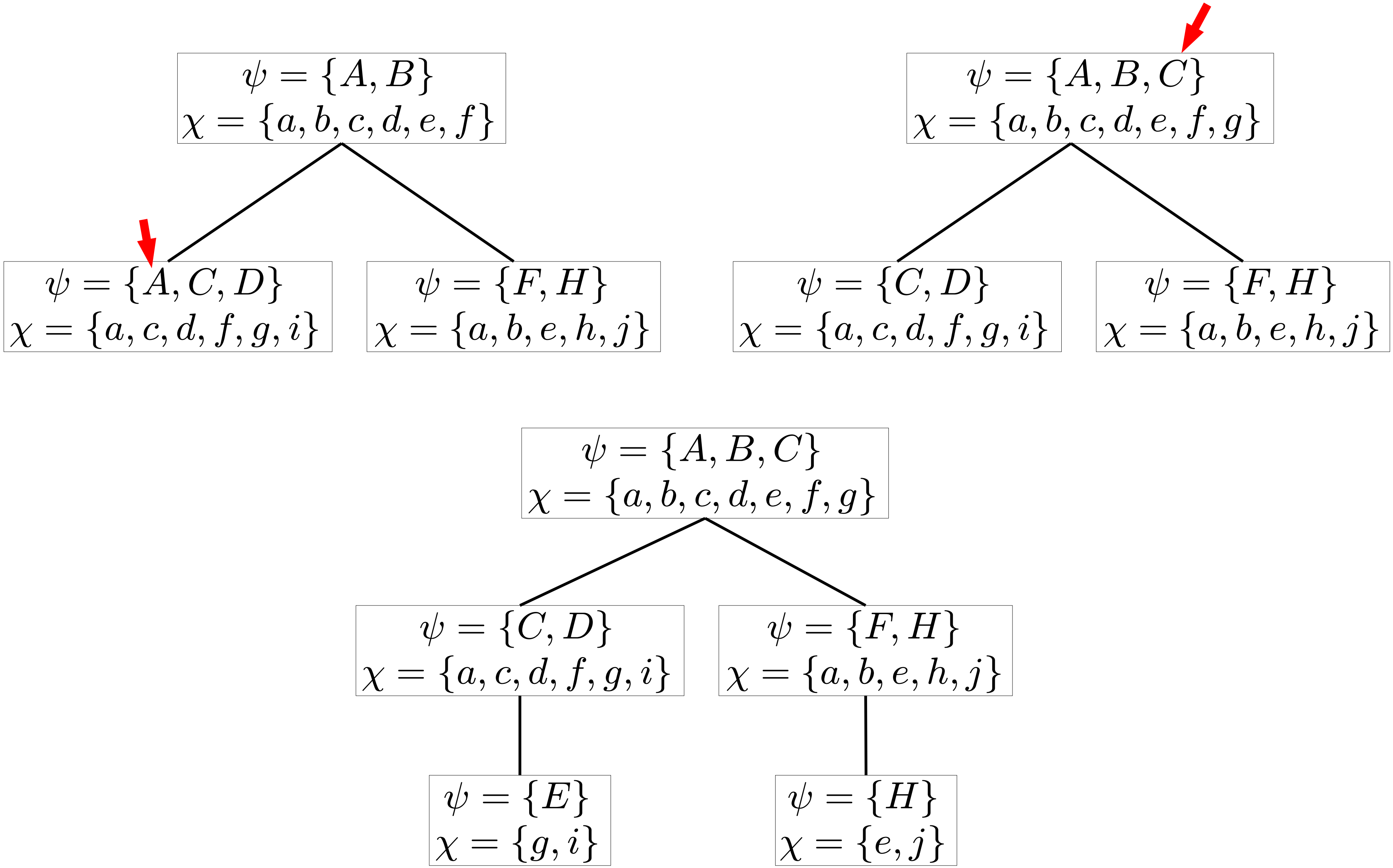
- Slight differences in the definitions
  - Dechter: “Each hyperedge assigned to *exactly one* cluster”.
  - Gottlob: “Hyperedges can be assigned to multiple clusters or none at all.”





E, G ?

Alternative valid decompositions



Algorithm *k*-decomp (1)

- Gottlob et. al. propose a nondeterministic algorithm for checking and finding a hypertree decomposition:

Algorithm 1 $k\text{-}decomp(HGraph)$

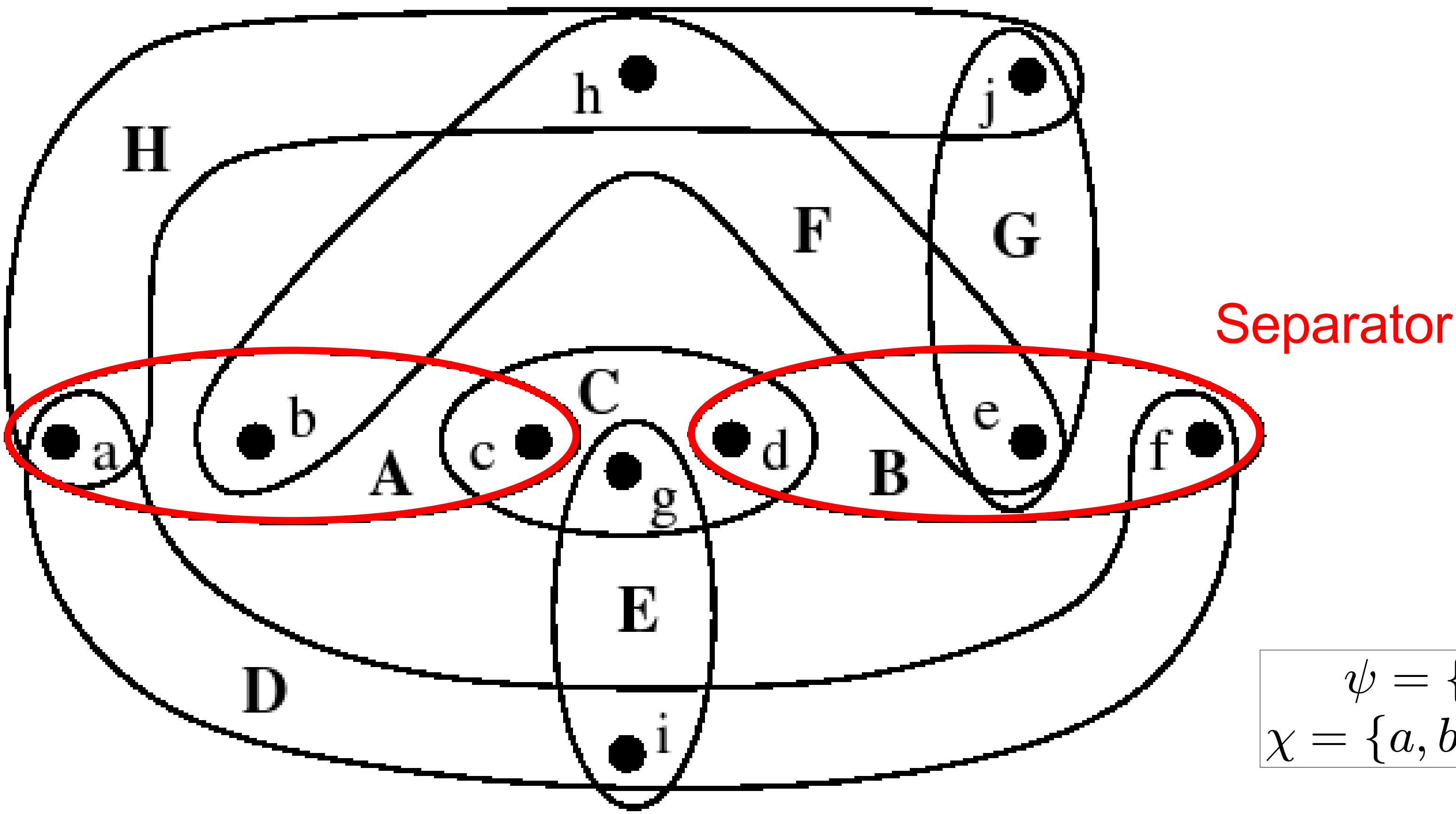
$HTree := k\text{-}decomposable(edges(HGraph), \emptyset);$
return $HTree;$

Algorithm *k*-decomp (2)

Algorithm 2 $k\text{-}decomposable(Edges, OldSep)$

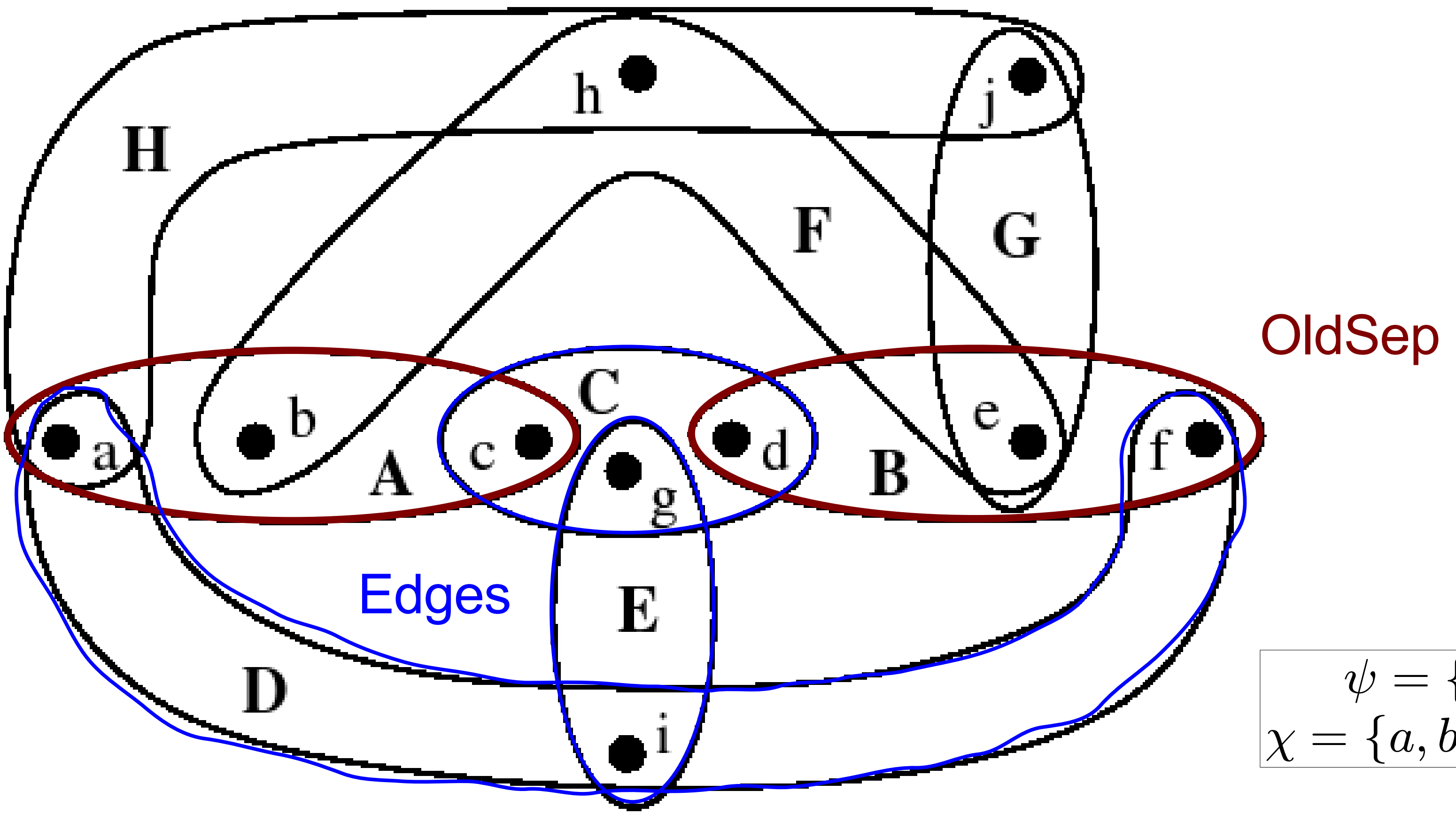
guess $Separator \subseteq edges(HGraph)$ **such that** $|Separator| \leq k$;
check that the following two conditions hold:
    $\bigcup Edges \cap \bigcup OldSep \subseteq \bigcup Separator$;
    $Separator \cap Edges \neq \emptyset$;
if one of these checks fails **then return** $NULL$;
$Components := separate(Edges, Separator)$;
$Subtrees := \emptyset$;
for each $Comp \in Components$ **do**
    $HTree := k\text{-}decomposable(Comp, Separator)$;
    if $HTree = NULL$ **then**
        return $NULL$;
    else
        $Subtrees := Subtrees \cup \{HTree\}$;
    endif
endfor
$Chi := (\bigcup Edges \cap \bigcup OldSep) \cup \bigcup (Separator \cap Edges)$;
$HTree := getHTNode(Separator, Chi, Subtrees)$;
return $HTree$;

Example



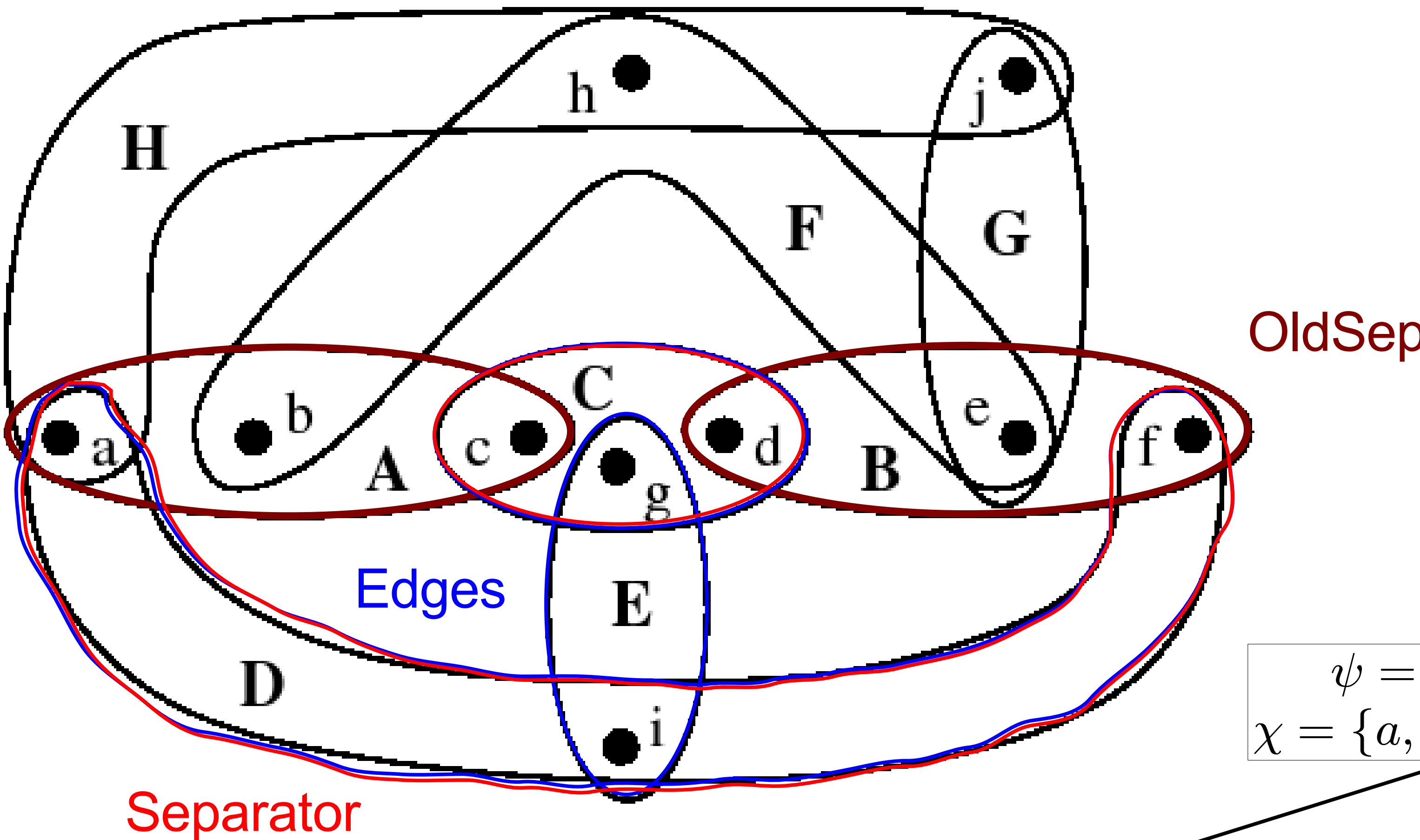
$$\psi = \{A, B\}$$
$$\chi = \{a, b, c, d, e, f\}$$

Example



$$\psi = \{A, B\}$$
$$\chi = \{a, b, c, d, e, f\}$$

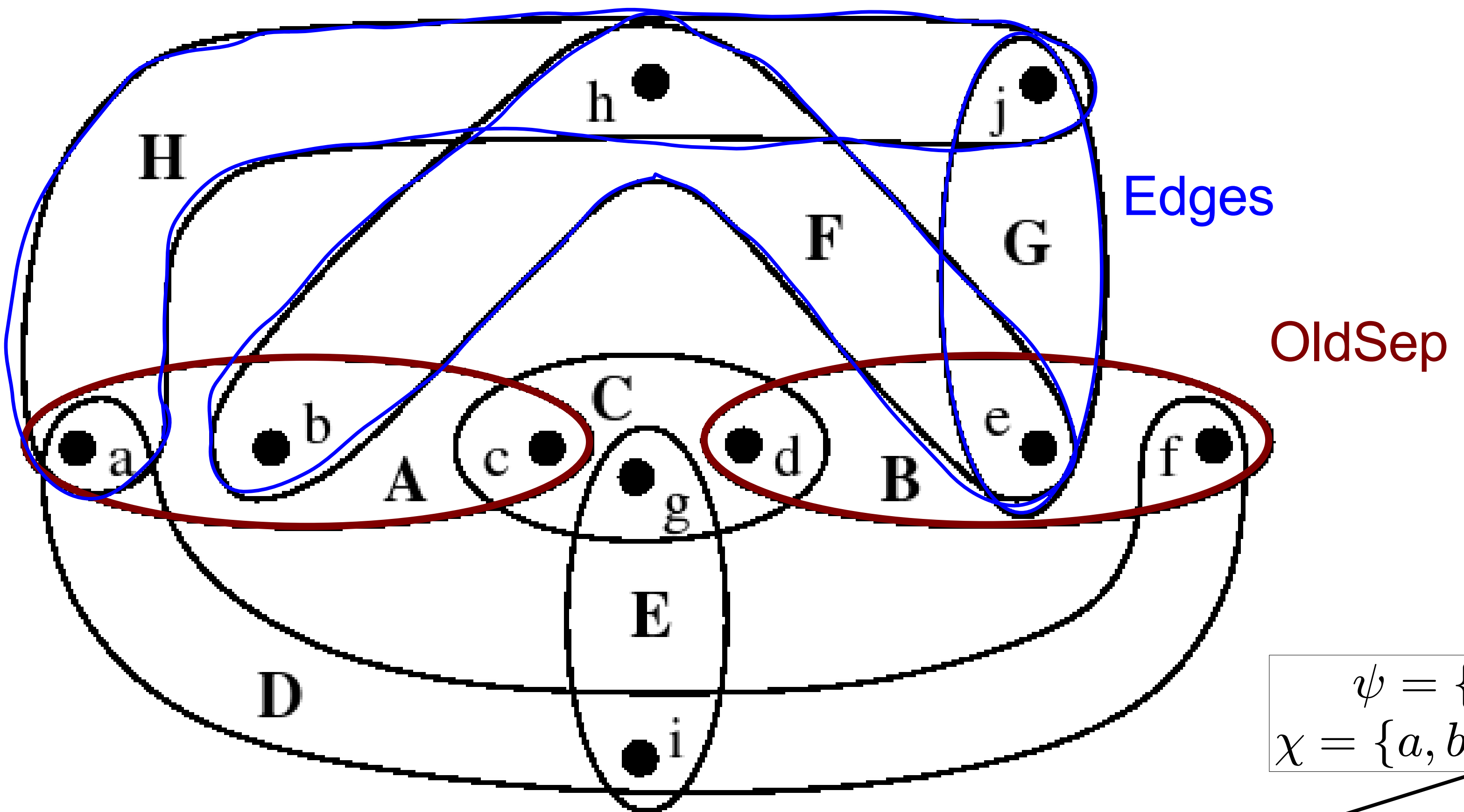
Example



$\psi = \{A, B\}$
$\chi = \{a, b, c, d, e, f\}$

$\psi = \{C, D\}$
$\chi = \{a, c, d, f, i, g\}$

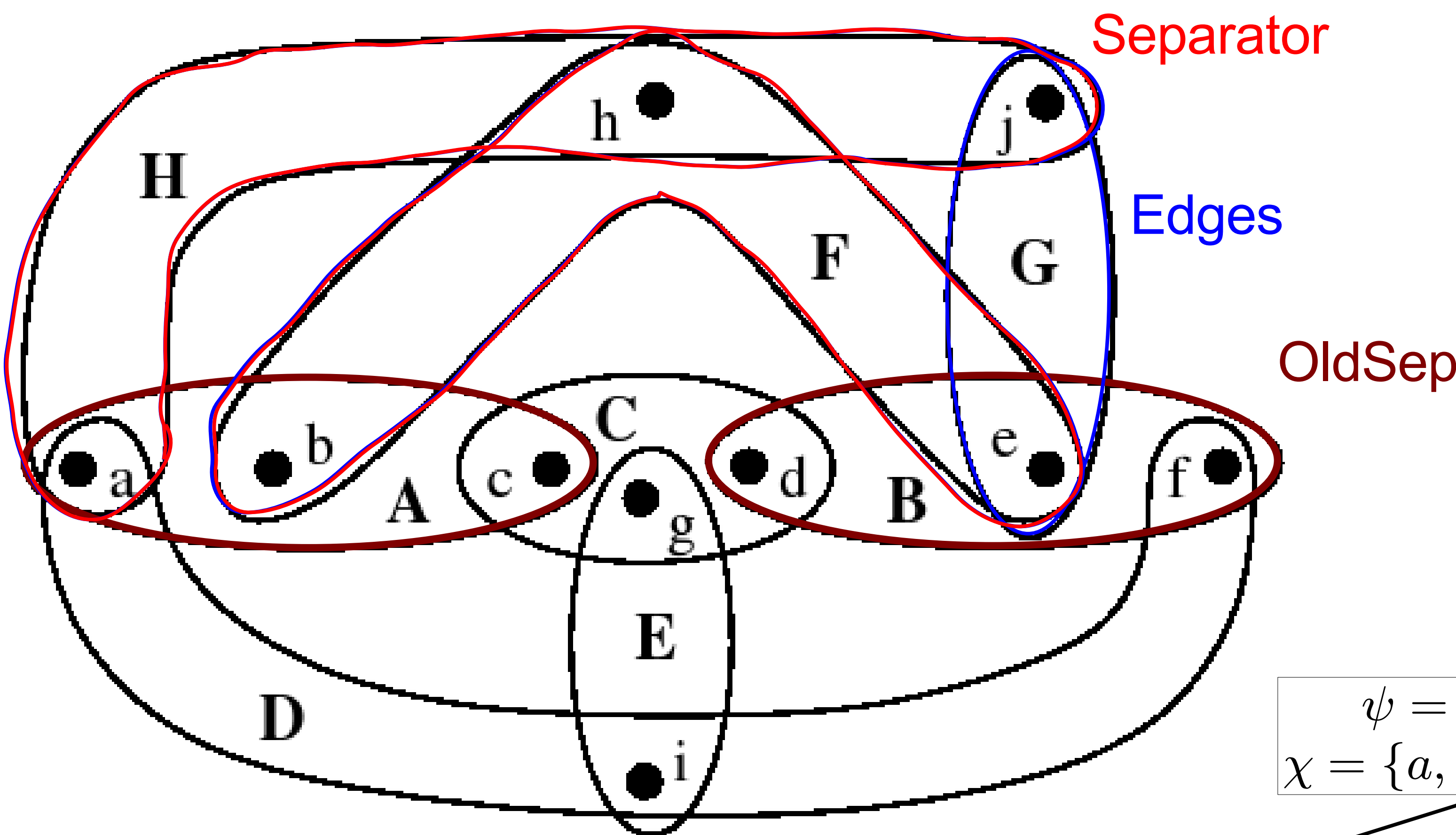
Example



$\psi = \{A, B\}$
$\chi = \{a, b, c, d, e, f\}$

$\psi = \{C, D\}$
$\chi = \{a, c, d, f, i, g\}$

Example



$$\psi = \{A, B\}$$
$$\chi = \{a, b, c, d, e, f\}$$

$$\psi = \{C, D\}$$
$$\chi = \{a, c, d, f, i, g\}$$

$$\psi = \{H, F\}$$
$$\chi = \{a, b, e, h, j\}$$

To be continued

- Nondeterminism:
  - Cannot be implemented, only theoretical interest:
    - Gottlob et al. show that problem of deciding whether a problem's hypertree width is bounded by k is in P.
- Next time:
  - Transform k-decomp into a deterministic algorithm with polynomial runtime:
    - Replace “guess and check” (lines 1-4) by backtracking-based search.
  - Benchmark results

A Backtracking-Based Algorithm for Computing Hypertree-Decompositions

Georg Gottlob and Marko Samer
Draft

(Part 2)

Algorithm det-*k*-decomp (1)

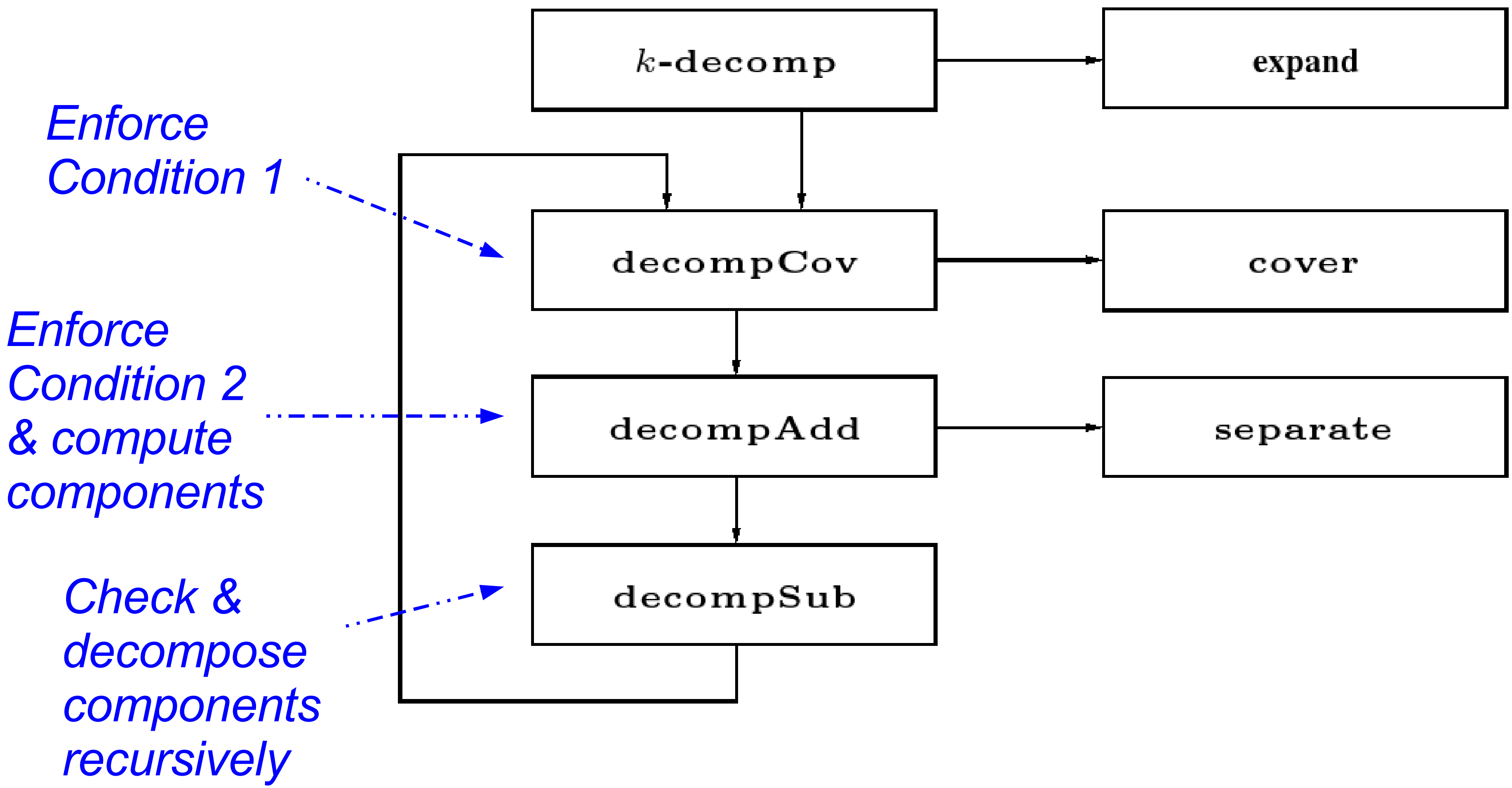
- Replace "guess and check":
  - Heuristic backtrack search, keeping track of failed and succeeded decompositions
  - Key: Find *Separator* that satisfies two conditions:
    1. $\bigcup Edges \cap \bigcup OldSep \subseteq \bigcup Separator$ => Connectivity
    2. $Separator \cap Edges \neq \emptyset$ => Monotonicity

Algorithm 3 $det\text{-}k\text{-}decomp(HGraph)$

```
FailSeps := ∅;
SuccSeps := ∅;
HTree := decompCov(edges(HGraph), ∅);
if HTree ≠ NULL then
    HTree := expand(HTree);
endif
return HTree;
```

Algorithm det-*k*-decomp (2)

- Algorithm outline:



Algorithm det-*k*-decomp (3)

- *decompCov* enforces condition 1:
  - $\bigcup Edges \cap \bigcup OldSep \subseteq \bigcup Separator$

Algorithm 4 $decompCov(Edges, Conn)$

```
if |Edges| ≤ k then
    HTree := getHTNode(Edges, ⋃Edges, ∅);
    return HTree;
endif
BoundEdges := {e ∈ edges(HGraph) | e ∩ Conn ≠ ∅};
for each CovSep ∈ cover(Conn, BoundEdges) do
    HTree := decompAdd(Edges, Conn, CovSep);
    if HTree ≠ NULL then
        return HTree;
    endif
endfor
return NULL;
```

Algorithm det-*k*-decomp (4)

- *decompAdd* enforces condition 2 and decomposes
  - condition 2: $Separator \cap Edges \neq \emptyset$

Algorithm 5 $decompAdd(Edges, Conn, CovSep)$

```
InCovSep := CovSep ∩ Edges;
if InCovSep ≠ ∅ or k − |CovSep| > 0 then
    if InCovSep = ∅ then AddSize := 1 else AddSize := 0 endif;
    for each AddSep ⊆ Edges s.t. |AddSep| = AddSize do
        Separator := CovSep ∪ AddSep;
        Components := separate(Edges, Separator);
        if ∀Comp ∈ Components. ⟨Separator, Comp⟩ ∉ FailSeps then
            Subtrees := decompSub(Components, Separator);
            if Subtrees ≠ ∅ then
                Chi := Conn ∪ ⋃(InCovSep ∪ AddSep);
                HTree := getHTNode(Separator, Chi, Subtrees);
                return HTree;
            endif
        endif
    endfor
endif
return NULL;
```

Algorithm det-*k*-decomp (5)

- *decompSub* recursively decomposes the components
  - checks for previous processing of components

Algorithm 6 *decompSub*(*Components*, *Separator*)

```
Subtrees := ∅;
for each Comp ∈ Components do
    ChildConn := ⋃ Comp ∩ ⋃ Separator;
    if ⟨Separator, Comp⟩ ∈ SuccSeps then
        HTree := getHTNode(Comp, ChildConn, ∅);
    else
        HTree := decompCov(Comp, ChildConn);
        if HTree = NULL then
            FailSeps := FailSeps ∪ {⟨Separator, Comp⟩};
            return ∅;
        else
            SuccSeps := SuccSeps ∪ {⟨Separator, Comp⟩};
        endif
    endif
    Subtrees := Subtrees ∪ {HTree};
endfor
return Subtrees;
```

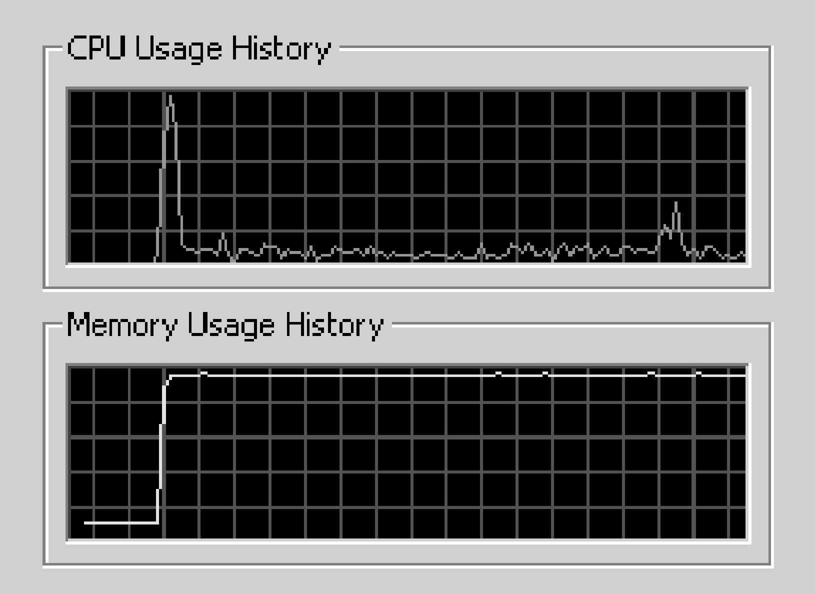
Complexity analysis

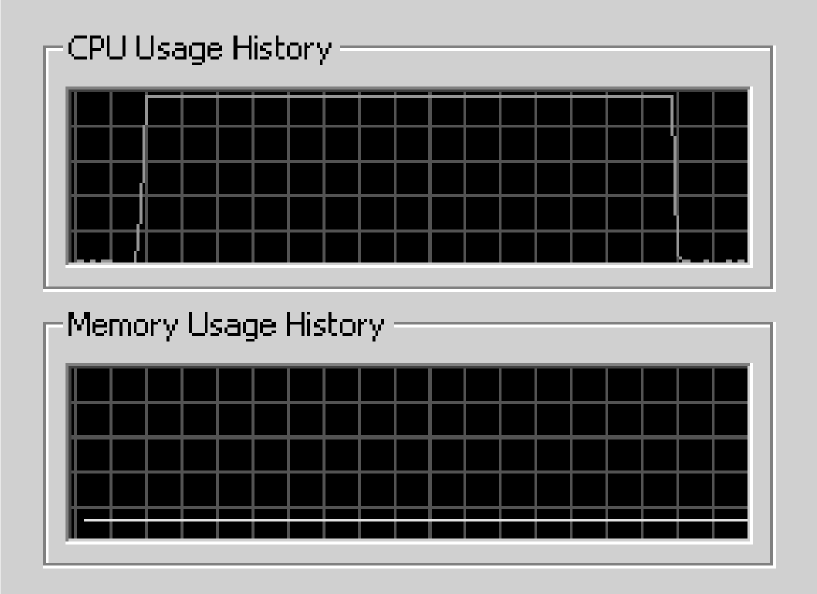
- Bounds:
  - Number of recursive calls:
    - Number of separators bounded by $\Psi = \sum_{i=1}^{k} \binom{n}{i} = \sum_{i=1}^{k} \frac{n!}{i!(n-i)!}$
    - At most *m* subcomponents each time.
    - Number of recursive calls thus bounded by $\mathcal{O}(\Psi\, m)$.
  - Each recursive call:
    - Loops in *decompCov* bounded by $\Phi = \sum_{i=1}^{k} \binom{\min(n,ck)}{i} = \sum_{i=1}^{k} \frac{\min(n,ck)!}{i!(\min(n,ck)-i)!}$
    - Loops in *decompAdd* bounded by *n* .
    - Loops in *decompSub* bounded by *m* .
    - Single recursive call therefore bounded bounded by $\mathcal{O}(\Phi n m)$.
- Total complexity bound:

$$\mathcal{O}(\Psi\,\Phi\,n\,m^2) = \mathcal{O}(n^{k+1}\,\min(n,ck)^k\,m^2)$$

Comparison

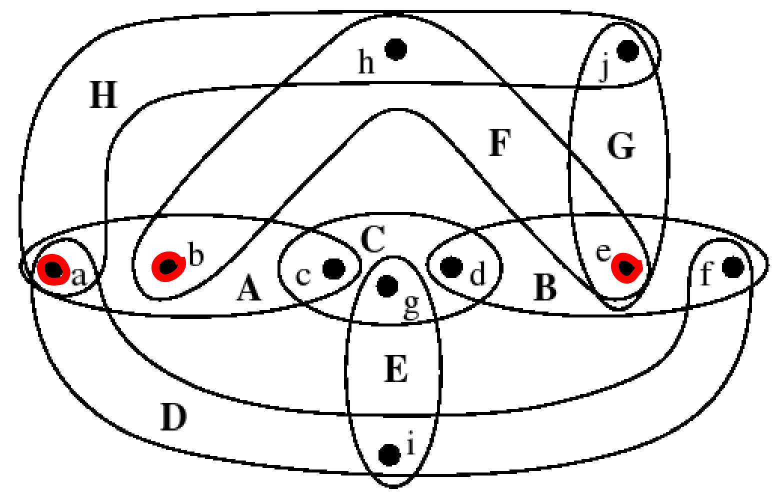
- Compare to algorithm opt-*k*-decomp:
  - Complexity $\mathcal{O}(n^{2k}\, m^2)$.
  - Often *ck* << *n*, hence det-*k*-decomp is $\mathcal{O}(n^{k+1}\,(ck)^k\, m^2)$.
  - Lower memory usage

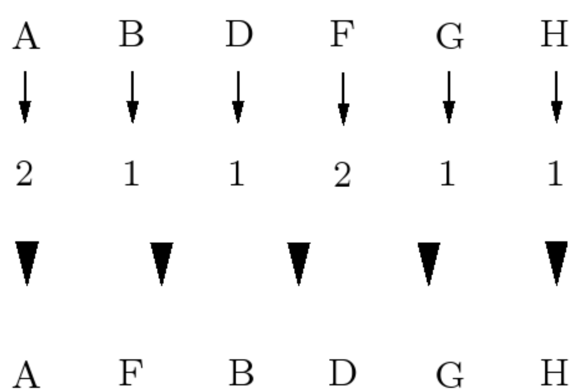


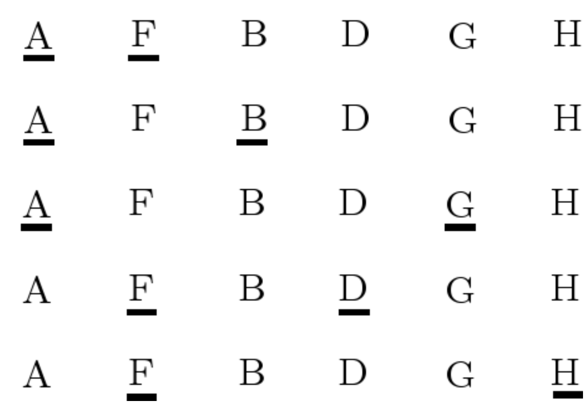


Heuristic for procedure *cover*

- Choosing *CovSep* candidates:
  - Assign weights to *BoundEdges:*
    - Number of vertices in Conn each edge contains
  - Order by decreasing weight
  - Greedily cover from first to last.



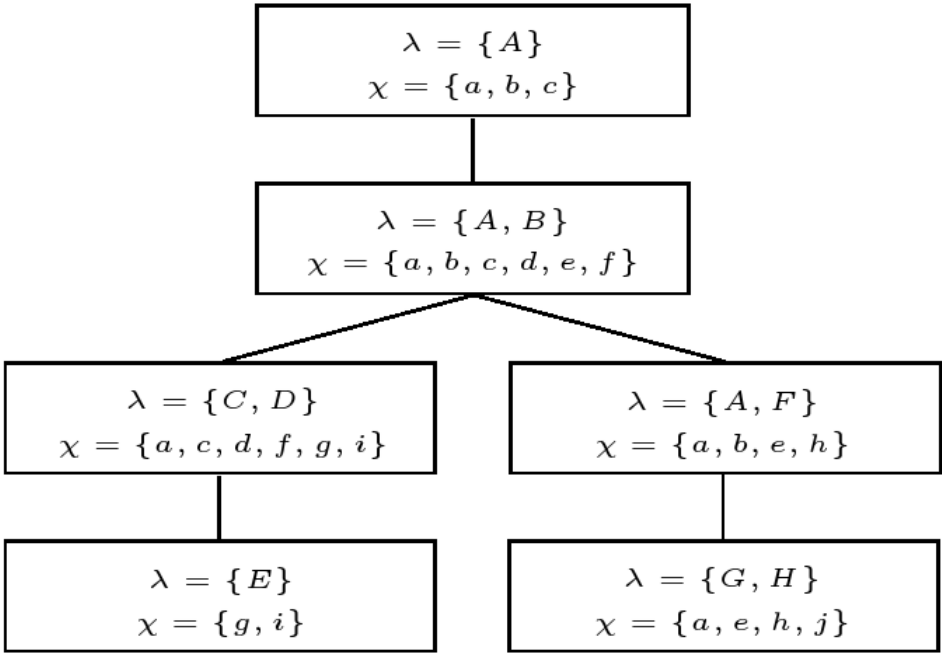




Example

- Run det-*k*-decomp on familiar example:

$$decompCov(\{A,B,C,D,E,F,G,H\},\emptyset)$$
$$decompAdd(\{A,B,C,D,E,F,G,H\},\emptyset,\emptyset)$$
$$decompSub(\{\{B,C,D,E,F,G,H\}\},\{A\})$$
$$decompCov(\{B,C,D,E,F,G,H\},\{a,b,c\})$$
$$decompAdd(\{B,C,D,E,F,G,H\},\{a,b,c\},\{A\})$$
$$decompSub(\{\{C,D,E\},\{F,G,H\}\},\{A,B\})$$
$$decompCov(\{C,D,E\},\{a,c,d,f\})$$
$$decompAdd(\{C,D,E\},\{a,c,d,f\},\{C,D\})$$
$$decompSub(\{\{E\}\},\{C,D\})$$
$$decompCov(\{E\},\{g,i\})$$
$$decompCov(\{F,G,H\},\{a,b,e\})$$
$$decompAdd(\{F,G,H\},\{a,b,e\},\{A,F\})$$
$$decompSub(\{\{G,H\}\},\{A,F\})$$
$$decompCov(\{G,H\},\{a,e,h\})$$



Experimental results

- Compare performance:
  - det-*k*-decomp
  - Bucket Elimination heuristics
  - opt-*k*-decomp
- Report smallest hypertree width obtained within 1 hour.

Results (1)

- Benchmarks from Daimler Chrysler (adder circuits etc.)

Instance *(Atoms / Variables)*	**Min**	**opt-k-decomp**		**BE**		**det-k-decomp**	
		Width	*Time*	*Width*	*Time*	*Width*	*Time*
adder_15 (76 / 106)	**2**	**2**	2	**2**	0	**2**	0
adder_25 (126 / 176)	**2**	**2**	20	**2**	0	**2**	0
adder_50 (251 / 351)	**2**	—	—	**2**	0	**2**	0
adder_75 (376 / 526)	**2**	—	—	**2**	0	**2**	0
adder_99 (496 / 694)	**2**	—	—	**2**	1	**2**	0
bridge_15 (137 / 137)	**2**	**2**	9	**3**	0	**2**	0
bridge_25 (227 / 227)	**2**	**2**	69	**3**	0	**2**	0
bridge_50 (452 / 452)	**2**	**2**	1105	**3**	1	**2**	0
bridge_75 (677 / 677)	**2**	—	—	**3**	1	**2**	0
bridge_99 (893 / 893)	**2**	—	—	**3**	2	**2**	1
NewSystem1 (84 / 142)	**3**	—	—	**3**	0	**3**	0
NewSystem2 (200 / 345)	**3**	—	—	**4**	0	**3**	0
NewSystem3 (278 / 474)	—	—	—	**5**	1	**4**	0
NewSystem4 (418 / 718)	—	—	—	**5**	2	**4**	0
atv_partial_system (88 / 125)	**3**	—	—	**3**	0	**3**	0

Results (2)

- Hypergraphs extracted from 2D grids
  - hypertree width known from construction

Instance *(Atoms / Variables)*	Min	opt-k-decomp		BE		det-k-decomp	
		Width	*Time*	*Width*	*Time*	*Width*	*Time*
grid2d_10 (50 / 50)	**4**	—	—	**5**	0	**4**	0
grid2d_15 (112 / 113)	**6**	—	—	**8**	0	**6**	3
grid2d_20 (200 / 200)	**7**	—	—	**12**	0	**7**	3140
grid2d_25 (312 / 313)	**9**	—	—	**15**	3	**10**	2000
grid2d_30 (450 / 450)	**11**	—	—	**19**	7	**13**	1566
grid2d_35 (612 / 613)	**12**	—	—	**23**	15	**15**	1905
grid2d_40 (800 / 800)	**14**	—	—	**26**	28	**17**	2530
grid2d_45 (1012 / 1013)	**16**	—	—	**31**	51	**21**	2606
grid2d_50 (1250 / 1250)	**17**	—	—	**33**	86	**24**	2786
grid2d_60 (1800 / 1800)	**21**	—	—	**41**	204	**31**	2984
grid2d_70 (2450 / 2450)	**24**	—	—	**48**	474	**42**	2161
grid2d_75 (2812 / 2813)	**26**	—	—	**48**	631	**45**	2881

Results (3)

- ISCAS89
  - extracted from circuits
  - examples from practice

Instance *(Atoms / Variables)*	Min	opt-k-decomp		BE		det-k-decomp	
		Width	*Time*	*Width*	*Time*	*Width*	*Time*
s27 (13 / 17)	**2**	**2**	0	**2**	0	**2**	0
s208 (104 / 115)	$\geq$ **3**	—	—	**7**	0	**6**	0
s298 (133 / 139)	$\geq$ **3**	—	—	**5**	0	**4**	462
s344 (175 / 184)	$\geq$ **3**	—	—	**7**	0	**5**	730
s349 (176 / 185)	$\geq$ **3**	—	—	**7**	0	**5**	4
s382 (179 / 182)	$\geq$ **3**	—	—	**5**	0	**5**	722
s386 (165 / 172)	—	—	—	**8**	1	**7**	1824
s400 (183 / 186)	$\geq$ **3**	—	—	**6**	0	**5**	273
s420 (212 / 231)	$\geq$ **3**	—	—	**9**	0	**8**	454
s444 (202 / 205)	$\geq$ **3**	—	—	**6**	0	**5**	385
s510 (217 / 236)	$\geq$ **3**	—	—	**23**	1	**20**	2082
s526 (214 / 217)	$\geq$ **3**	—	—	**8**	1	**7**	1715
s641 (398 / 433)	—	—	—	**7**	1	**7**	1611
s713 (412 / 447)	—	—	—	**7**	1	**7**	1800
s820 (294 / 312)	$\geq$ **3**	—	—	**13**	3	**12**	2846
s832 (292 / 310)	$\geq$ **3**	—	—	**12**	3	**11**	2575
s838 (422 / 457)	$\geq$ **3**	—	—	**16**	1	**15**	2046
s953 (424 / 440)	$\geq$ **3**	—	—	**40**	8	—	—
s1196 (547 / 561)	—	—	—	**35**	11	—	—
s1238 (526 / 540)	—	—	—	**34**	13	—	—
s1423 (731 / 748)	—	—	—	**18**	3	—	—
s1488 (659 / 667)	—	—	—	**23**	18	—	—
s1494 (653 / 661)	—	—	—	**24**	19	—	—
s5378 (2958 / 2993)	—	—	—	**85**	141	—	—

Conclusion

- Performance:
  - Significantly outperforms opt-*k*-decomp.
    - Time- and memory-wise
  - Results better than or comparable to BE heuristic.
    - Only when time is not the issue and graphs are “not too large and complicated”