YewPar: Skeletons for Exact Combinatorial Search (Operational Semantics Supplement)

Blair Archibald
University of Glasgow
Blair.Archibald@glasgow.ac.uk

Patrick Maier
University of Stirling
Patrick.Maier@stir.ac.uk

Robert Stewart
Heriot-Watt University
R.Stewart@hw.ac.uk

Phil Trinder
University of Glasgow
Phil.Trinder@glasgow.ac.uk

Abstract

This note presents an expanded version of Section 3 of the paper *YewPar: Skeletons for Exact Combinatorial Search* [2]. It is intended to supplement that paper with a more detailed account of the operational semantics, including detailed proof sketches and examples that were omitted for lack of space.

3 Formalising Parallel Tree Search

The dynamic behaviour of the YewPar framework is modeled by a novel operational semantics formalising multi-threaded backtracking search. The semantics presented here generalises the semantics published in Archibald's thesis [1] by classifying search types in terms of the monoids they use to accumulate information, and by providing correctness proofs.

Section 3.1 specifies search tree generation and traversal. Section 3.2 introduces the monoids used to accumulate information in enumeration, optimisation and decision searches, and Sections 3.3ff. define a parallel operational semantics for multi-threaded backtracking search, including pruning (Section 3.5) and work generation (Section 3.6). Section 3.7 proves correctness of the semantics, i.e. search always terminates and returns a correct/optimal result.

3.1 Trees

To represent search trees, we formalise trees and tree traversals. Let X be a non-empty alphabet. X^* denotes the set of finite words over X, and $\preceq$ denotes the prefix order on X^*.

A *tree* T is a non-empty finite prefix-closed subset of X^*. We call words $w \in T$ *nodes* of T. A node w is a *parent* of a node u, and u is a *child* of w, if there is $a \in X$ such that $u = wa$. Nodes u and v are *siblings* if they share the same parent. The *root* of T is the empty word ϵ, and the *depth* of a node w in T is the length of word w, denoted by $|w|$.

We assume that a tree T is *ordered*, that is, equipped with a partial order $\lessdot_T$ that linearly orders siblings and does not order non-siblings; we call $\lessdot_T$ a *sibling order.* (A sibling order is a disjoint union of total orders, each of which linearly orders the children of one node.) We define the *traversal order* $\ll_T$ for T as a linear extension of both the prefix order $\prec$ and the sibling order $\lessdot_T$ as follows; Proposition 3.1 shows that this definition is well-formed.

$$u \ll_T v \ \text{ iff } \begin{cases} u \prec v \text{ or} \\ \exists\, w, u', v' \in X^* \ \exists\, a, b \in X \text{ such that} \\ \quad u = wau' \text{ and } v = wbv' \text{ and } wa \lessdot_T wb \end{cases}$$

We will omit the subscript if the tree T is obvious from the context. Traversing T in $\ll$ order amounts to a depth-first traversal that visits the children of each node in sibling order.

Proposition 3.1. *$\ll$ is a linear extension of both $\prec$ and $\lessdot$.*

Proof sketch. We have to show that $\ll$ is a strict linear order that extends both $\prec$ and $\lessdot$. Irreflexivity and totality of $\ll$ are obvious from the definition. It is also obvious that $\ll$ extends (i.e. is a superset of) both $\prec$ and $\lessdot$ since the relations $\prec$ and $\lessdot$ are disjoint.

It remains to prove transitivity. Assume $u \ll v \ll w$. We have to show $u \ll w$. According to the definition of $\ll$ there are four cases to consider.

1. Assume that $u \prec v$ and $v \prec w$.
   By transitivity of $\prec$, we have $u \prec w$, that is $u \ll w$.
2. Assume there are $x, u', v' \in X^*$ and $a, b \in X$ such that $u = xau'$ and $v = xbv'$ and $xa \lessdot xb$. Assume that $v \prec w$, so there is $w' \in X^*$ such that $w = vw'$.
   Hence $u = xau'$ and $w = vw' = xbv'w'$ and $xa \lessdot xb$, that is $u \ll w$.
3. Assume that $u \prec v$, so $v = uav'$ for some $a \in X$ and $v' \in X^*$. Assume there are $y, v'', w'' \in X^*$ and $c, d \in X$ such that $v = ycv''$ and $w = ydw''$ and $yc \lessdot yd$. There are two cases.
   - Case $|u| \leq |y|$: Then $u \preceq y$ because $v = uav' = ycv''$. Hence $u \preceq y \prec w$, that is $u \ll w$.
   - Case $|u| > |y|$: Because $v = uav' = ycv''$ we have $u = ycu'$ for some $u' \in X^*$. Hence $u = ycu'$ and $w = ydw''$ and $yc \lessdot yd$, that is $u \ll w$.
4. Assume there are $x, u', v' \in X^*$ and $a, b \in X$ such that $u = xau'$ and $v = xbv'$ and $xa \lessdot xb$. Assume there are $y, v'', w'' \in X^*$ and $c, d \in X$ such that $v = ycv''$ and $w = ydw''$ and $yc \lessdot yd$. There are three cases.
   - Case $|x| = |y|$: Then $x = y$ and $b = c$ because $v = xbv' = ycv''$. Hence we have $u = xau'$ and

$w = ydw'' = xdw''$ and $xa \lessdot xb = yc \lessdot yd = xd$, which implies $xa \lessdot xd$ by transitivity of $\lessdot$. Thus $u \ll w$.

- Case $|x| < |y|$: Then $y = xby'$ for some $y' \in X^*$ because $v = xbv' = ycv''$. Hence we have $u = xau'$ and $w = ydw'' = xby'dw''$ and $xa \lessdot xb$, that is $u \ll w$.
- Case $|x| > |y|$: Then $x = ycx'$ for some $x' \in X^*$ because $v = xbv' = ycv''$. Hence we have $u = xau' = ycx'au'$ and $w = ydw''$ and $yc \lessdot yd$, that is $u \ll w$. □

A subset S of X^* is a *subtree* if S has a least element w.r.t. the prefix order, the *root* u, and is prefix-closed above the root, i.e. whenever $u \preceq v \preceq w$ and $w \in S$ then $v \in S$; we call S a subtree *rooted at* u. If a subtree S is a subset of a tree T then S inherits T's sibling order $\lessdot_T$ and traversal order $\ll_T$. Note that every tree is a subtree but not every subtree is a tree (because the root of a tree must be ϵ).

Let S be a subtree rooted at u, and let $v \in S$. We define $children(S, v) = \{va \mid a \in X \text{ and } va \in S\}$ as the set of nodes in S that are children of v. We define $subtree(S, v)$, the subtree of S rooted at v as $subtree(S, v) = \{w \mid w \in S \text{ and } v \preceq w\}$. If $v \neq u$ then the complement $S \setminus subtree(S, v)$ is a subtree rooted at u. Note that $S \setminus subtree(S, v)$ is strictly contained in S because $v \notin S \setminus subtree(S, v)$.

We define $succ(S, v) = \{w \mid w \in S \text{ and } v \ll w\}$ to be the set of nodes in S that follow v in traversal order. We define $next(S, v) = \min_{\ll} succ(S, v)$ as the node in S that immediately follows v in traversal order; we write $next(S, v) = \bot$ if no such node exists, i.e. if $succ(S, v) = \emptyset$.

We define $lowest(S, v)$ to be the subset of nodes in $succ(S, v)$ at minimum depth, that is closest to the root of S; formally:

$$lowest(S, v) = \{w \in succ(S, v) \mid \forall x \in succ(S, v) : |x| \geq |w|\}$$

We define $nextLowest(S, v) = \min_{\ll} lowest(S, v)$ as the "first" (in traversal order) of the minimum depth nodes in $succ(S, v)$; we write $nextLowest(S, v) = \bot$ if there is no such node, i.e. if $succ(S, v) = \emptyset$.

Tree generators. For the purpose of formalising semantics we assume a fully materialised search tree. (In reality, the search tree is not materialised but generated on demand.) We call a function $g : X^* \to X^*$ an *ordered tree generator* if all images of g are *isograms*, i.e. have no repeating letters. We define T_g, *the* tree generated by such a g as the smallest subset of X^* that contains ϵ and is closed under g, that is, if $u \in T_g$ and $g(u) = a_1 \dots a_n$, $a_i \in X$, then all $ua_i \in T_g$. (Clearly, T_g is a tree iff it is finite.) We equip T_g with *the* sibling order $\lessdot_{T_g}$ induced by g, defined as $ua_i \lessdot_{T_g} ua_j$ iff $g(u) = a_1 \dots a_n$ and $i < j$. This defines a total order on siblings because images of g are isograms.

3.2 Search types

YewPar supports enumeration, optimisation and decision searches. All three search types can be characterised in terms of a *commutative monoid M* for accumulating information and an *objective function h* for mapping search tree nodes into the monoid.

Enumeration. search traverses the entire search tree and gathers information by summing up the value of the objective function. Formally, such a search is defined by a commutative monoid $\langle M, +, 0\rangle$ and an objective function $h : X^* \to M$, and searching an initial tree S_0 amounts to computing the sum $\sum\{h(v) \mid v \in S_0\}$.

Examples. A simple enumeration search example counts the nodes of a search tree. The monoid M is the natural numbers with addition, and the objective function h is the constant function $h(v) = 1$. Counting the nodes at a given depth d requires the same monoid M but a different objective function h, where $h(v) = 1$ if the depth of v is d, i.e. $|v| = d$, and $h(v) = 0$ otherwise.

The monoid for collecting the set of all nodes of a tree is $\langle X^*, \cup, \emptyset\rangle$ and the objective function h maps nodes to singleton sets $h(v) = \{v\}$. In this case, the sum $\sum\{h(v) \mid v \in S\}$ equals S.

Optimisation. search computes the maximal value of the objective function while traversing the search tree. To this end, the commutative monoid $\langle M, +, 0\rangle$ must also be a total order $\langle M, \sqsubseteq\rangle$ with least element 0. That is, + must be idempotent (so $\langle M, +, 0\rangle$ is a semilattice) and the induced order $\sqsubseteq$ must be total (so the operations + and max coincide). Consequently the sum $\sum\{h(v) \mid v \in S_0\}$ (computed by enumeration search) is the same as $\max\{h(v) \mid v \in S_0\}$.

Thus, optimisation search can be reduced to enumeration. However, enumeration accumulates the maximal value of the objective function but cannot provide a witness for that value. Computing a witness requires the tracking of incumbents, and also provides opportunities for pruning the search tree (Section 3.5). Note that there may be many witnesses; optimisation search may pick one nondeterministically.

Example. A simple optimisation search computes the depth of a tree. The monoid M is the natural numbers with maximum, which induces the usual total order, and the objective function h maps each node to its depth, i.e. $h(v) = |v|$.

Decision. search, like optimisation search, computes the maximal value of the objective function while traversing the search tree. (Like optimisation search, decision search typically returns a nondeterministically chosen node witnessing that maximum.) However, decision search requires the total order $\langle M, \sqsubseteq\rangle$ to be bounded, and it will short-circuit and stop as soon as the objective function reaches the greatest element.

Example. A simple decision search decides whether a tree is at least k levels deep. The bounded total order is the set

$\{0, \ldots, k\}$ with the usual order, i.e. k is the greatest element. The objective function h maps each node to its depth, cut off at level k, i.e. $h(v) = \min\{|v|, k\}$.

3.3 Configurations

The semantics captures the current state of a parallel search in a *configuration* of the form $\langle \sigma, \mathit{Tasks}, \theta_1, \ldots, \theta_n \rangle$, where $n \geq 1$ is the fixed number of parallel threads. The components of a configuration are:

- θ_i is the state of the i^{th} *thread*. It is either $\perp$ to denote an idle thread, or $\langle S, v \rangle^k$ to denote an active thread that is currently executing task S, that is, searching subtree S in traversal order. The node currently being explored is v, and the superscript k records how often the search of S has backtracked; we may omit k when counting backtracks does not matter.
- *Tasks* is a queue of pending *tasks*, that is, subtrees yet to be searched. We use list notation, so [] is an empty queue, $[S]$ is a singleton queue, $S{:}\mathit{Tasks}$ is a queue with S at the head, and $\mathit{Tasks}{:}S$ is a queue with S at the tail.
- σ is the current *global knowledge*, which is either of the form $\langle x \rangle$ or $\{u\}$, depending on the search type (Section 3.2). For enumeration searches, the *accumulator* $\langle x \rangle$ is an element of a commutative monoid that sums up the current knowledge. For optimisation and decision searches, the *incumbent* $\{u\}$ is a search tree node that currently maximises the objective function h.

Search begins with all threads idle, a singleton task queue and global knowledge being either the root node incumbent or the zero accumulator. That is, an initial configuration takes the form $\langle \sigma_0, [S_0], \perp, \ldots, \perp \rangle$, where S_0 is the entire search tree and σ_0 is of the form $\{\epsilon\}$ or $\langle 0 \rangle$. Note that all tasks in later configurations (Section 3.4) are subtrees of S_0 and thus inherit the traversal order of S_0.

Search ends when the task queue is empty and all threads idle, i.e. final configurations take the form $\langle \sigma, [], \perp, \ldots, \perp \rangle$, where σ is a search result.

3.4 Reduction rules

Figure 2 lists the reduction rules of the multi-threaded semantics. The rules are divided into four categories and define reduction relations $\rightarrow_i^T$, $\rightarrow_i^N$, $\rightarrow_i^P$ and $\rightarrow_i^S$ for tree traversal, node processing, pruning and spawning, respectively. The subscript i indicates the active thread performing a reduction step. The per-thread and overall reduction relations $\rightarrow_i$ and $\rightarrow$ are defined as follows.

$$\begin{aligned} \rightarrow_i &= (\rightarrow_i^T \circ \rightarrow_i^N) \cup \rightarrow_i^P \cup \rightarrow_i^S \\ \rightarrow &= \rightarrow_1 \cup \cdots \cup \rightarrow_n \end{aligned}$$

Every $\rightarrow$ reduction is a per-thread reduction for some thread i, which is either a spawn reduction, a prune reduction, or a traversal reduction followed immediately by a node processing reduction.

Prune and spawn reduction rules are explained in subsequent sections. The traversal rules encode standard backtracking, searching a subtree S in traversal order, starting at the root of S (schedule), expanding the current branch (expand), backtracking to another branch (backtrack), and terminating once S is explored (terminate). The search type determines which node processing rules are applicable. Enumeration searches accumulate the value of the objective function using the monoid addition + (accumulate). Optimisation and decision searches update the incumbent after comparing its objective value to the current node using the total order $\sqsubseteq$ (strengthen/skip). The (noop) rule prevents node processing getting stuck after (terminate).

We observe that $\rightarrow$ reductions do not get stuck except on final configurations because every non-final configuration can make progress with at least one traversal rule followed by a node processing rule. In particular, a sequence of reductions starting with an initial configuration and using only traversal and node processing rules (thereby corresponding to a single-threaded execution without pruning) cannot get stuck. Moreover, as such a reduction sequence explores a finite search tree node by node in traversal order, the sequence will eventually terminate in a final configuration.

3.5 Pruning

Optimisation and decision searches admit *pruning* the search tree, i.e. removing subtrees that can never improve the current incumbent. Semantically, this is reflected by the (prune) rule. Note that the rule removes the subtree rooted at v but not v itself. This is a technicality owing to the fact that node v is needed to determine the next node in traversal order.

What to prune is decided by search-specific heuristics. For the purpose of the semantics, the heuristics are abstracted to a binary pruning relation $\triangleright$ on search tree nodes; write $u \triangleright v$ to express that *u justifies pruning v*. The pruning relation must satisfy the following admissibility conditions w.r.t. the objective function h and the total order $\langle M, \sqsubseteq \rangle$.

1. For all u and v, if $u \triangleright v$ then $h(u) \sqsupseteq h(v)$.
2. For all u', u and v, if $h(u') \sqsupseteq h(u)$ and $u \triangleright v$ then $u' \triangleright v$.
3. For all u, v and v', if $u \triangleright v$ and $v \leq v'$ then $u \triangleright v'$.

Condition 1 states correctness of pruning w.r.t. maximising the objective function: if u justifies pruning v then $h(u)$ dominates $h(v)$. Condition 2 allows strengthening of incumbents: if u justifies pruning v then any stronger incumbent u' also will. Condition 3 allows pruning entire subtrees: if u justifies pruning v then any descendent of v can also be pruned.

It is reasonable to also demand that no node justifies pruning itself, i.e. that $\triangleright$ is irreflexive. This leads to Proposition 3.2 observing that $\triangleright$ partially orders search tree nodes. In fact, the pruning relation $\triangleright$ can be thought of as a strict partial

Tree traversal rules

$$(\text{schedule}_i)\ \frac{v = \text{root of } S}{\langle \sigma,\ S{:}\mathit{Tasks},\ \ldots,\ \bot,\ \ldots \rangle \rightarrow_i^T \langle \sigma,\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle^0,\ \ldots \rangle}$$

$$(\text{expand}_i)\ \frac{v' = \mathit{next}(S, v) \qquad v' \neq \bot \qquad v \leq v'}{\langle \sigma,\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle^k,\ \ldots \rangle \rightarrow_i^T \langle \sigma,\ \mathit{Tasks},\ \ldots,\ \langle S, v'\rangle^k,\ \ldots \rangle}$$

$$(\text{backtrack}_i)\ \frac{v' = \mathit{next}(S, v) \qquad v' \neq \bot \qquad v \not\leq v'}{\langle \sigma,\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle^k,\ \ldots \rangle \rightarrow_i^T \langle \sigma,\ \mathit{Tasks},\ \ldots,\ \langle S, v'\rangle^{k+1},\ \ldots \rangle}$$

$$(\text{terminate}_i)\ \frac{\mathit{next}(S, v) = \bot}{\langle \sigma,\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle^k,\ \ldots \rangle \rightarrow_i^T \langle \sigma,\ \mathit{Tasks},\ \ldots,\ \bot,\ \ldots \rangle}$$

Node processing rules

$$(\text{accumulate}_i)\ \frac{}{\langle \langle x\rangle,\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle,\ \ldots \rangle \rightarrow_i^N \langle \langle x + h(v)\rangle,\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle,\ \ldots \rangle}$$

$$(\text{strengthen}_i)\ \frac{h(v) \sqsupset h(u)}{\langle \{u\},\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle,\ \ldots \rangle \rightarrow_i^N \langle \{v\},\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle,\ \ldots \rangle}$$

$$(\text{skip}_i)\ \frac{h(v) \sqsubseteq h(u)}{\langle \{u\},\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle,\ \ldots \rangle \rightarrow_i^N \langle \{u\},\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle,\ \ldots \rangle}$$

$$(\text{noop}_i)\ \frac{}{\langle \sigma,\ \mathit{Tasks},\ \ldots,\ \bot,\ \ldots \rangle \rightarrow_i^N \langle \sigma,\ \mathit{Tasks},\ \ldots,\ \bot,\ \ldots \rangle}$$

Prune rules

$$(\text{prune}_i)\ \frac{u \triangleright v \qquad S' = \mathit{subtree}(S, v) \setminus \{v\} \qquad S' \neq \emptyset}{\langle \{u\},\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle,\ \ldots \rangle \rightarrow_i^P \langle \{u\},\ \mathit{Tasks},\ \ldots,\ \langle S \setminus S', v\rangle,\ \ldots \rangle}$$

$$(\text{shortcircuit}_i)\ \frac{h(u) \text{ is greatest element}}{\langle \{u\},\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle,\ \ldots \rangle \rightarrow_i^P \langle \{u\},\ [],\ \bot,\ \ldots,\ \bot \rangle}$$

Spawn rules

$$(\text{spawn}_i)\ \frac{u \in S \qquad v \ll u \qquad S_u = \mathit{subtree}(S, u)}{\langle \sigma,\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle,\ \ldots \rangle \rightarrow_i^S \langle \sigma,\ \mathit{Tasks}{:}S_u,\ \ldots,\ \langle S \setminus S_u, v\rangle,\ \ldots \rangle}$$

$$(\text{spawn-depth}_i)\ \frac{|v| < d_{\mathit{cutoff}} \qquad \{u_1, \ldots, u_m\} = \mathit{children}(S, v) \neq \emptyset \qquad u_1 \ll \cdots \ll u_m \qquad S_1 = \mathit{subtree}(S, u_1)\ \ldots\ S_m = \mathit{subtree}(S, u_m)}{\langle \sigma,\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle,\ \ldots \rangle \rightarrow_i^S \langle \sigma,\ \mathit{Tasks}{:}S_1{:}\ldots{:}S_m,\ \ldots,\ \langle S \setminus S_1 \setminus \cdots \setminus S_m, v\rangle,\ \ldots \rangle}$$

$$(\text{spawn-budget}_i)\ \frac{k \geq k_{\mathit{budget}} \qquad \{u_1, \ldots, u_m\} = \mathit{lowest}(S, v) \neq \emptyset \qquad u_1 \ll \cdots \ll u_m \qquad S_1 = \mathit{subtree}(S, u_1)\ \ldots\ S_m = \mathit{subtree}(S, u_m)}{\langle \sigma,\ \mathit{Tasks},\ \ldots,\ \langle S, v\rangle^k,\ \ldots \rangle \rightarrow_i^S \langle \sigma,\ \mathit{Tasks}{:}S_1{:}\ldots{:}S_m,\ \ldots,\ \langle S \setminus S_1 \setminus \cdots \setminus S_m, v\rangle^0,\ \ldots \rangle}$$

$$(\text{spawn-stack}_i)\ \frac{u = \mathit{nextLowest}(S, v) \neq \bot \qquad S_u = \mathit{subtree}(S, u)}{\langle \sigma,\ [],\ \ldots,\ \langle S, v\rangle,\ \ldots \rangle \rightarrow_i^S \langle \sigma,\ [S_u],\ \ldots,\ \langle S \setminus S_u, v\rangle,\ \ldots \rangle}$$

Figure 2. Reduction rules of the operational semantics.

order that extends the pre-order induced by the objective function h.

Proposition 3.2. *Any irreflexive binary relation $\triangleright$ on X^* that satisfies the admissibility conditions is a strict partial order.*

Proof sketch. It suffices to show that any irreflexive $\triangleright$ is transitive. Assume $u \triangleright v \triangleright w$. By admissibility condition (1), the first part of the chain implies $h(u) \sqsupseteq h(v)$. This fact and the second part of the chain imply $u \triangleright w$ by admissibility condition (2). ☐

Example: Maximum clique search. Let $G = \langle V, E\rangle$ be an undirected graph. Given a vertex $u \in V$, we denote its set of adjacent vertices by $E(u)$.

A *maximum clique search* finds a largest set of pairwise adjacent vertices. We will define the search space for maximum clique search (be defining an ordered tree generator), the objective function, and a simple pruning relation.

A maximum clique search in G can be performed by searching a tree over the alphabet $X = V \times 2^V$ whose nodes w at depth k are words of the form $\langle v_1, V_1\rangle \ldots \langle v_k, V_k\rangle$ such that $\{v_1, \ldots, v_k\}$ is a k-clique in G and V_k is a set of candidate vertices each of which extends the k-clique to a $(k+1)$-clique. Formally, we define the ordered tree generator $g : X^* \longrightarrow X^*$ by $g(\langle v_1, V_1\rangle \ldots \langle v_k, V_k\rangle) = \langle u_1, U_1\rangle \ldots \langle u_n, U_n\rangle$ such that

- the u_i enumerate the candidate set V_k, and
- $U_i = (V_k \setminus \{u_1, \ldots, u_{i-1}\}) \cap E(u_i)$

where we assume $V_0 = V$ in order to define g for the empty word, i.e. when $k = 0$. Typically, the $\langle u_i, U_i\rangle$ are ordered such that the size of U_i decreases as i increases; this order supports a more efficient implementation of pruning, allowing to prune all nodes to "to the right" when pruning the current node is justified.

A maximum clique search is an optimisation search. Its total order $\langle M, \sqsubseteq\rangle$ are the natural numbers with their usual order, and its objective function h maps search tree nodes w to their depth $|w|$.

A simple heuristic prunes all nodes where the size of the current clique plus the size of the candidate set does not exceed the size of the incumbent clique.[1] Formally, $u \triangleright w$ if and only if $u \neq w$ and $|u| \geq |w| + \#V_k$ where $w = \langle v_1, V_1\rangle \ldots \langle v_k, V_k\rangle$ and the # operator returns the size of a set. The $\triangleright$ relation is clearly irreflexive and satisfies admissibility conditions 1 and 2. To establish condition 3 note that if $w' = \langle v_1, V_1\rangle \ldots \langle v_k, V_k\rangle\langle v_{k+1}, V_{k+1}\rangle$ is a child of w then $|w| + \#V_k = k + \#V_k \geq (k+1) + \#V_{k+1} = |w'| + \#V_{k+1}$ because $\#V_k > \#V_{k+1}$. Hence by induction, all descendants of w will satisfy the inequality defining the pruning relation $\triangleright$.

[1]Stronger pruning can be achieved by replacing the size of the candidate set with an upper bound estimate of the largest clique amongst the candidates; such estimates can be found by greedy graph colouring algorithms.

3.6 Spawning

The semantics includes a (spawn) rule to model space-splitting parallel search. A (spawn) reduction hives off some subtree S_u of the current thread $\langle S, v\rangle$ into a new task, which is added to the task queue (and later scheduled by another thread). S_u must be *unexplored*, i.e. its root u is visited after the current node v in traversal order $\ll$.

The search coordination of YewPar skeletons implement more complex space-splitting behaviours, selecting specific groups of subtrees to spawn as tasks, in a specific order. We model these coordination behaviours as derived spawn rules. Semantically, these rules are redundant — their reductions steps can be translated into (spawn) reduction sequences. However, these rules allow to faithfully model the coordination behaviour of a particular YewPar skeleton by restricting spawn reductions to use only the appropriate derived spawn rule.

The (spawn-depth) rule. models Depth-Bounded search coordination. The rule fires if the depth of the current node v is less than the d_{cutoff} parameter. The rule spawns all subtrees of S rooted at children of v and queues them in traversal order. In effect, the rule causes eager spawning of the top d_{cutoff} levels of the search tree, queued in heuristic search order.

The (spawn-budget) rule. models Budget search coordination. The rule fires if the backtrack counter k of the current thread exceeds the k_{budget} parameter. The rule spawns all unexplored subtrees of the current thread $\langle S, v\rangle$ at lowest depth, i.e. closest to the root of S. New tasks are queued in traversal order, and the backtrack counter of the current thread is reset to 0. In effect, the rule periodically generates new tasks from threads that contain significant amounts of work (since search has not completed within the backtrack budget). Spawning at lowest depth (i.e. closest to the root) prioritises large tasks, and queuing tasks in traversal order respects the heuristic search order.

The (spawn-stack) rule. models Stack-Stealing search coordination. It differs from the other rules in that it only fires when the task queue is empty, and only spawns one new task. That task is the first (in traversal order) of the unexplored lowest-depth subtrees of the current thread $\langle S, v\rangle$. The (spawn-stack) rule is designed to split the search space on demand, generating a single new task to be stolen by an idle thread, and prioritising large tasks close to the root.

The YewPar implementation of Stack-Stealing lets a thief steal directly from the victim. Semantically, this behaviour corresponds to a (spawn-stack$_i$) reduction followed by a (schedule$_j$) reduction, modelling idle thread j stealing from victim thread i, with the task queue acting as a transit buffer for the stolen task.

3.7 Correctness

The semantics is correct if every sequence of reductions starting at a given search tree computes the same sum or maximum of the objective function, independent of the particular reduction sequence. For enumeration searches, correctness amounts to termination and confluence of the reduction relation, but optimisation and decision searches may nondeterministically return any optimal witness, hence the reduction relation cannot be confluent in general. The following theorems formalise this statement of correctness.

Theorem 3.3. *Let S_0 be a search tree for an enumeration search. If $\langle\langle 0\rangle, [S_0], \bot, \ldots, \bot\rangle \rightarrow^* \langle\langle x\rangle, [], \bot, \ldots, \bot\rangle$ then $x = \sum\{h(v) \mid v \in S_0\}$.*

Proof sketch. The correctness argument boils down to showing that every node in S_0 is processed by the (accumulate) rule exactly once. In the absence of spawn reductions, this property follows from the strict alternation between traversal $\rightarrow_i^T$ and node processing $\rightarrow_i^N$ reductions, and from the fact that traversal visits all nodes of a subtree S in traversal order $\ll$. Thus, the (terminate) rule fires when all nodes in S have been processed exactly once.

To argue for the correctness of enumeration in the presence of spawning, we note that spawn reductions do not change the total set of nodes in the configuration. To be precise, we define a function $\mathcal{N}$ mapping configurations to the set of all their nodes.

$$\begin{aligned}
\mathcal{N}(\langle\sigma, \mathit{Tasks}, \theta_1, \ldots, \theta_n\rangle) &= \mathcal{N}(\mathit{Tasks}) \cup \mathcal{N}(\theta_1) \cup \cdots \cup \mathcal{N}(\theta_n) \\
\mathcal{N}(S{:}\mathit{Tasks}) &= S \cup \mathcal{N}(\mathit{Tasks}) \\
\mathcal{N}([]) &= \emptyset \\
\mathcal{N}(\langle S, v\rangle) &= S \\
\mathcal{N}(\bot) &= \emptyset
\end{aligned}$$

It is easy to verify that all rules except (terminate) keep $\mathcal{N}$ invariant, and all nodes removed by (terminate) have been processed exactly once by the (accumulate) rule as outlined above. □

Theorem 3.4. *Let S_0 be a search tree for an optimisation or decision search. If $\langle\{\epsilon\}, [S_0], \bot, \ldots, \bot\rangle \rightarrow^* \langle\{\hat{u}\}, [], \bot, \ldots, \bot\rangle$ then $h(\hat{u}) = \max\{h(v) \mid v \in S_0\}$.*

Proof sketch. Correctness is easy to show if the (shortcircuit) rule fires. It can only fire in the very last step and only if $h(\hat{u})$ is the greatest element. Hence $h(\hat{u}) = \max\{h(v) \mid v \in S_0\}$ because the final incumbent $\hat{u}$ is a node in S_0.

It remains to show correctness for reductions that do not end in a (shortcircuit) step. The (strengthen) rule is the only one that updates the incumbent, and it only fires to increase the objective value of the imcumbent. Thus an induction on the length of reductions shows that the h-value of incumbents increases monotonically over time, hence $h(\hat{u}) \sqsupseteq h(u)$, where u is the incumbent in some configuration along the reduction. We will argue that nodes removed by rules (terminate) and (prune) cannot beat the current incumbent u, let alone the final incumbent $\hat{u}$.

For the correctness of (terminate) we observe that, as in the proof of Theorem 3.3, traversal reductions visit the nodes of a subtree in traversal order $\ll$. So when (terminate) fires in configuration $\langle\{u\}, \mathit{Tasks}, \dots, \langle S, v\rangle, \dots\rangle$ then $\mathit{next}(S, v) = \bot$, i.e. v is the last node in $\ll$ order in S. This means that all nodes in S have been visited, and hence have been processed either by the (strengthen) or the (skip) rule. By induction on the number of node processing reductions, we can show that $h(\hat{u}) \sqsupseteq h(u) \sqsupseteq \max\{h(v) \mid v \in S\}$.

To show correctness of the (prune) rule, assume it fires in configuration $\langle\{u\}, \mathit{Tasks}, \dots, \langle S, v\rangle, \dots\rangle$, that is $u \triangleright v$ holds and $S' = \mathit{subtree}(S, v) \setminus \{v\}$. The (prune) rule removes only nodes in S', i.e. nodes v' which are strict descendants of v. Hence we have $u \triangleright v$ and $v \prec v'$ which implies $u \triangleright v'$ and $h(u) \sqsupseteq h(v')$ by admissibility conditions 3 and 1, respectively. Thus we have $h(\hat{u}) \sqsupseteq h(u) \sqsupseteq \max\{h(v') \mid v' \in S'\}$.

Since every node in the entire search tree S_0 is eventually removed by (terminate) or (prune), the arguments above are sufficient to show that $h(\hat{u})$ is an upper bound on the h-value of any node $v \in S_0$. Hence $h(\hat{u}) = \max\{h(v) \mid v \in S_0\}$ because the final incumbent $\hat{u}$ is also a node in S_0. □

Theorem 3.5. *The reduction relation $\longrightarrow$ is terminating.*

Proof sketch. It suffices to map configurations into a *measure* (i.e. a well-founded partially ordered set) such that each $\longrightarrow$ reduction step strictly decreases the measure.

We choose as measure finite multisets of natural numbers, ordered by the well-founded multiset extension $>_{\mathrm{mul}}$ of the usual order on natural numbers, as defined by Dershowitz and Manna [3]. That is, $\mathfrak{M} >_{\mathrm{mul}} \mathfrak{M}'$ if multiset $\mathfrak{M}'$ is obtained from multiset $\mathfrak{M}$ by removing one or more numbers from $\mathfrak{M}$ and then inserting zero or more numbers such that all inserted numbers are strictly smaller than at least one of the removed numbers.

We define a measuring function $\mathcal{M}$ mapping configurations to multisets of task and thread sizes. More precisely, $\mathcal{M}$ returns a multiset containing the number of nodes of each task and the number of as-yet-unexplored nodes of each thread. In the formal definition below, $\emptyset$ denotes the empty multiset, $\uplus$ denotes the sum of two multisets, and the # operator returns the size of a finite set.

$$\begin{aligned}
\mathcal{M}(\langle\sigma, \mathit{Tasks}, \theta_1, \dots, \theta_n\rangle) &= \mathcal{M}(\mathit{Tasks}) \uplus \mathcal{M}(\theta_1) \uplus \cdots \uplus \mathcal{M}(\theta_n) \\
\mathcal{M}(S{:}\mathit{Tasks}) &= \#S \uplus \mathcal{M}(\mathit{Tasks}) \\
\mathcal{M}([]) &= \emptyset \\
\mathcal{M}(\langle S, v\rangle) &= \#\{u \in S \mid v \ll u\} \\
\mathcal{M}(\bot) &= \emptyset
\end{aligned}$$

It remains to show that every $\longrightarrow$ reduction step decreases the measure.

- *Tree traversal and node processing steps.* Rule (schedule) removes a task and adds a thread; the measure decreases because the thread size is less than the task size as the root node v is already explored. Rules (expand) and (backtrack) decrease the measure because they decrease the size of the current thread by one. The (terminate) rule decreases the measure because it removes a thread.
  By definition of the per-thread reduction relation $\longrightarrow_i$, every tree traversal step must be followed by exactly one node processing step. As the rules (accumulate), (strengthen), (skip) and (noop) do not change the measure, node processing steps do not affect termination.
- *Pruning steps.* The (prune) rule decreases the measure because it removes as-yet-unexplored nodes from the current thread. The (shortcircuit) rule removes all tasks and all threads, decreasing the measure to the empty multiset.
- *Spawning steps.* The (spawn) rule removes thread $\langle S, v\rangle$ and adds in its place thread $\langle S \setminus S_u, v\rangle$ and task S_u. This reduces the measure because both the sizes of task S_u and of thread $\langle S \setminus S_u, v\rangle$ are strictly smaller than the size of thread $\langle S, v\rangle$, thanks to the fact that u is an as-yet-unexplored node of $\langle S, v\rangle$. Similar arguments show that the other spawn rules also decrease the measure.

It follows that $\longrightarrow$ must be terminating, as otherwise there would be an infinite descending chain of measures, in contradiction to the well-foundedness of $>_{\mathrm{mul}}$. □

References

[1] Blair Archibald. 2018. *Skeletons for Exact Combinatorial Search at Scale.* Ph.D. Dissertation. University of Glasgow. http://theses.gla.ac.uk/id/eprint/31000

[2] Blair Archibald, Patrick Maier, Robert Stewart, and Phil Trinder. 2020. YewPar: Skeletons for Exact Combinatorial Search. In *Proceedings of the 25th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2020, San Diego, CA, USA, February 22-26, 2020.* ACM, New York, NY, USA. https://doi.org/10.1145/3332466.3374537

[3] Nachum Dershowitz and Zohar Manna. 1979. Proving Termination with Multiset Orderings. *Comm. ACM* 22, 8 (1979), 465–476. https://doi.org/10.1145/359138.359142