Acta Informatica (2007) 44:427–461
DOI 10.1007/s00236-007-0054-z

ORIGINAL ARTICLE

BRNGLR: a cubic Tomita-style GLR parsing algorithm

Elizabeth Scott · Adrian Johnstone · Rob Economopoulos

Received: 29 July 2004 / Accepted: 4 June 2007 / Published online: 4 August 2007

Abstract Tomita-style generalised LR (GLR) algorithms extend the standard LR algorithm to non-deterministic grammars by performing all possible choices of action. Cubic complexity is achieved if all rules are of length at most two. In this paper we shall show how to achieve cubic time bounds for all grammars by *binarising* the search performed whilst executing reduce actions in a GLR-style parser. We call the resulting algorithm Binary Right Nulled GLR (BRNGLR) parsing. The binarisation process generates run-time behaviour that is related to that shown by a parser which pre-processes its grammar or parse table into a binary form, but without the increase in table size and with a reduced run-time space overhead. BRNGLR parsers have worst-case cubic run time on all grammars, linear behaviour on LR(1) grammars and produce, in worst-case cubic time, a cubic size binary SPPF representation of all the derivations of a given sentence.

1 The generalised parsing landscape

Parsing of context free languages is perhaps one of the most intensively studied formal aspects of computing, yet the asymptotic complexity of the problem is still unknown. Valiant [33] showed how a *recognition matrix* which encodes all possible derivations of a string may be constructed using boolean matrix multiplication (BMM). At the time, this gave a bound of $2^{2.81}$, the best presently known lower bound on BMM is $2^{2.376}$ [6]. Lee has further shown that the reverse of Valiant's observation holds, and that parsers may be converted into BMM algorithms implying that 'practical parsers running in significantly lower than cubic time are unlikely to exist' [23].

E. Scott (✉) · A. Johnstone · R. Economopoulos
Department of Computer Science, Royal Holloway, University of London,
Egham TW20 0EX, Surrey, UK
e-mail: E.Scott@rhul.ac.uk

A. Johnstone
e-mail: A.Johnstone@rhul.ac.uk

The overheads of using BMM based general context free parsers are high so they only become practical for very long strings: we know of no applications of Valiant's algorithm. Fortunately a variety of practical cubic time algorithms are known. The CYK algorithm [36] constructs a recognition matrix in cubic time. Earley's algorithm [9] constructs 'Earley sets' in cubic time on ambiguous grammars and quadratic time on unambiguous grammars. However, both of these algorithms deliver derivations only indirectly: a single derivation may be 'read off' from the recognition matrix or the Earley sets. There may be exponentially many derivations (or an infinite number if cycles are allowed in the grammar) and thus enumerating all derivations may be expensive.

Of course, deterministic parsing algorithms are available that admit large subsets of the context free grammars, and one approach to general parsing is to take a deterministic algorithm and extend it to simulate any nondeterminism encountered during a parse. Many authors have described backtracking and lookahead extensions to LR and LL parsing [4,7], which in general yield exponential worst-case parse times; although the addition of a *well formed substring table* to cache intermediate results can be used to reduce the runtime to $O(n^{k+1})$ where k is the length of the longest rule.

Tomita extended LR parsing, not by backtracking and lookahead but by a breadth-first simulation of multiple LR parsers spawned by nondeterminism in the LR table. The algorithm maintains a Graph Structured Stack (GSS) (really, a stack-structured graph, but the terminology is now standard) to represent the multiple stacks created by a nondeterministic LR automaton. The GSS nodes may be organised into $n + 1$ levels (where n is the length of the input), and edges are used to create pathways corresponding to different stacks. It is easy to see that stacks with common prefixes may be represented as trees. More importantly, the context free nature of the interaction between parallel parses means that, at a given level, stacks with the same state on the top may be merged, hence each level contains at most one node *per* LR state. An important of Tomita's algorithm is that it performs like an LR parser on LR grammars, so the average time complexity on grammars with significant LR(1) components is linear. For general grammars, the time complexity of Tomita's algorithm is $O(n^{k+1})$ where k is the length of the longest rule, and is thus cubic for grammars with no rules of length greater than two.

Tomita-style algorithms can construct a Shared Packed Parse Forest (SPPF) in which subtrees are re-used and packed nodes are used to represent ambiguous derivations. SPPFs form a structural representation that generalises the traditional parse tree and so easily support conventional attribute evaluators or *ad hoc* semantics evaluation processes. Johnson [12] showed that any parser which constructs a Tomita-style SPPF will need at least $O(n^{k+1})$ time.

Tomita's basic algorithm is for ϵ-free grammars, and his attempt to extend the algorithm to handle ϵ-rules fails to terminate for hidden left recursion. Farshi [26] gave a different extension to ϵ grammars that uses brute-force searching of the GSS to ensure that all possible derivation steps are found, but this fix is expensive [17]. The Right Nulled GLR (RNGLR) [30] algorithm uses what is essentially Tomita's original ϵ-free algorithm on a modified (RN) parse table. The RN tables have extra reductions corresponding to rules with suffixes that can match ϵ (right-nullable rules). These reductions effectively short circuit the parsing automaton, avoiding the ϵ problem and also improving performance [29].

The RNGLR algorithm is also $O(n^{k+1})$. In this paper we shall show how to achieve cubic time bounds for all grammars by *binarising* the search performed whilst executing reduce actions in a GLR-style parser. We call the resulting algorithm Binary Right Nulled GLR (BRNGLR) parsing. The binarisation process generates run-time behaviour that is related to that shown by a parser which pre-processes its grammar or parse table into a binary form, but without the increase in table size and with a reduced run-time space overhead.

BRNGLR parsers have linear behaviour on LR(1) grammars and produce, in worst-case cubic time, a cubic size binary-SPPF representation of all the derivations of a given sentence.

1.1 Exploiting binarisation

Any general context free parser which builds Tomita-style SPPFs is of unbounded polynomial order, but it is known that there are forms of SPPF that require at most cubic space. If the grammar rules have length at most two then these are also the SPPFs produced by Tomita's algorithm. For general grammars, the cubic SPPFs can be obtained by 'sharing' right-hand children of different SPPF nodes (see [5]). There are three points in the parser generation process at which we might exploit this observation:

(a) by pre-processing the grammar into binary form;
(b) by retaining the original grammar but transforming the parse table; or
(c) by retaining the original grammar and table but modifying the algorithm to produce cubic size SPPFs.

Option (c) is the main topic of this paper. We report on our investigations into option (b) in [31]. We begin by using option (a) to motivate our approach to modifying the RNGLR algorithm.

We can make the RNGLR algorithm cubic by adding an automated preprocessing stage, factoring each rule and adding new non-terminals. The factoring is carried out in a straightforward fashion because the resulting SPPF will describe derivations in terms of the new non-terminals, and we need to be able to 'walk' the SPPF to recover the derivations with respect to the original grammar in a direct manner. Thus we replace rules $A ::= x_1 x_2 \cdots x_m$ with

$$A ::= x_1 A_1 \quad A_1 ::= x_2 A_2 \quad \cdots \quad A_{m-2} ::= x_{m-1} x_m$$

This results in the addition of $m-2$ new non-terminals for each production of length greater than 2 which increases the size of the resulting table and the size of the runtime GSS. In detail, the SLR(1) automaton gains one extra state for each new non-terminal corresponding to a new accepting state containing a reduction by that non-terminal. A straightforward representation of an LR table requires $S \times N$ cells where S is the number of states in the underlying automaton and N is the number of terminals and non-terminals in the grammar plus one. If the factored grammar requires M new non-terminals, then the size of the table goes up from $S \times N$ to $(S + M) \times (N + M)$ cells. For example, the SLR(1) table for our Cobol grammar has 2.8×10^6 cells, while the table for the binarised version of the grammar has 6.2×10^6 cells. For LR(1) grammars the number of new states is higher because items which have the same rule can have different local FOLLOW sets (that is, the follow sets associated with the individual LR(1) items). The LR(1) table for ANSI-C has 2.9×10^5 cells, while the LR(1) table for the binarised version of the grammar has 8.0×10^5 cells.

If we examine the DFA underlying the parse table of a binarised grammar, we find that the extra states associated with the new non-terminals show a very simple structure, and in fact we can simply add rows to an existing (non-binarised) LR table mechanically, corresponding to option (b) above. Since we cannot tell which rules in the table will be used by the GLR algorithm, we need to apply this transformation to all rules.

If we compare the GSSs arising from parses with such a binarised table to those constructed from the original grammar, we find that the extra parts of the GSS are themselves very repetetive. A reduction in the original grammar by a rule of length j triggers a search for a path in the GSS of length j. In the binarised case, this is converted into $j - 1$ searches of length 2, each of which results in a new reduction edge being added to one of the new states in the level. (These edges allow repeated searches to be aborted after just two steps.) This GSS transformation can be performed easily just as we can mechanically transform the table directly, but more interestingly we can perform the transformation efficiently on-the-fly. Done naïvely, this saves space in the table but the GSS levels will still need space for $S + M$ states. However, it turns out that instead of needing $k - 2$ new states for each rule we can make do with $h - 2$ new states *per* non-terminal A, where h is the length of the longest rule for A. This reduces the potential worst-case size of the GSS, as we shall demonstrate in our experimental section. For LR(1) grammars we still only need $h - 2$ new states *per* non-terminal, even though the binarised LR(1) table requires more additional states than the SLR(1) table.

Now we give a brief overview of some related theoretical work and existing implementations of GLR algorithms, and then give the definitions we shall need and review the RNGLR algorithm. We then give the BRNGLR algorithm which we prove correctly constructs all the traversals of an RN LR table on a given input, is of at most cubic order and requires at most quadratic space. We then add the necessary actions to turn the algorithm into a parser, and show that the parser has at most cubic order and that it produces an SPPF which requires at most cubic space. Finally we give some experimental results that illustrate the practical aspects of the performance of our algorithm.

1.2 Related work

Lang [22] studied the general problem of simulating the behaviour of nondeterministic PDAs which perform at most one push and one pop action at each step. The obvious way of obtaining a PDA from the FA that underlies an LR parse table yields pop sequences whose lengths correspond to the lengths of the alternates in the grammar rules. While Lang's algorithm is cubic, its natural extension to LR-based PDAs which pop multiple symbols has complexity $O(n^{k+1})$ where k is the length of the longest rule. Thus it is necessary to binarise the grammar or modify the parse table, as described under options (a) and (b) in the above section, to obtain a cubic algorithm. Once the grammar or table has been modified, Lang's algorithm can generate a cubic size grammar which encodes all derivations [22] or cubic size SPPFs [5].

A variety of developments of Tomita's work exist. We have already mentioned Farshi's brute force correction and our own RNGLR solution. Kipps [20] described what he claimed was cubic variant of Tomita's algorithm. Like Tomita's algorithm, Kipps' algorithm does not terminate on grammars with hidden left recursion and hence is not general or actually cubic. Furthermore, Kipp's algorithm is a recogniser only. The assumption is that Tomita's method would be used to produce derivation trees and, as noted in [12], Tomita-style SPPFs are of supra-cubic order for some classes of grammars. Rekers [28] used Farshi's algorithm as a basis for developing a more compact SPPF format and Visser [35] investigated the integration of lexical rules directly into the main grammar, yielding a scannerless parser. These algorithms inherit the order of Farshi's algorithm, and hence of Tomita's algorithm.

Aycock and Horspool [1,2] described an automaton in which stack activity is reduced by directly incorporating some LR reductions as edges from the state containing the reduction to the 'goto' state of the reduction. Such edges can only be added if the reduction rule does not generate self-embedding. Aycock and Horspool developed techniques for detecting and removing embedded recursion from grammars, creating a set of DFAs that recognise regular sub-languages and 'call' each other when embedded recursion is encountered. They use a

GSS to simulate the resulting nondeterministic PDA. Although they describe their approach as being a modification of Tomita's GLR algorithm it behaves differently to Tomita's even on LR grammars. We have described a closely related algorithm that does not require Aycock and Horspool's restriction to grammars that are free from hidden left recursion [15]. These techniques are interesting, but the tables are very large and their practicality is borderline. As recognisers, they run in cubic time, but the parser versions producing SPPFs once again display $O(n^{k+1})$ time complexity, where k is the length of the longest rule.

Despite the worst-case order of existing GLR algorithms, there has been a resurgence of interest in the use of generalised parsing even for programming language type applications, typified by the use of backtracking extensions in parser generators such as Java-CC [11], ANTLR-2 [27] and BTYacc; but perhaps more interestingly Bison [10] (the present shift-reduce parser generator of choice) has acquired a GLR mode. Sadly, it is not a full GLR implementation (stack recombination does not take place) and this can result in exponential runtimes [8, p. 16]; and rather than producing SPPFs an attempt is made to merge derivation trees during the parse into a single preferred derivation. Elkhound [25] is a more successful implementation of GLR parsing which runs almost as fast as Bison on deterministic *parts* of grammars; that is as well as exploiting the linear asymptotic behaviour of GLR parsers on LR grammars Elkhound achieves constants of proportionality that are close to the simple deterministic parser. The ASF+SDF tool [34], which has been used extensively for re-engineering Cobol code, has a GLR parser that is based on Farshi's algorithm, using the developments made by Rekers and Visser mentioned above. The tool has support for SPPF generation, and includes sophisticated techniques for disambiguating the forest.

2 Basic definitions

In this section we introduce the standard concepts we require. Further details on standard deterministic parsing techniques can be found in [3].

A *context free grammar* (CFG) consists of a set $\mathbf{N}$ of non-terminal symbols, a set $\mathbf{T}$ of terminal symbols, an element $S \in \mathbf{N}$ called the start symbol, and a set of grammar rules of the form $A ::= \alpha$ where $A \in \mathbf{N}$ and α is a (possibly empty) string of terminals and non-terminals.

A *derivation step* is an element of the form $\gamma A\delta \Rightarrow \gamma\alpha\delta$ where γ and δ are strings of terminals and non-terminals, and $A ::= \alpha$ is a grammar rule. A *derivation* of τ from σ is a sequence of derivation steps $\sigma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \cdots \Rightarrow \beta_{n-1} \Rightarrow \tau$. We also write $\sigma \overset{*}{\Rightarrow} \tau$ and $\sigma \overset{n}{\Rightarrow} \tau$ and we may write $\sigma \overset{+}{\Rightarrow} \tau$ if $n > 0$. We use ϵ to denote the empty string.

A grammar rule $A ::= \alpha\beta$ is said to be *right nullable* if $\beta \neq \epsilon$ but $\beta \overset{*}{\Rightarrow} \epsilon$. A grammar is said to have *hidden left recursion* if there is a non-terminal A and strings τ, σ such that $\tau \overset{+}{\Rightarrow} \epsilon$ and $A \overset{*}{\Rightarrow} \tau A \sigma$.

A *sentential form* is any string α such that $S \overset{*}{\Rightarrow} \alpha$ and a *sentence* is a sentential form which contains only elements of $\mathbf{T}$. The set, $L(\Gamma)$, of sentences which can be derived from the start symbol of a grammar Γ, is defined to be the *language* generated by Γ.

For a grammar symbol x and a string γ we define

$$\text{FIRST}_{\mathbf{T}}(x) = \{t \in \mathbf{T} \mid \text{for some string } \beta, x \overset{*}{\Rightarrow} t\beta\}$$

A finite state automaton (FA) is a directed graph whose nodes are called states, one of which is designated as the start state and some of which are designated as accepting states, and whose arrows, called transitions, are labelled with either an input symbol or the empty string, ϵ. Accepting states are denoted with double lines. An FA is *deterministic* (a DFA) if it has no ϵ transitions, and if for each grammar symbol x and each node v there is at most one transition from v labelled x.

Given an integer valued function $f : \mathbb{N} \to \mathbb{N}$ we define $O(f)$ to be the set of functions which are eventually dominated by some scalar multiple of f. So

$$O(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \text{for some } c, N > 0,\ cf(i) \geq g(i) \quad \text{for all } i \geq N\}.$$

We say that a function g has order at most n^k, and that g is at most $O(n^k)$, if $g \in O(n^k)$, i.e. if there is a polynomial $f(n) = b_k n^k + \cdots + b_1 n + b_0$ with $b_k > 0$ such that for large enough n, $f(n) \geq g(n)$. We say that a function g has order at least n^k, and that g is at least $O(n^k)$, if $n^k \in O(g)$, i.e. if there is a polynomial $f(n) = b_k n^k + \cdots + b_1 n + b_0$ with $b_k > 0$ such that for large enough n, $g(n) \geq f(n)$.

We say that a parsing algorithm has unbounded polynomial order if, for any $k \in \mathbb{N}$, there is some grammar for which the algorithm has order at least $O(n^k)$. If, for all grammars, the algorithm has order at most $O(n^3)$ then we say that it has cubic order.

3 Nondeterministic shift-reduce parsers

A bottom up parser searches a sentential form for a substring that corresponds to a right hand side of some rule $A ::= \alpha$, replacing the handle α with its corresponding non-terminal A to form a new sentential form. Acceptance of the string is signalled by the appearance of a sentential form containing only the start symbol.

A shift-reduce parser is a stack based mechanism which, in conjunction with a rectangular table indexed by state number and the grammar's alphabet, computes configurations. Loosely, the operation of the parser may be visualised as the *shifting* of input characters onto the stack until the right hand side of a rule $A ::= \alpha$ is detected, at which point the stack is *reduced* by popping the elements of α off the stack and then pushing the non-terminal A. It is important to note that the symbols being popped need not be examined since the table construction algorithm guarantees that the top m elements (where m is the length of α) will be the elements of α: in effect the only stack operations needed are to push a single symbol whose value is supplied and to discard m elements, both of which require only unit time for the traditional stack implementation in which an array is indexed by a stack pointer. After a reduction, the stack holds part of a new sentential form: further reductions may be immediately available or the parser may need to push more of the string on before reductions can occur. If the string is in the language the parser will eventually perform a reduce by the start symbol which indicates acceptance.

In practice, the shift-reduce parser may encounter nondeterminism in the form of a choice of reduction possibilities and/or a choice between shift and reduce actions. The LR-style parse tables pre-compute some part of the prefix recognition which may be described using regular grammars and thus recognised using finite automata. The effect is to reduce the amount of nondeterminism in the stack based mechanism essentially by applying the subset construction to the underlying automaton. For the remaining nondeterminism to be simulated on a sequential machine we must replicate parser configurations and allow them to evolve independently. A shift-reduce parser's configuration is completely encoded in its stack so it suffices to replicate stacks. Tomita proposed that each stack be arranged as a chain of nodes with elements pointing to their predecessors down the stack. This has the advantage that stacks with the same prefix may share nodes, but the disadvantage is that a reduction by m symbols requires m time steps.

If the nondeterministic parser executes all reductions in a configuration before any shift, then the stacks will be *synchronised by level*, where each level corresponds to a position in the input string. This synchronisation gives the algorithm the character of a breadth-first simulation of the nondeterministic parse, whereas backtracking yields depth-first simulation. After executing reductions, all stack tops are examined for configurations that allow a valid shift: such stacks will be extended to the next level, corresponding to the consumption of an input terminal.

The multiple stacks are merged into a single GSS which has one outstanding property. Since a complete parser configuration is encoded in its stack, and since there is no advantage in having more than one (simulated) nondeterministic parser instance working on each configuration, we can *merge* stacks having the same state on top. The context free nature of stack activations ensures that no false paths are introduced by this merging process, which then bounds the number of nodes in the GSS to at most $n \times H$ where n is the length of the string and H is the number of states in the parser.

A GSS, then, is a directed graph with the nodes arranged in levels each containing a number of nodes bounded by H. Potentially there is an edge from a node in level i to every node in every level j, $0 \leq j \leq i$. In the sequential parser, a reduction requires us to merely discard the top m elements of the stack. In the GSS representation a reduction requires us to find the set of nodes that are connected to a particular node via a path of length m edges. Since the out-degree of each node in level i is potentially $H \times (i+1)$, there may be of the order of i^m paths to explore. Standard Tomita-style algorithms, therefore, display $O(n^{m+1})$ time complexity.

Our approach converts a reduction of length m into $m-1$ searches of length two. We can apply the transformation at the level of the table or we can change the way that the algorithm constructs the GSS. Both approaches add at most $O(n)$ additional states to the GSS. Before introducing these approaches, we review the RNGLR algorithm on which they are based.

4 LR(1) DFAs and RN tables

We assume that the grammar Γ has an augmented start rule, $S' ::= S$, so that S' does not appear on the right hand side of any grammar rule. We construct the LR(1) DFA for Γ in the standard way, see for example [3], such that the DFA states are sets of *items* of the form $(X ::= \alpha \cdot \beta, a)$, where $a \in \mathbf{T}$ or $a = \$$, the special end-of-string symbol. The states k are *closed* in the sense that if $(X ::= \alpha \cdot Y\beta, g) \in k$ then for all rules $Y ::= \gamma$ and for all $f \in \text{FIRST}_{\mathbf{T}}(\beta g)$ (and for $f = \$$ if $\beta \overset{*}{\Rightarrow} \epsilon$ and $g = \$$) $(Y ::= \cdot\gamma, f) \in k$. Also, for each state k, if $(X ::= \alpha \cdot x\beta, a) \in k$ then there is a state h which is the closure of the set which contains all the items $(Z ::= \mu x \cdot \nu, b)$, where $(Z ::= \mu \cdot x\nu, b) \in k$. In this case there is a *transition* labelled x from k to h. We say that the item $(X ::= \alpha \cdot \beta, b)$ *is in the DFA state* k, and that $(X ::= \alpha \cdot \beta, b) \in k$, if $(X ::= \alpha \cdot \beta, b)$ belongs to the set of items which label k. State 0, the *start state* of the DFA, is the state obtained by forming the closure on the item $(S' ::= \cdot S, \$)$, and the *accepting state* is the (unique) state l which is the target of the transition labelled S from 0. (An example of an LR(1) DFA is given in Sect. 5.)

We construct a *right nulled* (RN) table, $\mathcal{T}$, from the LR(1) DFA of a CFG as follows. The rows of $\mathcal{T}$ are indexed by the DFA states, the columns are indexed by symbols $x \in \mathbf{T} \cup (\mathbf{N} \backslash \{S'\}) \cup \{\$\}$, and the entries are sets of actions. We write $\mathcal{T}(k, x)$ for the entry in row k, column x of $\mathcal{T}$. If there is a transition labelled x from k to h in the DFA, then $\mathcal{T}(k, x)$

contains the action 'shift h', traditionally these are written sh if x is a terminal and gh if x is a non-terminal, but we shall just write ph (push h) in both cases. A *right nulled reduction* is an item of the form $(X ::= \alpha \cdot \beta, g)$, where $\beta \overset{*}{\Rightarrow} \epsilon$ and $X \neq S'$. The items $X ::= \alpha \cdot \beta$ are numbered, and if a state k contains reduction (q, k) then $\mathcal{T}(k, g)$ contains the action 'reduce by q', written rq. If l is the accepting state of the LR(1) DFA then $\mathcal{T}(l, \$)$ contains the action 'accept', which we shall write as acc, and if $S \overset{*}{\Rightarrow} \epsilon$ then $\mathcal{T}(0, \$)$ also contains acc.

Note: The RN table has the same dimensions as the corresponding LR table and, if the grammar does not contain any right nullable rules then the RN table is the same as the LR table. Also, this table is slightly different to that given in [30] because, to ensure an $O(n^3)$ parser, we need to know the reduction q applied at each step.

4.1 Traversing an RN PDA

The RN push down automaton (RN PDA) for a grammar Γ consists of the RN table, $\mathcal{T}$, for Γ and a stack. A *configuration* of an RN PDA is a set $\{\mathcal{S}, a\}$ where $\mathcal{S}$ is a stack whose entries are row numbers of $\mathcal{T}$ and a is an input symbol. The configuration should be thought of as the current stack and the lookahead (next input) symbol, and the state at the top of the current stack is the *current state*. The configuration $\{(0), a_1\}$ is the *initial configuration* for an input string of the form $a_1\alpha$. For all input strings the configuration $\{(0, \ldots, l), \$\}$ is an *accepting configuration*, if l is the accepting state of the LR(1) DFA, and if $S \overset{*}{\Rightarrow} \epsilon$ then $\{(0), \$\}$ is also an accepting configuration.

Given an RN PDA with RN table, $\mathcal{T}$, and an input string, $a_1 \cdots a_m$, we define an *execution step*, θ, as follows. There is an input configuration $\{(k_0, \ldots, k_n), a_j\}$, an action from $\mathcal{T}(k_n, a_j)$, and a resulting configuration defined by:

- If the action is ph then θ results in $\{(k_0, \ldots, k_n, h), a_{j+1}\}$.
- If the action is rt, where t is $A ::= x_1 \cdots x_m \cdot \omega$, then θ results in the configuration $\{(k_0, \ldots, k_{n-m}, h), a_j\}$ where $ph \in \mathcal{T}(k_{n-m}, A)$.
- If the input configuration is an accepting configuration and if the action is acc then the result is 'accept'. In this case we say that *the execution step results in success.*

We shall often refer to an execution step θ as being a 'shift' or a 'reduction' or an 'accept' when that is the action associated with θ.

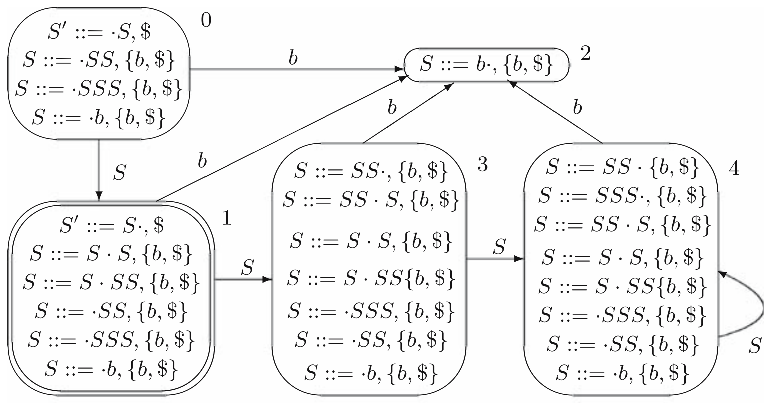
An *execution path* on $a_1 \cdots a_n$ through an RN table $\mathcal{T}$ is a sequence, $\theta_1 \cdots \theta_q$, of execution steps where the input configuration of θ_1 is the initial configuration for a_1 and for $i \geq 2$, the input configuration of θ_i is the resulting configuration of θ_{i-1}. The *result of the path* $\theta_1 \cdots \theta_q$ is the result of θ_q. We have shown [30] that $u \in L(\Gamma)$ if and only if there is an execution path on u through the RN table, which results in 'accept'.

5 Review of the RNGLR algorithm

The RNGLR algorithm [30] constructs all the execution paths on a given input string through the RN table for a given CFG. A full discussion of the operation of and motivation for RNGLR, together with some examples, can also be found in [32] (where it is called the GRMLR algorithm). In this section we illustrate the algorithm using an example on which it displays quartic behaviour. The example is one of the class of examples used by Johnson [12] to show that Tomita's original algorithm is of unbounded polynomial order. (Johnson's argument is based on the size of the output which is a shared packed parse forest, but it is not hard to see [31] that the recogniser is also of quartic order. Furthermore, the grammar $S ::= b \mid S^{N-1}$ triggers $O(n^N)$ behaviour in the RNGLR recogniser.)

Example 1 Consider the grammar, Γ_1, which has the following rules, LR(1) DFA and RN table, $\mathcal{T}_1$.

$$1.\ S ::= b \quad 2.\ S ::= \ SS \quad 3.\ S ::= SSS$$


0
$S' ::= \cdot S, \$$
$S ::= \cdot SS, \{b, \$\}$
$S ::= \cdot SSS, \{b, \$\}$
$S ::= \cdot b, \{b, \$\}$
b
2
$S ::= b\cdot, \{b, \$\}$
b
b
S
b
1
$S' ::= S\cdot, \$$
$S ::= S \cdot S, \{b, \$\}$
$S ::= S \cdot SS, \{b, \$\}$
$S ::= \cdot SS, \{b, \$\}$
$S ::= \cdot SSS, \{b, \$\}$
$S ::= \cdot b, \{b, \$\}$
S
3
$S ::= SS\cdot, \{b, \$\}$
$S ::= SS \cdot S, \{b, \$\}$
$S ::= S \cdot S, \{b, \$\}$
$S ::= S \cdot SS\{b, \$\}$
$S ::= \cdot SSS, \{b, \$\}$
$S ::= \cdot SS, \{b, \$\}$
$S ::= \cdot b, \{b, \$\}$
S
4
$S ::= SS \cdot \{b, \$\}$
$S ::= SSS\cdot, \{b, \$\}$
$S ::= SS \cdot S, \{b, \$\}$
$S ::= S \cdot S, \{b, \$\}$
$S ::= S \cdot SS\{b, \$\}$
$S ::= \cdot SSS, \{b, \$\}$
$S ::= \cdot SS, \{b, \$\}$
$S ::= \cdot b, \{b, \$\}$
S

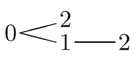
	\$	b	S
0		$p2$	$p1$
1	acc	$p2$	$p3$
2	$r1$	$r1$	
3	$r2$	$r2/p2$	$p4$
4	$r2/r3$	$r2/r3/p2$	$p4$

We compute the execution paths on bbb through $\mathcal{T}_1$. The strategy that we apply is that stacks are replicated until all valid reductions have been applied, then the next input symbol is read and the corresponding state is pushed on to the appropriate stacks, the other stacks being terminated.

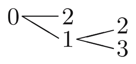
The initial configuration is $\{(0), b\}$ and since $p2$ is the only entry in $\mathcal{T}_1(0, b)$, the symbol b is read and we move to the configuration $\{(0, 2), b\}$. We have $r1 \in \mathcal{T}_1(2, b)$ so we pop one state off the stack and, since $p1 \in \mathcal{T}_1(0, S)$, we push 1 onto the stack, generating another stack $(0, 1)$ and corresponding configuration. There are no reductions in $\mathcal{T}_1(1, b)$ so the current stacks are $(0, 2)$ and $(0, 1)$. Because these stacks have the same first element we combine them, branching at the point where they differ.

$$\begin{tikzcd}[row sep=tiny] & 2 \\ 0 \arrow[ur, dash] \arrow[dr, dash] & \\ & 1 \end{tikzcd}$$

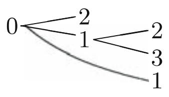
We read the next input symbol, b, and look for actions pk in column b of $\mathcal{T}_1$. There is an action $p2 \in \mathcal{T}_1(1, b)$ so we generate a new stack (0, 1, 2), but there is no entry of the form pk in $\mathcal{T}_1(2, b)$ so the configuration whose stack is (0, 2) is terminated.



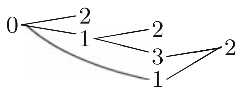
We then apply the reductions to the stack (0, 1, 2). We have $r1 \in \mathcal{T}_1(2, b)$ and $p3 \in \mathcal{T}_1(1, S)$ so we generate a new stack (0, 1, 3). This is merged with (0, 1, 2) up to the point of difference.



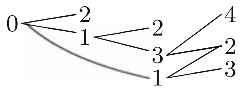
Applying the reductions to the stack (0, 1, 3) we have $r2 \in \mathcal{T}_1(3, b)$ and $p1 \in \mathcal{T}_1(0, b)$ so we generate the stack (0, 1).



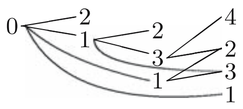
There are no further applicable reductions so we read the last input symbol, b, and set the lookahead symbol to be \$. From the p actions we generate the stacks (0, 1, 3, 2) and (0, 1, 2) from (0, 1, 3) and (0, 1), respectively, and terminate the configuration whose stack is (0, 1, 2). The two stacks with the same state, 2, on top can be recombined.



Since $r1 \in \mathcal{T}_1(2, \$)$ we generate two further stacks (0, 1, 3, 4) and (0, 1, 3).

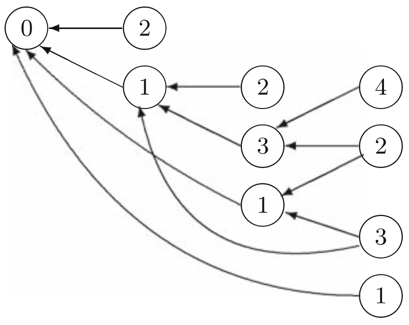


There are two reductions $r3, r2 \in \mathcal{T}_1(4, \$)$ and applying these and the reduction $r2 \in \mathcal{T}_1(3, \$)$ generates the stacks (0, 1) and (0, 1, 3). (The latter can be recombined with the other stack (0, 1, 3) since they both have the same state on the top.)



$\mathcal{T}_1(1, \$)$ contains no reductions and all the input has been read, so no further actions can be applied. Since the accepting configuration {(0, 1), \$} is a current configuration, the string *bbb* is accepted.

We represent the combined stacks as a GSS. The nodes of the graph are the states on the stack and there is an arrow from each state to the state(s) immediately below it on some stack. The following GSS corresponds to the combined stacks in the above example.



6 Binary RN push down automata

As mentioned in Sect. 1.1, we can make the RNGLR algorithm worst-case cubic order by adding a preprocessing step that factorises the grammar rules using new non-terminals to generate rules of length two. For example, we could rewrite Γ_1 from Sect. 5 in the form

$$S ::= SS \mid SA \mid b \qquad A ::= SS$$

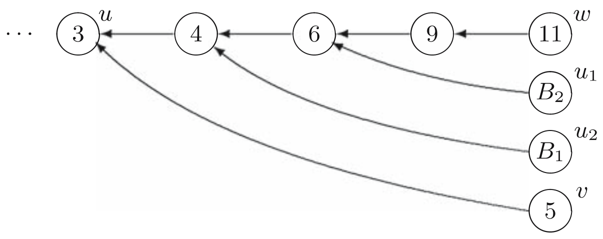
resulting in a grammar whose relationship is close enough to the original grammar to allow us to generate derivations with respect to that grammar. However, this can substantially increase both the number of rows and the number of columns in the RN table.

Instead of modifying the grammar, we can modify the RN table directly so that a GSS is produced in which only paths of length 2 need to be retraced, and on which the RNGLR algorithm is at most $O(n^3)$. The resulting BRN table will have only N_X extra rows for each non-terminal X, where $(N_X + 2)$ is the length of the longest alternate of rule X, and it will have no extra columns at all. If $N_X \geq 1$ then we add N_X new states, $X_1, \ldots, X_{N_X}$, called *additional states*, to the table. We also introduce a new type of action, rX_j, which pops the top two elements off the stack and pushes X_j onto the stack, and rX that pops two elements off the stack and then pushes the goto state in the usual way. We illustrate this with the following example.

Example 2 Consider the grammar, Γ_2, which has right nulled reductions

1. $S ::= A\ B\ a$	4. $B ::= A\ b\ C\ C$	7. $C ::= a\ a$
2. $A ::= b$	5. $B ::= A\ b\ C \cdot C$	8. $C ::= a$
3. $A ::= \epsilon$	6. $B ::= A\ b \cdot C\ C$	9. $C ::= \epsilon$

When we need to apply the reduction $B ::= AabCC$ from a node w, say, we trace back along a path of length 4 to a node u, say, and then create a node v and an edge (u, v). In the binary version, the idea is that we trace back along a path of length 2 and then create a new node, u_1 say. From this node we then trace back 2 and create u_2, say, and from u_2 we trace back 2 and create the node v.



To achieve this we can create a new BRN table which has additional states B_2 and B_1. The RN table, $\mathcal{T}_2$, for Γ_2, and the BRN table, $\mathcal{T}_2'$, obtained by replacing certain reduction entries (in this case in columns \$ and a only) are shown below. We replace $r4 \in \mathcal{T}_2(11, a)$ with the reduction rB_2, and we add rB_1 to $\mathcal{T}_2'(B_2, a)$ and rB to $\mathcal{T}_2'(B_1, a)$. In the same way we replace the reduction $B ::= AbC \cdot C$ with rB_1 and we add a state, S_1, for the reduction of length 3 on S. We shall return to this example when we consider the parser in Sect. 8.3.

The GSS generated by the BRN table will, in general, be larger than the GSS generated by the RN table, although both graphs have $O(n)$ nodes and $O(n^2)$ edges. But the GSS from the BRN table can always be constructed in at worst cubic time, see [31].

	$\mathcal{T}_2$		$\mathcal{T}_2'$						
	\$	a	\$	a	b	S	A	B	C
0					$p2/r3$	$p1$	$p3$		
1	acc		acc						
2					$r2$				
3					$p2/r3$		$p4$	$p5$	
4					$p6$				
5		$p7$		$p7$					
6		$p8/r6/r9$		$p8/r6/r9$					$p9$
7	$r1$		rS_1						
8		$p10/r8$		$p10/r8$					
9		$p8/r5/r9$		$p8/rB_1/r9$					$p11$
10		$r7$		$r7$					
11		$r4$		rB_2					
B_2				rB_1					
B_1				rB					
S_1			rS						

The ‘empty’ error entries in rows of the BRN table labelled with the additional states will never be examined, because the error will be discovered earlier, and the non-error entries in a given row will all be the same. So we do not need to increase the size of the table by actually adding the new rows. We now give a modified version of the RNGLR algorithm which takes as input the original RN table and constructs the additional GSS states ‘on-the-fly’.

7 The BRNGLR algorithm

In this section we give a formal definition of the BRNGLR algorithm, a proof that the BRNGLR algorithm correctly computes all the execution paths through an RN table, and show that it has order at most $O(n^3)$ and takes at most $O(n^2)$ space.

We number all of the grammar rules and the items that correspond to right nulled reductions. Then for rule number q we use $|q|$ to denote the length of the reduction, i.e. the length of the right hand side reduction string. So, if q is the reduction $X ::= \alpha \cdot \omega$ then $|q| = |\alpha|$.

The algorithm builds a GSS for the input string, $a_1 \cdots a_n$. The nodes of the GSS are divided into disjoint levels, $U_0, \ldots, U_n$, where level U_i contains the nodes which are at the top of any current stack constructed after a_i is read and before a_{i+1} is read. For each node

w in U_i with label k say, if $rq \in \mathcal{T}(k, a_{i+1})$ we need to apply this reduction down all paths of length $|q|$ from w. These pending reductions are stored in a set $\mathcal{R}$ in the form of a triple $(v, q, |q|)$ where v is the target of the first edge down which the reduction is to be applied. For $m \geq 3$, when an element (v, q, m) is processed, for each child u of v a GSS edge is added from a node labelled X_m to u, where q is $X ::= \alpha \cdot \omega$, and $(u, q, m-1)$ is added to $\mathcal{R}$. In this way reductions of length greater than 2 are traversed in steps of length 2 and a record is kept, in the form of the GSS *book-keeping node* labelled X_m, to ensure that this path is not traversed again. The non-book-keeping GSS nodes are called *state nodes*. The algorithm also maintains a set $\mathcal{Q}$ of pairs (v, h) where v is a node in U_i with label k and $ph \in \mathcal{T}(k, a_{i+1})$. These are the 'shift' actions which will be applied when the construction of U_i is complete.

7.1 The algorithm

Input: RN table $\mathcal{T}$ and an input string $a_1 \cdots a_n$

```
PARSER {
 **if** $n = 0$ { **if** $acc \in \mathcal{T}(0, \$)$ report  SUCCESS
               **else** report  FAILURE }
 **else** {
   create a node $v_0$ labelled 0
   set $U_0 = \{v_0\}$, $\mathcal{R} = \emptyset$, $\mathcal{Q} = \emptyset$, $a_{n+1} = \$$, $U_1 = \emptyset, \ldots, U_n = \emptyset$
   **if** $pk \in \mathcal{T}(0, a_1)$ add $(v_0, k)$ to $\mathcal{Q}$
   **forall** $rq \in \mathcal{T}(0, a_1)$ add $(v_0, q, 0)$ to $\mathcal{R}$
   **for** $i = 0$ to $n$  **while** $U_i \neq \emptyset$ do {  **while** $\mathcal{R} \neq \emptyset$ do REDUCER($i$)
                                                   SHIFTER($i$)  }
   **if** the label of some $w \in U_n$ contains $(S' ::= S\cdot, \$)$ report  SUCCESS
   **else** report  FAILURE } }

REDUCER($i$) {
 remove $(v, q, m)$ from $\mathcal{R}$
 let $q$ be $X ::= \alpha \cdot \omega$
 **if** $m = 0$ {
    let $k$ be the label of $v$ and let $pl \in \mathcal{T}(k, X)$
    **if** there is a node $w \in U_i$ with label $l$  {
      **if** there is not an edge $(w, v)$ create an edge $(w, v)$}
    **else** {
        create a GSS node $w$ labelled $l$ and an edge $(w, v)$
        **if** $ph \in \mathcal{T}(l, a_{i+1})$ add $(w, h)$ to $\mathcal{Q}$
        **forall** $rt \in \mathcal{T}(l, a_{i+1})$ with $|t| = 0$ add $(w, t, 0)$ to $\mathcal{R}$ } }
 **if** $m = 1$ COMPLETE_REDUCTION($v, X, i$)
 **if** $m = 2$ {
   **forall** children $u$ of $v$ COMPLETE_REDUCTION($u, X, i$) }
 **if** $m \geq 3$ {
   **if** there is not a GSS node $w \in U_i$ with label $X_m$ create one
   **forall** children $u$ of $v$ {
     **if** there is not an edge $(w, u)$  {
         create an edge $(w, u)$
         add $(u, q, m-1)$ to $\mathcal{R}$ } } } }
```

```
SHIFTER($i$) {
 if $i \neq n$ {
   set $\mathcal{Q}' = \emptyset$
   while $\mathcal{Q} \neq \emptyset$ do {
     remove an element $(v, k)$ from $\mathcal{Q}$
     if there is $w \in U_{i+1}$ with label $k$ {
       create an edge $(w, v)$
       forall $rt \in \mathcal{T}(k, a_{i+2})$ with $|t| \neq 0$, add $(v, t, |t|)$ to $\mathcal{R}$ }
     else {
       create a node $w \in U_{i+1}$ labelled $k$ and an edge $(w, v)$
       if $ph \in \mathcal{T}(k, a_{i+2})$ add $(w, h)$ to $\mathcal{Q}'$
       forall $rt \in \mathcal{T}(k, a_{i+2})$ with $|t| \neq 0$, add $(v, t, |t|)$ to $\mathcal{R}$
       forall $rt \in \mathcal{T}(k, a_{i+2})$ with $|t| = 0$, add $(w, t, 0)$ to $\mathcal{R}$ } }
   copy $\mathcal{Q}'$ into $\mathcal{Q}$ } }

COMPLETE_REDUCTION($u, X, i$) {
     let $k$ be the label of $u$ and let $pl \in \mathcal{T}(k, X)$
   if there is a node $w \in U_i$ with label $l$ {
     if there is not an edge $(w, u)$ {
       create an edge $(w, u)$
       forall $rt \in \mathcal{T}(l, a_{i+1})$ with $|t| \neq 0$ add $(u, t, |t|)$ to $\mathcal{R}$ } }
   else {
     create a GSS node $w$ labelled $l$ and an edge $(w, u)$
     if $ph \in \mathcal{T}(l, a_{i+1})$ add $(w, h)$ to $\mathcal{Q}$
     forall $rt \in \mathcal{T}(l, a_{i+1})$ with $|t| \neq 0$ add $(u, t, |t|)$ to $\mathcal{R}$
     forall $rt \in \mathcal{T}(l, a_{i+1})$ with $|t| = 0$ add $(w, t, 0)$ to $\mathcal{R}$ } }
```

The GSS generated by the BRNGLR algorithm from Γ_1 and input string *bbb* can be found in Sect. 8 below.

7.2 The order of the BRNGLR algorithm

For each non-terminal A define N_A such that $(N_A + 2)$ is the length of the longest rule whose left hand side is A, or $N_A = 0$ if the longest rule has length at most 2. We let N denote the sum of all the N_A, we let H denote the number of states (rows) in the RN table, $\mathcal{T}$, we let K denote the maximum number of reductions in an entry of $\mathcal{T}$, and we suppose that we have an input string of length n.

Looking at the algorithm it can be seen, by induction on the order in which elements are created, that for a GSS node u if $(u, t, m) \in \mathcal{R}$ or $(u, k) \in \mathcal{Q}$ then u cannot be a book-keeping node. Furthermore, since a GSS edge whose target is a node w is only created as the result of an action ph where h is the label of w, there are no edges in the GSS whose target is a book-keeping node.

Each level, U_i, of the GSS contains at most one node with each state label and at most N book-keeping nodes, thus the maximum number of nodes in the GSS is $(n + 1)(H + N)$. There is at most one edge from each node in U_i to each of the state (non-book-keeping) nodes in U_j, where $0 \leq j \leq i$. Thus there are at most

$$\sum_{i=0}^{n}(H + N)H(i + 1) = \frac{(H + N)H(n + 1)(n + 2)}{2}$$

edges in the GSS. So the size of the structure is at most $O(n^2)$.

There are at most H elements in $\mathcal{Q}$ and at most K entries in $\mathcal{T}(k, a_{i+2})$ so the order of SHIFTER(i) is constant.

Apart from at the beginning of the algorithm, an element (u, t, m) is added to $\mathcal{R}$ during step i of the algorithm only as a result of creating an edge in the GSS from a node in U_i. So there are at most $K(H+N)H(i+1)$ elements added to $\mathcal{R}$ at step i of the algorithm.

There are at most $H(j+1)$ children of a node $v \in U_j$, where $j \leq i$, so the bodies of cases $m = 2$ and $m \geq 3$ are executed at most $H(i+1)$ times in REDUCER(i).

We assume that the edges from each node $u \in U_i$ are stored in such a way that a given edge (w, u) can be accessed in constant time with respect to i. (This may involve storing the edges in an array which is the size of the number of nodes currently in the GSS as discussed in Sect. 9.3). These arrays are only needed for the nodes in the current U_i and their total size is at most $(H+N)H(i+1)$.) So the order of COMPLETE_REDUCTION is constant, the order of REDUCER(i) is at most i, and at each step of the algorithm REDUCER(i) is executed once for each element added to $\mathcal{R}$, which is at most $K(H+N)H(i+1)$ times. Thus, since there are at most $n+1$ steps in the algorithm, the BRNGLR algorithm is at most $O(n^3)$.

7.3 The correctness of the BRNGLR algorithm

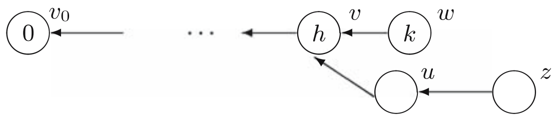
We define the BRNGLR algorithm to be *correct* if, given any CFG Γ and any input string u, it terminates and reports success if $u \in L(\Gamma)$ and terminates and reports failure otherwise. We have already shown that the algorithm is at most $O(n^3)$ so it terminates. The correctness proof relies on the fact, which is proved in [30], that the RN table is correct in the sense that $u \in L(\Gamma)$ if and only if u is accepted by the RN table for Γ. We begin by proving a lemma which characterises the situation in which there can be an edge between two GSS nodes which lie in the same U_i.

Lemma 1 *Let G be a GSS constructed by the BRNGLR algorithm from an RN table $\mathcal{T}$ and input $a_1 \cdots a_n$, and suppose that there is an edge (w, v) in G, where $w \in U_i$. Then $v \in U_i$ if and only if the edge (w, v) was created by the REDUCER when processing an element $(v, q, 0)$ (so w is a state node and q is of the form $X ::= \cdot\delta$). Furthermore, in this case if $rp \in \mathcal{T}(h, a_{i+1})$ where h is the label of w and p is $Y ::= \alpha x \cdot \omega$, then $x = X$ and $rt \in \mathcal{T}(k, a_{i+1})$, where k is the label of v and t is $Y ::= \alpha \cdot X\omega$.*

Proof Suppose that $v \in U_i$, so the edge (w, v) must have been constructed as a result of processing an element (u, q, m). We prove that $m = 0$, and hence $v = u$, by induction on the order in which the GSS edges are constructed.

If (w, v) is the first edge to be constructed then we have $v = u = v_0$ and (u, q, m) must have been added to $\mathcal{R}$ at the start of the algorithm, so, since $rq \in \mathcal{T}(0, a_1)$, we have $m = 0$ as required.

If $m \neq 0$, (u, q, m) was added to $\mathcal{R}$ when an edge (z, u) was created.



(If $m = 1$ then $u = v$.) Since there is a path from u to v and $v \in U_i$ we must have $u \in U_i$ and, similarly, $z \in U_i$. The edge (z, u) was created before the edge (w, v), and so, by induction, it was created as a result of processing an element (u, q', g) from $\mathcal{R}$, where $g = 0$. But, looking at the definition of the REDUCER, we see that for (u, q, m) to have been added to $\mathcal{R}$ when (u, q', q) was processed we must have $g > 0$. Thus the edge (z, u) does not exist, and we have $m = 0$, as required.

Conversely, if (w, v) was created when processing an element $(u, q, 0)$ in the REDUCER we must have $u = v$ and, since this element would only have been added to $\mathcal{R}$ when u was created, that $v \in U_i$.

Now, if $rp \in \mathcal{T}(h, a_{i+1})$ then, since there is an edge (w, v) generated by $X ::= \cdot\delta$, from the LR(1) DFA construction we must have $x = X$ and the state k must contain the item $(Y ::= \alpha \cdot X\omega, a_{i+1})$. Since q is a reduction, $X \overset{*}{\Rightarrow} \epsilon$, so $Y ::= \alpha \cdot X\omega$ is a reduction and hence $rt \in \mathcal{T}(k, a_{i+1})$, as required.

Theorem 1 *Given any CFG Γ and any input string, $a_1 \cdots a_n$, the BRNGLR algorithm given in Sect. 7.1 returns success if and only if $u \in L(\Gamma)$.*

Proof If $n = 0$ then the BRNGLR algorithm returns success if and only if $S \overset{*}{\Rightarrow} \epsilon$, as required. Thus we shall suppose that $n \geq 1$ and show that the BRNGLR algorithm results in success if and only if there is an execution path on $a_1 \cdots a_n$ through the RN table $\mathcal{T}$ for Γ which results in accept. The result then follows from the correctness of the RN table.

Let G be the GSS constructed from $\mathcal{T}$ on input $a_1 \cdots a_n$, using the BRNGLR algorithm, let $a_{n+1} = \$$, and let v_0 be the base node of G, the first node constructed. We shall show that there is a path

$$\begin{tikzcd} \overset{v_0}{0} & \overset{v_1}{h_1} \arrow[l] & \cdots & \overset{v_{j-1}}{h_{j-1}} \arrow[l] & \overset{v_j}{h_j} \arrow[l] \end{tikzcd}$$

in G where $v_j \in U_i$ is a state node (so h_j is a row number of $\mathcal{T}$) if and only if there is an execution path $\theta_1 \cdots \theta_d$ through $\mathcal{T}$ on $a_1 \cdots a_i$ which results in the configuration $\{(0, h_1, \ldots, h_j), a_{i+1}\}$.

($\Rightarrow$): We suppose that there is a path $(v_0, \ldots, v_j)$ in G where $v_j \in U_i$ is a state node. There are no edges in G whose target node is a book-keeping node, thus h_q is an LR(1) DFA state for $1 \leq q \leq j$. The proof is by induction on the order in which the edges in G are created. If the path has no edges then we must have $v_j = v_0$ and the empty execution path results in the initial configuration which is of the required form.

Now suppose that the edge (v_q, v_{q-1}) was the last edge to be created, so $v_q \in U_i$, and that the result is true for all paths which contain only edges created before (v_q, v_{q-1}). Note, for $q \leq c \leq j$ there is a path from v_c to v_j, so $v_c \in U_i$.

If the edge (v_q, v_{q-1}) was created by the SHIFTER then, since it was the last edge in the path to be created, we must have $j = q$, $v_{j-1} \in U_{i-1}$ and $ph_j \in \mathcal{T}(h_{j-1}, a_i)$. By induction there is an execution path $\theta_1 \cdots \theta_{d-1}$ on $a_1 \cdots a_{i-1}$ which results in $\{(0, \ldots, h_{j-1}), a_i\}$ and we can take θ_d to be the step whose action is ph_j.

Thus we may suppose that the edge (v_q, v_{q-1}) was created by REDUCER(i) while processing an element (z_1, t, m) where z_1 has label k_1 and t is $Y ::= \alpha \cdot \omega$, say. Since v_q is a state node we must have $m \leq 2$ and $ph_q \in \mathcal{T}(h_{q-1}, Y)$. First we show that there is an execution path through $\mathcal{T}$ which results in $\{(0, \ldots, h_{q-1}, h_q), a_{i+1}\}$.

If $m \leq 1$ then $z_1 = v_{q-1}$. If $m = 0$ then $rt \in \mathcal{T}(h_{q-1}, a_{i+1})$. By induction there is an execution path through $\mathcal{T}$, which results in $\{(0, \ldots, h_{q-1}), a_{i+1}\}$, which can be extended to result in the required configuration with the step whose action is rt.

If $m \geq 1$ then there is an edge (u_1, z_1) which was created when (z_1, t, m) was added to $\mathcal{R}$. Let l_1 be the label of u_1. If $m = 1$ then $z_1 = h_{q-1}$, u_1 must be a state node and

$rt \in \mathcal{T}(l_1, a_{i+1})$. By induction, there is an execution path Φ through $\mathcal{T}$ which results in $\{(0, \dots, h_{q-1}, l_1), a_{i+1}\}$ and we can extend Φ with the step whose action is rt. Thus we may suppose that $m = 2$, and hence that v_{q-1} is a child of z_1.

$$\begin{tikzcd}
\overset{v_0}{0} & \overset{v_1}{h_1} \arrow[l] & \cdots & \overset{v_{q-1}}{h_{q-1}} \arrow[l] & \overset{v_q}{h_q} \arrow[l] & \\
 & & & & \overset{z_1}{k_1} \arrow[ul] & \overset{u_1}{\bigcirc} \arrow[l]
\end{tikzcd}$$

If u_1 is a book-keeping node $(z_1, t, 2)$ was added to $\mathcal{R}$ while processing an element $(z_2, t, 3)$, where z_2 has label k_2, say, and child z_1. There must also be an edge (u_2, z_2) which was created when $(z_2, t, 3)$ was added to $\mathcal{R}$. If u_2 is a book-keeping node we can carry on in this way constructing book-keeping nodes u_g such that there are processes $(z_{g+1}, t, g+2)$. Since $(g+2)$ must be at most the length of the right hand side of the longest rule for Y, there must be some c such that u_c is a state node, with label l say. (If u_1 is a state node then $c = 1$.)

$$\begin{tikzcd}
\overset{v_0}{0} & \cdots & \overset{v_{q-1}}{h_{q-1}} \arrow[l] & \overset{z_1}{k_1} \arrow[l] & \overset{z_2}{k_2} \arrow[l] & \cdots & \overset{z_c}{k_c} \arrow[l] \\
 & & & \overset{u_1}{\bigcirc} \arrow[u] & \overset{u_2}{\bigcirc} \arrow[u] & & \overset{u_c}{l} \arrow[u]
\end{tikzcd}$$

Since all the edges in the above path were created before (v_q, v_{q-1}), by induction there is an execution path Θ through $\mathcal{T}$ which results in the configuration $\{(0, \dots, h_{q-1}, k_1, \dots, k_c, l), a_{i+1}\}$. Since u_c is a state node and $(z_c, t, c+1)$ is added to $\mathcal{R}$ at this point, we must have $rt \in \mathcal{T}(l, a_{i+1})$. Thus we can extend Θ with a step whose action is rt to result in $\{(0, \dots, h_q), a_{i+1}\}$.

Now, let Θ' be an execution path through $\mathcal{T}$ which results in $\{(0, \dots, h_q), a_{i+1}\}$. We have $v_q, \dots, v_j \in U_i$ and so, by Lemma 1, for $q < c \le j$ the edges (v_c, v_{c-1}) were created when elements of the form $(v_{c-1}, t_c, 0)$ were processed. So $rt_c \in \mathcal{T}(h_{s-1}, a_{i+1})$ and we can take θ_s to have action rt_c. Then $\Theta'\theta_{q+1} \cdots \theta_j$ results in the required configuration.

($\Leftarrow$): We suppose that there is an execution path $\theta_1 \cdots \theta_d$ through $\mathcal{T}$ which results in $\{(0, \dots, h_j), a_{i+1}\}$, and prove the result by induction on d. If $d = 0$ then we must have $j = i = 0$ and the path with just the root node, $v_0 \in U_0$, is the required path. Thus we suppose that the result is true for $\Theta' = \theta_1 \cdots \theta_{d-1}$.

If the action associated with θ_d is acc then Θ' results in $\{(0, \dots, h_j), \$\}$ and by induction there is a path of the required form in the GSS.

If the action associated with θ_d is pl then we must have $l = h_j$, $ph_j \in \mathcal{T}(h_{j-1}, a_i)$ and Θ' results in $\{(0, \dots, h_{j-1}), a_i\}$. By induction there is a path

$$\begin{tikzcd}
\overset{v_0}{0} & \overset{v_1}{h_1} \arrow[l] & \cdots & \overset{v_{j-1}}{h_{j-1}} \arrow[l]
\end{tikzcd}$$

in the GSS where $v_{j-1} \in U_{i-1}$. When the node v_{j-1} is created (v_{j-1}, h_j) will be added to $\mathcal{Q}$, and then eventually the edge (v_j, v_{j-1}) will be created, as required.

Now suppose that the action associated with θ_d is rt, where t is $Y ::= x_1 \cdots x_m \cdot \omega$, so that Θ' results in $\{(0, \dots, h_{j-1}, k_1, \dots, k_m), a_{i+1}\}$, $rt \in \mathcal{T}(k_m, a_{i+1})$ and $ph_j \in \mathcal{T}(h_{j-1}, Y)$. By induction there is a path

$$\begin{tikzcd}
\overset{v_0}{0} & \overset{v_1}{h_1} \arrow[l] & \cdots & \overset{v_{j-1}}{h_{j-1}} \arrow[l] & \overset{u_1}{k_1} \arrow[l] & \cdots & \overset{u_{m-1}}{k_{m-1}} \arrow[l] & \overset{u_m}{k_m} \arrow[l]
\end{tikzcd}$$

in G with $u_m \in U_i$. Let $u_0 = v_{j-1}$.

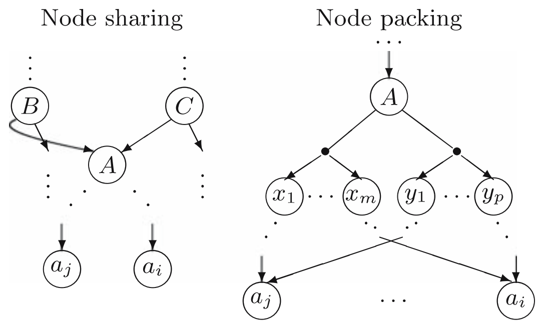
If $v_{j-1} \in U_i$ then $u_c \in U_i$ for $1 \leq c \leq m$. Since $rt \in \mathcal{T}(k_m, a_{i+1})$ by Lemma 1 we have $rt_c \in \mathcal{T}(k_c, a_{i+1})$, where $t_m = t$ and t_c is $Y ::= x_1 \cdots x_c \cdots x_m \omega$, and $rt_0 \in \mathcal{T}(h_{j-1}, a_{i+1})$. Thus $(v_{j-1}, t_0, 0)$ will be added to $\mathcal{R}$ when v_{j-1} is created and when this element is processed the edge (v_j, v_{j-1}) will be constructed.

Otherwise let g be the smallest integer such that $u_g \in U_i$. Then, as above, since $u_c \in U_i$, $g \leq c \leq m$, we have $rt_g \in \mathcal{T}(k_g, a_{i+1})$. Since u_{g-1} does not lie in U_i, by Lemma 1 the edge (u_g, u_{g-1}) must have been created when an element (u', q', f), $f \geq 1$, is processed by REDUCER(i) and at this point (u_{g-1}, t_g, g) will be added to $\mathcal{R}$. Furthermore, all the edges (u_b, u_{b-1}), $1 \leq b \leq g-1$, must have been created before step i of the algorithm. When (u_{g-1}, t_g, g) is processed, if $g = 1$ then $v_{j-1} = u_{g-1}$ and if $g = 2$ then $v_{j-1} = u_{g-2}$. In either case, (v_j, v_{j-1}) will be added to G by COMPLETE_REDUCTION. If $g \geq 3$ then when (u_{g-1}, t_g, g) is processed, since (u_{g-1}, u_{g-2}) must exist, $(u_{g-2}, t_g, g-1)$ will be added to $\mathcal{R}$. Carrying on in this way, eventually $(u_1, t_g, 2)$ will be added to $\mathcal{R}$ and when this element is processed the edge (v_j, v_{j-1}) will be constructed, as required.

8 The BRNGLR parser

We turn the BRNGLR algorithm into a parser by constructing a shared packed parse forest (SPPF) in the same way as for the RNGLR algorithm, see [30].

An SPPF is a representation designed to reduce the space required to represent multiple derivation trees for an ambiguous sentence. In an SPPF, nodes which have the same tree below them are shared and nodes which correspond to different derivations of the same substring are combined by creating a packed node for each *family* of children.

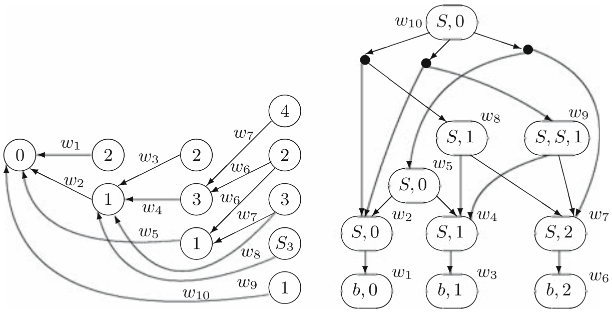


The addition of book-keeping nodes to the GSS means that there are some additional corresponding *intermediate* SPPF nodes. If such an SPPF node, z, is constructed at step i then we have a rule $A ::= x_1 \cdots x_p \beta$, where the subtree of the SPPF which has root z corresponds to a derivation $\beta \overset{*}{\Rightarrow} a_{j+1} \cdots a_i$.

Nodes can be packed only if their yields correspond to the same portion of the input string. Thus, to make it easier to determine whether two alternates can be packed under a given node, we store with the node the index of the left-most portion of the input string to which it corresponds. So the nodes in the SPPF are labelled with a pair (x, j) where x is a grammar symbol (or a left context for intermediate nodes) and j is an integer.

To construct an SPPF at the same time as the GSS we use an approach based on that developed by Rekers [28] for his implementation of Farshi's algorithm. Each edge in the GSS is associated with a node in the SPPF; edges created by SHIFTER(i) are associated with the leaf node labelled (a_{i+1}, i) and edges created by REDUCER(i) are associated with internal SPPF nodes.

Recall Example 1 from Sect. 5. The GSS and SPPF constructed by the BRNGLR parser for this example are shown below. The SPPF node w_9 is an intermediate node, created because the reduction $S ::= SSS$ is applied in two steps. The edges of the GSS are labelled with the corresponding SPPF nodes. The three packed nodes under w_{10} correspond to the three different derivations of *bbb*.



For an intermediate SPPF node the situation with respect to packed nodes is slightly different. For a rule $A ::= \alpha\beta$, where $|\alpha| \geq 1$ and $|\beta| \geq 2$, if $\beta \overset{*}{\Rightarrow} a_{j+1} \cdots a_i$ then from a parse containing this derivation the SPPF will have a corresponding intermediate node constructed at step i. If there is another rule $A ::= \alpha\gamma$ where $\gamma \overset{*}{\Rightarrow} a_{j+1} \cdots a_i$, then this derivation can be attached to the same SPPF node. However, another rule $B ::= \delta\beta$ which is also applicable at this point may not share the node. In other words, right hand portions of reduction rules have left contexts and are thus are not context-free. Thus we pack SPPF intermediate nodes only if the left contexts of the rules are the same.

To achieve this we label intermediate nodes with the left hand context of the rule and an integer, that is triples (A, α, j) where $0 \leq j \leq n$ and α is such that there is a rule of the form $A ::= \alpha\beta$, where $|\alpha| \geq 1$ and $|\beta| \geq 2$.

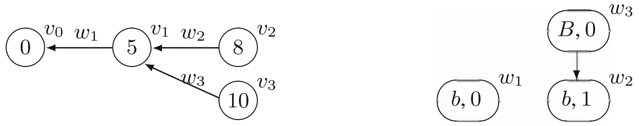
Example 3 Consider the grammar Γ_3 which has the following rules and RN table.

1. $S ::= A\ B$	2. $S ::= a\ B$	3. $S ::= b\ B\ b$
4. $S ::= b\ b\ b$	5. $A ::= a$	6. $B ::= b$

	\$	a	b	A	B	S
0		$p2$	$p5$	$p4$		$p1$
1	*acc*					
2			$r5$		$p7$	
3	$r6$					
4			$p3$		$p6$	
5			$p8$		$p10$	
6	$r1$					
7	$r2$					
8			$r6/p9$			
9	$r4$					
10			$p11$			
11	$r3$					

Parsing the string *bbb*, we begin by constructing GSS nodes v_0 labelled 0 and v_1 labelled 5, an SPPF node w_1 labelled $(b, 0)$ and an edge (v_0, v_1) labelled w_1. Applying $p8$ from

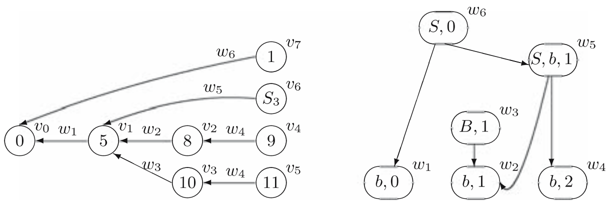
v_1, we create a new SPPF node w_2 labelled (b, 1), a GSS node v_2 labelled 8, and an edge (v_2, v_1) labelled w_2. From v_2 we apply $r6$, create an SPPF node w_3 labelled w_3, a GSS node v_3 labelled 10 and an edge (v_3, v_1) labelled (B, 0).



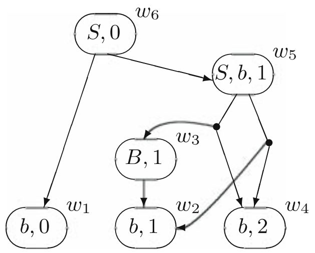
Then we read the final input symbol, b, create an SPPF node, w_4, labelled (b, 1), GSS nodes v_4 and v_5 labelled 9 and 11, and edges (v_4, v_2) and (v_5, v_3) labelled w_4.

From v_4 we apply the reduction $r4$ that has length 3, so we create an intermediate SPPF node w_5 labelled (S, b, 1), a new GSS node v_6 labelled S_3 and an edge (v_6, v_1) labelled w_5.

We then apply the second step of the reduction by $r4$, from v_6, we create a new SPPF node w_6 labelled (S, 0), a GSS node v_7 labelled 1 and an edge (v_7, v_0) labelled w_6.



When we apply $r3$ from v_5 we find that there is already node v_6 labelled S_3 and an edge (v_6, v_1), labelled w_5. So we simply add a new family of children (w_3, w_4) to w_5. This completes the parse and results in the SPPF



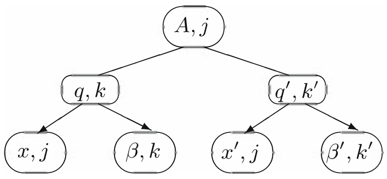
8.1 Sharing nodes in the SPPF

In order to share SPPF nodes we need to check for the prior existence of a node to which an additional family of children can be added. Checking avoids duplication of nodes, but, for input of length n, there are potentially $O(n^2)$ nodes in a BRNGLR generated SPPF and each node can have $O(n)$ families of children. Thus care needs to be taken when checking for the existence of a particular node or a particular family of children.

In the RNGLR parser [30] we maintain a set $\mathcal{N}$ of SPPF nodes constructed at the current step of the algorithm. Correspondingly, the BRNGLR parser maintains two sets $\mathcal{N}$ and $\mathcal{I}$ containing symbol and intermediate SPPF nodes, respectively. To achieve constant search time, our GTB implementation of the BRNGLR parser $\mathcal{N}$ and $\mathcal{I}$ are represented as dynamically allocated arrays. For more details of this see Sect. 9.2.

This still leaves the problem of searching the families of children of an existing node. Even for the SPPFs constructed by the BRNGLR algorithm the number of families of children can be of order n, this is the case, for example, for Γ_1. Thus the checking could make parser version of the BRNGLR algorithm have order $O(n^4)$.

For an SPPF node constructed by the BRNGLR algorithm, a family of children is defined by the creating reduction and the mid-index k which labels the second child in the set. (If $\beta = \epsilon$ then we set k to be the current step number.) So, for example, if there are two rules $A ::= x\beta$ and $A ::= x'\beta', q$ and q' say, with $x\overset{*}{\Rightarrow}a_{j+1}\cdots a_k$, $\beta\overset{*}{\Rightarrow}a_{k+1}\cdots a_i$, $x'\overset{*}{\Rightarrow}a_{j+1}\cdots a_{k'}$, $\beta'\overset{*}{\Rightarrow}a_{k'+1}\cdots a_i$, we have



Each packed node is labelled with the number of reduction that generated its children and the mid-index k. Thus all the possible families of children can be represented in an array of order n^2, allowing constant search time. There are other issues to be addressed in any actual implementation of the algorithm. For example, to ensure that the algorithm runs in linear time on LR(1) grammars, the arrays should only be allocated when they are needed. Additional discussion of our GTB BRNGLR implementation can be found in Sect. 9.2, but the full technical details required for practical implementations are not discussed in this, primarily theoretically oriented, paper.

8.2 Right nullable rules

For grammars that contain right nullable rules the full SPPF will not always be constructed directly because the nullable right hand ends are short circuited. Thus we pre-construct SPPFs for the nullable right hand ends of rules and add these in the appropriate places once the GSS construction has been completed. For a nullable string ω we call the SPPF of all the derivations $\omega\overset{*}{\Rightarrow}\epsilon$ the ϵ-SPPF for ω. We provide ϵ-SPPFs for all the nullable non-terminals and for nullable strings ω such that $|\omega| > 1$ and there is a grammar rule of the form $A ::= \alpha\omega$ where $\alpha \neq \epsilon$. We call the latter strings *required nullable parts*. (For rules of the form $A ::= \omega$ we use the ϵ-SPPF for A.) We share common nodes, merging these SPPFs into a single ϵ-SPPF.

Given a grammar we index, starting at 1, the nullable non-terminals and the required nullable parts. We let I be this indexing function and we define $I(\epsilon) = 0$. We label $u_{I(\omega)}$ the node of the ϵ-SPPF which is the root node of the ϵ-SPPF for ω, and u_0 is the node labelled ϵ.

8.3 The BRNGLR parsing algorithm

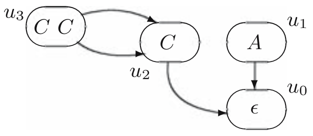
Input: an RN table $\mathcal{T}$, input string $a_1\cdots a_n$ and the ϵ-SPPF, whose nodes are labelled $u_{I(\omega)}$ as described above.

```
PARSER {
 **if** $n = 0$ { **if** $acc \in \mathcal{T}(0, \$)$ {
            report SUCCESS
            output the SPPF whose root node is $u_{I(S)}$ }
          **else** report FAILURE }
 **else** {
  create a node $v_0$ labelled 0
  set $U_0 = \{v_0\}$, $\mathcal{R} = \emptyset$, $\mathcal{Q} = \emptyset$, $a_{n+1} = \$$, $U_1 = \emptyset, ..., U_n = \emptyset$
  if $pk \in \mathcal{T}(0, a_1)$ add $(v_0, k)$ to $\mathcal{Q}$
  forall $rq \in \mathcal{T}(0, a_1)$ add $(v_0, q, 0, \epsilon)$ to $\mathcal{R}$
  for $i = 0$ to $n$  while $U_i \neq \emptyset$ do {
          set $\mathcal{N} = \emptyset$    $\mathcal{I} = \emptyset$
          while $\mathcal{R} \neq \emptyset$ do REDUCER($i$)
          SHIFTER($i$) }
  if the label of some $w \in U_n$ contains $(S' ::= S\cdot, \$)$ {
    let root be the SPPF node that labels the GSS edge $(w, v_0)$
    remove nodes not reachable from root and report SUCCESS }
  else report FAILURE } }

REDUCER($i$) {
 remove $(v, q, m, y)$ from $\mathcal{R}$
 let $q$ be $X ::= \alpha \cdot \omega$
 if $m = |\alpha|$ let $f = I(\omega)$ else let $f = 0$
 /* so $u_f$ is the root of the $\epsilon$-SPPF for $\omega$ */
 if $m = 0$ {
   let $k$ be the label of $v$ and let $pl \in \mathcal{T}(k, X)$
   if there is a node $w \in U_i$ with label $l$ {
     if there is not an edge $(w, v)$ {
       create an edge $(w, v)$ labelled $u_{I(X)}$ } }
   else {
     create a GSS node $w$ labelled $l$ and an edge $(w, v)$ labelled $u_{I(X)}$
     if $ph \in \mathcal{T}(l, a_{i+1})$ add $(w, h)$ to $\mathcal{Q}$
     forall $rt \in \mathcal{T}(l, a_{i+1})$ with $|t| = 0$ add $(w, t, 0, \epsilon)$ to $\mathcal{R}$ } }
 if $m = 1$ {
  suppose that $v \in U_{j'}$
  COMPLETE_REDUCTION($v, X, i, j'$)
  let $z$ be the node in $\mathcal{N}$ labelled $(X, j')$
  ADD_CHILDREN($z$, $(y)$, $f$) }
 if $m = 2$ {
  forall children $u$ of $v$ {
      suppose that $u \in U_j$
      COMPLETE_REDUCTION($u, X, i, j$)
      let $x$ be the label of the edge $(v, u)$
      let $z$ be the node in $\mathcal{N}$ labelled $(X, j)$
      ADD_CHILDREN($z$, $(x, y)$, $f$) } }
 if $m \geq 3$ {
  if there is not a GSS node $w \in U_i$ with label $X_m$ create one
  forall children $u$ of $v$ {
    let $x$ be the label of $(v, u)$
    suppose that $u \in U_j$
    let $\alpha'$ be the left hand prefix of $\alpha$ of length $m - 2$
    if there is no node $z \in \mathcal{I}$ labelled $(X, \alpha', j)$ {
        create an SPPF node $z$ labelled $(X, \alpha', j)$
        add $z$ to $\mathcal{I}$ }
    let $z$ be the node in $\mathcal{I}$ labelled $(X, \alpha', j)$
    if there is not an edge $(w, u)$ {
        create an edge $(w, u)$ labelled $z$
        add $(u, q, m - 1, z)$ to $\mathcal{R}$ }
    ADD_CHILDREN($z$, $(x, y)$, $f$) } }

SHIFTER($i$) {
 if $i \neq n$ {
   set $\mathcal{Q}' = \emptyset$
   create a new SPPF node $z$ labelled $(a_{i+1}, i)$
   while $\mathcal{Q} \neq \emptyset$ do {
     remove an element $(v, k)$ from $\mathcal{Q}$
     if there is $w \in U_{i+1}$ with label $k$ {
       create an edge $(w, v)$ labelled $z$
       forall $rt \in \mathcal{T}(k, a_{i+2})$ with $|t| \neq 0$, add $(v, t, |t|, z)$ to $\mathcal{R}$ }
     else {
       create a node $w \in U_{i+1}$ labelled $k$ and an edge $(w, v)$ labelled $z$
       if $ph \in \mathcal{T}(k, a_{i+2})$ add $(w, h)$ to $\mathcal{Q}'$
       forall $rt \in \mathcal{T}(k, a_{i+2})$ with $|t| \neq 0$, add $(v, t, |t|, z)$ to $\mathcal{R}$
       forall $rt \in \mathcal{T}(k, a_{i+2})$ with $|t| = 0$, add $(w, t, 0, z)$ to $\mathcal{R}$ } }
  copy $\mathcal{Q}'$ into $\mathcal{Q}$ } }

COMPLETE_REDUCTION($u$, $X$, $i$, $j$) {
    let $k$ be the label of $u$ and let $pl \in \mathcal{T}(k, X)$
    if there is no node $z$ in $\mathcal{N}$ labelled $(X, j)$ {
              create an SPPF node $z$ labelled $(X, j)$
              add $z$ to $\mathcal{N}$ }
   if there is a node $w \in U_i$ with label $l$ {
     /* if $(w, u)$ exists it will be labelled $z$ */
     if there is not an edge $(w, u)$ {
       create an edge $(w, u)$ labelled $z$
       forall $rt \in \mathcal{T}(l, a_{i+1})$ with $|t| \neq 0$ add $(u, t, |t|, z)$ to $\mathcal{R}$ } }
   else {
     create a GSS node $w$ labelled $l$ and an edge $(w, u)$ labelled $z$
     if $ph \in \mathcal{T}(l, a_{i+1})$ add $(w, h)$ to $\mathcal{Q}$
     forall $rt \in \mathcal{T}(l, a_{i+1})$ with $|t| \neq 0$ add $(u, t, |t|, z)$ to $\mathcal{R}$
     forall $rt \in \mathcal{T}(l, a_{i+1})$ with $|t| = 0$ add $(w, t, 0, \epsilon)$ to $\mathcal{R}$ } }

ADD_CHILDREN($z$, $\Delta$, $f$) {
  if $f \neq 0$ let $\Upsilon = (\Delta, u_f)$
  else let $\Upsilon = \Delta$
  if $z$ has no children then add edges from $z$ to each node in $\Upsilon$
  if $z$ has a child which is not a packed node {
    if the children of $z$ are not the sequence $\Upsilon$ {
      create a new packed node $p$ and a tree edge from $z$ to $p$
      add tree edges from $p$ to all the other children of $z$
      remove all tree edges from $z$ apart from the one to $p$
      create a new packed node $p'$ and a tree edge from $z$ to $p'$
      add tree edges from $p'$ to each node in $\Upsilon$ } }
  if $z$ has a child which is a packed node {
    if $z$ does not have child with a sequence of children labelled $\Upsilon$ {
      create a new packed node $p'$ and a new tree edge from $z$ to $p'$
      create new edges from $p'$ to each node in $\Upsilon$ } } }
```

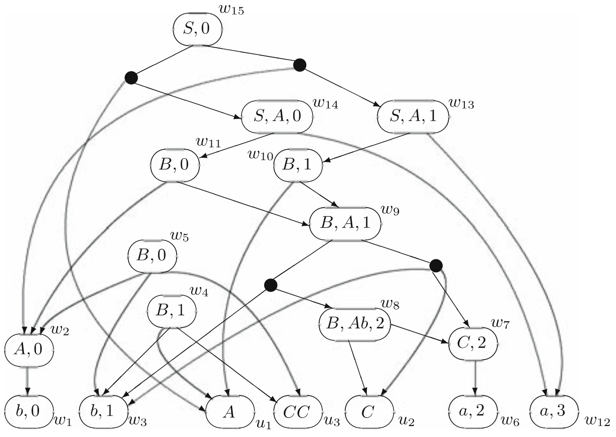
For example, consider again the grammar Γ_2, from Sect. 6

$$S ::= A\ B\ a \quad A ::= b \mid \epsilon \quad B ::= A\ b\ C\ C \quad C ::= a\ a \mid a \mid \epsilon$$

whose required nullable part is CC. We have the following pre-compiled ϵ-SPPF, where $I(A) = 1$, $I(C) = 2$ and $I(CC) = 3$.



Running the algorithm with $\mathcal{T}_2$ and input $bbaa$ results in the following SPPF, from which the final SPPF can be obtained by substituting the nodes u_1 and u_2 and removing the unreachable nodes w_4, w_5 and u_3.



8.4 The order of the BRNGLR parser

Although we have shown that the BRNGLR recogniser is at most $O(n^3)$, as we have discussed above, it does not follow directly that the corresponding parser is cubic. In this section we shall show that, since the modification that we have made to the algorithm to produce GSS searches of length at most 2 also produces, automatically, SPPFs which are in some sense binary, the size of the SPPF is always at most $O(n^3)$. We shall also show that the additional actions required to construct the SPPF need not increase the order of the BRNGLR algorithm. We assume that the input string is $a_1 \cdots a_n$ and that the input table is the RN table for a grammar Γ.

The SPPF contains four types of node: *packed nodes*, *nullable nodes* which are nodes from the ϵ-SPPF, *symbol nodes* with a label of the form (x, j) where x is a grammar symbol, and *intermediate nodes* labelled (A, α, j). Note, if a symbol node (x, j) is constructed at step i of the algorithm and labels the edge (w, v) then $w \in U_i$ and $v \in U_j$. Furthermore, the use of the sets $\mathcal{N}$ and $\mathcal{I}$ ensure that there is at most one node in the SPPF with a given label constructed at each step.

A GSS leaf node which is not an ϵ-SPPF node, is labelled $(a_i, i-1)$ and there are clearly n of these. Interior SPPF nodes are either ϵ-SPPF nodes or are labelled (A, j) or (A, α, j). Non-ϵ-SPPF nodes must be constructed by REDUCER(i) where $j \leq i$. In fact, using Lemma 1, it can be seen that we must have $j < i$. Let M denote the number of non-terminals in Γ and let G denote the number of pairs (A, α) where there is some rule $A ::= \alpha\beta$ with $|\alpha| \geq 1$ and $|\beta| \geq 2$. The number of ϵ-SPPF nodes is a constant, E say. So, if $K(n)$ is the number of non-ϵ-SPPF packed nodes, the SPPF contains at most the following number of symbol nodes

$$n + E + \sum_{i=0}^{n} i(M+G) + K(n) = n + \frac{(M+G)n(n+1)}{2} + K(n)$$

We now put an upper bound on the number of SPPF edges. The ϵ-SPPF has at most E^2 edges, and the nodes labelled $(a_l, l-1)$ do not have any children. Looking at ADD_CHILDREN we see that packed nodes have at most three children. An interior symbol node has either at most three children, none of which is a packed node, or all of its children are packed nodes. Each packed node has only one parent thus there are at most 4 edges whose source or target node is a given packed node. The other edges must have source nodes which are symbol nodes with at most three children, thus there are at most

$$E^2 + \frac{3(M+G)n(n+1)}{2} + 4K(n)$$

edges in the SPPF, where $K(n)$ is the number of non-ϵ-SPPPF packed nodes.

We show that the SPPF has size at most $O(n^3)$ by showing that $K(n)$ is at most $O(n^3)$. Each packed node is the child of some symbol or intermediate node, z, labelled (A, j) or (A, α, j), and there are $O(n^2)$ of these. We show that such a node constructed at step i has at most $O(i)$ non-ϵ-SPPF packed nodes as children.

Each packed node is uniquely defined by its children, which must be of one of the forms (y), (y, u_f), (x, y) or (x, y, u_f), where u_f is a nullable node. Looking at REDUCER(i), for $(u, q, m, y) \in \mathcal{R}$ we have that y is an intermediate node only if $m \geq 2$. Furthermore, intermediate nodes cannot label GSS edges which have predecessors. Thus intermediate nodes can only arise as the second child of an SPPF node. Also, the right-most non-ϵ-SPPF child of any node must have been constructed at the current step. Thus for children of the form (y) and (y, u_f) the label of y is either $(a_i, i-1)$ or (A, j) and thus, since $j < i$, there are at most $(iM + 1)$ choices for y. For children of the form (x, y) or (x, y, u_f) we have that x is a symbol node labelled (A, j) or (a_{j+1}, j) or an ϵ-SPPF node, and y is a symbol or intermediate node labelled (B, k), (B, β, k) or $(a_i, i-1)$, for some k such that $j \leq k < i$. If x is an ϵ-SPPF node then $k = j$ and if x is (a_{j+1}, j) then $k = j+1$, so in both cases there are at most $M + G$ choices for y. If x is a symbol node then $j < k < i$ so there are $(i - j - 1)(M + G)$ choices for y. This gives a total of

$$((iM + 1) + (E + 1)(M + G) + M(i - j - 1)(M + G))(1 + E)$$

possible families of children of a node labelled (A, j) or (A, α, j) constructed at step i.

The only significant differences between the BRNGLR recogniser and parser are that REDUCER and COMPLETE_REDUCTION maintain the sets $\mathcal{N}$ and $\mathcal{I}$, and make calls to the ADD_CHILDREN function. A call to function ADD_CHILDREN by REDUCER(i) results in the creation of at most two packed nodes and eight edges, and the removal of at most three edges, thus the only aspect of the function which may not be constant with respect to i is the check for existing families of children of the node z and the membership of z in $\mathcal{N}$

or $\mathcal{I}$. As we discussed in Sect. 8.1 the sets $\mathcal{N}$ and $\mathcal{I}$ can be implemented as arrays of order at most n^2 which allow constant search time. So we have the following result.

Theorem 2 *The order of the BRNGLR parser is at most $O(n^3)$, where n is the length of the input string, it requires at most $O(n^2)$ space for its internal operation, and it constructs an SPPF representation of size at most $O(n^3)$ of the set of derivations of a sentence.*

Note: We do not want to construct internal structures of order $O(n^2)$ which will not be needed, for example when the grammar is LR(1). Thus the algorithm should be implemented so that the array corresponding to multiple families of child nodes of a particular node is only initialised when the second set of children for that node is encountered.

9 Experimental results

This paper is primarily a theoretical paper, presenting a parsing algorithm which is proved to be both correct and of worst-case cubic order, and is of linear order on deterministic grammars. Of course, this would be of little interest if the constants of proportionality made the algorithm impractical, but our claim is that, on contemporary hardware, BRNGLR is a practical alternative to conventional LALR parsers such as those generated by YACC; and yet provides fully general context free parsing power. In this section we give experimental data to support this claim.

Both the BRNGLR and RNGLR algorithms have been implemented in Java and in C, following closely the theoretical description. The Java implementation is part of the PAT tool [19], designed with dynamic graph visualisation for pedagogic purposes in mind. The C implementation is part of the GTB tool [18] and focuses on the efficient implementation of the GSS, and the $\mathcal{N}$ and $\mathcal{I}$ data structures. Both of these implementations should be treated as prototypes: we discuss below straightforward changes to the memory management regimes that will deliver significant speed-ups in future versions.

As example grammars we use Γ_1 from Sect. 5 because it generates worst case behaviour for the BRNGLR algorithm and supra-cubic behaviour for the RNGLR algorithm. We also use the grammar from the ANSI-C standard, and the IBM VS-COBOL grammar extracted by Steven Klusener and Ralf Lämmel using the techniques described in [24]. The ANSI-C grammar is very close to deterministic, exhibiting only the well known `if-then-else` and `typedef` conflicts. The COBOL grammar is highly non-deterministic and might be thought of as a middle example between the YACC-friendly ANSI-C grammar and the highly ambiguous Γ_1.

We instrumented the algorithms so as to report the sizes of the data structures constructed. Actual run times for any algorithm are heavily dependent on the implementation and the hardware configuration whereas the order of an algorithm is a theoretical property based on the number of 'elementary operations' carried out. For GLR algorithms the order is dominated by the tracing of paths in the GSS, so we concentrate on the total number of GSS edge traversals, and show that this number is of cubic order for Γ_1. To show that cubic order of the BRNGLR algorithm is not cancelled out by other, possibly hidden, implementation costs we have also included actual timings generated using the GTB implementations of the algorithms.

9.1 BRNGLR parsing compared to YACC generated LALR parsers

Our reference implementation is part of the Grammar ToolBox (GTB) which is publicly available from [16]. We compiled GTB with the Intel compiler bundled with the Borland 5.01 C++ development suite and ran experiments on a 1.6 GHz Pentium-M (in full power mode) with 256 MByte of memory running under Windows XP patched to June 2006.

The ANSI-C test source includes `bool` (a Quine-McCluskey boolean minimisation tool which contains 1,081 lines yielding 4,291 lexemes) and the complete source code for GTB itself (which totals 36,827 lines of code, yielding 36,827 lexemes). See [13] for the source code for `bool`; [16] for the source code of GTB and [32] for more details of the tokenisation.

The GTB BRNGLR implementation parses `bool` in 0.06s and the full GTB source in 0.67s.

The widely used GNU `gcc` and `g++` compilers use a deterministic LALR parser generated by Bison, the successor to YACC. We could imagine a version of the GNU tools that used our BRNGLR parser instead of the Bison generated equivalent, but how much would `g++` slow down if it used a BRNGLR general parser?

We compiled the `bool` source code using `gcc` version 3.3.3 under Cygwin on the same system and used the Cygwin bash shell time function to measure runtimes, and compiled the GTB source code using the same system but with the `g++` compiler. We measured the time taken when using the `-O0` (default, no optimisation) options and the `-O3` (full optimisation) options: this table shows the number of CPU seconds for these compilations, and contrasts it with the time taken for the BRNGLR parse.

Source	`g++ -O0`	`g++-O3`	BRNGLR
`bool`	0.278	1.766	0.06
GTB	5.56	11.60	0.67

So, if the YACC-generated LALR parser within GNU C were replaced with a BRNGLR style parser, the user experience would not be markedly different. To reinforce this: the `g++` figures given here are average of a series of runs which show variations of up to 0.2 CPU seconds, probably as a result of disk cache history (no other user processes were running during these tests), and this effect dominates over the additional cost of the BRNGLR algorithm for medium sized compilation units of around 2,000 lines of ANSI-C.

We should also note here that there is considerable scope for improving the performance of our reference implementation since, at the moment, we use the general purpose graph handler from our RDP toolkit [14] to construct the GSS and SPPF data structures. The RDP graph handler is very general, and provides a great deal of support for debugging dynamic data structures which is a costly overhead in production code. Furthermore, use of the library means that each node and edge creation requires a separate call to the C++ standard run time heap manager. In a future implementation, we shall allocate one large buffer for these structures and reserve space for each node an edge simply by advancing a free-store pointer, and we believe that this will yield significant speed-ups.

9.2 The implementation of SPPF searches

Two kinds of SPPF search are required: (i) to see if the required SPPF node already exists and (ii) to see if a given SPPF node already has the required sequence of children, either directly or as the immediate children of a child packed node. In the BRNGLR parser the searching is limited to nodes created within the current frontier (as opposed to the whole SPPF) through the maintenance of two sets $\mathcal{N}$ and $\mathcal{I}$, see Sect. 8.1.

In detail, referring to the algorithm in Sect. 8.3, we can identify four kinds of operations related to $\mathcal{N}$ and $\mathcal{I}$.

1. Clearing both sets: $\mathcal{I} = \emptyset, \mathcal{N} = \emptyset$; PARSER inner loop line 12; executed n times.
2. Testing for membership:

   (a) to $\mathcal{N}$; COMPLETE_REDUCTION line 3; executed once for each full reduction processed.
   (b) to $\mathcal{I}$; REDUCER case $m \geq 3$ line 7; executed once for each intermediate reduction processed.
3. Adding an element:
   (a) to $\mathcal{N}$; COMPLETE_REDUCTION line 5; executed once for each full reduction processed.
   (b) to $\mathcal{I}$; REDUCER case $m \geq 3$ line 9; executed once for each intermediate reduction processed.
4. Testing to see if a sequence Υ is already a member of the family of SPPF node z; ADD_CHILDREN line 4.

Although the use of $\mathcal{N}$ and $\mathcal{I}$ limits searching to SPPF nodes created at the current main step, a naïve implementation of these operations (say, using linked lists to hold the sets) would impact the asymptotic performance since the cardinality of $\mathcal{N}$ and $\mathcal{I}$ at level i is potentially $O(i)$. Furthermore, each of these nodes can potentially have $O(i)$ families of children. Either of these two factors gives a potential search space of size $O(n)$.

Thus, we use tables to handle the $O(n)$ components of the SPPF search and a linked list for those searches whose worst-case time dependency is independent of n. We keep a table of booleans that records nodes that have been constructed at the current level, in which we can look up a particular set of children for a given node in $O(1)$ time. The children of an SPPF node are already organised as a linked list, and we arrange for these lists to be grouped by level of the destination SPPF node in the corresponding reduction.

The sets $\mathcal{I}$ and $\mathcal{N}$ are implemented within a single SPPF node table which is a two dimensional array of pointers to SPPF nodes; the array being indexed by $0 \leq j \leq n$ and $0 \leq x \leq |M| + |G|$, where M and G are as defined in Sect. 8.4.

The node table needs to be cleared at the start of each level, and this is itself an expensive operation if each individual bit needs to be reset. Instead, we keep an integer *valid* field in each second dimension entry, which must contain i (the number of the current level) for the bit vector to be valid. At the start of each level, the algorithm increments i, which has the effect of invalidating all edge table bit vectors. We also use the valid field to encode the reduction number of the first family added to an SPPF node. Once a second family is added, packed nodes are added between the parent SPPF node and the SPPF nodes in the family and the corresponding reduction number is encoded in the packed node directly. The pointers in the table reference the families of children that have a particular mid-index k (see the end of Sect. 8.1), and thus we can directly walk the sub-list of nodes for a particular mid-index. There can only be as many nodes in the sub-list as there are reductions, so the family check can be performed in $O(1)$ time.

9.3 The implementation of GSS edge searches

Whenever an edge is about to be added to an existing node in the GSS, a test is performed to check whether this edge already exists: see line 9 of COMPLETE_REDUCTION and line 11 of case $m \geq 3$ in REDUCER. In the worst-case there may be $O(n^3)$ such tests carried out during algorithm execution. There are worst-case $O(n)$ edges from a given GSS node. Therefore, if the edges are maintained on a list attached to each node, the total searching can be $O(n^4)$. We can avoid this by using tables analogous to the SPPF node tables described above.

An edge connects $w \in U_i$ to $u \in U_j$, say, and GSS nodes are labelled with a DFA state or are book-keeping nodes. We therefore need a table of maximum size $s^2 \times n$ where s is the number of DFA states plus the number of possible book-keeping nodes, and n is the length of the string. Simply allocating a three-dimensional table of this size is impractical for typical programming language applications. Instead, we initially allocate only a fixed length base vector of n pointers, and then allocate individual fixed-length vectors of s pointers and stretchable vectors of bits to handle the second and third dimensions.

As for the SPPF node table, we use a valid field to avoid having to clear the entire table at the start of each level. When the algorithm looks up an entry, if the valid field is less than i, then the lookup immediately returns false. When the algorithm creates a new edge, the corresponding valid field is first checked, and if it does not contain i then the bit vector is cleared before the new edge's bit is set.

9.4 GSS generation for Γ_1

Both the BRNGLR and the RNGLR algorithms were run on the grammar Γ_1 with input strings $b, b^2, \dots, b^{100}$ and b^{200}.

The use of the intermediate nodes increases the size of the GSS for the BRNGLR algorithm so, to compare the size of the two structures, the number of edges and nodes were recorded for each GSS. To compare the efficiency of the GSS construction, the number of edge visits made during reduction applications were counted. An edge is *visited* only when it is traversed as part of the path tracing required for the application of a reduction. In particular, the creation of an edge is not counted as a visit, and the first edge in any reduction path is not visited because of the way pending reductions are stored in the set $\mathcal{R}$. We have also given timings generated by our GTB implementation discussed in Sect. 9.1. The timings include SPPF as well as GSS construction. The results are given in the following table and the edge visits are also displayed graphically. There are more nodes and edges in the BRNGLR GSS, as expected, but it is only an increase by a small scalar multiple. The timings show a clear advantage for BRNGLR.

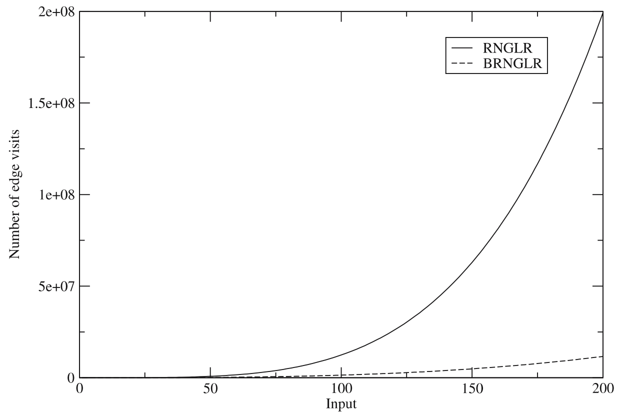
b^n	RNGLR GSS				BRNGLR GSS			
n	Edge visits	Nodes	Edges	CPUs	Edge visits	Nodes	Edges	CPUs
5	56	18	34	0	51	21	44	0
10	1,091	38	144	0	776	46	229	0
20	18,961	78	589	0.02	8,676	96	1,049	0
30	98,106	118	1,334	0.19	32,676	146	2,469	0.01
40	313,026	158	2,379	1.282	81,776	196	4,489	0.04
50	768,221	198	3,724	7.290	164,976	246	7,109	0.09
60	1,493,876	238	5,369	28.34	291,276	296	10,329	0.19
70	2,800,936	278	7,314	82.638	469,676	346	14,149	0.33
80	5,070,456	318	9,559	202.802	709,176	396	18,569	0.561
90	8,131,751	358	12,104	442.577	1,018,776	446	23,589	0.851
100	12,405,821	398	14,949	884.141	1,407,476	496	29,209	1.232
200	199,289,146	798	59,899	–	11,624,976	996	118,409	31.175

Because of the regularity of Γ_1 we can compute, by hand, the number of edge visits made by the BRNGLR algorithm on b^n. For $n \geq 3$, the number is

$$\frac{3n^3}{2} - \frac{19n^2}{2} + 25n - 24.$$

The experimental results obtained do indeed fit this formula. Microsoft Excel was used to generate two polynomial trend-lines for the experimental data gathered for both algorithms. Using the RNGLR trend-line we were able to generate the following formula for the number of RNGLR edge visits, which matches exactly the 101 data points. As expected, it is a quartic polynomial.

$$\frac{n^4}{8} - \frac{n^3}{12} - \frac{9n}{8} + \frac{49n}{12} - 4.$$



It is worth noting here that our unoptimised BRNGLR parser parsed b^{200} in less than 40 seconds, while the RNGLR parser ran out of space when building the SPPF for b^{200}, and the GLR version of Bison could not parse b^{20}.

9.5 SPPF generation for Γ_1

To illustrate the cubic order of the BRNGLR parser, data were also collected about the space required for the SPPFs of Γ_1 on the strings $b, \ldots, b^{100}$. The number of symbol nodes, intermediate nodes, packed nodes and edges in the SPPF were recorded. We expect the size of the SPPF generated for Γ_1 by the RNGLR algorithm to be quartic and cubic for the BRNGLR algorithm. (The timings for SPPF generation are included in the previous section as the SPPF is constructed concurrently with the GSS.)

b^n	BOTH	RNGLR SPPF		BRNGLR SPPF		
n	Symbol nodes	Packed nodes	Edges	Interm. nodes	Packed nodes	Edges
5	20	31	121	6	33	118
10	65	486	1,816	36	388	1,208
20	230	7,296	27,931	171	3,573	10,813
30	495	35,931	139,346	406	12,558	37,818
40	860	111,891	437,061	741	30,343	91,223
50	1,325	270,676	1,062,076	1,176	59,928	180,028
60	1,890	557,786	2,195,391	1,711	104,313	313,233
70	2,555	1,028,721	4,058,006	2,346	166,498	499,838
80	3,320	1,748,981	6,910,921	3,081	249,483	748,843
90	4,185	2,794,066	11,05,136	3,916	356,268	1,069,248
100	5,150	4,249,476	16,831,655	4,851	489,853	1,470,053
200	20,300	–	–	19,701	3,959,703	11,880,103

Again, data-generated trend-lines allowed us to calculate the following formulae (for $n \geq 3$) for the total number of SPPF nodes, symbol, intermediate and packed, created by the RNGLR and the BRNGLR algorithms, respectively

$$\frac{n^4}{24} + \frac{n^3}{12} + \frac{11n^2}{24} + \frac{5n}{12} + 1, \quad \frac{n^3}{2} - \frac{3n}{2} + 4.$$

9.6 Experimental results for ANSI-C and Cobol

Whilst the BRNGLR algorithm improves the worst-case performance, it is of course important that this is not at the expense of average case behaviour. To demonstrate that the BRNGLR algorithm is not less practical than the standard Tomita-style algorithms we have run both BRNGLR and RNGLR on a grammar for ANSI-C [21] and a grammar for Cobol.

The C test programs comprise the 4,291 lexeme Quine-McCluskey boolean minimiser, discussed in Sect. 9.1, modules with 5,805, 6,066, 10,109 and 16,610 lexemes which are part of the RDP implementation, and modules with 3,632, 4,747, and 10,324 lexemes from the GTB implementation. The 2,197 lexeme Cobol test program is the concatenation of a series of small test examples designed to provide good parser coverage.

	RNGLR GSS				BRNGLR GSS			
ANSI-C	Edge visits	Nodes	Edges	CPUs	Edge visits	Nodes	Edges	CPUs
3,632	3,954	23,516	23,676	0.04	1,930	25,446	25,659	0.05
4,291	4,502	28,048	28,172	0.05	4,498	25,446	25,659	0.06
4,747	4,502	28,048	28,172	0.06	5,036	33,387	33,670	0.07
5,805	6,278	43,993	44,304	0.07	6,276	46,777	47,302	0.10
6,066	6,405	40,626	40,793	0.07	6,405	43,538	43,856	0.09
10,109	16,155	84,852	86,955	0.14	16,155	89,681	96,747	0.19
10,324	12,689	80,959	81,868	0.13	12,685	86,041	888,549	0.18
16,610	21,412	127,007	129,557	0.22	21,400	135,082	141,101	0.28
Cobol	3,581	12,057	13,512	0.08	3,487	13,017	14,517	0.09

	BOTH	RNGLR SPPF		BRNGLR SPPF		
ANSI-C	Symbol nodes	Packed nodes	Edges	Interm. nodes	Packed nodes	Edges
3,632	23,238	54	23,622	1,983	54	24,287
4,291	27,870	58	28,146	2,062	58	30,197
4,747	30,689	100	31,108	2,428	100	33,529
5,805	43,934	30	44,437	2,998	30	47,433
6,066	40,476	40	40,871	3,063	40	43,934
10,109	86,569	4	92,621	9,792	4	102,413
10,314	81,192	98	79,929	6,681	98	90,366
16,610	128,162	81	132,964	11,544	81	144,498
Cobol	10,793	306	13,121	1,005	316	14,082

As we expected there are a similar number of edge visits for both algorithms and the sizes of the BRNGLR GSS and SPPF are comparable with the sizes of the corresponding RNGLR structures.

9.7 Grammar binary factorisation

We also compare the BRNGLR algorithm with what we shall call the BFRNGLR algorithm, which initially binary-factors the grammar so that all the rules have length at most two and then runs the RNGLR algorithm. The results in this and the previous section are for the C LALR(1) tables and for the Cobol SLR(1) tables. The following table shows the comparative sizes of the LR tables for the original grammars (used by both the RNGLR and BRNGLR algorithms) and for the factored grammars (used by the BFRNGLR algorithm). For ANSI-C the factored grammar has 154 additional non-terminals, and for Cobol the factored grammar has 767 additional non-terminals. As we discussed in Sect. 1, factoring the grammar in order to make the RNGLR algorithm cubic results in much larger parsers than the standard LR tables used by the BRNGLR algorithm. For ANSI-C the table has approximately 2.7 times as many cells and for Cobol the number of table cells goes up from 2.8×10^6 to 6.2×10^6.

Grammar	Original grammar	Factored grammar
C LALR(1) rows × columns	382 × 156	533 × 310
Cobol SLR(1) rows × columns	2692 × 1026	3459 × 1793

The run-time structures are comparable, although both the GSS and SPPF contain slightly more nodes and edges in the BFRNGLR case.

	BFRNGLR GSS				BFRNGLR SPPF		
ANSI-C	Edge visits	Nodes	Edges	CPUs	Symbol	Packed	Edges
3,632	3,951	25,469	25,671	0.04	25,234	54	25,613
4,291	4,596	30,417	30,761	0.06	25,869	58	30,483
4,747	5,059	33,458	33,778	0.07	33,188	100	33,620
5,805	6,277	46,826	47,303	0.09	46,933	30	47,435
6,066	6,405	43,591	43,855	0.08	43,539	40	43,934
10,109	16,155	89,939	96,747	0.19	96,361	4	102,413
10,324	12,688	86,167	88,552	0.18	87,876	98	90,372
16,610	21,409	135,408	141,108	0.30	139,713	82	144,513
Cobol	3,521	13,051	14,556	0.10	11,864	316	14,184

Finally we note that, although the table sizes for the binarised grammars are significantly larger, the GSS sizes for the binarised version of C and Cobol are only very slightly larger than those for the BRNGLR algorithm. This is to be expected because the *run time* space savings of the BRNGLR algorithm over the binarised grammar version are obtained when different reductions are performed down GSS paths which have common left hand ends, and this happens only in the non-deterministic parts of a grammar. To demonstrate GSS size reduction we can consider, for example, the grammar

$$S ::= a^k b \mid a^k B \qquad B ::= b \mid c$$

and input $a^k b$. The BRNGLR algorithm constructs a GSS with $2k + 3$ nodes and $2k + 2$ edges, while the BFRNGLR algorithm constructs a GSS with $3k + 2$ nodes and $3k + 1$ edges.

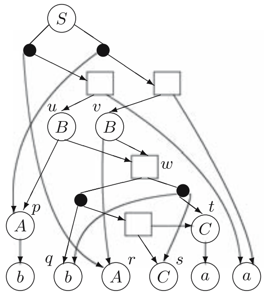
10 Conclusions

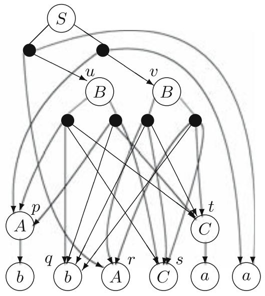
We have presented a GLR parsing algorithm and proved that it is correct and has worst-case cubic order. The algorithm is linear on LR(1) grammars. We have also presented experimental data to support our claims for the practicality of the algorithm.

Although we have a proof that the parser is $O(n^3)$ it is perhaps not obvious why the BRNGLR algorithm is of lower order than RNGLR and other Tomita-style algorithms, especially as it can construct larger data structures. The GSS constructed by any of these algorithms has at most $O(n)$ nodes and $O(n^2)$ edges, it is the searching of paths in the GSS which can cause GLR algorithms to be supra-cubic. When searching a GSS, a GLR algorithm can only tell that it has already searched paths of length k from a given node when it gets to the end of the path. For the grammar $S ::= \epsilon | S^k$, $k \geq 3$, there are nodes for which there are $O(n^{k-2})$ paths of length $k - 1$ which have the same final edge. When a reduction of length k is applied from such a node (the second node on the path) the final edge on the path is traversed $O(n^{k-2})$ times. For the BRNGLR algorithm, all reduction paths have length at most two, so the final edge is also the only edge and hence it is traversed only once. Intuitively then, we can see how the searching is reduced by n^{k-2}, from $O(n^{k+1})$ to $O(n^3)$.

With regard to tree construction, Tomita-style SPPFs can have $O(n^k)$ nodes, for any given integer k. For the Rekers-style SPPFs produced by the RNGLR algorithm, there are at most $O(n^2)$ symbol and intermediate nodes, but there can be $O(n^k)$ packed nodes. However, the sequence of children for two different packed nodes may be the same from the right of some position. The SPPFs produced by BRNGLR group the children together in pairs from the right, using intermediate nodes. Since intermediate nodes can be shared, nodes can effectively share common right-hand-sequences of children. As intermediate nodes can have packed nodes below them this sharing can reduce the number of packed nodes. This reduces the bound to $O(n^3)$.

We can see the effects of this sharing in the node counts for Γ_1 presented in Sect. 9.5. A small reduction in the number of packed nodes is seen when $n = 10$, but for $n = 100$ the effect is substantial. Unfortunately even for $n = 10$ the SPPFs are too large to reproduce here. However, we can illustrate the packed node reduction using Γ_2 and the string *bbaa* discussed in Sect. 8.3. The pruned BRNGLR-generated and RNGLR-generated SPPFs are shown below.





In the standard SPPF (on the right) the nodes u, v each have two families of children $\{p, q, s, t\}$, $\{p, q, t, s\}$, $\{r, q, s, t\}$, $\{r, q, t, s\}$, respectively. However, both nodes have the same two sub-families $\{q, s, t\}$, $\{q, t, s\}$ from the right of the first position. The intermediate node w in the BRNGLR SPPF groups these children together, and then both u and v can have just one family of children so the sub-families are not duplicated.

Of course, the BRNGLR SPPF contains additional, intermediate, nodes. However, the number of these is bounded by $O(n^2)$, the same as for the symbol nodes in both types of SPPF, as is illustrated in the table in Sect. 9.5. Thus the asymptotic SPPF size is not affected by the intermediate nodes.

References

1. Aycock, J., Horspool, N.: Faster generalised LR parsing. In: Compiler Construction, 8th International Conference, CC'99, vol. 1575 of Lecture Notes in Computer Science, pp. 32–46. Springer, Heidelberg (1999)
2. Aycock, J., Nigel Horspool, R., Janousek, J., Melichar, B.: Even faster generalized LR parsing. Acta Informatica **37**(8), 633–651 (2001)
3. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques and tools. Addison-Wesley, Reading (1986)
4. Breuer, P.T., Bowen, J.P.: A PREttier Compiler-Compiler: Generating higher-order parsers in C. Softw. Pract. Exp. **25**(11), 1263–1297 (1995)
5. Billot, S., Lang, B.: The structure of shared forests in ambiguous parsing. In: Proceedings of the 27th conference on Association for Computational Linguistics, pp. 143–151. Association for Computational Linguistics (1989)
6. Coppersmith, D.: Rectangular matrix multiplication revisited. J. Complex. **13**, 42–49 (1997)
7. Dodd, C., Maslov, V.: http://www.siber.com/btyacc. June (2002)
8. Donnelly, C., Stallman, R.: Bison, the YACC compatible parser generator. Version 2.1 manual., September (2005)
9. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM **13**(2), 94–102 (1970)
10. Eggert, P.: http://compilers.iecc.com/comparch/article/03-01-042 (2003)
11. JAVACC home page http://www.metamata.com/JavaCC, 2000.
12. Johnson, M.: The computational complexity of GLR parsing. In: Masaru, T. (ed.) Generalized LR parsing, pp. 35–42. Kluwer, The Netherlands (1991)
13. Johnstone, A.: `bool`—a boolean function minimiser. Technical Report CSD–TR–93–25, Computer Science Department, Royal Holloway, University of London, London (1993)
14. Johnstone, A., Scott, E.: `rdp`—an iterator based recursive descent parser generator with tree promotion operators. SIGPLAN notices, **33**(9) (1998)
15. Johnstone, A., Scott, E.: Generalised regular parsers. In: Gorel, H. (ed.) Compiler Construction, 12th International Conference CC'03, vol. 2622 of Lecture Notes in Computer Science, pp. 232–246. Springer, Berlin (2003)
16. Johnstone, A., Scott, E.: RHUL compiler group home page. http://www.cs.rhul.ac.uk/research/languages/ (2006)
17. Johnstone, A., Scott, E., Economopoulos, G.: Generalised parsing: some costs. In: Evelyn, D. (ed.) Compiler Construction, 13th International Conference CC'04, vol. 2985 of Lecture Notes in Computer Science, pp. 89–103. Springer, Berlin (2004)
18. Johnstone, A., Scott, E., Economopoulos, G.: The grammar tool box: a case study comparing GLR parsing algorithms. Electron. Notes Theoret. Comput. Sci. **110**, 97–133 (2004)
19. Johnstone, A., Scott, E., Economopoulos, G.: The GTB and PAT tools. In: Gorel, H., Eric Van, W. (eds.) Proceedings of the 4th Workshop on Language Descriptions, Tools and Applications LDTA2004. INRIA (2004)
20. Kipps, J.R.: GLR parsing in time O(n^3). In: Masaru, T. (ed.) Generalized LR parsing, pp. 43–59. Kluwer, The Netherlands (1991)
21. Kernighan, B.W., Ritchie, D.M.: The C Programming Language, second edition. Prentice-Hall, Englewood Cliffs (1988)
22. Lang, B.: Deterministic techniques for efficient non-deterministic parsers. In: Automata, Lanugages and Programming: 2nd Colloquium, vol. 14 of Lecture Notes in Computer Science, pp. 255–269. Springer, Heidelberg (1974)
23. Lee, L.: Fast context-free grammar parsing requires fast boolean matrix multiplication. J. ACM **49**, 1–15 (2002)

24. Lämmel, R., Verhoef, C.: Semi-automatic Grammar Recovery. Softw. Pract. Exp. **31**(15), 1395–1438 (2001)
25. McPeak, S., Necula, G.: Elkhound: a fast, practical GLR parser generator. In: Evelyn, D. (ed.) Compiler Construction, 13th International Conference CC'04, Lecture Notes in Computer Science. Springer, Berlin (2004)
26. Nozohoor-Farshi, R.: GLR parsing for ϵ-grammars. In: Masaru, T. (ed.) Generalized LR Parsing, pp. 60–75. Kluwer, The Netherlands (1991)
27. Parr, T.: ANTLR home page. http://www.antlr.org, Last visited: Dec (2004)
28. Rekers, J.G.: Parser generation for interactive environments. PhD thesis, Universty of Amsterdam (1992)
29. Scott, E., Johnstone, A.: Reducing non-determinism in right nulled GLR parsers. Acta Informatica **40**, 459–489 (2004)
30. Scott, E., Johnstone, A.: Right nulled GLR parsers. ACM Trans. Program. Lang. Syst. pp. 1–43 (2006)
31. Scott, E.A., Johnstone, A.I.C., Economopoulos, G.R.: BRN-table based GLR parsers. Technical Report TR-03-06, Computer Science Department, Royal Holloway, University of London, London (2003)
32. Scott, E., Johnstone, A., Hussain, S.S.: Tomita-style generalised LR parsers. Updated Version. Technical Report TR-00-12, Computer Science Department, Royal Holloway, University of London, London, December (2000)
33. Valiant, L.: General context-free recognition in less than cubic time. J. Comput. Syst. Sci. **10**, 308–315 (1975)
34. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the ASF+SDF compiler. ACM Trans. Program. Lang. Syst. **24**(4), 334–368 (2002)
35. Visser, E.: Syntax definition for language prototyping. PhD thesis, Universty of Amsterdam (1997)
36. Younger, D.H.: Recognition of context-free languages in time n^3. Inform. Control **10**(2), 189–208 (1967)